2019103555 PRANAVA RAMAN B M S 25/10/2021

**WEEK - 08 - MLP**

1. **Use Pima Indians Diabetes dataset.**

**Implement MLP. Use the sigmoidal activation function used in the algorithm in your text book. Fix the number of hidden neurons based on experimentation. With the same number of hidden neurons experiment using three different activation functions.**

Definition of class MLP:-

import numpy as np

def tanh(z):

    return (np.exp(z) - np.exp(-z)) / (np.exp(z) + np.exp(-z))

class mlp:

    """A Multi-Layer Perceptron"""

    def \_\_init\_\_(

        self, inputs, targets, nhidden, beta=1, momentum=0.9, outtype="logistic"

    ):

        """Constructor"""

        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=10000

    ):

        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

        print("No. of neurons in hidden layers = ", self.nhidden)

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

            (old\_val\_error2 - old\_val\_error1) > 0.001

        ):

            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, error = ", new\_val\_error)

        return new\_val\_error

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

        """Train the neural network"""

        # 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)

            # Different types of output neurons and their activation functions

            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

                )

            elif self.outtype == "tanh":

                deltao = (

                    (self.outputs - targets)

                    \* (1.0 - np.power(self.outputs, 2))

                )

            else:

                print("error")

            # hidden network delta

            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

            return error

    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)

        elif self.outtype == "tanh":

            return tanh(outputs)

        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)

                )

        output = cm

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

        return output

**Helper functions:-**

To display confusion matrix-

import seaborn as sn

import pandas as pd

import matplotlib.pyplot as plt

def displayConfusionMatrix(cm, plt):

    out\_cm = np.array(cm)

    df\_cm = pd.DataFrame(out\_cm)

    plt.figure(figsize=(10,7))

    sn.set(font\_scale=1)  # for label size

    sn.heatmap(df\_cm, annot=True, annot\_kws={"size": 14})  # font size

To show classification report:-

from sklearn.metrics import classification\_report

def printClassificationReport(network, test, testt):

    targets=testt

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

    nclasses = np.shape(targets)[1]

    output = network.mlpfwd(inputs)

    if nclasses == 1:

        nclasses = 2

        output = np.where(output > 0.5, 1, 0)

    else:

        # 1-of-N encoding

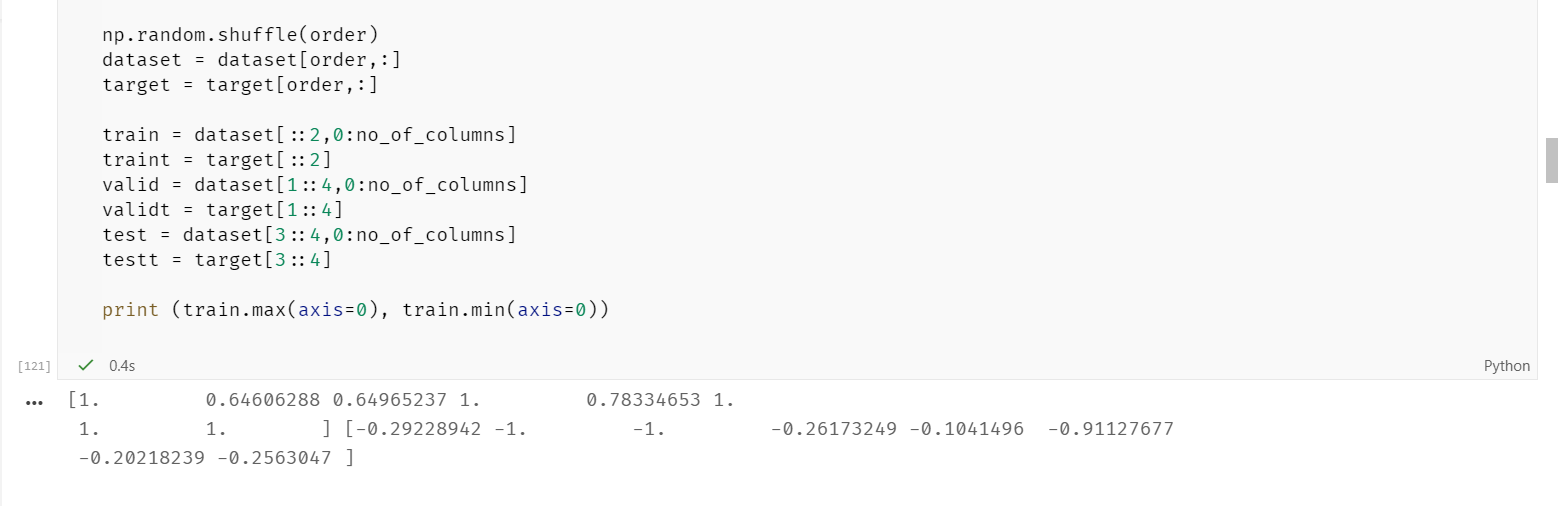
        output = np.argmax(output, 1)

