6CS4-22: Machine Learning Lab **(Date : 11/03/2020)**

Experiment No. : 01

Instructor: Mr. Katkar Atish R. *Name:* Anjali Malav, *Roll number:* 17EEJCS002

AIM : Implementation of k - Nearest Neighbors (KNNs) on Synthetic data using Python

INTRODUCTION TO KNN

K-Nearest Neighbor (KNN) algorithm is a type of supervised learning algorithm. KNN is used for both regression and classification tasks. When KNN is used for regression problems the prediction is based on the mean or the median of the K-most similar instances. Similarly, when KNN is used for classification, the output can be calculated as the class with the highest frequency from the K-most similar instances. Each instance in essence votes for their class and the class with the most votes is taken as the prediction. Basically, KNN is a prediction algorithm. The predictions are made by training dataset directly. Predictions are made for a new instance (x) by searching through the entire training set for the K most similar instances (the neighbors) and summarizing the output variable for those K instances. For regression this might be the mean output variable, in classification this might be the mode (or most common) class value.

To find out which of the K instances in the training dataset are most similar to a new input a distance measure is used. For real-valued input variables, the most popular distance measure is Euclidean distance. The Euclidean distance is represented by the following formula :

$$EuclideanDistance(x, x_i) = \sqrt{\sum (x_j - x_{ij})^2}$$

In above formula, the Euclidean distance is calculated as the square root of the sum of the squared differences between a new point (x) and an existing point (xi) across all input attributes j. The value for K can be found by algorithm tuning. It can be chosen by trying many different values for K (odd values) depending upon the size of the dataset and selected best value which suits for our problem.

KNN works well with a small number of input variables (p), but struggles when the number of inputs is very large. Each input variable can be considered a dimension of a p-dimensional input space. For example, if you had two input variables x1 and x2, the input space would be 2-dimensional. In high dimensions, points that may be similar may have very large distances. All points will be far away from each other and our intuition for distances in simple 2 and 3-dimensional spaces breaks down. This might feel unintuitive at first, but this general problem is called the “Curse of Dimensionality”.

PSEUDO CODE OF KNN

KNN can be implemented with the help of following steps :

1. **Load the data**
2. **Initialise the value of k**
3. **For getting the predicted class, iterate from 1 to total number of training data points**
   (a) **Calculate the Euclidean distance between test data and each row of training data.**
   (b) **Sort the calculated distances in ascending order based on distance values**
   (c) **Get top k rows from the sorted array**
   (d) **Get the most frequent class of these rows**
   (e) **Return the predicted class**

DATASET DESCRIPTION

The dataset identified for this experimental study is Synthetic dataset. The dataset is broadly categorized into 3 parts namely Linearly Separable, Non - Linearly Separable and Overlapping Data. These categories are further divided into several groups where in each group we are given with separate training and testing files. Testing and training data is also divided into different class files. Each class contain two features with many feature vectors and both features contain only numeric value.

