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)

In [1]:
```
# 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"""
```

OUTPUT :-

In [2]:
```
>>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data', header=None)
>>>
>>> df.tail()
```

Out[2]:

	0	1	2	3	4
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

In [3]:
```
>>> df.iloc[145:150, 0:5]
```

Out[3]:

	0	1	2	3	4
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

```
In [4]: >>> import matplotlib.pyplot as plt
        >>> import numpy as np
        >>>
        >>> y = df.iloc[0:100, 4].values
        >>> y
```

```
Out[4]: array(['Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
               'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
               'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
               'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
               'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
               'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
               'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
               'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
               'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
               'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
               'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
               'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
               'Iris-setosa', 'Iris-setosa', 'Iris-versicolor', 'Iris-versicolor',
               'Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor',
               'Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor',
               'Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor',
               'Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor',
               'Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor',
               'Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor',
               'Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor',
               'Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor',
               'Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor',
               'Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor',
               'Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor',
               'Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor',
               'Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor',
               'Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor',
               'Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor',
               'Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor'],
              dtype=object)
```

```
In [5]: >>> y = np.where(y == 'Iris-setosa', -1, 1)
        >>> y
```

```
Out[5]: array([-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
               -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
               -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,  1,
                1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
                1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
                1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1])
```

```
In [6]: >>> X = df.iloc[0:100, [0, 2]].values
        >>> X
```

```
Out[6]: 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],
```

Out6 full :-

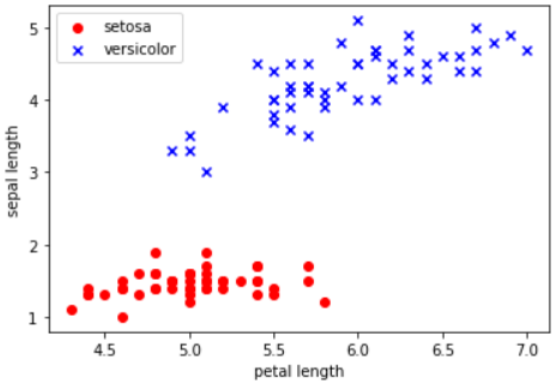
```
array([[5.1, 1.4],
       [4.9, 1.4],
       [4.7, 1.3],
       [4.6, 1.5],
       [5. , 1.4],
       [5.4, 1.7],
       [4.6, 1.4],
       [5. , 1.5],
       [4.4, 1.4],
       [4.9, 1.5],
       [5.4, 1.5],
       [4.8, 1.6],
       [4.8, 1.4],
       [4.3, 1.1],
       [5.8, 1.2],
       [5.7, 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 :-

```
In [7]: >>> plt.scatter(X[:50, 0], X[:50, 1], color='red', marker='o', label='setosa')
        >>> plt.scatter(X[50:100, 0], X[50:100, 1], color='blue', marker='x', label='versicolor')
        >>> plt.xlabel('petal length')
        >>> plt.ylabel('sepal length')
        >>> plt.legend(loc='upper left')
        >>> plt.show()
```

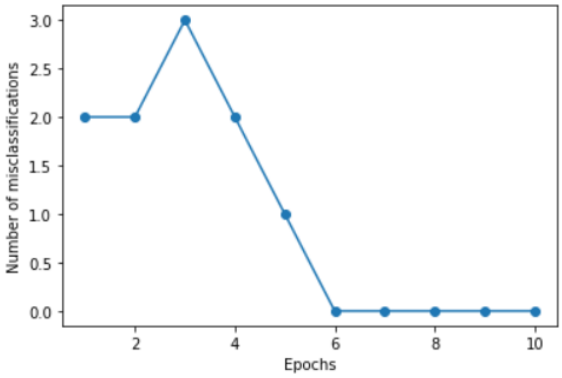


Confusion :-

```
In [16]: from sklearn.metrics import confusion_matrix
         from sklearn.metrics import classification_report

         y_true = y
         y_pred = pn.predict(X)
         print(confusion_matrix(y_true, y_pred))

         # outcome values order in sklearn
         tp, fn, fp, tn = confusion_matrix(y_true,y_pred,labels=[-1,1]).reshape(-1)
         print('Outcome values : \n', tp, fn, fp, tn)

         # classification report for precision, recall f1-score and accuracy
         matrix = classification_report(y_true,y_pred,labels=[-1,1])
         print('Classification report : \n',matrix)
```

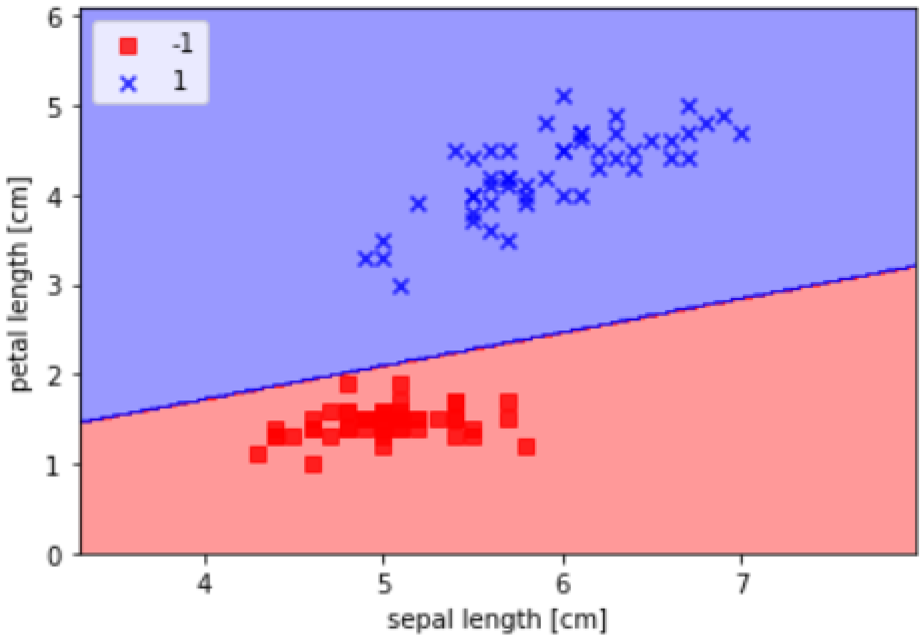
```
[[50  0]
 [ 0 50]]
Outcome values :
 50 0 0 50
Classification report :
               precision    recall  f1-score   support

