MACHINE LEARNING

LAB-5

2018103073 **SURYA N**

1. MULTI LAYER PERCEPTRON:

Importing the required libraries.

```
[1]: import numpy as np
```

```
[2]: def sigmoid (x):
         return 1/(1 + np.exp(-x))
```

```
[3]: def sigmoid_derivative(x):
         return x * (1 - x)
```

```python
def mlp(inputs,expected_output,epochs=10000,lr=0.5,inputLayerNeurons=2,hiddenLayerNeurons=2,outputLayerNeurons=1):
    hidden_weights = np.random.uniform(size=(inputLayerNeurons,hiddenLayerNeurons))
    hidden_bias =np.random.uniform(size=(1,hiddenLayerNeurons))
    output_weights = np.random.uniform(size=(hiddenLayerNeurons,outputLayerNeurons))
    output_bias = 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='')
    print(*output_weights)
    print("Initial output biases: ",end='')
    print(*output_bias)

    for _ in range(epochs):
        hidden_layer_activation = np.dot(inputs,hidden_weights)
        hidden_layer_activation += hidden_bias
        hidden_layer_output = sigmoid(hidden_layer_activation)

        output_layer_activation = np.dot(hidden_layer_output,output_weights)
        output_layer_activation += output_bias
        predicted_output = sigmoid(output_layer_activation)

        error = expected_output - predicted_output
        d_predicted_output = error * sigmoid_derivative(predicted_output)

        error_hidden_layer = d_predicted_output.dot(output_weights.T)
        d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)

        output_weights += hidden_layer_output.T.dot(d_predicted_output) * lr
        output_bias += np.sum(d_predicted_output,axis=0,keepdims=True) * lr
        hidden_weights += inputs.T.dot(d_hidden_layer) * lr
        hidden_bias += np.sum(d_hidden_layer,axis=0,keepdims=True) * lr
    return hidden_weights,hidden_bias,output_weights,output_bias,predicted_output
```

Random array is created and MLP is applied to it.

```python
inputs = np.array([[0,0],[0,1],[1,0],[1,1]])
expected_output = np.array([[0],[1],[1],[0]])
```

```python
hidden_weights,hidden_bias,output_weights,output_bias,predicted_output = mlp(inputs,expected_output)
```

```
Initial hidden weights: [0.09069827 0.64224379] [0.46950825 0.27398655]
Initial hidden biases: [0.06397014 0.94643633]
Initial output weights: [0.81775456] [0.53614158]
Initial output biases: [0.21095038]
```

7]:
```python
print("Final hidden weights: ",end='')
print(hidden_weights)
print("Final hidden bias: ",end='')
print(hidden_bias)
print("Final output weights: ",end='')
print(output_weights)
print("Final output bias: ",end='')
print(output_bias)
```

```
Final hidden weights: [[4.58927602 6.44884421]
 [4.59113773 6.45643385]]
Final hidden bias: [[-7.04340524 -2.8635691 ]]
Final output weights: [[-10.25943731]
 [  9.57294339]]
Final output bias: [[-4.43135451]]
```

8]:
```python
print("\nOutput from neural network after 10,000 epochs: ",end='')
print(predicted_output)
```

```

Output from neural network after 10,000 epochs: [[0.01938924]
 [0.9832398 ]
 [0.98323143]
 [0.01737316]]
```

The accuracy is been calculated.

2. MULTI LAYER PERCEPTRON USING XOR:

Importing the required libraries.

If the input patterns for XOR gate are plotted according to their outputs, it is seen that these points are not linearly separable. Hence the neural network has to be modeled to separate these input patterns using decision planes. This is achieved by using the concept of hidden layers. To Implement XOR gate, we will be using a Sigmoid Neuron as nodes in the neural network.

```
import numpy as np
import pandas as pd
```

```
data = pd.read_csv("diabetes2.csv")
data = data.sample(frac=1).reset_index(drop=True)
data.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	7	159	64	0	0	27.4	0.294	40	0
1	2	95	54	14	88	26.1	0.748	22	0
2	8	105	100	36	0	43.3	0.239	45	1
3	0	94	70	27	115	43.5	0.347	21	0
4	0	86	68	32	0	35.8	0.238	25	0

The data is converted into array.

