PESIT Bangalore South Campus

1Km before Electronic City, Hosur Road, Bangalore-560100.

DEPARTMENT OF INFORMATION SCIENCE ENGINEERING

VII SEMESTER

LAB MANUAL

SUBJECT: MACHINE LEARNING LABORATORY

SUBJECT CODE: 15CSL76

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

***** Create Excel file Weather.csv and save it in same path**

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

def loadCsv(filename):
        lines = csv.reader(open(filename, "rt"))
        dataset = list(lines)
        for i in range(len(dataset)):
                dataset[i] = dataset[i]
        return dataset

attributes = ['Sky','Temp','Humidity','Wind','Water','Forecast']
print(attributes)
num_attributes = len(attributes)

filename = "Weather.csv"
dataset = loadCsv(filename)
print(dataset)

target=['Yes','Yes','No','Yes']
print(target)

hypothesis=['0'] * num_attributes
print(hypothesis)

print("The Hypothesis are")
for i in range(len(target)):

    if(target[i] == 'Yes'):
        for j in range(num_attributes):
            if(hypothesis[j]=='0'):
                hypothesis[j] = dataset[i][j]
            if(hypothesis[j]!= dataset[i][j]):
                hypothesis[j]='?'

    print(i+1,'=',hypothesis)

print("Final Hypothesis")
print(hypothesis)
```

OUTPUT:

```
['Sky', 'Temp', 'Humidity', 'Wind', 'Water', 'Forecast']
[['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']]
['Yes', 'Yes', 'No', 'Yes']
['0', '0', '0', '0', '0', '0']

The Hypothesis are
1 = ['Sunny ', 'Warm', 'Normal', 'Strong ', 'Warm', 'Same']
2 = ['Sunny ', 'Warm', '?', 'Strong ', 'Warm', 'Same']
3 = ['Sunny ', 'Warm', '?', 'Strong ', 'Warm', 'Same']
4 = ['Sunny ', 'Warm', '?', 'Strong ', '?', '?']

Final Hypothesis
['Sunny ', 'Warm', '?', 'Strong ', '?', '?']
```

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.

***** Create Excel file Training_examples.csv and save it in same path**

Sky	Air	Humidity	Wind	Water	Forecast	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 numpy as np
import pandas as pd

# Loading Data from a CSV File
data = pd.DataFrame(data=pd.read_csv('Training_examples.csv'))

# Separating concept features from Target
concepts = np.array(data.iloc[:,0:-1])

# Isolating target into a separate DataFrame
#copying last column to target array
target = np.array(data.iloc[:,-1])


def learn(concepts, target):

'''    learn() function implements the learning method of the Candidate elimination
algorithm.
    Arguments:
    concepts - a data frame with all the features
    target - a data frame with corresponding output values
    '''
    # Initialise S0 with the first instance from concepts
    # .copy() makes sure a new list is created instead of just pointing to the same
memory location
    specific_h = concepts[0].copy()
    print("initialization of specific_h and general_h")
    print(specific_h)
    general_h = [["?" for i in range(len(specific_h))] for i in range(len(specific_h))]
    print(general_h)
    # The learning iterations
    for i, h in enumerate(concepts):

        # Checking if the hypothesis has a positive target
        if target[i] == "Yes":
            for x in range(len(specific_h)):

                # Change values in S & G only if values change
                if h[x] != specific_h[x]:
```

```
                specific_h[x] = '?'
                general_h[x][x] = '?'

        # Checking if the hypothesis has a positive target
        if target[i] == "No":
            for x in range(len(specific_h)):

                # For negative hyposthesis change values only  in G
                if h[x] != specific_h[x]:
                    general_h[x][x] = specific_h[x]
                else:
                    general_h[x][x] = '?'
        print(" steps of Candidate Elimination Algorithm",i+1)
        print(specific_h)
        print(general_h)
    # find indices where we have empty rows, meaning those that are unchanged
    indices = [i for i, val in enumerate(general_h) if val == ['?', '?', '?', '?', '?', '?']]
    for i in indices:
        # remove those rows from general_h
        general_h.remove(['?', '?', '?', '?', '?', '?'])

    # Return final values
    return specific_h, general_h
s_final, g_final = learn(concepts, target)
print("Final Specific_h:", s_final, sep="\n")
print("Final General_h:", g_final, sep="\n")
```

OUTPUT:

```
initialization of specific_h and general_h
['Sunny' 'Warm' 'High' 'Strong' 'Warm' 'Same']
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'],
['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]
 Steps of Candidate Elimination Algorithm 1
['Sunny' 'Warm' 'High' 'Strong' 'Warm' 'Same']
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'],
['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]
 Steps of Candidate Elimination Algorithm 2
['Sunny' 'Warm' 'High' 'Strong' 'Warm' 'Same']
[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?',
'?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', 'Same']]
 Steps of Candidate Elimination Algorithm 3
['Sunny' 'Warm' 'High' 'Strong' '?' '?']
[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?',
'?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Final Specific_h:
['Sunny' 'Warm' '?' 'Strong' '?' '?']

Final General_h:
[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?']]
```

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.