IMPLEMENTATION CODE FOR KNN

```python

# Package imported for different libraries
import math
import numpy as np
import matplotlib.pyplot as plt
import operator
import csv
import random
from pandas import *


# Loading .txt data from computer using numpy
loaddata_train1=np.loadtxt('/home/asus/Desktop/prerana_ws/synthetic_data/synthetic_data/
    nonlinearlySeparable/group4/class1_train.txt')
loaddata_train2=np.loadtxt('/home/asus/Desktop/prerana_ws/synthetic_data/synthetic_data/
    nonlinearlySeparable/group4/class2_train.txt')
loaddata_test1=np.loadtxt('/home/asus/Desktop/prerana_ws/synthetic_data/synthetic_data/
    nonlinearlySeparable/group4/class1_test.txt')
loaddata_test2=np.loadtxt('/home/asus/Desktop/prerana_ws/synthetic_data/synthetic_data/
    nonlinearlySeparable/group4/class2_test.txt')
loaddata_train=np.concatenate((loaddata_train1, loaddata_train2), axis=0)
loaddata_test=np.concatenate((loaddata_test1, loaddata_test2), axis=0)


# Labeling training data
label1=np.ones((loaddata_train1.shape[0], 1))
label2=np.ones((loaddata_train2.shape[0], 1)) * -1
r1=np.append(label1, loaddata_train1, axis=1)
r2=np.append(label2, loaddata_train2, axis=1)
train_data=np.concatenate((r1,r2))



# Labeling testing data
label3=np.ones((loaddata_test1.shape[0], 1))
label4=np.ones((loaddata_test2.shape[0], 1)) * -1
r3=np.append(label3, loaddata_test1, axis=1)
r4=np.append(label4, loaddata_test2, axis=1)
test_data=np.concatenate((r3,r4))


# Ploting Data
plt.scatter(loaddata_train1[:,0], loaddata_train1[:,1], marker='o', label='Class1')
plt.scatter(loaddata_train2[:,0], loaddata_train2[:,1], marker='x', label='Class2')
plt.legend()
plt.show()
# Function to find Euclidean distances
def euclideanDistance(instance1, instance2, length):
    distance = 0
    for x in range(length):
        distance += pow((float(instance1[x]) - float(instance2[x])), 2)
    return math.sqrt(distance)


# Function to find neighbours by sorting Euclidean distances
def getKNeighbors(train_data, testInstance, k):
    distances = []
    length = len(testInstance)-1
    for x in range(len(train_data)):
        dist = euclideanDistance(testInstance, train_data[x], length)
        distances.append((train_data[x], dist))
    distances.sort(key=operator.itemgetter(1))
    neighbors = []
```

```
    for x in range(k):
        neighbors.append(distances[x][0])
    return neighbors


# Calculating label of neighbours and assign it to test instance


def getResponse(neighbors):
    classVotes = {}
    for x in range(len(neighbors)):
        response = neighbors[x][0]
        if response in classVotes:
            classVotes[response] += 1
        else:
            classVotes[response] = 1
    sortedVotes = sorted(classVotes.items(), key=operator.itemgetter(1), reverse=True)
    return sortedVotes[0][0]


# Main function
def main():
    l00 = 0
    l01 = 0
    l10 = 0
    l11 = 0
    split = 0.70
    print ('Train: ' + repr(len(train_data)))
    print ('Test: ' + repr(len(test_data)))
    #generate predictions
    predictions = []
    k = input("Enter value of k: ")
    k = int(k)
    list_pred = []
    list_act = []
    for x in range(len(test_data)):
        neighbors = getKNeighbors(train_data, test_data[x], k)
        result = getResponse(neighbors)
        predictions.append(result)
        list_pred.append(result)
        list_act.append(test_data[x][0])
    for i in range(0, len(test_data)):
        x = list_pred[0]
        y = list_act[0]
        if int(list_pred[i]) == -1 and int(list_act[i]) == -1:
            l00 += 1
        elif int(list_pred[i]) == -1 and int(list_act[i]) == 1:
            l01 += 1
        elif int(list_pred[i]) == 1 and int(list_act[i]) == -1:
            l10 += 1
        elif int(list_pred[i]) == 1 and int(list_act[i]) == 1:
            l11 += 1
        i+=1

    a = np.array([[l00, l01],
        [l10, l11]])
    print('Confusion Matrix: ')
    print(DataFrame(a, columns = ['class_0', 'class_1'], index = ['class_0', 'class_1']))
    prec_0 = (l00/float(l00+l10))
    prec_1 = (l11/float(l01+l11))

    acc = (l00+l11)*100/(l00+l01+l10+l11)
    print('Accuracy: ' + repr(acc))

    print('Precision: ')
    print('Precision for class 0: ' + repr(prec_0))
    print('Precision for class 1: ' + repr(prec_1))
    print('Average Precision: ' + repr((prec_0+prec_1)/2))

    rec_0 = (l00/float(l00+l01))
    rec_1 = (l11/float(l10+l11))
    print('Recall: ')
    print('Recall for class 0: ' + repr(rec_0))
```

```
    print('Recall for class 1: ' + repr(rec_1))
    print('Average Recall: ' + repr((rec_0+rec_1)/2))

    f0 = (2*(prec_0*rec_0)/(prec_0+rec_0))
    f1 = (2*(prec_1*rec_1)/(prec_1+rec_1))
    print('F1 Score: ')
    print('F1 Score for class 0: ' + repr(f0))
    print('F1 Score for class 1: ' + repr(f1))
    print('Average F1 Score: ' + repr((f0+f1)/2))

main()
```