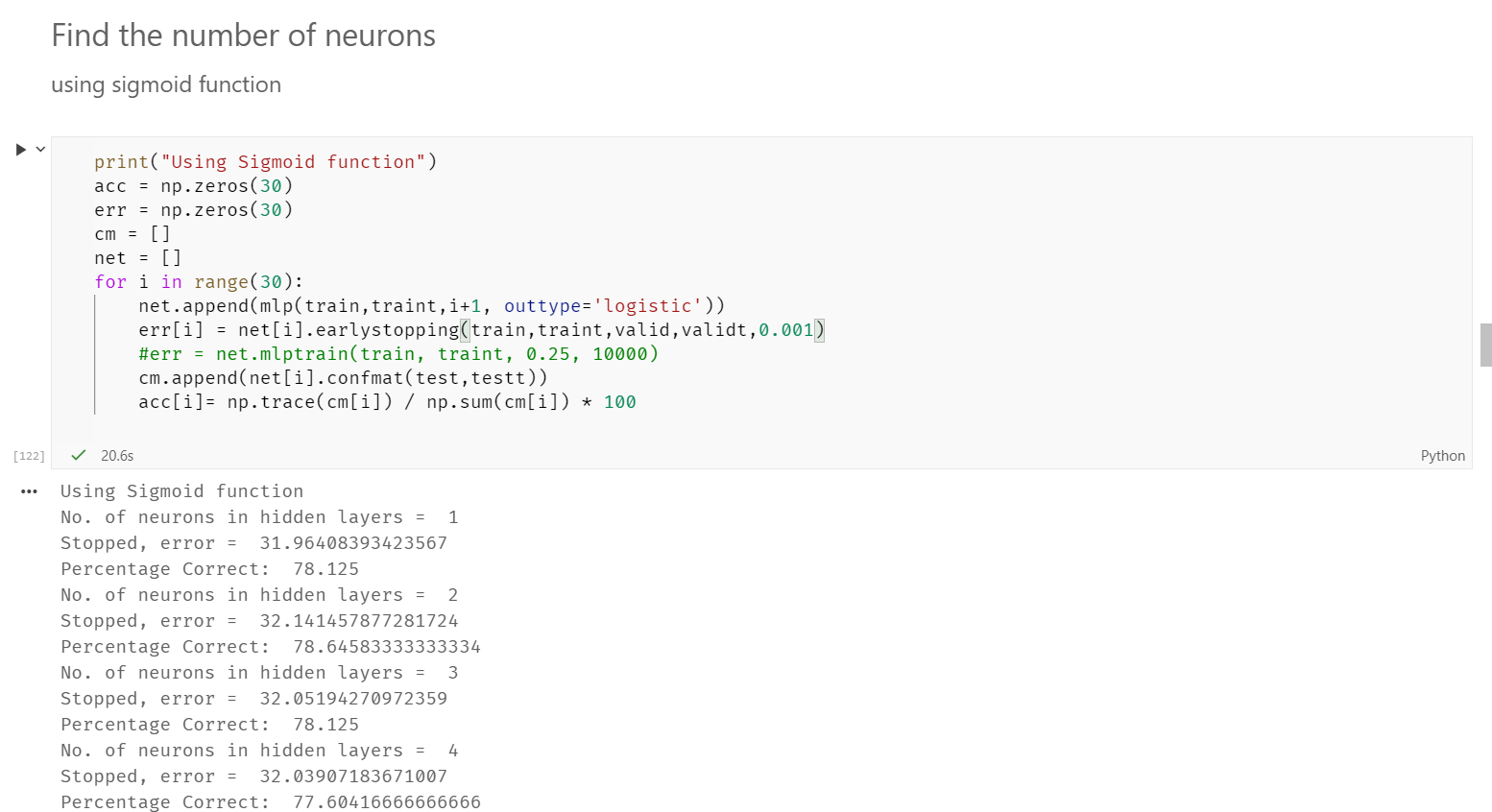
        targets = np.argmax(targets, 1)