***** Create Excel file 'playtennis.csv' and save it in same path**

0	0	0	0	0
0	0	0	1	0
1	0	0	0	1
2	1	0	0	1
2	2	1	0	1
2	2	1	1	0
1	2	1	1	1
0	1	0	0	0
0	2	1	0	1
2	1	1	0	1
0	1	1	1	1
1	1	0	1	1
1	0	1	0	1
2	1	0	1	0
0	1	1	1	1
1	1	1	1	1

```
import pandas as pd
import numpy as np
#Import the dataset and define the feature as well as the target datasets / columns

dataset = pd.read_csv('playtennis.csv',
                      names=['outlook','temperature','humidity','wind','class',])

#Import all columns omitting the fist which consists the names of the animals
#We drop the animal names since this is not a good feature to split the data on

attributes =('Outlook','Temperature','Humidity','Wind','PlayTennis')
def entropy(target_col):

"""  Calculate the entropy of a dataset.
  The only parameter of this function is the target_col parameter which specifies the
target column   """

    elements,counts = np.unique(target_col,return_counts = True)

    entropy = np.sum([(-counts[i]/np.sum(counts))*np.log2(counts[i]/np.sum(counts)) for
i in range(len(elements))])
    #print('Entropy =', entropy)
    return entropy

def InfoGain(data,split_attribute_name,target_name="class"):
  #Calculate the entropy of the total dataset
    total_entropy = entropy(data[target_name])

  ##Calculate the entropy of the dataset

    #Calculate the values and the corresponding counts for the split attribute
```

```
            vals,counts= np.unique(data[split_attribute_name],return_counts=True)

            #Calculate the weighted entropy

            Weighted_Entropy =
        np.sum([(counts[i]/np.sum(counts))*entropy(data.where(data[split_attribute_name]==va
        ls[i]).dropna()[target_name]) for i in range(len(vals))])

            #Calculate the information gain

            Information_Gain = total_entropy - Weighted_Entropy
            return Information_Gain


        def ID3(data,originaldata,features,target_attribute_name="class",parent_node_class =
        None):

#Define the stopping criteria --> If one of this is satisfied, we want to return a leaf node#

    #If all target_values have the same value, return this value

            if len(np.unique(data[target_attribute_name])) <= 1:
                return np.unique(data[target_attribute_name])[0]

    #If the dataset is empty, return the mode target feature value in the original dataset

            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

    #If none of the above holds true, grow the tree!

            else:
     #Set the default value for this node --> The mode target feature value of the current
node
                parent_node_class = np.unique(data[target_attribute_name])
        [np.argmax(np.unique(data[target_attribute_name],return_counts=True)[1])]

  #Select the feature which best splits the dataset

          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]

 #Create the tree structure. The root gets the name of the feature (best_feature) with the
maximum information gain in the first run
                tree = {best_feature:{}}


#Remove the feature with the best inforamtion gain from the feature space
                features = [i for i in features if i != best_feature]
```

#Grow a branch under the root node for each possible value of the root node feature

```
        for value in np.unique(data[best_feature]):
            value = value
```
#Split the dataset along the value of the feature with the largest information gain and there with create sub_datasets
```
            sub_data = data.where(data[best_feature] == value).dropna()
```

#Call the ID3 algorithm for each of those sub_datasets with the new parameters --> Here the recursion comes in!
```
            subtree =
ID3(sub_data,dataset,features,target_attribute_name,parent_node_class)
```

#Add the sub tree, grown from the sub_dataset to the tree under the root node
```
            tree[best_feature][value] = subtree

        return(tree)

def predict(query,tree,default = 1):
```

#1.
```
    for key in list(query.keys()):
        if key in list(tree.keys()):
```
#2.
```
            try:
                result = tree[key][query[key]]
            except:
                return default
```
#3.
```
            result = tree[key][query[key]]
```
#4.

```
            if isinstance(result,dict):
                return predict(query,result)
            else:
                return result

def train_test_split(dataset):
    training_data = dataset.iloc[:14].reset_index(drop=True)
```
#We drop the index respectively relabel the index
#starting form 0, because we do not want to run into errors regarding the row labels / indexe #testing_data = dataset.iloc[10:].reset_index(drop=True)

```
    return training_data
```
#,testing_data

```
def test(data,tree):
```

#Create new query instances by simply removing the target feature column from the original #dataset and Convert it to a dictionary

```
queries = data.iloc[:,:-1].to_dict(orient = "records")
```

#Create a empty DataFrame in whose columns the prediction of the tree are stored
```
    predicted = pd.DataFrame(columns=["predicted"])
```

```
#Calculate the prediction accuracy
        for i in range(len(data)):
            predicted.loc[i,"predicted"] = predict(queries[i],tree,1.0)

        print('The prediction accuracy is: ',(np.sum(predicted["predicted"] ==
    data["class"])/len(data))*100,'%')

"""
Train the tree, Print the tree and predict the accuracy
"""
    XX = train_test_split(dataset)
    training_data=XX
    #testing_data=XX[1]
    tree = ID3(training_data,training_data,training_data.columns[:-1])
    print(' Display Tree',tree)
    print('len=',len(training_data))
    test(training_data,tree)
```

OUTPUT:

```
Display Tree {'outlook': {0: {'humidity': {0.0: 0.0, 1.0: 1.0}}, 1: 1.0, 2: {'wind': {0.0:
1.0, 1.0: 0.0}}}}
len= 14
The prediction accuracy is:  100.0 %
```

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

```
from math import exp
from random import seed
from random import random

# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
	network = list()
	hidden_layer = [{'weights':[random() for i in range(n_inputs + 1)]} for i in
range(n_hidden)]
	network.append(hidden_layer)
	output_layer = [{'weights':[random() for i in range(n_hidden + 1)]} for i in
range(n_outputs)]
	network.append(output_layer)
	return network

# Calculate neuron activation for an input
def activate(weights, inputs):
	activation = weights[-1]
	for i in range(len(weights)-1):
		activation += weights[i] * inputs[i]
	return activation

# Transfer neuron activation
def transfer(activation):
	return 1.0 / (1.0 + exp(-activation))

# Forward propagate input to a network output
def forward_propagate(network, row):
	inputs = row
	for layer in network:
		new_inputs = []
		for neuron in layer:
			activation = activate(neuron['weights'], inputs)
			neuron['output'] = transfer(activation)
			new_inputs.append(neuron['output'])
		inputs = new_inputs
	return inputs

# Calculate the derivative of an neuron output
def transfer_derivative(output):
	return output * (1.0 - output)

# Backpropagate error and store in neurons
def backward_propagate_error(network, expected):
	for i in reversed(range(len(network))):
		layer = network[i]
		errors = list()
		if i != len(network)-1:
			for j in range(len(layer)):
				error = 0.0
				for neuron in network[i + 1]:
```

```
                                        error += (neuron['weights'][j] * neuron['delta'])
                                errors.append(error)
                else:
                        for j in range(len(layer)):
                                neuron = layer[j]
                                errors.append(expected[j] - neuron['output'])
                for j in range(len(layer)):
                        neuron = layer[j]
                        neuron['delta'] = errors[j] * transfer_derivative(neuron['output'])
```

```
# Update network weights with error
def update_weights(network, row, l_rate):
        for i in range(len(network)):
                inputs = row[:-1]
                if i != 0:
                        inputs = [neuron['output'] for neuron in network[i - 1]]
                for neuron in network[i]:
                        for j in range(len(inputs)):
                                neuron['weights'][j] += l_rate * neuron['delta'] * inputs[j]
                        neuron['weights'][-1] += l_rate * neuron['delta']

# Train a network for a fixed number of epochs
def train_network(network, train, l_rate, n_epoch, n_outputs):
        for epoch in range(n_epoch):
                sum_error = 0
                for row in train:
                        outputs = forward_propagate(network, row)
                        expected = [0 for i in range(n_outputs)]
                        expected[row[-1]] = 1
                        sum_error += sum([(expected[i]-outputs[i])**2 for i in
range(len(expected))])
                        backward_propagate_error(network, expected)
                        update_weights(network, row, l_rate)
                print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))

# Test training backprop algorithm
seed(1)
dataset = [[2.7810836,2.550537003,0],
             [1.465489372,2.362125076,0],
        [3.396561688,4.400293529,0],
        [1.38807019,1.850220317,0],
        [3.06407232,3.005305973,0],
        [7.627531214,2.759262235,1],
        [5.332441248,2.088626775,1],
        [6.922596716,1.77106367,1],
        [8.675418651,-0.242068655,1],
        [7.673756466,3.508563011,1]]
n_inputs = len(dataset[0]) - 1
n_outputs = len(set([row[-1] for row in dataset]))
network = initialize_network(n_inputs, 2, n_outputs)
train_network(network, dataset, 0.5, 20, n_outputs)
for layer in network:
        print(layer)
```

