MACHINE LEARNING LAB RECORD

Submitted To,

Mrs. Shabari Shedthi B

Asst Prof Gd II,Dept of CSE,NMAMIT

Submitted By:

K Prahlad Bhat

4NM18CS072

VI sem

Sec: ‘B’

Program 1:

Implement and demonstrate the *FIND-S algorithm* for finding the most specific hypothesis based on a given set of training data samples. Read the training data from a .CSV file.

Dataset used:

A	B	C	D	E	F	G
sky	airtemp	humidity	wind	water	forcast	enjoysport
sunny	warm	normal	strong	warm	same	yes
sunny	warm	high	strong	warm	same	yes
rainy	cold	high	strong	warm	change	no
sunny	warm	high	strong	cool	change	yes

```
import csv
a = []

with open('./dataset/sport.csv','r') as csvfile:
    for row in csv.reader(csvfile):
        a.append(row)
        print(row)
num = len(a[0])-1
hy = ['0']*num
for i in range(len(a)):
    if a[i][num] == 'yes':
        for j in range(num):
            if hy[j] == '0' or hy[j] == a[i][j]:
                hy[j] = a[i][j]
            else:
                hy[j] = "?"


print('*'*30)
print(hy)
```

OUTPUT:

```
In [1]: runfile('G:/ML Lab Programs/finds.py', wdir='G:/ML Lab Programs')
['sky', 'air_temp', 'humidity', 'wind', 'water', 'forecast', 'enjoy_sport']
['sunny', 'warm', 'normal', 'strong', 'warm', 'same', 'yes']
['sunny', 'warm', 'high', 'strong', 'warm', 'same', 'yes']
['rainy', 'cold', 'high', 'strong', 'warm', 'change', 'no']
['sunny', 'warm', 'high', 'strong', 'cool', 'change', 'yes']
******************************
['sunny', 'warm', '?', 'strong', '?', '?']

In [2]:
```

Program 2:

For a given set of training data examples stored in a .CSV file, implement and demonstrate the *Candidate-Elimination algorithm* to output a description of the set of all hypotheses consistent with the training examples.

Dataset used:

A	B	C	D	E	F	G
sky	airtemp	humidity	wind	water	forcast	enjoysport
sunny	warm	normal	strong	warm	same	yes
sunny	warm	high	strong	warm	same	yes
rainy	cold	high	strong	warm	change	no
sunny	warm	high	strong	cool	change	yes

```
import pandas as pd
import numpy as np



data = pd.DataFrame(data=pd.read_csv('./dataset/sport.csv'))
concepts = np.array(data.iloc[:,0:-1])
print("Instances are: ")
for i in concepts:
    print(i)
target = np.array(data.iloc[:,-1])
print("Target: ",target)



def learn(concepts, target):
    specific_h = concepts[0].copy()
```

```
    print("\nInitialization of specific_h and genearal_h")
    print("\nSpecific Boundary: ", specific_h)
    general_h = [["?" for i in range(len(specific_h))] for i in
range(len(specific_h))]
    print("\nGeneric Boundary: ",general_h)

    for i, h in enumerate(concepts):
        print("\nInstance", i+1 , "is ", h)
        if target[i] == "yes":
            for x in range(len(specific_h)):
                if h[x]!= specific_h[x]:
                    specific_h[x] ='?'
                    general_h[x][x] ='?'

        if target[i] == "no":
            for x in range(len(specific_h)):
                if h[x]!= specific_h[x]:
                    general_h[x][x] = specific_h[x]
                else:
                    general_h[x][x] = '?'

        print("Specific Boundary after ", i+1, "Instance is ", specific_h)
        print("Generic Boundary after ", i+1, "Instance is ", general_h)
        print("\n")

    indices = [i for i, val in enumerate(general_h) if val == ['?', '?',
'?', '?', '?', '?']]
    for i in indices:
        general_h.remove(['?', '?', '?', '?', '?', '?'])
    return specific_h, general_h

s_final, g_final = learn(concepts, target)

print("\n\n\nFinal Specific_h: ", s_final, sep="\n")
print("\nFinal General_h: ", g_final, sep="\n")
```