OUTPUT

1. Output for Linearly Separable Data

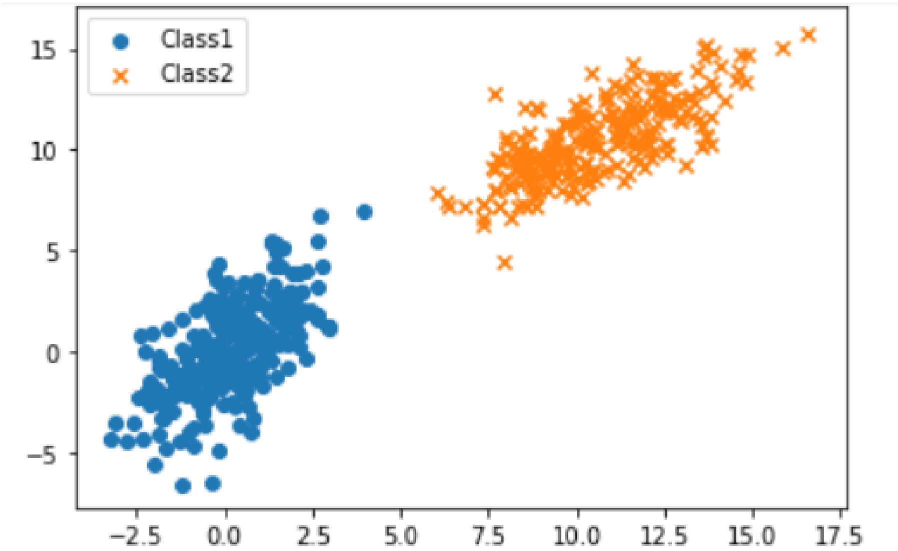

FIGURE 1: Plotting of Non-Linearly Separable Data

```
Anjali_Malav/Machine_Learning \$
Train: 500
Test: 200
Enter value of k:  9
Confusion Matrix:
         class_0  class_1
class_0      100        0
class_1        0      100
Accuracy: 100.0
Precision:
Precision for class 0: 1.0
Precision for class 1: 1.0
Average Precision: 1.0
Recall:
Recall for class 0: 1.0
Recall for class 1: 1.0
Average Recall: 1.0
F1 Score:
F1 Score for class 0: 1.0
F1 Score for class 1: 1.0
Average F1 Score: 1.0
```