OUTPUT:

```
>epoch=0, lrate=0.500, error=6.350
>epoch=1, lrate=0.500, error=5.531
>epoch=2, lrate=0.500, error=5.221
>epoch=3, lrate=0.500, error=4.951
>epoch=4, lrate=0.500, error=4.519
>epoch=5, lrate=0.500, error=4.173
>epoch=6, lrate=0.500, error=3.835
>epoch=7, lrate=0.500, error=3.506
>epoch=8, lrate=0.500, error=3.192
>epoch=9, lrate=0.500, error=2.898
>epoch=10, lrate=0.500, error=2.626
>epoch=11, lrate=0.500, error=2.377
>epoch=12, lrate=0.500, error=2.153
>epoch=13, lrate=0.500, error=1.953
>epoch=14, lrate=0.500, error=1.774
>epoch=15, lrate=0.500, error=1.614
>epoch=16, lrate=0.500, error=1.472
>epoch=17, lrate=0.500, error=1.346
>epoch=18, lrate=0.500, error=1.233
>epoch=19, lrate=0.500, error=1.132
[{'weights': [-1.4688375095432327, 1.850887325439514, 1.0858178629550297], 'output': 0.029980305604426185, 'delta': -0.0059546604162323625}, {'weights': [0.37711098142462157, -0.0625909894552989, 0.2765123702642716], 'output': 0.9456229000211323, 'delta': 0.0026279652850863837}]
[{'weights': [2.515394649397849, -0.3391927502445985, -0.9671565426390275], 'output': 0.23648794202357587, 'delta': -0.04270059278364587}, {'weights': [-2.5584149848484263, 1.0036422106209202, 0.42383086467582715], 'output': 0.7790535202438367, 'delta': 0.03803132596437354}]
```

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.

***** Create Excel file DBetes.csv and save it in same path**

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

#1.Load Data

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

#Split the data into Training and Testing randomly

```
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]
```

#Seperatedata by Class

```
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
```

#Calculate Mean

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

#Calculate Standard Deviation

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

#Summarize the data

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

```
            return summaries
```

#Summarize Attributes by Class

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

#Calculate Gaussian Probability Density Function

```
    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
```

#Calculate Class Probabilities

```
    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
```

#Make a Prediction

```
    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
```

#return a list of predictions for each test instance.

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

#calculate accuracy ratio.

```
    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 = 'DBetes.csv'
    splitRatio = 0.70
```

```
    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:

```
Split 250 rows into train=175 and test=75 rows
2
{1.0: [(5.188405797101449, 3.144908875135665), (141.1159420289855,
30.431473757532896), (72.44927536231884, 18.13635950878467),
(19.855072463768117, 17.342802679327338), (113.08695652173913,
159.1615660015684)], 0.0: [(3.2735849056603774, 2.792960603162459),
(109.0754716981132, 26.201671380061143), (69.5, 16.88405841530491),
(19.358490566037737, 15.185951326799056), (68.72641509433963,
111.65606485725267)]}
Accuracy: 68.0%
```

6. Assuming a set of documents that need to be classified, use the **naïve Bayesian Classifier** model to perform this task. Built-in Java classes/API can be used to write the program. Calculate the accuracy, precision, and recall for your data set.

```
from sklearn.datasets import fetch_20newsgroups
twenty_train = fetch_20newsgroups(subset='train', shuffle=True)
print("lenth of the twenty_train--------->", len(twenty_train))
#print(twenty_train.target_names)  #prints all the categories

print("***First Line of the First Data File***")
#print("\n".join(twenty_train.data[0].split("\n")[:5]))#prints first line of the first data file

#2 Extracting features from text files
from sklearn.feature_extraction.text import CountVectorizer
count_vect = CountVectorizer()
X_train_counts = count_vect.fit_transform(twenty_train.data)
print('dim=',X_train_counts.shape)

#3 TF-IDF
from sklearn.feature_extraction.text import TfidfTransformer
tfidf_transformer = TfidfTransformer()
X_train_tfidf = tfidf_transformer.fit_transform(X_train_counts)
print(X_train_tfidf.shape)

# Machine Learning
#4 Training Naive Bayes (NB) classifier on training data.
from sklearn.naive_bayes import MultinomialNB
clf = MultinomialNB().fit(X_train_tfidf, twenty_train.target)

# Building a pipeline: We can write less code and do all of the above, by building a pipeline as follows:
# The names 'vect' , 'tfidf' and 'clf' are arbitrary but will be used later.
# We will be using the 'text_clf' going forward.
from sklearn.pipeline import Pipeline
text_clf = Pipeline([('vect', CountVectorizer()), ('tfidf', TfidfTransformer()), ('clf', MultinomialNB())])
text_clf = text_clf.fit(twenty_train.data, twenty_train.target)

# Performance of NB Classifier
import numpy as np
twenty_test = fetch_20newsgroups(subset='test', shuffle=True)
predicted = text_clf.predict(twenty_test.data)
accuracy=np.mean(predicted == twenty_test.target)
print("Predicted Accuracy = ",accuracy)