OUTPUT:

```
Initialization of specific_h and genearal_h

Specific Boundary:  ['sunny' 'warm' 'normal' 'strong' 'warm' 'same']

Generic Boundary:  [['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?',
'?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Instance 1 is  ['sunny' 'warm' 'normal' 'strong' 'warm' 'same']
Specific Boundary after  1 Instance is  ['sunny' 'warm' 'normal' 'strong' 'warm' 'same']
Generic Boundary after  1 Instance is  [['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?',
'?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]


Instance 2 is  ['sunny' 'warm' 'high' 'strong' 'warm' 'same']
Specific Boundary after  2 Instance is  ['sunny' 'warm' '?' 'strong' 'warm' 'same']
Generic Boundary after  2 Instance is  [['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?',
'?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]


Instance 3 is  ['rainy' 'cold' 'high' 'strong' 'warm' 'change']
Specific Boundary after  3 Instance is  ['sunny' 'warm' '?' 'strong' 'warm' 'same']
Generic Boundary after  3 Instance is  [['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?',
'?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', 'same']]


Instance 4 is  ['sunny' 'warm' 'high' 'strong' 'cool' 'change']
Specific Boundary after  4 Instance is  ['sunny' 'warm' '?' 'strong' '?' '?']
Generic Boundary after  4 Instance is  [['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?',
'?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]



Final Specific_h:
['sunny' 'warm' '?' 'strong' '?' '?']

Final General_h:
[['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?']]
```


Program 3:

Write a program to demonstrate the working of the ***decision tree based ID3 algorithm***. Use an appropriate data set for building the decision tree and apply this knowledge to classify a new sample

Dataset used:

day	outlook	temp	humidity	wind	play
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

```
import pandas as pd
import numpy as np

dataset = pd.read_csv("dataset/tennis.csv")
print(dataset)


def entropy(target_col):
    elements, counts = np.unique(target_col, return_counts=True)
    # print(elements,counts)
    entropy = np.sum(
        [(-counts[i] / np.sum(counts)) * np.log2(counts[i] / np.sum(counts))
for i in range(len(elements))])
    # print(entropy)
    return entropy


def InfoGain(data, split_attribute_name, target_name="Play Tennis"):
    total_entropy = entropy(data[target_name])
    vals, counts = np.unique(data[split_attribute_name], return_counts=True)
    Weighted_Entropy = np.sum(
        [(counts[i] / np.sum(counts)) *
entropy(data.where(data[split_attribute_name] ==
vals[i]).dropna()[target_name])
         for i in range(len(vals))])
    Information_Gain = total_entropy - Weighted_Entropy
    return Information_Gain


def ID3(data, originaldata, features, target_attribute_name="Play Tennis",
parent_node_class=None):
    if len(np.unique(data[target_attribute_name])) <= 1:
        return np.unique(data[target_attribute_name])[0]
    elif len(data) == 0:
        return np.unique(originaldata[target_attribute_name])[
            np.argmax(np.unique(originaldata[target_attribute_name],
return_counts=True)[1])]
    elif len(features) == 0:
        return parent_node_class
    else:
        parent_node_class

np.unique(data[target_attribute_name])[np.argmax(np.unique(data[target_attri
bute_name], return_counts=True)[1])]
        item_values = [InfoGain(data, feature, target_attribute_name)
                       for feature in features]  # Return the information
gain values for the features in the dataset
```

```
        best_feature_index = np.argmax(item_values)
        best_feature = features[best_feature_index]
        tree = {best_feature: {}}
        features = [i for i in features if i != best_feature]
        for value in np.unique(data[best_feature]):
            value = value
            sub_data = data.where(data[best_feature] == value).dropna()
            subtree = ID3(sub_data, dataset, features,
target_attribute_name, parent_node_class)
            tree[best_feature][value] = subtree
        return (tree)


tree = ID3(dataset, dataset, dataset.columns[1:-1])
print(' \nDisplay Tree\n', tree)
```