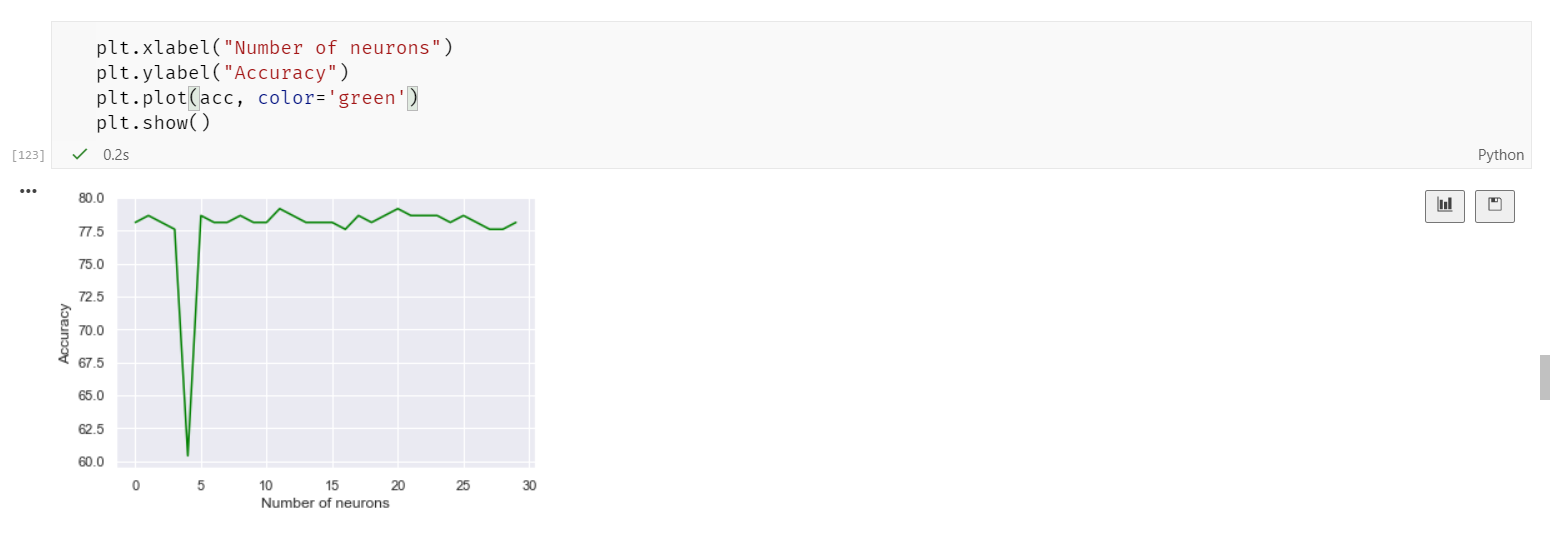
    print(classification\_report(targets, output))