          -1       1.00      1.00      1.00        50
           1       1.00      1.00      1.00        50

    accuracy                           1.00       100
   macro avg       1.00      1.00      1.00       100
weighted avg       1.00      1.00      1.00       100
```

In [14]:
```
pn = Perceptron(0.1, 10)
pn.fit(X, y)
plt.plot(range(1, len(pn.errors) + 1), pn.errors, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Number of misclassifications')
plt.show()
```



Decision :-

In [17]:
```
from matplotlib.colors import ListedColormap
def plot_decision_regions(X, y, classifier, resolution=0.02):
    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:,  0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
    np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # plot class samples
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
        alpha=0.8, c=cmap(idx),
        marker=markers[idx], label=cl)

plot_decision_regions(X, y, classifier=pn)
plt.xlabel('sepal length [cm]')
plt.ylabel('petal length [cm]')
plt.legend(loc='upper left')
plt.show()
```



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)
```

In [21]:
```
import numpy as np
def sigmoid (x):
    return 1/(1 + np.exp(-x))
def sigmoid_derivative(x):
    return x * (1 - x)
def mlp(inputs,expected_output,epochs=3500,lr=0.25,inputLayerNeurons=2,hiddenLayerNeurons=2,
        outputLayerNeurons=1):
    hidden_weights = np.random.uniform(size=(inputLayerNeurons,hiddenLayerNeurons))
    hidden_bias =-1*np.random.uniform(size=(1,hiddenLayerNeurons))
    output_weights = np.random.uniform(size=(hiddenLayerNeurons,outputLayerNeurons))
    output_bias =-1*np.random.uniform(size=(1,outputLayerNeurons))
    print("Initial hidden weights: ",end='')
    print(*hidden_weights)
    print("Initial hidden biases: ",end='')
    print(*hidden_bias)
    print("Initial output weights: ",end='')
```

OUTPUT :-

```
Initial hidden weights: [0.81452573 0.07171115] [0.96725412 0.97048616]
Initial hidden biases: [-0.52163582 -0.87876933]
Initial output weights: [0.60110026] [0.43969818]
Initial output biases: [-0.80451746]

Final hidden weights: [5.71025716 3.79671337] [5.7507781  3.80550097]
Final hidden bias: [-2.38344014 -5.81978818]
Final output weights: [7.46779392] [-8.02325461]
Final output bias: [-3.39685104]

Output from neural network after 3,500 epochs:
[[0]
 [1]
 [1]
 [0]]
```

In [22]:
```python
import numpy as np
import mlp
```

In [23]:
```python
xordata = np.array([[0,0,0],[0,1,1],[1,0,1],[1,1,0]])
```

In [24]:
```python
q = mlp.mlp(xordata[:,0:2],xordata[:,2:3],2,outtype='logistic')
q.mlptrain(xordata[:,0:2],xordata[:,2:3],0.25,5001)
q.confmat(xordata[:,0:2],xordata[:,2:3])
```

```
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
```

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: 0.00079294018306812
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?

In [28]:
```
import pandas as pd
from sklearn.datasets import load_iris
iris = load_iris()
```

In [29]:
```
from sklearn.model_selection import train_test_split
datasets = train_test_split(iris.data, iris.target,
                            test_size=0.2)
train_data, test_data, train_labels, test_labels = datasets
```

In [30]:
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(train_data)
train_data = scaler.transform(train_data)
test_data = scaler.transform(test_data)
print(train_data[:3])
```

```
[[-1.25605453 -0.13238368 -1.31801426 -1.15697292]
 [-1.01450558  0.53885753 -1.31801426 -1.28758134]
 [-0.41063321 -1.69861319  0.1448831   0.14911128]]
```

In [31]:
```
from sklearn.neural_network import MLPClassifier
mlp = MLPClassifier(hidden_layer_sizes=(10, 5), max_iter=1000)
mlp.fit(train_data, train_labels)
```

Out[31]:
```
MLPClassifier(hidden_layer_sizes=(10, 5), max_iter=1000)
```

In [32]:
```
from sklearn.metrics import accuracy_score
predictions_train = mlp.predict(train_data)
print(accuracy_score(predictions_train, train_labels))
predictions_test = mlp.predict(test_data)
print(accuracy_score(predictions_test, test_labels))
```

```
0.975
1.0
```

In [33]:
```
from sklearn.metrics import confusion_matrix
confusion_matrix(predictions_train, train_labels)
```

Out[33]:
```
array([[41,  0,  0],
       [ 0, 37,  1],
       [ 0,  2, 39]])
```

In [34]:
```
confusion_matrix(predictions_test, test_labels)
```

Out[34]:
```
array([[ 9,  0,  0],
       [ 0, 11,  0],
       [ 0,  0, 10]])
```

In [35]:
```
from sklearn.metrics import classification_report
print(classification_report(predictions_test, test_labels))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         9
           1       1.00      1.00      1.00        11
           2       1.00      1.00      1.00        10

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```