#To Calculate Accuracy,Precision,Recall
from sklearn import metrics
print("Accuracy= ",metrics.accuracy_score(twenty_test.target,predicted))
print("Precision=",metrics.precision_score(twenty_test.target,predicted,average=None))
print("Recall=",metrics.recall_score(twenty_test.target,predicted,average=None))
print(metrics.classification_report(twenty_test.target,
predicted,target_names=twenty_test.target_names))
```

OUTPUT:

Downloading 20news dataset. This may take a few minutes.
Downloading dataset from https://ndownloader.figshare.com/files/5975967 (14 MB)
lenth of the twenty_train---------> 6
First Line of the First Data File
dim= (11314, 130107)
(11314, 130107)
Predicted Accuracy = 0.7738980350504514
Accuracy= 0.7738980350504514
Precision= [0.80193237 0.81028939 0.81904762 0.67180617 0.85632184 0.88955224
0.93127148 0.84651163 0.93686869 0.92248062 0.89170507 0.59379845
0.83629893 0.92113565 0.84172662 0.43896976 0.64339623 0.92972973
0.95555556 0.97222222]
Recall= [0.52037618 0.64781491 0.65482234 0.77806122 0.77402597 0.75443038
0.69487179 0.91919192 0.9321608 0.89924433 0.96992481 0.96717172
0.59796438 0.73737374 0.89086294 0.98492462 0.93681319 0.91489362
0.41612903 0.13944223]

	precision	recall	f1-score	support
alt.atheism	0.80	0.52	0.63	319
comp.graphics	0.81	0.65	0.72	389
comp.os.ms-windows.misc	0.82	0.65	0.73	394
comp.sys.ibm.pc.hardware	0.67	0.78	0.72	392
comp.sys.mac.hardware	0.86	0.77	0.81	385
comp.windows.x	0.89	0.75	0.82	395
misc.forsale	0.93	0.69	0.80	390
rec.autos	0.85	0.92	0.88	396
rec.motorcycles	0.94	0.93	0.93	398
rec.sport.baseball	0.92	0.90	0.91	397
rec.sport.hockey	0.89	0.97	0.93	399
sci.crypt	0.59	0.97	0.74	396
sci.electronics	0.84	0.60	0.70	393
sci.med	0.92	0.74	0.82	396
sci.space	0.84	0.89	0.87	394
soc.religion.christian	0.44	0.98	0.61	398
talk.politics.guns	0.64	0.94	0.76	364
talk.politics.mideast	0.93	0.91	0.92	376
talk.politics.misc	0.96	0.42	0.58	310
talk.religion.misc	0.97	0.14	0.24	251
avg / total	0.82	0.77	0.77	7532

7. Write a program to construct a **Bayesian network** considering medical data. Use this model to demonstrate the diagnosis of heart patients using standard Heart Disease Data Set. You can use Java/Python ML library classes/API.

```
import bayespy as bp
import numpy as np
import csv
from colorama import init
from colorama import Fore, Back, Style
init()

# Define Parameter Enum values
#Age
ageEnum = {'SuperSeniorCitizen':0, 'SeniorCitizen':1, 'MiddleAged':2, 'Youth':3,
'Teen':4}
# Gender
genderEnum = {'Male':0, 'Female':1}
# FamilyHistory
familyHistoryEnum = {'Yes':0, 'No':1}
# Diet(Calorie Intake)
dietEnum = {'High':0, 'Medium':1, 'Low':2}
# LifeStyle
lifeStyleEnum = {'Athlete':0, 'Active':1, 'Moderate':2, 'Sedetary':3}
# Cholesterol
cholesterolEnum = {'High':0, 'BorderLine':1, 'Normal':2}
# HeartDisease
heartDiseaseEnum = {'Yes':0, 'No':1}
#heart_disease_data.csv
with open('heart_disease_data.csv') as csvfile:
    lines = csv.reader(csvfile)
    dataset = list(lines)
    data = []
    for x in dataset:

data.append([ageEnum[x[0]],genderEnum[x[1]],familyHistoryEnum[x[2]],dietEnum[x[
3]],lifeStyleEnum[x[4]],cholesterolEnum[x[5]],heartDiseaseEnum[x[6]]])
# Training data for machine learning todo: should import from csv
data = np.array(data)
N = len(data)

# Input data column assignment
p_age = bp.nodes.Dirichlet(1.0*np.ones(5))
age = bp.nodes.Categorical(p_age, plates=(N,))
age.observe(data[:,0])

p_gender = bp.nodes.Dirichlet(1.0*np.ones(2))
gender = bp.nodes.Categorical(p_gender, plates=(N,))
gender.observe(data[:,1])

p_familyhistory = bp.nodes.Dirichlet(1.0*np.ones(2))
familyhistory = bp.nodes.Categorical(p_familyhistory, plates=(N,))
familyhistory.observe(data[:,2])

p_diet = bp.nodes.Dirichlet(1.0*np.ones(3))
```

```
    diet = bp.nodes.Categorical(p_diet, plates=(N,))
    diet.observe(data[:,3])

    p_lifestyle = bp.nodes.Dirichlet(1.0*np.ones(4))
    lifestyle = bp.nodes.Categorical(p_lifestyle, plates=(N,))
    lifestyle.observe(data[:,4])

    p_cholesterol = bp.nodes.Dirichlet(1.0*np.ones(3))
    cholesterol = bp.nodes.Categorical(p_cholesterol, plates=(N,))
    cholesterol.observe(data[:,5])

# Prepare nodes and establish edges
# np.ones(2) ->  HeartDisease has 2 options Yes/No
# plates(5, 2, 2, 3, 4, 3)  ->  corresponds to options present for domain values
    p_heartdisease = bp.nodes.Dirichlet(np.ones(2), plates=(5, 2, 2, 3, 4, 3))
    heartdisease = bp.nodes.MultiMixture([age, gender, familyhistory, diet, lifestyle,
    cholesterol], bp.nodes.Categorical, p_heartdisease)
    heartdisease.observe(data[:,6])
    p_heartdisease.update()