Including ‘pima-indian-diabetes’ dataset:-

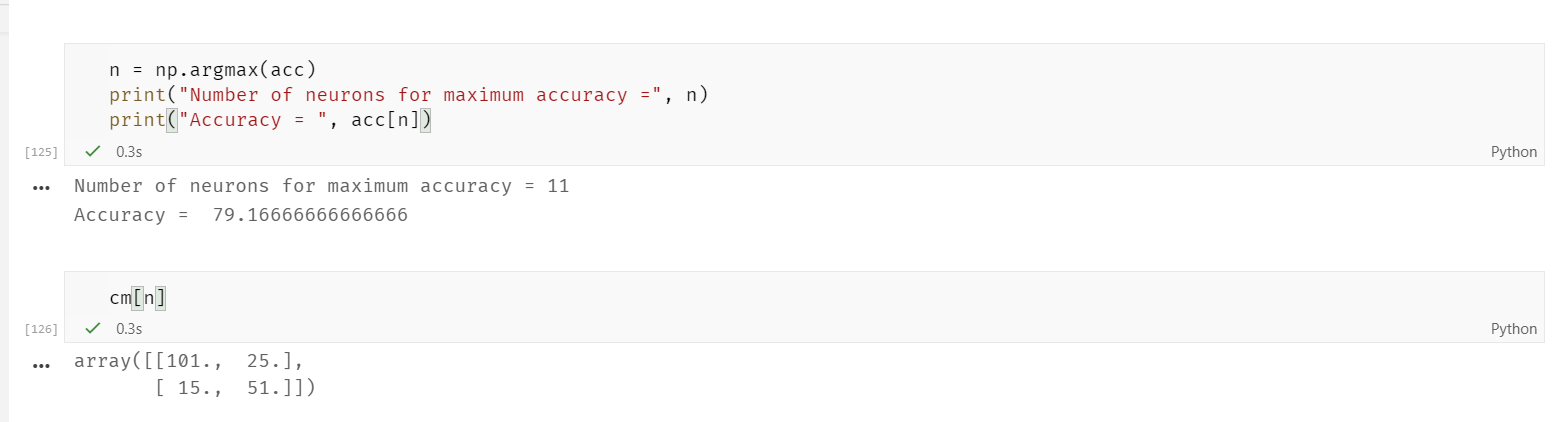




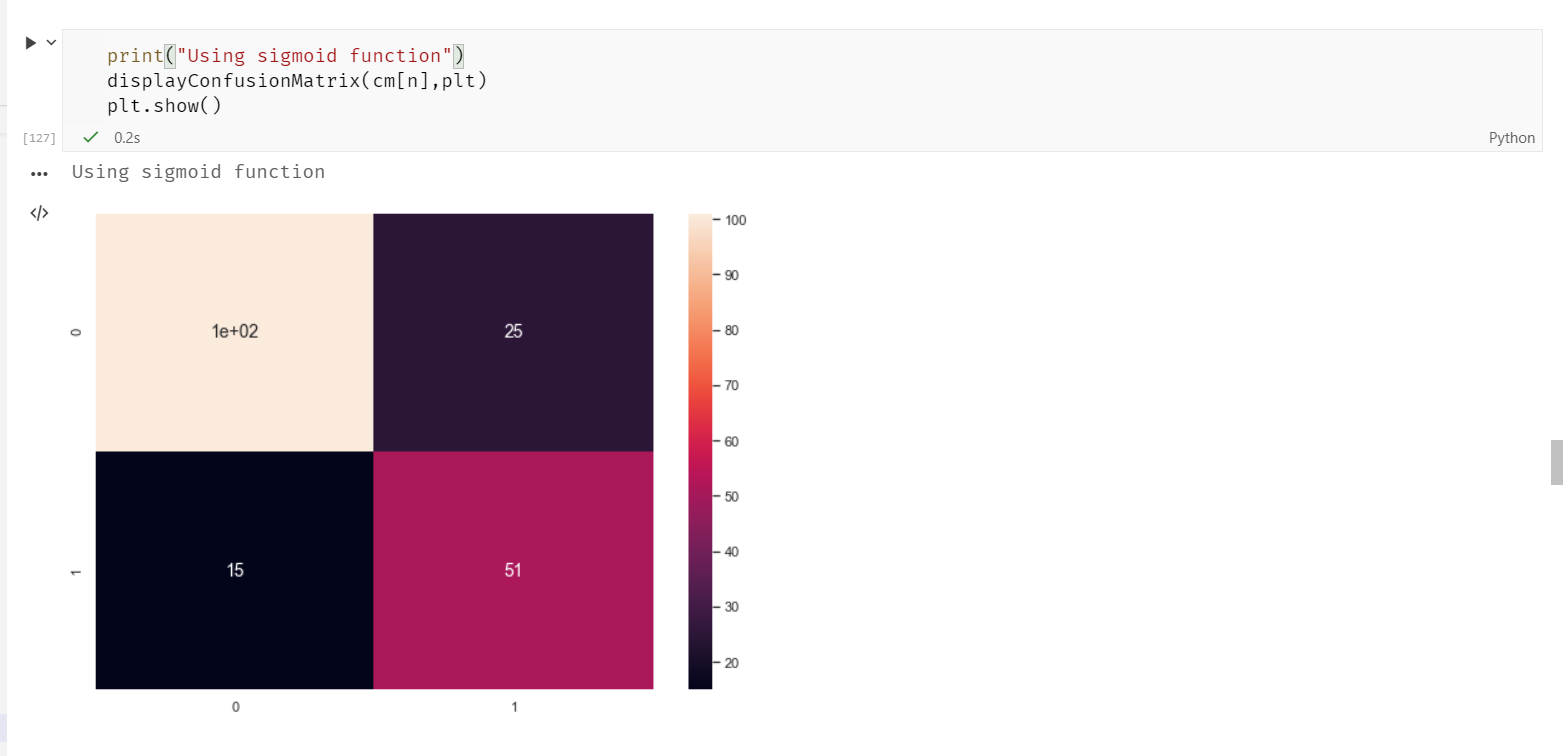


