12/4/23

EXPERIMENT - 6A

AIM

To implement the Linear Regression algorithm in Python

ALGORITHM

* For the given dataset, compute the values of x^2, xy and hence the summation of x, y, x^2 and xy values respectively
* Find slope 'm' & y-intercept 'c' using the formula: $m = \frac{n\sum xy - \sum x \sum y}{n\sum x^2 - (\sum x)^2}$, $c = \frac{\sum y - m\sum x}{n}$
* Form the linear regression line : $y = mx + c$
* Hence the model is ready; To check accuracy, compute SSR, SST, SSE and R^2 values

THEORY

Linear Regression is a supervised learning algorithm to predict a dependent variable value (y) based on a given independent variable (x). The regression line is the best fit line for the model.

OUTPUT

```
Data read from dataset.csv :

   Temperature  Ice Cream Sales
0           20               13
1           25               21
2           30               25
3           35               35
4           40               38

Linear Regression equation : y = ( 1.28 ) x + ( -12.0 )

Sum of Squares Total ( SST ) =  419.2
Sum of Squares Regression ( SSR ) =  409.6
Sum of Squares Error ( SSE ) =  9.6
R-squared =  0.977
```

CODE

```
import pandas as pd
import numpy as np

def calculate_Slope ( x , y , n ):
    numerator = ( n * np.sum ( x * y ) ) - ( np.sum ( x ) * np.sum ( y ) )
    denominator = ( n * np.sum ( x ** 2 ) ) - ( np.sum ( x ) ** 2 )
    return numerator / denominator

def calculate_Y_Intercept ( x , y , m , n ):
    return ( np.sum ( y ) - ( m * np.sum ( x ) ) ) / n

def evaluate_Model ( df ):
    n = len ( df )
    x = df [ df.columns [ 0 ] ]
    y = df [ df.columns [ 1 ] ]
    y_mean = np.mean ( y )
    m = round ( calculate_Slope ( x , y , n ) , 2 )
    c = round ( calculate_Y_Intercept ( x , y , m , n ) , 2 )
    df [ 'y_cap' ] = m * x + c
    SSR = round ( np.sum ( ( df [ 'y_cap' ] - y_mean ) ** 2 ) , 3 )
    SST = round ( np.sum ( ( y - y_mean ) ** 2 ) , 3 )
    SSE = round ( np.sum ( ( y - df [ 'y_cap' ] ) ** 2 ) , 3 )
    R_squared = round ( SSR / SST , 3 )
    return m , c , SSR , SST , SSE , R_squared

data = pd.read_csv ( 'LR_dataset.csv' )
print ( "\n Data read from dataset.csv :\n\n" , data )
df = pd.DataFrame ( data )
m , c , SSR , SST , SSE , R_squared = evaluate_Model ( df )
print ( "\n Linear Regression equation : y = (" , m , ") x + (" , c , ")" )
print ( "\n Sum of Squares Total ( SST ) = " , SST )
print ( " Sum of Squares Regression ( SSR ) = " , SSR )
print ( " Sum of Squares Error ( SSE ) = " , SSE )
print ( " R-squared = " , R_squared )
```

RESULT

Hence the Linear Regression algorithm has been successfully implemented

12/4/23

EXPERIMENT - 6B

AIM

To implement the perceptron algorithm in Python

ALGORITHM

* Set initial weights $w_1, w_2, \ldots, w_n$ and bias θ
* Compute the weighted sum and apply the activation function on the weighted sum (binary step)
* If the sum > threshold value, output the value as positive else output as negative
* Calculate the error by subtracting estimated output from the desired output

Error $e(t) = Y_{desired} - Y_{estimated}$

* Update the weights if there is an error:

$w_i = w_i + (\alpha \times e(t) \times x_i)$

* Repeat above steps for each epoch until there is no error

THEORY

Perceptron is a supervised learning algorithm that consists of 4 steps:

1. Inputs from other neurons
2. Weights and bias
3. Net sum
4. Activation function

OUTPUT

```
Single Layer Perceptron - AND Gate

EPOCH  1
x1    x2     Error   w1     w2
 0     0      0      0.3    -0.2
 0     1      0      0.3    -0.2
 1     0      0      0.3    -0.2
 1     1      1      0.5    0.0

EPOCH  2
x1    x2     Error   w1     w2
 0     0      0      0.5    0.0
 0     1      0      0.5    0.0
 1     0      -1     0.3    0.0
 1     1      1      0.5    0.2

EPOCH  3
x1    x2     Error   w1     w2
 0     0      0      0.5    0.2
 0     1      0      0.5    0.2
 1     0      -1     0.3    0.2
 1     1      0      0.3    0.2

EPOCH  4
x1    x2     Error   w1     w2
 0     0      0      0.3    0.2
 0     1      0      0.3    0.2
 1     0      0      0.3    0.2
 1     1      0      0.3    0.2

Final Weights : w1 = 0.30 , w2 = 0.20
```