# Sample Test with hardcoded values
#print("Sample Probability")
#print("Probability(HeartDisease|Age=SuperSeniorCitizen, Gender=Female,
FamilyHistory=Yes, DietIntake=Medium, LifeStyle=Sedetary, Cholesterol=High)")
#print(bp.nodes.MultiMixture([ageEnum['SuperSeniorCitizen'], genderEnum['Female'],
familyHistoryEnum['Yes'], dietEnum['Medium'], lifeStyleEnum['Sedetary'],
cholesterolEnum['High']], bp.nodes.Categorical, p_heartdisease).get_moments()[0]
[heartDiseaseEnum['Yes']])

# Interactive Test
    m = 0
    while m == 0:
      print("\n")
      res = bp.nodes.MultiMixture([int(input('Enter Age: ' + str(ageEnum))),
    int(input('Enter Gender: ' + str(genderEnum))), int(input('Enter FamilyHistory: ' +
    str(familyHistoryEnum))), int(input('Enter dietEnum: ' + str(dietEnum))),
    int(input('Enter LifeStyle: ' + str(lifeStyleEnum))), int(input('Enter Cholesterol: ' +
    str(cholesterolEnum)))], bp.nodes.Categorical, p_heartdisease).get_moments()[0]
    [heartDiseaseEnum['Yes']]
      print("Probability(HeartDisease) = " +  str(res))

 #print(Style.RESET_ALL)

     m = int(input("Enter for Continue:0, Exit :1  "))
```

OUTPUT:

```
Enter Age: {'SuperSeniorCitizen': 0, 'SeniorCitizen': 1, 'MiddleAged': 2, 'Youth': 3, 'Teen': 4}0

Enter Gender: {'Male': 0, 'Female': 1}0

Enter FamilyHistory: {'Yes': 0, 'No': 1}0

Enter dietEnum: {'High': 0, 'Medium': 1, 'Low': 2}0

Enter LifeStyle: {'Athlete': 0, 'Active': 1, 'Moderate': 2, 'Sedetary': 3}0

Enter Cholesterol: {'High': 0, 'BorderLine': 1, 'Normal': 2}0
Probability(HeartDisease) = 0.5

Enter for Continue:0, Exit :1 0


Enter Age: {'SuperSeniorCitizen': 0, 'SeniorCitizen': 1, 'MiddleAged': 2, 'Youth': 3, 'Teen': 4}4

Enter Gender: {'Male': 0, 'Female': 1}0

Enter FamilyHistory: {'Yes': 0, 'No': 1}0

Enter dietEnum: {'High': 0, 'Medium': 1, 'Low': 2}1

Enter LifeStyle: {'Athlete': 0, 'Active': 1, 'Moderate': 2, 'Sedetary': 3}3

Enter Cholesterol: {'High': 0, 'BorderLine': 1, 'Normal': 2}2
Probability(HeartDisease) = 0.13784165696493575

Enter for Continue:0, Exit :1 0

Enter Age: {'SuperSeniorCitizen': 0, 'SeniorCitizen': 1, 'MiddleAged': 2, 'Youth': 3, 'Teen': 4}3

Enter Gender: {'Male': 0, 'Female': 1}1

Enter FamilyHistory: {'Yes': 0, 'No': 1}0

Enter dietEnum: {'High': 0, 'Medium': 1, 'Low': 2}1

Enter LifeStyle: {'Athlete': 0, 'Active': 1, 'Moderate': 2, 'Sedetary': 3}0

Enter Cholesterol: {'High': 0, 'BorderLine': 1, 'Normal': 2}1
Probability(HeartDisease) = 0.2689414213699951

Enter for Continue:0, Exit :1
```

8. Apply **EM algorithm** to cluster a set of data stored in a .CSV file. Use the same dataset 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.

**EM algorithm

```
import numpy as np
from scipy import stats

np.random.seed(110)
```

 # for reproducible random results
 # set parameters

```
red_mean = 3
red_std = 0.8

blue_mean = 7
blue_std = 1
```

 # draw 40 samples from normal distributions with red/blue parameters

```
red = np.random.normal(red_mean, red_std, size=40)
blue = np.random.normal(blue_mean, blue_std, size=40)

both_colours = np.sort(np.concatenate((red, blue)))
```

#Since the colours are hidden from us, we will start the EM process
#Starting guesses are very critical because the EM Algorithm converges to
 # a local maxima. Hence we can get different answers with different starting points
#One reasonably good guess would be to take the value from a different but less
#robust algorithm
 # estimates for the mean

```
red_mean_guess = 2.1
blue_mean_guess = 6
```

 # estimates for the standard deviation

```
red_std_guess = 1.5
blue_std_guess = 0.8
```

#These are pretty bad guesses
#To continue with EM and improve these guesses, we compute the likelihood
#of each data point (regardless of its secret colour) appearing under
#these guesses for the mean and standard deviation

#The variable both_colours holds each data point. The function stats.norm computes
#the probability of the point under a normal distribution with the given parameters:

```
for i in range(10):
   likelihood_of_red = stats.norm(red_mean_guess, red_std_guess).pdf(both_colours)
   likelihood_of_blue = stats.norm(blue_mean_guess,
blue_std_guess).pdf(both_colours)
```

#Normalize these weights so that they can total 1

```
likelihood_total = likelihood_of_red + likelihood_of_blue

red_weight = likelihood_of_red / likelihood_total
blue_weight = likelihood_of_blue / likelihood_total
```

#With our current estimates and our newly-computed weights, we can now compute new,
#probably better, estimates for the parameters (step 4). We need a function for the
#mean and a function for the standard deviation:

```
def estimate_mean(data, weight):
    return np.sum(data * weight) / np.sum(weight)

def estimate_std(data, weight, mean):
    variance = np.sum(weight * (data - mean)**2) / np.sum(weight)
    return np.sqrt(variance)
```

 # new estimates for standard deviation

```
blue_std_guess = estimate_std(both_colours, blue_weight, blue_mean_guess)
red_std_guess = estimate_std(both_colours, red_weight, red_mean_guess)
```

 # new estimates for mean

```
red_mean_guess = estimate_mean(both_colours, red_weight)
blue_mean_guess = estimate_mean(both_colours, blue_weight)
```

#Lets print the model parameters (The means and the std deviation in our case)

```
print("red mean:", red_mean_guess, ":::::::::", "blue mean:", blue_mean_guess)
print("red std:", red_std_guess, ":::::::::", "blue std:", blue_std_guess)
```

#plot the data