```
X = np.array(data)[:,1:-1]
```

```
print(X)
```

```
[[1.59e+02 6.40e+01 0.00e+00 ... 2.74e+01 2.94e-01 4.00e+01]
 [9.50e+01 5.40e+01 1.40e+01 ... 2.61e+01 7.48e-01 2.20e+01]
 [1.05e+02 1.00e+02 3.60e+01 ... 4.33e+01 2.39e-01 4.50e+01]
 ...
 [1.06e+02 7.00e+01 2.80e+01 ... 3.42e+01 1.42e-01 2.20e+01]
 [1.17e+02 6.20e+01 1.20e+01 ... 2.97e+01 3.80e-01 3.00e+01]
 [1.05e+02 7.20e+01 2.90e+01 ... 3.69e+01 1.59e-01 2.80e+01]]
```

The accuracy is been calculated:

```
print("Testing Accuracy: {}".format(Accuracy(X_test, Y_test, weights, display = True)))
```

```
Input:
 [96.     64.     27.     87.     33.2    0.289 21.    ]
Predicted:
 [1, 0]
Actual:
 [1.0, 0.0]


Input:
 [129.      0.      0.      0.     38.5     0.304  41.    ]
Predicted:
 [1, 0]
Actual:
 [1.0, 0.0]


Input:
 [127.     80.     37.    210.     36.3     0.804  23.    ]
Predicted:
 [1, 0]
Actual:
 [1.0, 0.0]


Input:
 [130.     64.      0.      0.     23.1     0.314  22.    ]
Predicted:
 [1, 0]
Actual:
 [1.0, 0.0]
```

Calculating the final weights

```
print("Final weights:\n",weights)
```

```
Final weights:
 [matrix([[ 0.96065986,  3.2871523 , -2.35888122,  0.03552078,  1.82221295
         -0.97423125, -0.02374021, -0.91763199],
        [ 0.84348617,  0.56375397,  0.37839956,  0.74503813,  0.85311255,
         -0.82999787, -0.50727101,  0.87742271],
        [ 0.08245552, -0.39162808, -0.66480653,  0.94276714, -0.97409234,
          0.51380313,  0.6632706 , -0.83862125],
        [-0.46873168, -0.80135391,  0.6174883 , -0.48937134, -0.73985205,
          0.48861742,  0.08603182, -0.67767906],
        [ 0.07821509,  1.18252925,  1.08735476, -0.70994612,  0.41087396,
         -0.13666399,  0.06100686,  0.96486982]]), matrix([[ 0.52076355, -
          1.70253029],
        [-1.36121246,  2.52312713, -0.23143287, -0.6928267 ,  2.6383596 ,
         -1.56783576]])]
```

```
from sklearn.preprocessing import OneHotEncoder
one_hot_encoder = OneHotEncoder(sparse=False,categories='auto')

Y = np.array(data)[:,-1]
Y = one_hot_encoder.fit_transform(np.array(Y).reshape(-1, 1))
```

```python
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.25)
```

```python
def NeuralNetwork(X_train, Y_train, X_val=None, Y_val=None, epochs=10, nodes=[], lr=0.07):
    hidden_layers = len(nodes) - 1
    weights = InitializeWeights(nodes)

    for epoch in range(1, epochs+1):
        weights = Train(X_train, Y_train, lr, weights)

        if(epoch % 50 == 0):
            print("Epoch {}".format(epoch))
            print("Training Accuracy:{}".format(Accuracy(X_train, Y_train, weights)))
            if X_val is not None:
                print("Validation Accuracy:{}".format(Accuracy(X_val, Y_val, weights)))

    return weights
```

```python
def InitializeWeights(nodes):
    layers = len(nodes)
    weights = []

    for i in range(1, layers):
        w = [[np.random.uniform(-1, 1) for k in range(nodes[i-1] + 1)]
              for j in range(nodes[i])]
        weights.append(np.matrix(w))

    return weights
```

```python
def ForwardPropagation(x, weights, layers):
    activations, layer_input = [x], x
    for j in range(layers):
        activation = Sigmoid(np.dot(layer_input, weights[j].T))
        activations.append(activation)
        layer_input = np.append(1, activation)

    return activations
```

```python
def BackPropagation(y, activations, weights, layers):
    outputFinal = activations[-1]
    error = np.matrix(y - outputFinal)

    for j in range(layers, 0, -1):
        currActivation = activations[j]

        if(j > 1):
            prevActivation = np.append(1, activations[j-1])
        else:
            prevActivation = activations[0]

        delta = np.multiply(error, SigmoidDerivative(currActivation))
        weights[j-1] += lr * np.multiply(delta.T, prevActivation)