CODE

```
import numpy as np

def calculateStep ( x1 , x2 , w1 , w2 , theta ) :
    z = x1 * w1 + x2 * w2 - theta
    return 1 if z >= 0 else 0

def updateWeights ( alpha , error , x1 , x2 , w1 , w2 ) :
    w1 += alpha * error * x1
    w2 += alpha * error * x2
    return w1 , w2

def calculateEpoch ( truth_table , w1 , w2 , alpha , theta ) :
    errors = []
    print ( " x1 \tx2  \tError \t w1  \t w2" )
    for row in truth_table :
        x1 , x2 , expected = row
        y = calculateStep ( x1 , x2 , w1 , w2 , theta )
        error = expected - y
        if error != 0 :
            w1 , w2 = updateWeights ( alpha , error , x1 , x2 , w1 , w2 )
        errors.append ( error )
        print ( " " , x1 , " \t" , x2 , " \t" , error , " \t" , w1 , " \t" , w2 )
    return w1 , w2 , errors

def calculatePerceptron ( gate , w1 , w2 , alpha , theta ) :
    if gate == "AND" :
        truth_table = np.array ( [ [ 0 , 0 , 0 ] , [ 0 , 1 , 0 ] , [ 1 , 0 , 0 ] , [ 1 , 1 , 1 ] ] )
    elif gate == "OR" :
        truth_table = np.array ( [ [ 0 , 0 , 0 ] , [ 0 , 1 , 1 ] , [ 1 , 0 , 1 ] , [ 1 , 1 , 1 ] ] )
    else:
        print ( "INVALID Gate !" )
        return
    epoch = 0
    while True :
        epoch += 1
        print ( "\n EPOCH " , epoch )
        w1 , w2 , errors = calculateEpoch ( truth_table , w1 , w2 , alpha , theta )
        if errors.count ( 0 ) == len ( errors ) :
            break
    print ( "\n Final Weights : w1 = {:.2f} , w2 = {:.2f}".format ( w1 , w2 ) )

print ( "\n Single Layer Perceptron - AND Gate" )
calculatePerceptron ( "AND" , 0.3 , -0.2 , 0.2 , 0.4 )
```

RESULT

Hence the Perceptron algorithm has been implemented successfully

12/4/23

EXPERIMENT - 6C

AIM

To implement the Multi-Layer Perceptron algorithm in python

ALGORITHM

FORWARD PROPAGATION

* Calculate Input & Output in Input Layer
* Calculate Net Input and Output in the Hidden Layer & Output Layer
* Estimate error at the node in the output layer

BACKWARD PROPAGATION

* Calculate Error at each node in the Hidden and Output layers
* Update all weights and biases

$W_{ij} = W_{ij} + (\alpha \times Error_j \times O_i)$, $\theta_j = \theta_j + (\alpha \times Error_j)$

Finally display error reduced from both iterations using MLP

THEORY

A Multi-Layer perceptron is a type of Feed Forward Neural Network with multiple neurons arranged in layers. The MLP network learns with 2 phases called the forward and backward phases.

OUTPUT

```
x1 = 1 , x2 = 1 , x3 = 0 , x4 = 1 , w15 = 0.3 ,
w16 = 0.1 , w25 = -0.2 , w26 = 0.4 , w35 = 0.2 , w36 = -0.3 ,
w45 = 0.1 , w46 = 0.4 , w57 = -0.3 , w67 = 0.2 , O1 = 1 ,
O2 = 1 , O3 = 0 , O4 = 1 , Odes7 = 1 , theta5 = 0.2 ,
theta6 = 0.1 , theta7 = -0.3 ,

 ITERATION 1

 Forward Propagation ...

 Backward Propagation ...


 ITERATION 2

 Forward Propagation ...

 ITERATION 1 Error at Ouput Node  7  =  0.583

 ITERATION 2 Error at Ouput Node  7  =  0.529

 Therefore using MLP , Error reduced =  0.054
```