```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.mlab as mlab
```

#The two Gaussian distributions

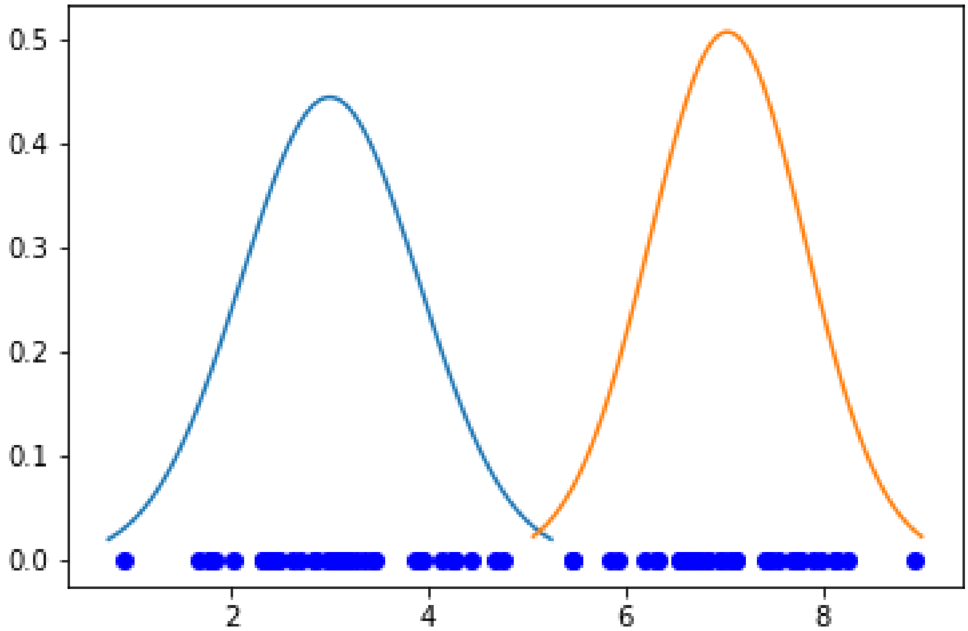
```
y = np.zeros(len(both_colours))
mured = red_mean_guess
sigmared = red_std_guess
x = np.linspace(mured - 2.5*sigmared, mured + 2.5*sigmared, 100)
plt.plot(x,mlab.normpdf(x, mured, sigmared))

mublue = blue_mean_guess
sigmablue = blue_std_guess
y = np.linspace(mublue - 2.5*sigmablue, mublue + 2.5*sigmablue, 100)
plt.plot(y,mlab.normpdf(y, mublue, sigmablue))
```

#The data points themselves

```
for i in range(len(both_colours)):
    plt.plot(both_colours[i],0,"bo")
plt.show()
```

OUTPUT:

```
red mean: 2.997142582038222 ::::::::: blue mean: 7.036259959647933
red std: 0.8992704481319626 ::::::::: blue std: 0.7882001074294297
```



****K-MEANS**

```
import pylab as pl
import numpy as np
from sklearn.cluster import KMeans

np.random.seed(110) # for reproducible random results
```

 # set parameters

```
        red_mean = 3
        red_std = 0.8

        blue_mean = 7
        blue_std = 1
```

 # draw 20 samples from normal distributions with red/blue parameters

```
        red = np.random.normal(red_mean, red_std, size=40)
        blue = np.random.normal(blue_mean, blue_std, size=40)

        both_colours = np.sort(np.concatenate((red, blue)))
        y = np.zeros(len(both_colours))
```

#We will need the elbow curve for calculating exact value of k
#But we will use 2 for now

```
        kmeans=KMeans(n_clusters=2)
        kmeansoutput=kmeans.fit(both_colours.reshape(-1,1))
```

#but what value of K was actually good?

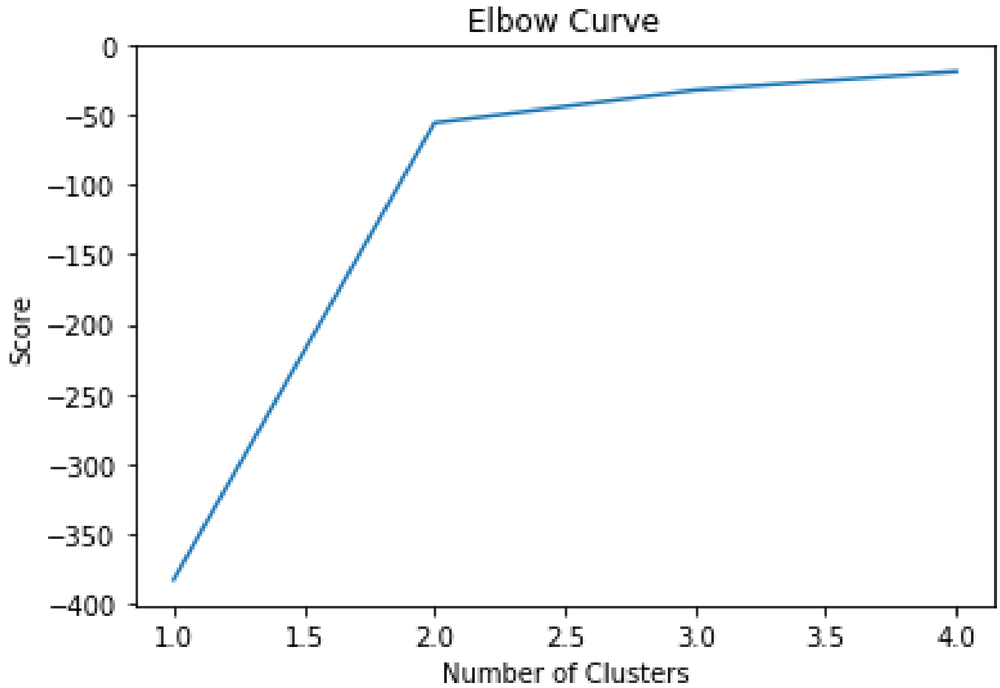
```
        Nc = range(1, 5)
```

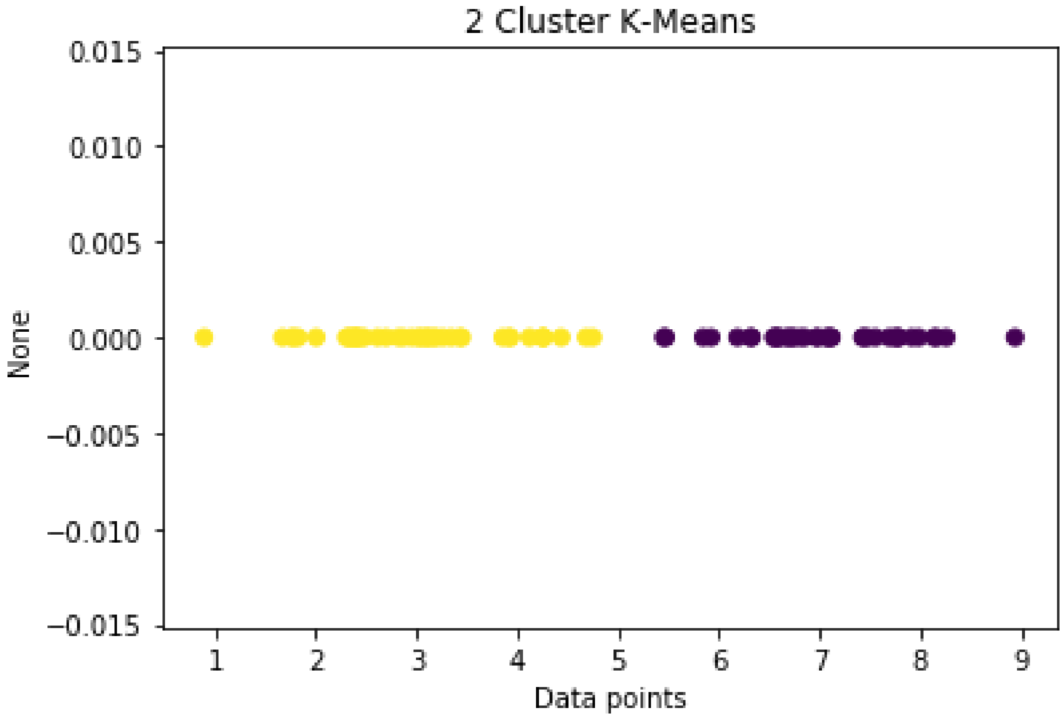
```
kmeans = [KMeans(n_clusters=i) for i in Nc]
score = [kmeans[i].fit(both_colours.reshape(-1,1)).score(both_colours.reshape(-1,1)) for
i in range(len(kmeans))]
pl.plot(Nc,score)