OUTPUT

```
In [3]: runfile('G:/ML Lab Programs/id3.py', wdir='G:/ML Lab Programs')
    day   Outlook Temperature Humidity    Wind Play Tennis
0    D1     Sunny         Hot     High    Weak          No
1    D2     Sunny         Hot     High  Strong          No
2    D3  Overcast         Hot     High    Weak         Yes
3    D4      Rain        Mild     High    Weak         Yes
4    D5      Rain        Cool   Normal    Weak         Yes
5    D6      Rain        Cool   Normal  Strong          No
6    D7  Overcast        Cool   Normal  Strong         Yes
7    D8     Sunny        Mild     High    Weak          No
8    D9     Sunny        Cool   Normal    Weak         Yes
9   D10      Rain        Mild   Normal    Weak         Yes
10  D11     Sunny        Mild   Normal  Strong         Yes
11  D12  Overcast        Mild     High  Strong         Yes
12  D13  Overcast         Hot   Normal    Weak         Yes
13  D14      Rain        Mild     High  Strong          No

Display Tree
 {'Outlook': {'Overcast': 'Yes', 'Rain': {'Wind': {'Strong': 'No', 'Weak': 'Yes'}}, 'Sunny': {'Humidity': {'High': 'No', 'Normal': 'Yes'}}}}
```

Program 4:

Build an Artificial Neural Network by implementing the *Backpropagation algorithm* and test the same using appropriate data sets.

```
import numpy as np

X = np.array(([2, 9], [1, 5], [3, 6],[4,8]))   # Hours Studied, Hours Slept
y = np.array(([92], [86], [89],[90]))   # Test Score

y = y / 100  # max test score is 100


# Sigmoid Function
def sigmoid(x):  # this function maps any value between 0 and 1
```

```
    return 1 / (1 + np.exp(-x))


# Derivative of Sigmoid Function
def derivatives_sigmoid(x):
    return x * (1 - x)



# Variable initialization
epoch = 10000  # Setting training iterations
lr = 0.1  # Setting learning rate
inputlayer_neurons = 2  # number of features in data set
hiddenlayer_neurons = 3  # number of hidden layers neurons
output_neurons = 1  # number of neurons of output layer

# weight and bias initialization
wh = np.random.uniform(size=(inputlayer_neurons, hiddenlayer_neurons))
bias_hidden = np.random.uniform(size=(1, hiddenlayer_neurons))  # bias
matrix to the hidden layer
weight_hidden = np.random.uniform(size=(hiddenlayer_neurons,
output_neurons))  # weight matrix to the output layer
bias_output = np.random.uniform(size=(1, output_neurons))  # matrix to the
output layer

for i in range(epoch):
    # Forward Propogation
    hinp1 = np.dot(X, wh)
    hinp = hinp1 + bias_hidden  # bias_hidden GRADIENT DISCENT
    hlayer_activation = sigmoid(hinp)

    outinp1 = np.dot(hlayer_activation, weight_hidden)
    outinp = outinp1 + bias_output
    output = sigmoid(outinp)

    # Backpropagation
    EO = y - output  # Compare prediction with actual output and calculate
the gradient of error (Actual - Predicted)

    outgrad = derivatives_sigmoid(output)  # Compute the slope/ gradient of
hidden and output layer neurons

    d_output = EO * outgrad  # Compute change factor(delta) at output layer,
dependent on the gradient of error multiplied by the slope of output layer
activation

    EH = d_output.dot(
        weight_hidden.T)  # At this step, the error will propagate back into
the network which means error at hidden layer. we will take the dot product
```

```
of output layer delta with weight parameters of edges between the hidden and
output layer (weight_hidden.T).

    hiddengrad = derivatives_sigmoid(hlayer_activation)  # how much hidden
layer weight contributed to error
    d_hiddenlayer = EH * hiddengrad

    # update the weights
    weight_hidden += hlayer_activation.T.dot(d_output) * lr  # dot product
of nextlayererror and currentlayerop
    bias_hidden += np.sum(d_hiddenlayer, axis=0, keepdims=True) * lr

    wh += X.T.dot(d_hiddenlayer) * lr
    bias_output += np.sum(d_output, axis=0, keepdims=True) * lr

print("Input: \n" + str(X))
print("Actual Output: \n" + str(y))
print("Predicted Output: \n", output)
```

OUTPUT:

```
In [4]: runfile('G:/ML Lab Programs/backprop_ann.py', wdir='G:/ML Lab Programs')
Input:
[[2 9]
 [1 5]
 [3 6]
 [4 8]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]
 [0.9 ]]
Predicted Output:
 [[0.89243733]
 [0.88148399]
 [0.89649414]
 [0.8984865 ]]

In [5]:
```