2. Output for Non-Linearly Separable Data

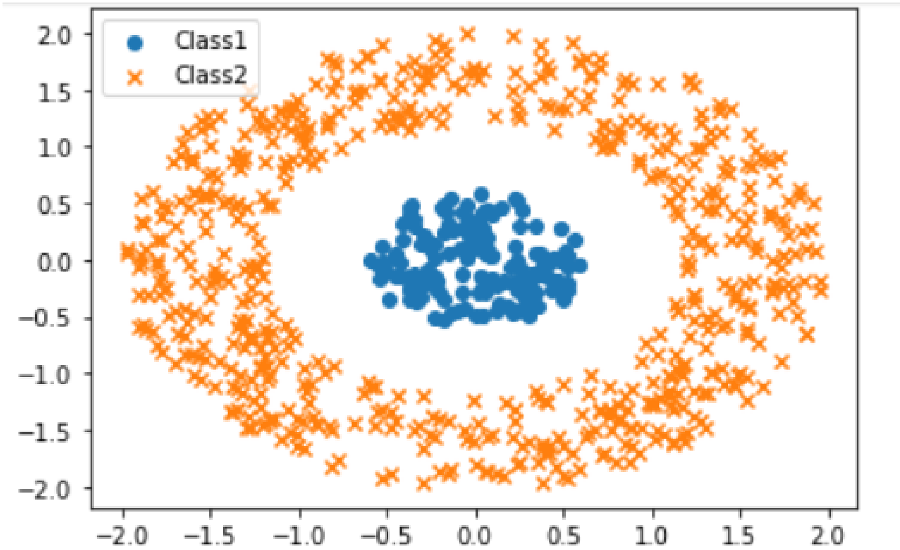

FIGURE 2: Plotting of Non-Linearly Separable Data

```
Anjali_Malav/Machine_Learning \$
Train: 750
Test: 300
Enter value of k:  7
Confusion Matrix:
          class_0  class_1
class_0      240        0
class_1        0       60
Accuracy: 100.0
Precision:
Precision for class 0: 1.0
Precision for class 1: 1.0
Average Precision: 1.0
Recall:
Recall for class 0: 1.0
Recall for class 1: 1.0
Average Recall: 1.0
F1 Score:
F1 Score for class 0: 1.0
F1 Score for class 1: 1.0
Average F1 Score: 1.0
```

3. Output for Over-lapping Data

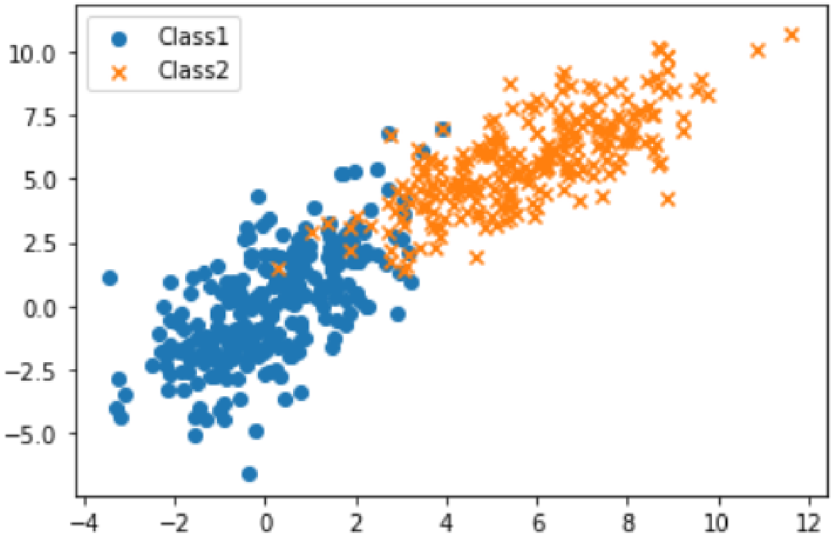

FIGURE 3: Plotting of Over-Lapping Data

```
Anjali_Malav/Machine_Learning \$
Train: 500
Test: 200
Enter value of k:  11
Confusion Matrix:
         class_0  class_1
class_0      100        0
class_1        0      100
Accuracy: 100.0
Precision:
Precision for class 0: 1.0
Precision for class 1: 1.0
Average Precision: 1.0
Recall:
Recall for class 0: 1.0
Recall for class 1: 1.0
Average Recall: 1.0
F1 Score:
F1 Score for class 0: 1.0
F1 Score for class 1: 1.0
Average F1 Score: 1.0
```

CONCLUSION

The KNN algorithm outperforms for the above cases. The implementation shows that KNN gives **100%** testing accuracy for linearly separable, non-linearly separable and over-lapping data respectively for used dataset.