pl.xlabel('Number of Clusters')
pl.ylabel('Score')
pl.title('Elbow Curve')
pl.show()
```

#plot the points themselves

```
pl.scatter(both_colours,y,c=kmeansoutput.labels_)
pl.xlabel('Data points')
pl.ylabel('None')
pl.title('2 Cluster K-Means')
pl.show()
```

OUTPUT:




2 Cluster K-Means
0.015
0.010
0.005
0.000
−0.005
−0.010
−0.015
None
1
2
3
4
5
6
7
8
9
Data points

9. Write a program to implement ***k*-Nearest Neighbour algorithm** to classify the iris data set. Print both correct and wrong predictions. Java/Python ML library classes can be used for this problem.

#1.Import Data

```
from sklearn.datasets import load_iris
iris = load_iris()
print("Feature Names:",iris.feature_names,"Iris Data:",iris.data,"Target
Names:",iris.target_names,"Target:",iris.target)
```

#2. Split the data into Test and Data

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
        iris.data, iris.target, test_size = .25)
```

#neighbors_settings = range(1, 11)
#for n_neighbors in neighbors_settings:
#3.Build The Model

```
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier()
clf.fit(X_train, y_train)
```

#4.Calculate Accuracy of the Test data with the trained data

```
print(" Accuracy=",clf.score(X_test, y_test))
```

#5 Calculate the prediction with the labels of the test data

```
print("Predicted Data")
print(clf.predict(X_test))

prediction=clf.predict(X_test)

print("Test data :")
print(y_test)
```

#6 To identify the miss classification

```
diff=prediction-y_test
print("Result is ")
print(diff)
print('Total no of samples misclassied =', sum(abs(diff)))
```