CODE

```
import math

def updateWeightsAndBiases () :
   for ip in range ( 1 , n_input + 1 ) :
      for hid in range ( n_input + 1 , n_input + n_hidden + 1 ) :
         inputs [ "w" + str ( ip ) + str ( hid ) ] += round ( alpha * inputs [ "Error" + str ( hid ) ] * inputs [
"O" + str ( ip ) ] , 3 )
   for hid in range ( n_input + 1 , n_input + n_hidden + 1 ) :
      inputs [ "theta" + str ( hid ) ] += round ( alpha * inputs [ "Error" + str ( hid ) ] , 3 )
      for op in range ( n_input + n_hidden + 1 , n_input + n_hidden + n_output + 1 ) :
         inputs [ "w" + str ( hid ) + str ( op ) ] += round ( alpha * inputs [ "Error" + str ( op ) ] * inputs [
"O" + str ( hid ) ] , 3 )
   for op in range ( n_input + n_hidden + 1 , n_input + n_hidden + n_output + 1 ) :
      inputs [ "theta" + str ( op ) ] += round ( alpha * inputs [ "Error" + str ( op ) ] , 3 )

def forwardPropagation ( call ) :
   for op in range ( n_input + n_hidden + 1 , n_input + n_hidden + n_output + 1 ) :
      inputs [ "I" + str ( op ) ] = inputs [ "theta" + str ( op ) ]
      for hid in range ( n_input + 1 , n_input + n_hidden + 1 ) :
         inputs [ "I" + str ( hid ) ] = round ( inputs [ "theta" + str ( hid ) ] , 3 )
         for inp in range ( 1 , n_input + 1 ) :
            inputs [ "I" + str ( hid ) ] += round ( inputs [ "x" + str ( inp ) ] * inputs [ "w" + str ( inp ) + str
( hid ) ] , 3 )
         inputs [ "O" + str ( hid ) ] = round ( 1 / ( 1 + math.exp ( - inputs [ "I" + str ( hid ) ] ) ) , 3 )
         inputs [ "I" + str ( op ) ] += round ( inputs [ "O" + str ( hid ) ] * inputs [ "w" + str ( hid ) + str (
op ) ] , 3 )
      inputs [ "O" + str ( op ) ] = round ( 1 / ( 1 + math.exp ( - inputs [ "I" + str ( op ) ] ) ) , 3 )
      inputs [ "Error" + str ( op ) ] = round ( inputs [ "Odes" + str ( op ) ] - inputs [ "O" + str ( op ) ] , 3
)
      inputs [ str ( call ) + "Error" + str ( op ) ] = inputs [ "Error" + str ( op ) ]

def backwardPropagation () :
   for op in range ( n_input + n_hidden + 1 , n_input + n_hidden + n_output + 1 ) :
      O_op = inputs [ "O" + str ( op ) ]
      inputs [ "Error" + str ( op ) ] = round ( O_op * ( 1 - O_op ) * ( inputs [ "Odes" + str ( op ) ] -
O_op ) , 3 )
   for hid in range ( n_input + 1 , n_input + n_hidden + 1 ) :
      O_hid = inputs [ "O" + str ( hid ) ]
      inputs [ "Error" + str ( hid ) ] = round ( O_hid * ( 1 - O_hid ) , 3 )
      err_wt = 0
      for op in range ( n_input + n_hidden + 1 , n_input + n_hidden + n_output + 1 ) :
         err_wt += inputs [ "Error" + str ( op ) ] * inputs [ "w" + str ( hid ) + str ( op ) ]
      inputs [ "Error" + str ( hid ) ] *= round ( err_wt , 3 )
   updateWeightsAndBiases ()

def calculateMLP () :
   print ( "\n\n ITERATION 1\n\n Forward Propagation ..." )
   forwardPropagation ( 1 )
   backwardPropagation ()
   print ( "\n Backward Propagation ...\n\n" ) #, inputs )
   print ( " ITERATION 2\n\n Forward Propagation ..." )
   forwardPropagation ( 2 )
```

```
for op in range ( n_input + n_hidden + 1 , n_input + n_hidden + n_output + 1 ) :
    err1 , err2 = inputs [ "1Error" + str ( op ) ] , inputs [ "2Error" + str ( op ) ]
    print ( "\n ITERATION 1 Error at Ouput Node " , op , " = " , err1 )
    print ( "\n ITERATION 2 Error at Ouput Node " , op , " = " , err2 )
    print ( "\n Therefore using MLP , Error reduced = " , round ( err1 - err2 , 3 ) )

alpha = 0.8
n_input , n_hidden , n_output = 4 , 2 , 1
inputs = { "x1" : 1 , "x2" : 1 , "x3" : 0 , "x4" : 1 ,
"w15" : 0.3 , "w16" : 0.1 , "w25" : -0.2 , "w26" : 0.4 , "w35" : 0.2 , "w36" : -0.3 ,
"w45" : 0.1 , "w46" : 0.4 , "w57" : -0.3 , "w67" : 0.2 ,
"O1" : 1 , "O2" : 1 , "O3" : 0 , "O4" : 1 , "Odes7" : 1 ,
"theta5" : 0.2 , "theta6" : 0.1 , "theta7" : -0.3
}

counter = 0
for key , value in inputs.items () :
    if counter % 5 == 0 :
        print ()
    print ( key , '=' , value , end = ' , ' )
    counter += 1

calculateMLP ()
```

RESULT

Hence the Multi-Layer Perceptron algorithm has been implemented successfully