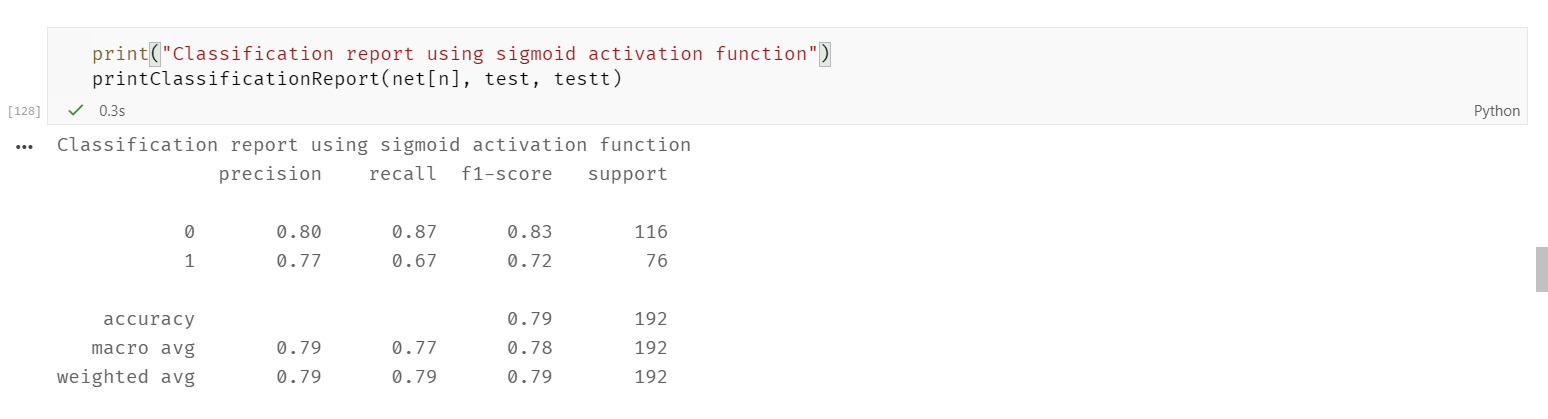






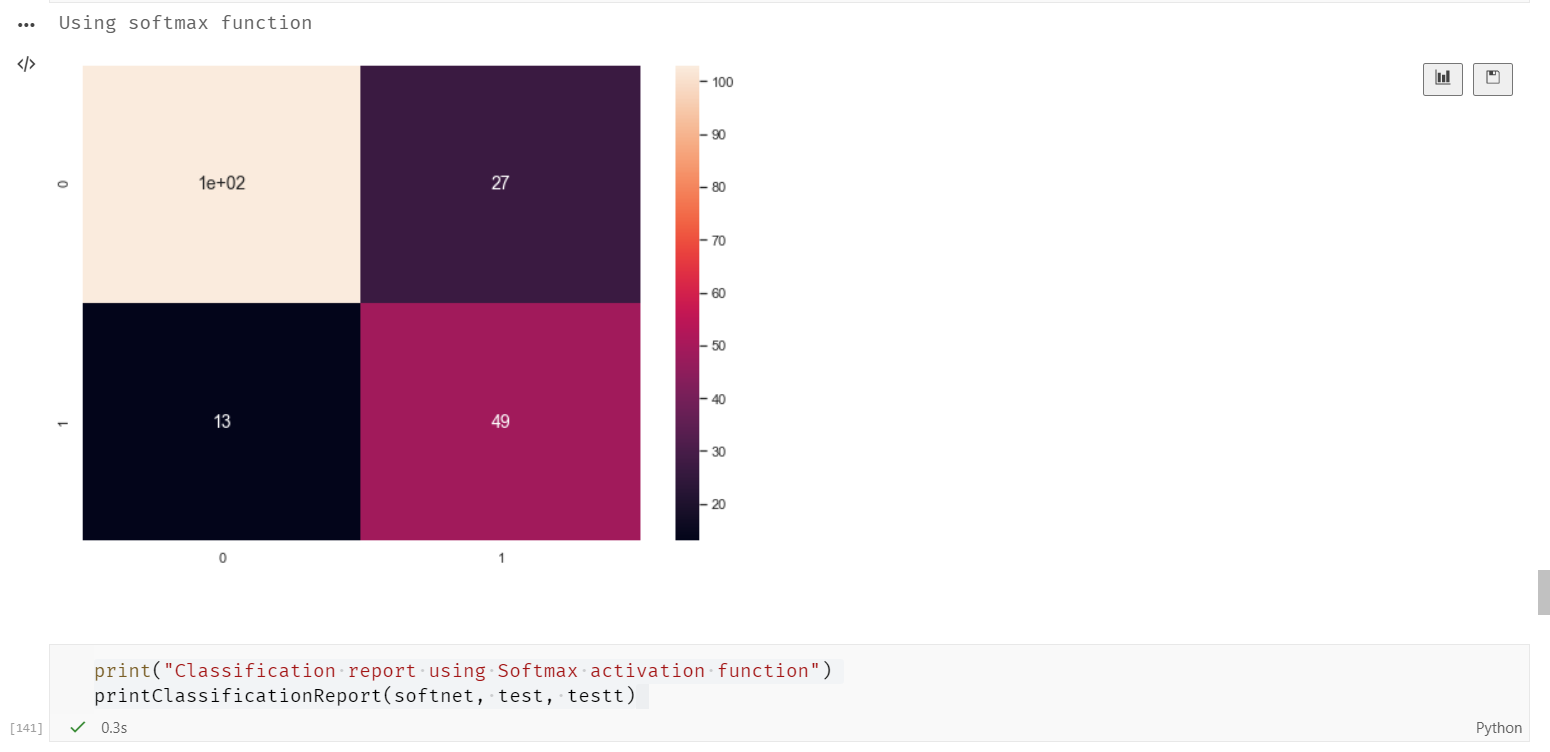
Using 11 neurons gave the maximum accuracy for sigmoid activation function.