Program 5:

Write a program to implement the *naïve Bayesian classifier* for a sample training data set stored as a .CSV file. Compute the accuracy of the classifier, considering few test data sets.

```
import csv
import random
import math
```

```
def loadCsv(filename):
    lines = csv.reader(open(filename))
    dataset = list(lines)
    for i in range(len(dataset)):
        dataset[i] = [float(x) for x in dataset[i]]
    return dataset


def splitDataset(dataset, splitRatio):
    trainSize = int(len(dataset) * splitRatio)
    trainSet = []
    copy = list(dataset)
    while len(trainSet) < trainSize:
        index = random.randrange(len(copy))
        trainSet.append(copy.pop(index))
    return [trainSet, copy]


def separateByClass(dataset):
    separated = {}
    for i in range(len(dataset)):
        vector = dataset[i]
        if (vector[-1] not in separated):
            separated[vector[-1]] = []
        separated[vector[-1]].append(vector)
    return separated


def mean(numbers):
    return sum(numbers) / float(len(numbers))


def stdev(numbers):
    avg = mean(numbers)
    variance = sum([pow(x - avg, 2) for x in numbers]) / float(len(numbers)
- 1)
    return math.sqrt(variance)


def summarize(dataset):
    summaries = [(mean(attribute), stdev(attribute)) for attribute in
zip(*dataset)]
    del summaries[-1]
    return summaries
```

```
def summarizeByClass(dataset):
    separated = separateByClass(dataset)
    summaries = {}
    for classValue, instances in separated.items():
        summaries[classValue] = summarize(instances)
    return summaries


def calculateProbability(x, mean, stdev):
    exponent = math.exp(-(math.pow(x - mean, 2) / (2 * math.pow(stdev, 2))))
    return (1 / (math.sqrt(2 * math.pi) * stdev)) * exponent


def calculateClassProbabilities(summaries, inputVector):
    probabilities = {}
    for classValue, classSummaries in summaries.items():
        probabilities[classValue] = 1
        for i in range(len(classSummaries)):
            mean, stdev = classSummaries[i]
            x = inputVector[i]
            probabilities[classValue] *= calculateProbability(x, mean,
stdev)
    return probabilities


def predict(summaries, inputVector):
    probabilities = calculateClassProbabilities(summaries, inputVector)
    bestLabel, bestProb = None, -1
    for classValue, probability in probabilities.items():
        if bestLabel is None or probability > bestProb:
            bestProb = probability
            bestLabel = classValue
    return bestLabel


def getPredictions(summaries, testSet):
    predictions = []
    for i in range(len(testSet)):
        result = predict(summaries, testSet[i])
        predictions.append(result)
    return predictions


def getAccuracy(testSet, predictions):
    correct = 0
    for i in range(len(testSet)):
        if testSet[i][-1] == predictions[i]:
            correct += 1
```

```
    return (correct / float(len(testSet))) * 100.0


filename = 'dataset/naivedata.csv'
splitRatio = 0.67
dataset = loadCsv(filename)
trainingSet, testSet = splitDataset(dataset, splitRatio)
print('Split {0} rows into train={1} and test={2} rows'.format(len(dataset),
len(trainingSet), len(testSet)))
    # prepare model
summaries = summarizeByClass(trainingSet)
    # test model
predictions = getPredictions(summaries, testSet)
accuracy = getAccuracy(testSet, predictions)
print('Accuracy: {0}%'.format(accuracy))
```

OUTPUT:

```
In [5]: runfile('G:/ML Lab Programs/naivebayes.py', wdir='G:/ML Lab Programs')
Split 768 rows into train=514 and test=254 rows
Accuracy: 76.77165354330708%
```

Program 6:

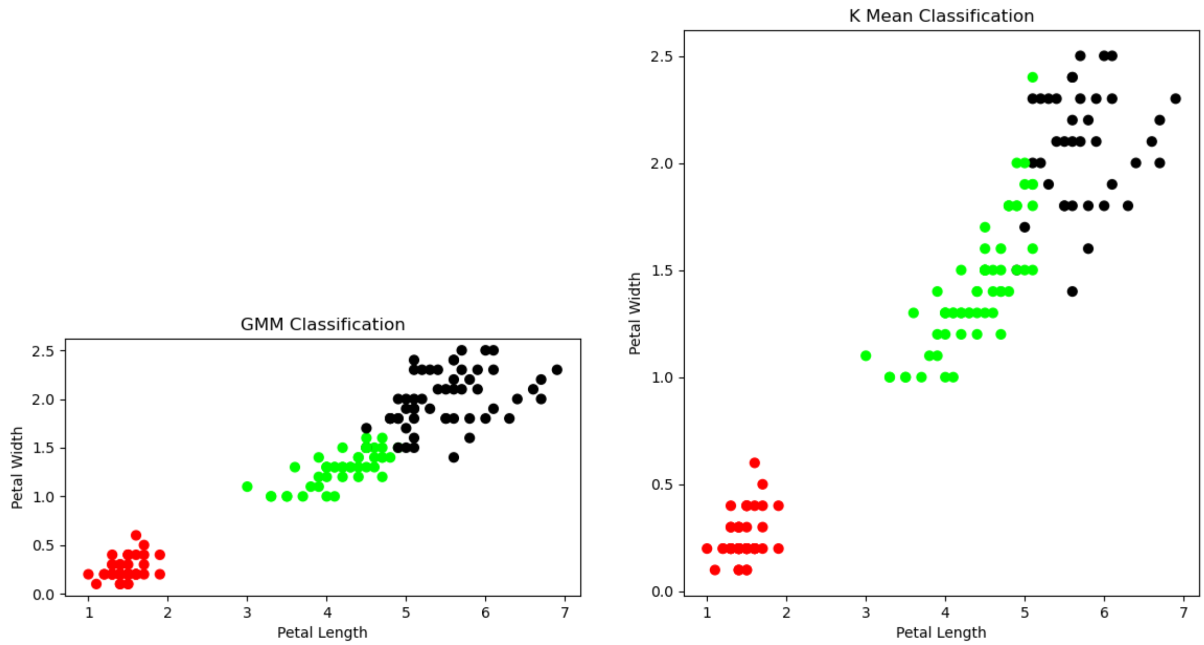
Apply *EM algorithm* to cluster a set of data stored in a .CSV file. Use the same data set for clustering using *k-Means algorithm*. Compare the results of these two algorithms and comment on the quality of clustering. You can add Java/Python ML library classes/API in the program.

