Machine Learning Lab Assessment

## KNN Algorithm

Name: Om Ashish Mishra

Registration Number: 16BCE0789

Slot: F2

1. Implement k-Nearest Neighbor algorithm for classifying a dataset.

**Dataset Used:** Iris Dataset

The data set consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor). Four features were measured from each sample: the length and the width of the sepals and petals, in centimetres. Based on the combination of these four features, Fisher developed a linear discriminant model to distinguish the species from each other.

The Code:

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

from sklearn.metrics import classification\_report,confusion\_matrix

url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"

names = ['sepal-length', 'sepal-width', 'petallength','petal-width', 'Class']

dataset = pd.read\_csv(url, names=names)

dataset.head()

X = dataset.iloc[:, :-1].values

y = dataset.iloc[:, 4].values

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y,test\_size=0.20)

from sklearn.preprocessing import StandardScaler

scaler =StandardScaler()

scaler.fit(X\_train)

X\_train = scaler.transform(X\_train)

X\_test = scaler.transform(X\_test)

from sklearn.neighbors import KNeighborsClassifier

for i in range(1,6):

print('for k = ',i)

classifier = KNeighborsClassifier(n\_neighbors=i)

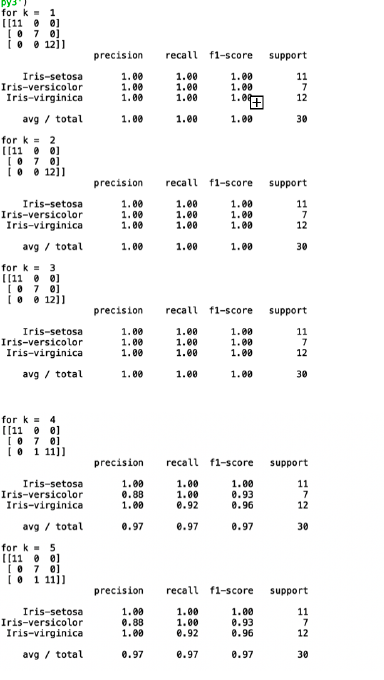
classifier.fit(X\_train, y\_train)

y\_pred = classifier.predict(X\_test)

print(confusion\_matrix(y\_test,y\_pred))

print(classification\_report(y\_test, y\_pred))

The Output:



1. Implement Multi-Layer Perceptron using a dataset from UCI repository.

The Code:

|  |
| --- |
|  |
|  | From numpy import exp, array, random, dot |
|  |  |
|  | class NeuronLayer(): |
|  | def \_\_init\_\_(self, number\_of\_neurons, number\_of\_inputs\_per\_neuron): |
|  | self.synaptic\_weights = 2 \* random.random((number\_of\_inputs\_per\_neuron, number\_of\_neurons)) - 1 |
|  |  |
|  |  |
|  | class NeuralNetwork(): |
|  | def \_\_init\_\_(self, layer1, layer2): |
|  | self.layer1 = layer1 |
|  | self.layer2 = layer2 |
|  |  |
|  | # The Sigmoid function, which describes an S shaped curve. |
|  | # We pass the weighted sum of the inputs through this function to |
|  | # normalise them between 0 and 1. |
|  | def \_\_sigmoid(self, x): |
|  | return 1 / (1 + exp(-x)) |
|  |  |
|  | # The derivative of the Sigmoid function. |
|  | # This is the gradient of the Sigmoid curve. |
|  | # It indicates how confident we are about the existing weight. |
|  | def \_\_sigmoid\_derivative(self, x): |
|  | return x \* (1 - x) |
|  |  |
|  | # We train the neural network through a process of trial and error. |
|  | # Adjusting the synaptic weights each time. |
|  | def train(self, training\_set\_inputs, training\_set\_outputs, number\_of\_training\_iterations): |
|  | for iteration in xrange(number\_of\_training\_iterations): |
|  | # Pass the training set through our neural network |
|  | output\_from\_layer\_1, output\_from\_layer\_2 = self.think(training\_set\_inputs) |
|  |  |
|  | # Calculate the error for layer 2 (The difference between the desired output |
|  | # and the predicted output). |
|  | layer2\_error = training\_set\_outputs - output\_from\_layer\_2 |
|  | layer2\_delta = layer2\_error \* self.\_\_sigmoid\_derivative(output\_from\_layer\_2) |
|  |  |
|  | # Calculate the error for layer 1 (By looking at the weights in layer 1, |
|  | # we can determine by how much layer 1 contributed to the error in layer 2). |
|  | layer1\_error = layer2\_delta.dot(self.layer2.synaptic\_weights.T) |
|  | layer1\_delta = layer1\_error \* self.\_\_sigmoid\_derivative(output\_from\_layer\_1) |
|  |  |
|  | # Calculate how much to adjust the weights by |
|  | layer1\_adjustment = training\_set\_inputs.T.dot(layer1\_delta) |
|  | layer2\_adjustment = output\_from\_layer\_1.T.dot(layer2\_delta) |
|  |  |
|  | # Adjust the weights. |
|  | self.layer1.synaptic\_weights += layer1\_adjustment |
|  | self.layer2.synaptic\_weights += layer2\_adjustment |
|  |  |
|  | # The neural network thinks. |
|  | def think(self, inputs): |
|  | output\_from\_layer1 = self.\_\_sigmoid(dot(inputs, self.layer1.synaptic\_weights)) |
|  | output\_from\_layer2 = self.\_\_sigmoid(dot(output\_from\_layer1, self.layer2.synaptic\_weights)) |
|  | return output\_from\_layer1, output\_from\_layer2 |
|  |  |
|  | # The neural network prints its weights |
|  | def print\_weights(self): |
|  | print " Layer 1 (4 neurons, each with 3 inputs): " |
|  | print self.layer1.synaptic\_weights |
|  | print " Layer 2 (1 neuron, with 4 inputs):" |
|  | print self.layer2.synaptic\_weights |
|  |  |
|  | if \_\_name\_\_ == "\_\_main\_\_": |
|  |  |
|  | #Seed the random number generator |
|  | random.seed(1) |
|  |  |
|  | # Create layer 1 (4 neurons, each with 3 inputs) |
|  | layer1 = NeuronLayer(4, 3) |
|  |  |
|  | # Create layer 2 (a single neuron with 4 inputs) |
|  | layer2 = NeuronLayer(1, 4) |
|  |  |
|  | # Combine the layers to create a neural network |
|  | neural\_network = NeuralNetwork(layer1, layer2) |
|  |  |
|  | print "Stage 1) Random starting synaptic weights: " |
|  | neural\_network.print\_weights() |
|  |  |
|  | # The training set. We have 7 examples, each consisting of 3 input values |
|  | # and 1 output value. |
|  | training\_set\_inputs = array([[0, 0, 1], [0, 1, 1], [1, 0, 1], [0, 1, 0], [1, 0, 0], [1, 1, 1], [0, 0, 0]]) |
|  | training\_set\_outputs = array([[0, 1, 1, 1, 1, 0, 0]]).T |
|  |  |
|  | # Train the neural network using the training set. |
|  | # Do it 60,000 times and make small adjustments each time. |
|  | neural\_network.train(training\_set\_inputs, training\_set\_outputs, 60000) |
|  |  |
|  | print "Stage 2) New synaptic weights after training: " |
|  | neural\_network.print\_weights() |
|  |  |
|  | # Test the neural network with a new situation. |
|  | print "Stage 3) Considering a new situation [1, 1, 0] -> ?: " |
|  | hidden\_state, output = neural\_network.think(array([1, 1, 0])) |
|  | print output |

The Output:

|  |
| --- |
| Stage 1) Random starting synaptic weights: |

Layer 1 (4 neurons, each with 3 inputs):

|  |
| --- |
| [[-0.16595599 0.44064899 -0.99977125 -0.39533485] |
| [-0.70648822 -0.81532281 -0.62747958 -0.30887855] |

[-0.20646505 0.07763347 -0.16161097 0.370439 ]]

Layer 2 (1 neuron, with 4 inputs):

[[-0.5910955 ] [ 0.75623487] [-0.94522481] [ 0.34093502]]

Stage 2) New synaptic weights after training:

Layer 1 (4 neurons, each with 3 inputs):

[[ 0.3122465 4.57704063 -6.15329916 -8.75834924]

[ 0.19676933 -8.74975548 -6.1638187 4.40720501]

[-0.03327074 -0.58272995 0.08319184 -0.39787635]]

Layer 2 (1 neuron, with 4 inputs):

[[ -8.18850925]

[ 10.13210706]

[-21.33532796]

[ 9.90935111]]

Stage 3) Considering a new situation [1, 1, 0] -> ?:

[ 0.0078876]