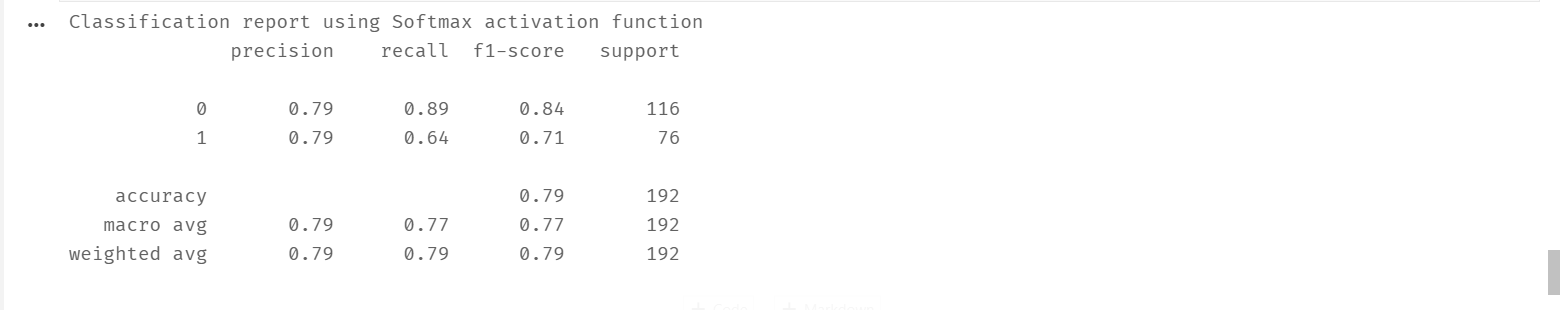




**Trying out different activation functions:-**

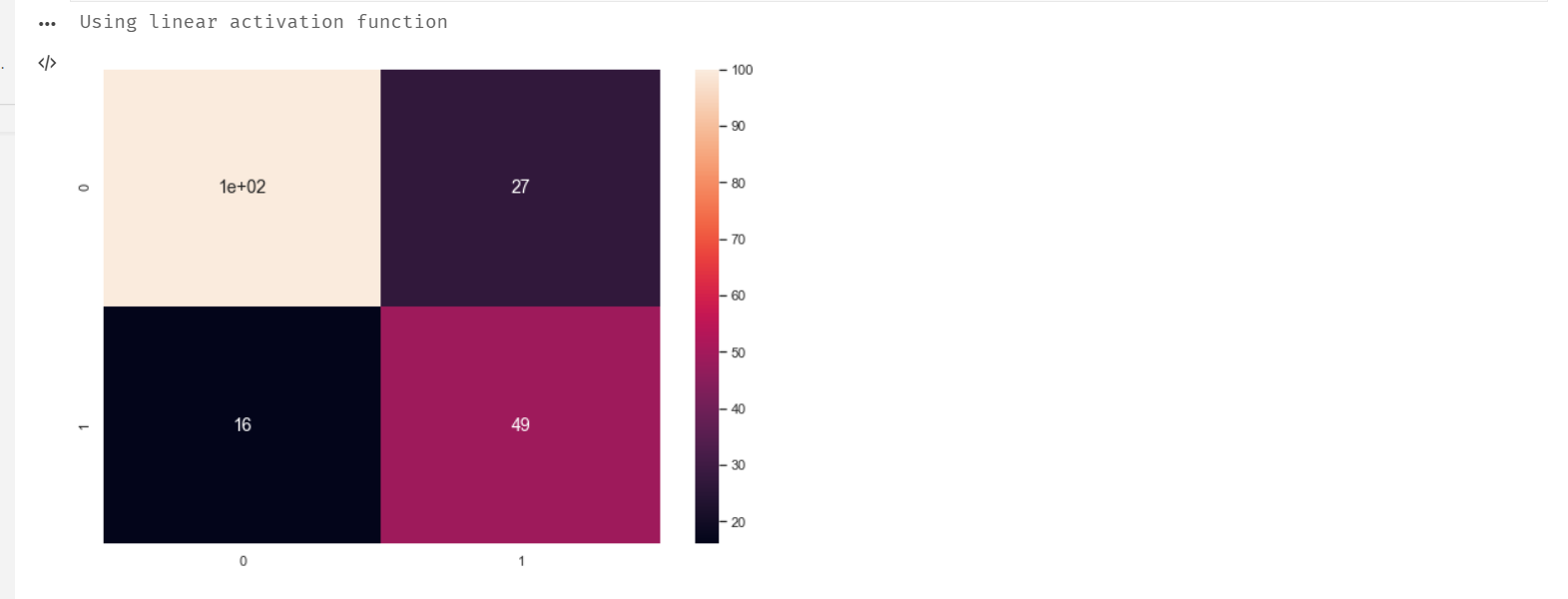
1. **Softmax activation function**

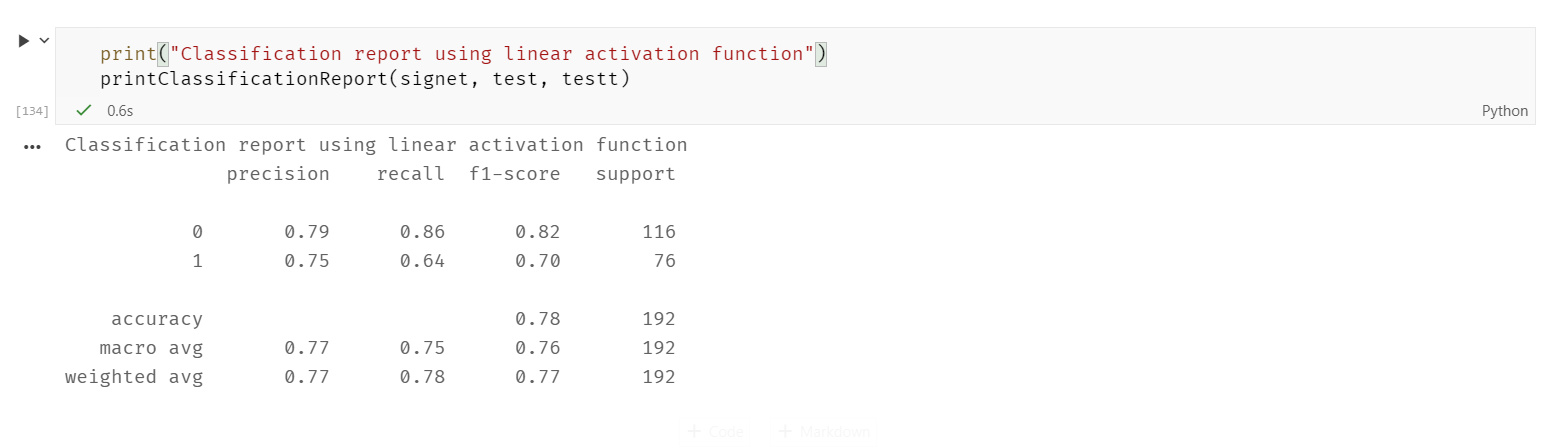