OUTPUT:

```
Feature Names: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
 Iris Data:
 [[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]
 [5.4 3.9 1.7 0.4]
 [4.6 3.4 1.4 0.3]
 [5.  3.4 1.5 0.2]
 [4.4 2.9 1.4 0.2]
 [4.9 3.1 1.5 0.1]
 [5.4 3.7 1.5 0.2]
 [4.8 3.4 1.6 0.2]
 [4.8 3.  1.4 0.1]
 [4.3 3.  1.1 0.1]
 [5.8 4.  1.2 0.2]
 [5.7 4.4 1.5 0.4]
 [5.4 3.9 1.3 0.4]
 [5.1 3.5 1.4 0.3]
 [5.7 3.8 1.7 0.3]
 [5.1 3.8 1.5 0.3]
 [5.4 3.4 1.7 0.2]
 [5.1 3.7 1.5 0.4]
 [4.6 3.6 1.  0.2]
 [5.1 3.3 1.7 0.5]
 [4.8 3.4 1.9 0.2]
 [5.  3.  1.6 0.2]
 [5.  3.4 1.6 0.4]
 [5.2 3.5 1.5 0.2]
 [5.2 3.4 1.4 0.2]
 [4.7 3.2 1.6 0.2]
 [4.8 3.1 1.6 0.2]
 [5.4 3.4 1.5 0.4]
 [5.2 4.1 1.5 0.1]
 [5.5 4.2 1.4 0.2]
 [4.9 3.1 1.5 0.1]
 [5.  3.2 1.2 0.2]
 [5.5 3.5 1.3 0.2]
 [4.9 3.1 1.5 0.1]
 [4.4 3.  1.3 0.2]
 [5.1 3.4 1.5 0.2]
 [5.  3.5 1.3 0.3]
 [4.5 2.3 1.3 0.3]
 [4.4 3.2 1.3 0.2]
 [5.  3.5 1.6 0.6]
 [5.1 3.8 1.9 0.4]
 [4.8 3.  1.4 0.3]
 [5.1 3.8 1.6 0.2]
 [4.6 3.2 1.4 0.2]
 [5.3 3.7 1.5 0.2]
 [5.  3.3 1.4 0.2]
 [7.  3.2 4.7 1.4]
 [6.4 3.2 4.5 1.5]
 [6.9 3.1 4.9 1.5]
```

```
[5.5 2.3 4.  1.3]
[6.5 2.8 4.6 1.5]
[5.7 2.8 4.5 1.3]
[6.3 3.3 4.7 1.6]
[4.9 2.4 3.3 1. ]
[6.6 2.9 4.6 1.3]
[5.2 2.7 3.9 1.4]
[5.  2.  3.5 1. ]
[5.9 3.  4.2 1.5]
[6.  2.2 4.  1. ]
[6.1 2.9 4.7 1.4]
[5.6 2.9 3.6 1.3]
[6.7 3.1 4.4 1.4]
[5.6 3.  4.5 1.5]
[5.8 2.7 4.1 1. ]
[6.2 2.2 4.5 1.5]
[5.6 2.5 3.9 1.1]
[5.9 3.2 4.8 1.8]
[6.1 2.8 4.  1.3]
[6.3 2.5 4.9 1.5]
[6.1 2.8 4.7 1.2]
[6.4 2.9 4.3 1.3]
[6.6 3.  4.4 1.4]
[6.8 2.8 4.8 1.4]
[6.7 3.  5.  1.7]
[6.  2.9 4.5 1.5]
[5.7 2.6 3.5 1. ]
[5.5 2.4 3.8 1.1]
[5.5 2.4 3.7 1. ]
[5.8 2.7 3.9 1.2]
[6.  2.7 5.1 1.6]
[5.4 3.  4.5 1.5]
[6.  3.4 4.5 1.6]
[6.7 3.1 4.7 1.5]
[6.3 2.3 4.4 1.3]
[5.6 3.  4.1 1.3]
[5.5 2.5 4.  1.3]
[5.5 2.6 4.4 1.2]
[6.1 3.  4.6 1.4]
[5.8 2.6 4.  1.2]
[5.  2.3 3.3 1. ]
[5.6 2.7 4.2 1.3]
[5.7 3.  4.2 1.2]
[5.7 2.9 4.2 1.3]
[6.2 2.9 4.3 1.3]
[5.1 2.5 3.  1.1]
[5.7 2.8 4.1 1.3]
[6.3 3.3 6.  2.5]
[5.8 2.7 5.1 1.9]
[7.1 3.  5.9 2.1]
[6.3 2.9 5.6 1.8]
[6.5 3.  5.8 2.2]
[7.6 3.  6.6 2.1]
[4.9 2.5 4.5 1.7]
[7.3 2.9 6.3 1.8]
[6.7 2.5 5.8 1.8]
```

```
 [7.2 3.6 6.1 2.5]
 [6.5 3.2 5.1 2. ]
 [6.4 2.7 5.3 1.9]
 [6.8 3.  5.5 2.1]
 [5.7 2.5 5.  2. ]
 [5.8 2.8 5.1 2.4]
 [6.4 3.2 5.3 2.3]
 [6.5 3.  5.5 1.8]
 [7.7 3.8 6.7 2.2]
 [7.7 2.6 6.9 2.3]
 [6.  2.2 5.  1.5]
 [6.9 3.2 5.7 2.3]
 [5.6 2.8 4.9 2. ]
 [7.7 2.8 6.7 2. ]
 [6.3 2.7 4.9 1.8]
 [6.7 3.3 5.7 2.1]
 [7.2 3.2 6.  1.8]
 [6.2 2.8 4.8 1.8]
 [6.1 3.  4.9 1.8]
 [6.4 2.8 5.6 2.1]
 [7.2 3.  5.8 1.6]
 [7.4 2.8 6.1 1.9]
 [7.9 3.8 6.4 2. ]
 [6.4 2.8 5.6 2.2]
 [6.3 2.8 5.1 1.5]
 [6.1 2.6 5.6 1.4]
 [7.7 3.  6.1 2.3]
 [6.3 3.4 5.6 2.4]
 [6.4 3.1 5.5 1.8]
 [6.  3.  4.8 1.8]
 [6.9 3.1 5.4 2.1]
 [6.7 3.1 5.6 2.4]
 [6.9 3.1 5.1 2.3]
 [5.8 2.7 5.1 1.9]
 [6.8 3.2 5.9 2.3]
 [6.7 3.3 5.7 2.5]
 [6.7 3.  5.2 2.3]
 [6.3 2.5 5.  1.9]
 [6.5 3.  5.2 2. ]
 [6.2 3.4 5.4 2.3]
 [5.9 3.  5.1 1.8]]
Target Names: ['setosa' 'versicolor' 'virginica']
Target: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
 Accuracy= 0.9473684210526315
Predicted Data
[2 1 2 1 0 1 1 1 2 2 0 2 1 1 1 2 2 1 2 2 1 1 1 2 0 0 1 0 1 0 2 0 1 1 0 1 1  2]
Test data :
[2 1 2 1 0 1 1 1 1 2 0 2 1 1 1 2 2 1 2 1 1 1 1 2 0 0 1 0 1 0 2 0 1 1 0 1 1 2]
Result is
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0]
Total no of samples misclassied = 2
```

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

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

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

#Weigh each point by its distance to the reference point. We are considering
# All points here. If KNN was the topic, we could restrict this to "K"
        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 = np.shape(xmat)
            ypred = np.zeros(m)
            for i in range(m):
 # predicted value y = wx. Here w = weights we have computed.
 # Remember that both w and x are vectors here (2*1 and 1*2 respectively)
 # Resultant value of y is a scalar
      ypred[i] = xmat[i]*localWeight(xmat[i],xmat,ymat,k)
            return ypred


# load data points
        data = pd.read_csv('LR.csv')
        colA = np.array(data.colA)
        colB = np.array(data.colB)

#preparing and add 1
#convert to matrix form
        mcolA = np.mat(colA)
        mcolB = np.mat(colB)
        m= np.shape(mcolA)[1]
        one = np.ones((1,m),dtype=int)

#horizontally stack
        X= np.hstack((one.T,mcolA.T))
        print(X.shape)

#set k here (0.5)
        ypred = localWeightRegression(X,mcolB,0.5)
        SortIndex = X[:,1].argsort(0)
        xsort = X[SortIndex][:,0]
```

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

OUTPUT:

(80, 2)