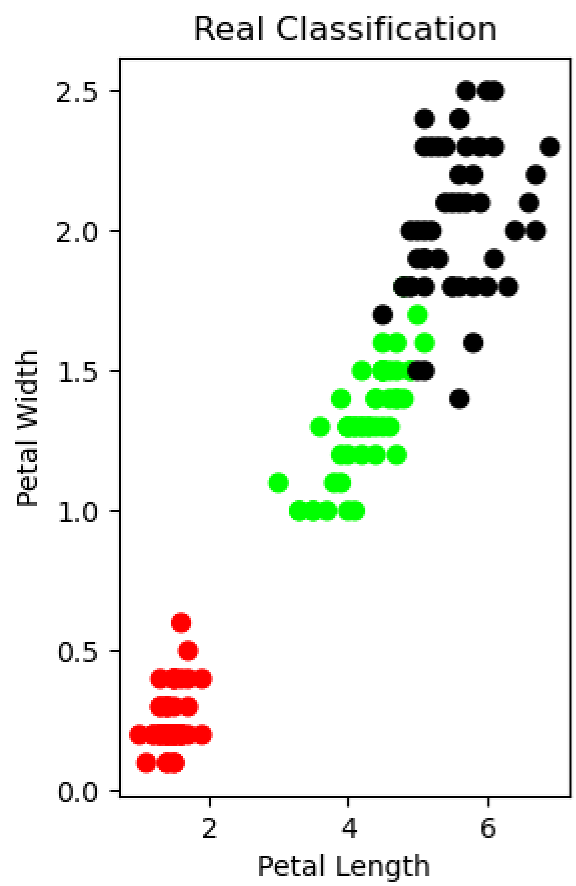
```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import KMeans
import sklearn.metrics as sm
import pandas as pd
import numpy as np

iris = datasets.load_iris()

X = pd.DataFrame(iris.data)
X.columns = ['Sepal_Length','Sepal_Width','Petal_Length','Petal_Width']

y = pd.DataFrame(iris.target)
y.columns = ['Targets']
```

```
model = KMeans(n_clusters=3)
model.fit(X)


plt.figure(figsize=(14,7))

colormap = np.array(['red', 'lime', 'black'])

# Plot the Original Classifications
plt.subplot(1, 2, 1)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y.Targets], s=40)
plt.title('Real Classification')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
plt.show()


# Plot the Models Classifications
plt.subplot(1, 2, 2)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[model.labels_], s=40)
plt.title('K Mean Classification')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
plt.show()
print('The accuracy score of K-Mean: ',sm.accuracy_score(y, model.labels_))
print('The Confusion matrixof K-Mean: ',sm.confusion_matrix(y,
model.labels_))


from sklearn import preprocessing
scaler = preprocessing.StandardScaler()
scaler.fit(X)
xsa = scaler.transform(X)
xs = pd.DataFrame(xsa, columns = X.columns)
#xs.sample(5)

from sklearn.mixture import GaussianMixture
gmm = GaussianMixture(n_components=3)
gmm.fit(xs)

y_gmm = gmm.predict(xs)
#y_cluster_gmm

plt.subplot(2, 2, 3)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y_gmm], s=40)
plt.title('GMM Classification')
plt.xlabel('Petal Length')
```

```
plt.ylabel('Petal Width')
plt.show()

print('The accuracy score of EM: ',sm.accuracy_score(y, y_gmm))
print('The Confusion matrix of EM: ',sm.confusion_matrix(y, y_gmm))
```

OUTPUT





Program 7:

Write a program to implement *k-Nearest Neighbour algorithm* to classify the iris data set.

```
# k-nearest neighbors on the Iris Flowers Dataset
from random import seed
from random import randrange
from csv import reader
from math import sqrt
import pandas as pd
import csv

#Load a CSV file
def load_csv(filename):
     dataset = list()
     with open(filename, 'r') as file:
          csv_reader = reader(file)
          for row in csv_reader:
               if not row:
                    continue
```

```
                dataset.append(row)
        return dataset
#dataset=pd.read_csv('irisdata.csv')
dataset=load_csv('dataset/iris.data')
# Convert string column to float
def str_column_to_float(dataset, column):
        for row in dataset:
                row[column] = float(row[column].strip())

# Convert string column to integer
def str_column_to_int(dataset, column):
        class_values = [row[column] for row in dataset]
        unique = set(class_values)
        lookup = dict()
        for i, value in enumerate(unique):
                lookup[value] = i
        for row in dataset:
                row[column] = lookup[row[column]]
        return lookup

# Find the min and max values for each column
def dataset_minmax(dataset):
        minmax = list()
        for i in range(len(dataset[0])):
                col_values = [row[i] for row in dataset]
                value_min = min(col_values)
