**Machine learning Lab**

**Experiment- 9**

**Implementation of KNN algorithm**

def Classify(nItem, k, Items):

if(k > len(Items)):

# k is larger than list

# length, abort

return "k larger than list length";

# Hold nearest neighbors.

# First item is distance,

# second class

neighbors = [];

for item in Items:

# Find Euclidean Distance

distance = EuclideanDistance(nItem, item);

# Update neighbors, either adding

# the current item in neighbors

# or not.

neighbors = UpdateNeighbors(neighbors, item, distance, k);

# Count the number of each

# class in neighbors

count = CalculateNeighborsClass(neighbors, k);

# Find the max in count, aka the

# class with the most appearances.

return FindMax(count);

def EuclideanDistance(x, y):

# The sum of the squared

# differences of the elements

S = 0;

for key in x.keys():

S += math.pow(x[key]-y[key], 2);

# The square root of the sum

return math.sqrt(S);

def UpdateNeighbors(neighbors, item, distance, k):

if(len(neighbors) > distance):

# If yes, replace the last

# element with new item

neighbors[-1] = [distance, item["Class"]];

neighbors = sorted(neighbors);

return neighbors;

def CalculateNeighborsClass(neighbors, k):

count = {};

for i in range(k):

if(neighbors[i][1] not in count):

# The class at the ith index

# is not in the count dict.

# Initialize it to 1.

count[neighbors[i][1]] = 1;

else:

# Found another item of class

# c[i]. Increment its counter.

count[neighbors[i][1]] += 1;

return count;

def FindMax(countList):

# Hold the max

maximum = -1;

# Hold the classification

classification = "";

for key in countList.keys():

if(countList[key] > maximum):

maximum = countList[key];

classification = key;

return classification, maximum;

# Python Program to illustrate

# KNN algorithm

# For pow and sqrt

import math

from random import shuffle

###\_Reading\_### def ReadData(fileName):

# Read the file, splitting by lines

f = open(fileName, 'r')

lines = f.read().splitlines()

f.close()

# Split the first line by commas,

# remove the first element and save

# the rest into a list. The list

# holds the feature names of the

# data set.

features = lines[0].split(', ')[:-1]

items = []

for i in range(1, len(lines)):

line = lines[i].split(', ')

itemFeatures = {'Class': line[-1]}

for j in range(len(features)):

# Get the feature at index j

f = features[j]

# Convert feature value to float

v = float(line[j])

# Add feature value to dict

itemFeatures[f] = v

items.append(itemFeatures)

shuffle(items)

return items

###\_Auxiliary Function\_### def EuclideanDistance(x, y):

# The sum of the squared differences

# of the elements

S = 0

for key in x.keys():

S += math.pow(x[key] - y[key], 2)

# The square root of the sum

return math.sqrt(S)

def CalculateNeighborsClass(neighbors, k):

count = {}

for i in range(k):

if neighbors[i][1] not in count:

# The class at the ith index is

# not in the count dict.

# Initialize it to 1.

count[neighbors[i][1]] = 1

else:

# Found another item of class

# c[i]. Increment its counter.

count[neighbors[i][1]] += 1

return count

def FindMax(Dict):

# Find max in dictionary, return

# max value and max index

maximum = -1

classification = ''

for key in Dict.keys():

if Dict[key] > maximum:

maximum = Dict[key]

classification = key

return (classification, maximum)

###\_Core Functions\_### def Classify(nItem, k, Items):

# Hold nearest neighbours. First item

# is distance, second class

neighbors = []

for item in Items:

# Find Euclidean Distance

distance = EuclideanDistance(nItem, item)

# Update neighbors, either adding the

# current item in neighbors or not.

neighbors = UpdateNeighbors(neighbors, item, distance, k)

# Count the number of each class

# in neighbors

count = CalculateNeighborsClass(neighbors, k)

# Find the max in count, aka the

# class with the most appearances

return FindMax(count)

def UpdateNeighbors(neighbors, item, distance,

k, ):

if len(neighbors) < k:

# List is not full, add

# new item and sort

neighbors.append([distance, item['Class']])

neighbors = sorted(neighbors)

else:

# List is full Check if new

# item should be entered

if neighbors[-1][0] > distance:

# If yes, replace the

# last element with new item

neighbors[-1] = [distance, item['Class']]

neighbors = sorted(neighbors)

return neighbors

###\_Evaluation Functions\_### def K\_FoldValidation(K, k, Items):

if K > len(Items):

return -1

# The number of correct classifications

correct = 0

# The total number of classifications

total = len(Items) \* (K - 1)

# The length of a fold

l = int(len(Items) / K)

for i in range(K):

# Split data into training set

# and test set

trainingSet = Items[i \* l:(i + 1) \* l]

testSet = Items[:i \* l] + Items[(i + 1) \* l:]

for item in testSet:

itemClass = item['Class']

itemFeatures = {}

# Get feature values

for key in item:

if key != 'Class':

# If key isn't "Class", add

# it to itemFeatures

itemFeatures[key] = item[key]

# Categorize item based on

# its feature values

guess = Classify(itemFeatures, k, trainingSet)[0]

if guess == itemClass:

# Guessed correctly

correct += 1

accuracy = correct / float(total)

return accuracy

def Evaluate(K, k, items, iterations):

# Run algorithm the number of

# iterations, pick average

accuracy = 0

for i in range(iterations):

shuffle(items)

accuracy += K\_FoldValidation(K, k, items)

print accuracy / float(iterations)

###\_Main\_### def main():

items = ReadData('data.txt')

Evaluate(5, 5, items, 100)

if \_\_name\_\_ == '\_\_main\_\_':

main()