        w = np.delete(weights[j-1], [0], axis=1)
        error = np.dot(delta, w)

    return weights
```

```python
def Train(X, Y, lr, weights):
    layers = len(weights)
    for i in range(len(X)):
        x, y = X[i], Y[i]
        x = np.matrix(np.append(1, x))
        activations = ForwardPropagation(x, weights, layers)
        weights = BackPropagation(y, activations, weights, layers)

    return weights
```

```python
def Sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

def SigmoidDerivative(x):
    return np.multiply(x, 1.0-x)
```

```python
def Predict(item, weights):
    layers = len(weights)
    item = np.append(1, item)

    activations = ForwardPropagation(item, weights, layers)

    outputFinal = activations[-1].A1
    m, index = outputFinal[0], 0
    for i in range(1, len(outputFinal)):
        if(outputFinal[i] > m):
            m, index = outputFinal[i], i

    y = [0 for i in range(len(outputFinal))]
    y[index] = 1
    return y
```

```python
def Accuracy(X, Y, weights, display=False):
    correct = 0

    for i in range(len(X)):
        x, y = X[i], list(Y[i])
        guess = Predict(x, weights)
        if display == True:
            print("\n\nInput:\n",x,"\nPredicted:\n",guess,"\nActual:\n",y)
        if(y == guess):
            correct += 1
        elif display == True:
            print("mispredicted")

    return correct / len(X)
```

```python
layers = [len(X[0]),5,len(Y[0])]
print(layers)
lr, epochs = 0.1,1000
weights = NeuralNetwork(X_train, Y_train, epochs=epochs, nodes=layers, lr=lr)
```

```
/srv/conda/envs/notebook/lib/python3.7/site-packages/ipykernel_launcher.py:2:
```

```
[7, 5, 2]
Epoch 50
Training Accuracy:0.6475694444444444
Epoch 100
Training Accuracy:0.6475694444444444
Epoch 150
Training Accuracy:0.6475694444444444
Epoch 200
Training Accuracy:0.6475694444444444
Epoch 250
Training Accuracy:0.6475694444444444
Epoch 300
Training Accuracy:0.6475694444444444
Epoch 350
Training Accuracy:0.6475694444444444
Epoch 400
Training Accuracy:0.6475694444444444
Epoch 450
Training Accuracy:0.6475694444444444
Epoch 500
```

```python
print("Final weights:\n",weights)
```

```
Final weights:
 [matrix([[ 0.96065986,  3.2871523 , -2.35888122,  0.03552078,  1.8222129
         -0.97423125, -0.02374021, -0.91763199],
        [ 0.84348617,  0.56375397,  0.37839956,  0.74503813,  0.85311255,
         -0.82999787, -0.50727101,  0.87742271],
        [ 0.08245552, -0.39162808, -0.66480653,  0.94276714, -0.97409234,
          0.51380313,  0.6632706 , -0.83862125],
        [-0.46873168, -0.80135391,  0.6174883 , -0.48937134, -0.73985205,
          0.48861742,  0.08603182, -0.67767906],
        [ 0.07821509,  1.18252925,  1.08735476, -0.70994612,  0.41087396,
         -0.13666399,  0.06100686,  0.96486982]]), matrix([[ 0.52076355, -
          1.70253029],
        [-1.36121246,  2.52312713, -0.23143287, -0.6928267 ,  2.6383596 ,
         -1.56783576]])]
```

```python
print("Testing Accuracy: {}".format(Accuracy(X_test, Y_test, weights, display = True)))
```

```
Input:
 [96.    64.    27.    87.    33.2    0.289 21.   ]
Predicted:
 [1, 0]
Actual:
 [1.0, 0.0]


Input:
 [129.      0.      0.      0.     38.5     0.304  41.    ]
Predicted:
 [1, 0]
Actual:
 [1.0, 0.0]


Input:
 [127.     80.     37.    210.     36.3     0.804  23.    ]
Predicted:
 [1, 0]
Actual:
 [1.0, 0.0]


Input:
 [130.     64.      0.      0.     23.1     0.314  22.    ]
Predicted:
 [1, 0]
Actual:
 [1.0, 0.0]
```

```python
import sklearn
Y_result = []
for x in X_test:
    guess = Predict(x,weights)
    Y_result.append(guess)
print("R2 score : %f" % sklearn.metrics.r2_score(Y_test,Y_result))
```

```
R2 score : -0.511811
```