                value_max = max(col_values)
                minmax.append([value_min, value_max])
        return minmax

# Rescale dataset columns to the range 0-1
def normalize_dataset(dataset, minmax):
        for row in dataset:
                for i in range(len(row)):
                        row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] -
minmax[i][0])

# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
        dataset_split = list()
        dataset_copy = list(dataset)
        fold_size = int(len(dataset) / n_folds)
        for _ in range(n_folds):
                fold = list()
                while len(fold) < fold_size:
                        index = randrange(len(dataset_copy))
                        fold.append(dataset_copy.pop(index))
                dataset_split.append(fold)
```

```
    return dataset_split

# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    return correct / float(len(actual)) * 100.0

# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
    folds = cross_validation_split(dataset, n_folds)
    scores = list()
    for fold in folds:
        train_set = list(folds)
        train_set.remove(fold)
        train_set = sum(train_set, [])
        test_set = list()
        for row in fold:
            row_copy = list(row)
            test_set.append(row_copy)
            row_copy[-1] = None
        predicted = algorithm(train_set, test_set, *args)
        actual = [row[-1] for row in fold]
        accuracy = accuracy_metric(actual, predicted)
        scores.append(accuracy)
    return scores

# Calculate the Euclidean distance between two vectors
def euclidean_distance(row1, row2):
    distance = 0.0
    for i in range(len(row1)-1):
        distance += (row1[i] - row2[i])**2
    return sqrt(distance)

# Locate the most similar neighbors
def get_neighbors(train, test_row, num_neighbors):
    distances = list()
    for train_row in train:
        dist = euclidean_distance(test_row, train_row)
        distances.append((train_row, dist))
    distances.sort(key=lambda tup: tup[1])
    neighbors = list()
    for i in range(num_neighbors):
        neighbors.append(distances[i][0])
    return neighbors
```

```
# Make a prediction with neighbors
def predict_classification(train, test_row, num_neighbors):
     neighbors = get_neighbors(train, test_row, num_neighbors)
     output_values = [row[-1] for row in neighbors]
     prediction = max(set(output_values), key=output_values.count)
     return prediction

# kNN Algorithm
def k_nearest_neighbors(train, test, num_neighbors):
     predictions = list()
     for row in test:
          output = predict_classification(train, row, num_neighbors)
          predictions.append(output)
     return(predictions)

# Test the kNN on the Iris Flowers dataset
seed(1)

for i in range(len(dataset[0])-1):
     str_column_to_float(dataset, i)
# convert class column to integers
str_column_to_int(dataset, len(dataset[0])-1)
# evaluate algorithm
n_folds = 5
num_neighbors = 5
scores = evaluate_algorithm(dataset, k_nearest_neighbors, n_folds,
num_neighbors)
print('Scores: %s' % scores)
print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

OUTPUT:

```
In [6]: runfile('G:/ML Lab Programs/prog7KNN.py', wdir='G:/ML Lab Programs')
Scores: [96.66666666666667, 96.66666666666667, 100.0, 90.0, 100.0]
Mean Accuracy: 96.667%

In [7]:
```

Program 8:

Implement the non-parametric *Locally Weighted Regression algorithm* in order to fit data points. Select appropriate data set for your experiment and draw graphs

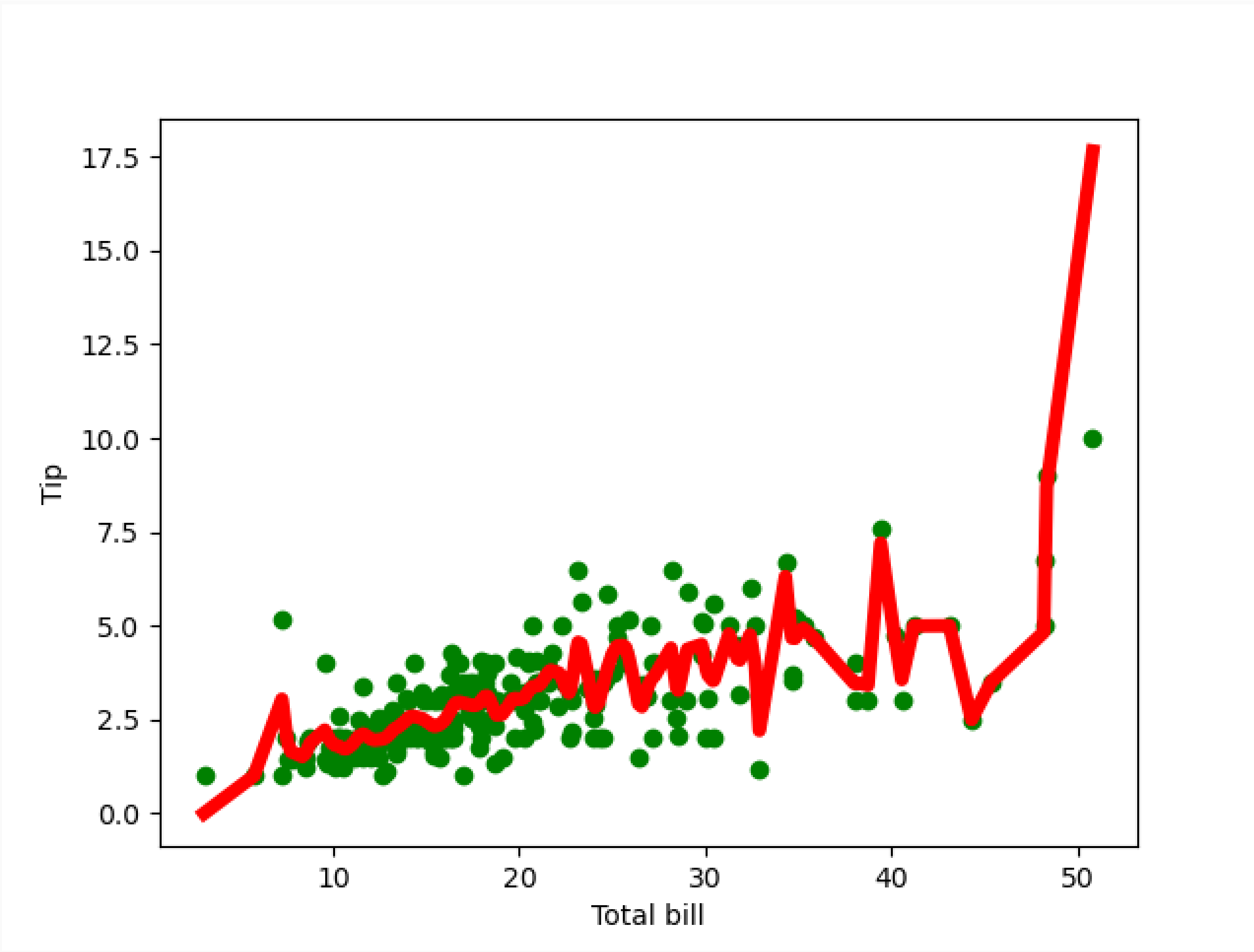
DATASET USED:

	A	B	C	D	E	F	G
1	total_bill	tip	sex	smoker	day	time	size
2	16.99	1.01	Female	No	Sun	Dinner	2
3	10.34	1.66	Male	No	Sun	Dinner	3
4	21.01	3.5	Male	No	Sun	Dinner	3
5	23.68	3.31	Male	No	Sun	Dinner	2
6	24.59	3.61	Female	No	Sun	Dinner	4
7	25.29	4.71	Male	No	Sun	Dinner	4
8	8.77	2	Male	No	Sun	Dinner	2
9	26.88	3.12	Male	No	Sun	Dinner	4

```
from numpy import *
from os import listdir
import matplotlib
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np1
import numpy.linalg as np
from scipy.stats.stats import pearsonr


def kernel(point, xmat, k):
    m, n = np1.shape(xmat)
    weights = np1.mat(np1.eye((m)))
    for j in range(m):
        diff = point - X[j]
        weights[j, j] = np1.exp(diff * diff.T / (-2.0 * k ** 2))
    return weights


def localWeight(point, xmat, ymat, k):
    wei = kernel(point, xmat, k)
    W = (X.T * (wei * X)).I * (X.T * (wei * ymat.T))
    return W


def localWeightRegression(xmat, ymat, k):
    m, n = np1.shape(xmat)
    ypred = np1.zeros(m)
    for i in range(m):
        ypred[i] = xmat[i] * localWeight(xmat[i], xmat, ymat, k)
    return ypred


# load data points
data = pd.read_csv('dataset/tips.csv')
```

```
bill = np1.array(data.total_bill)
tip = np1.array(data.tip)

# preparing and add 1 in bill
mbill = np1.mat(bill)
mtip = np1.mat(tip)  # mat is used to convert to n dimesiona to 2
dimensional array form
m = np1.shape(mbill)[1]
# print(m) 244 data is stored in m
one = np1.mat(np1.ones(m))
X = np1.hstack((one.T, mbill.T))  # create a stack of bill from ONE
# print(X)
# set k here
ypred = localWeightRegression(X, mtip, 0.3)
SortIndex = X[:, 1].argsort(0)
xsort = X[SortIndex][:, 0]

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.scatter(bill, tip, color='green')
ax.plot(xsort[:, 1], ypred[SortIndex], color='red', linewidth=5)
plt.xlabel('Total bill')
plt.ylabel('Tip')
plt.show();
```

OUTPUT



THANK YOU.