1. **Linear activation function**

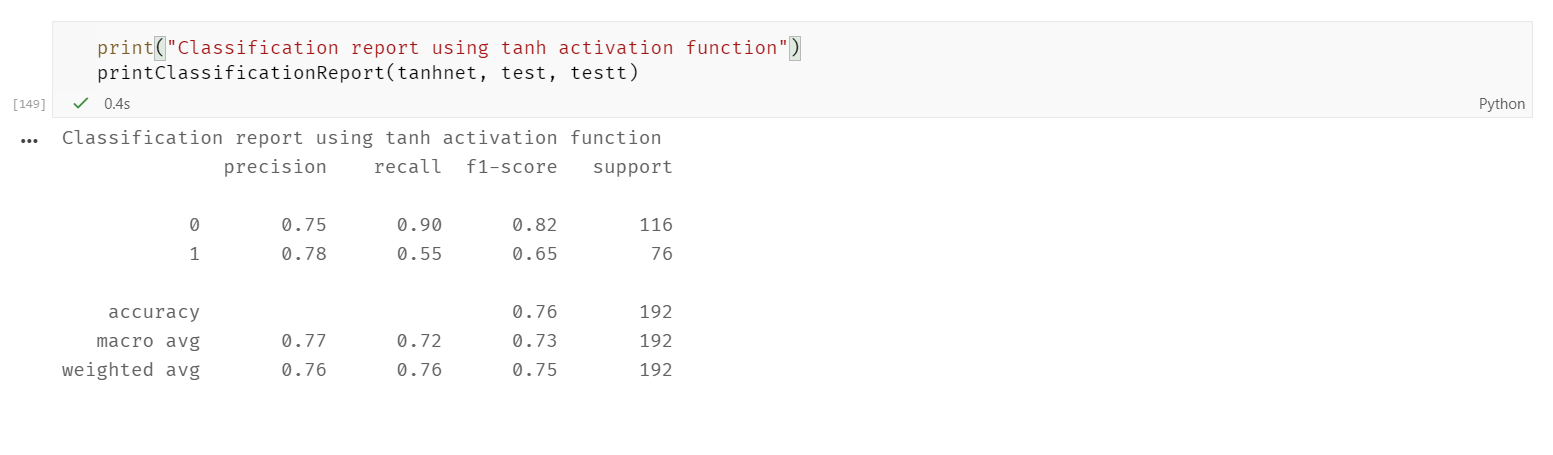
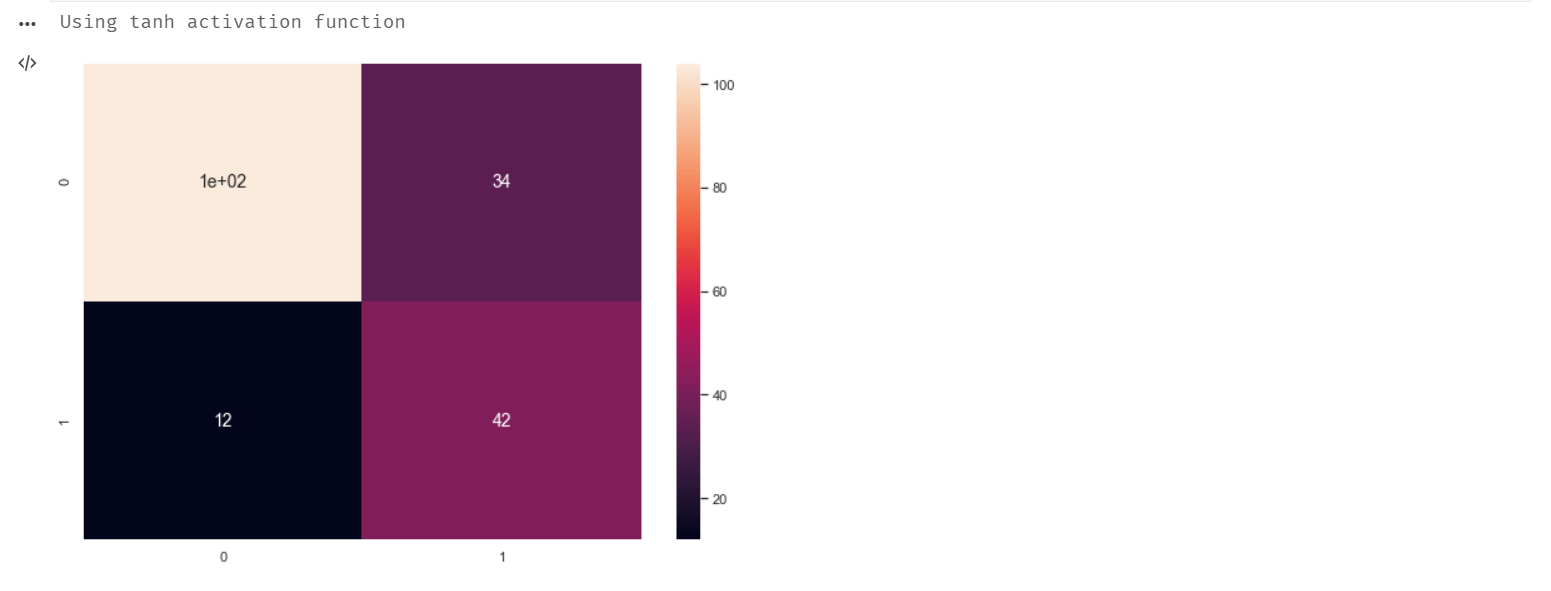






1. **Tanh activation function**





Result:-

For the given dataset, both Sigmoid and Softmax give the highest accuracy of about 79% while linear activation gives 77% accuracy and tanh follows very closely by 76% of test accuracy.

1. **Choose a dataset suitable for regression and apply regression using MLP**

Using real estate dataset,

