Implement A* Search algorithm.

```
class Graph:

    def __init__(self,adjac_lis):

        self.adjac_lis = adjac_lis

    def get_neighbours(self,v):

        return self.adjac_lis[v]

    def h(self,n):

        H={'A':1,'B':1, 'C':1,'D':1}

        return H[n]

    def a_star_algorithm(self,start,stop):

        open_lst = set([start])

        closed_lst = set([])

        dist ={}

        dist[start] = 0

        prenode ={}

        prenode[start] =start

        while len(open_lst)>0:

            n = None

            for v in open_lst:

                if n==None or
dist[v]+self.h(v)<dist[n]+self.h(n):

                    n=v;

            if n==None:

                print("path doesnot exist")
```

```
        return None
    if n==stop:
         reconst_path=[]
         while prenode[n]!=n:
             reconst_path.append(n)
             n = prenode[n]
         reconst_path.append(start)
         reconst_path.reverse()
         print("path found:{}".format(reconst_path))
         return reconst_path
    for (m,weight) in self.get_neighbours(n):
        if m not in open_lst and m not in closed_lst:
            open_lst.add(m)
            prenode[m] = n
            dist[m] = dist[n]+weight
        else:
            if dist[m]>dist[n]+weight:
                dist[m] = dist[n]+weight
                prenode[m]=n
                if m in closed_lst:
                    closed_lst.remove(m)
                    open_lst.add(m)
    open_lst.remove(n)
    closed_lst.add(n)
```

```
        print("Path doesnot exist")

        return None

adjac_lis
={'A':[('B',1),('C',3),('D',7)],'B':[('D',5)],'C':[('D',12)]}

graph1=Graph(adjac_lis)

graph1.a_star_algorithm('A', 'D')
```

Output

path found:['A', 'B', 'D']

['A', 'B', 'D']

2. Implement AO* Algorithm

```
# Cost to find the AND and OR path
def Cost(H, condition, weight = 1):
	cost = {}
	if 'AND' in condition:
		AND_nodes = condition['AND']
		Path_A = ' AND '.join(AND_nodes)
		PathA = sum(H[node]+weight for node in AND_nodes)
		cost[Path_A] = PathA

	if 'OR' in condition:
		OR_nodes = condition['OR']
		Path_B =' OR '.join(OR_nodes)
		PathB = min(H[node]+weight for node in OR_nodes)
		cost[Path_B] = PathB
	return cost

# Update the cost
def update_cost(H, Conditions, weight=1):
	Main_nodes = list(Conditions.keys())
	Main_nodes.reverse()
	least_cost= {}
	for key in Main_nodes:
		condition = Conditions[key]
		print(key,':', Conditions[key],'>>>', Cost(H, condition, weight))
		c = Cost(H, condition, weight)
```

```
        H[key] = min(c.values())

        least_cost[key] = Cost(H, condition, weight)

    return least_cost


# Print the shortest path

def shortest_path(Start,Updated_cost, H):

    Path = Start

    if Start in Updated_cost.keys():

        Min_cost = min(Updated_cost[Start].values())

        key = list(Updated_cost[Start].keys())

        values = list(Updated_cost[Start].values())

        Index = values.index(Min_cost)


        # FIND MINIMIMUM PATH KEY

        Next = key[Index].split()

        # ADD TO PATH FOR OR PATH

        if len(Next) == 1:

            Start =Next[0]

            Path += '<--' +shortest_path(Start, Updated_cost, H)

        # ADD TO PATH FOR AND PATH

        else:

            Path +='<--('+key[Index]+') '

            Start = Next[0]

            Path += '[' +shortest_path(Start, Updated_cost, H) + ' + '


            Start = Next[-1]

            Path += shortest_path(Start, Updated_cost, H) + ']'
```

```
        return Path

H = {'A': -1, 'B': 5, 'C': 2, 'D': 4, 'E': 7, 'F': 9, 'G': 3, 'H': 0, 'I':0, 'J':0}

Conditions = {
'A': {'OR': ['B'], 'AND': ['C', 'D']},
'B': {'OR': ['E', 'F']},
'C': {'OR': ['G'], 'AND': ['H', 'I']},
'D': {'OR': ['J']}
}
# weight
weight = 1
# Updated cost
print('Updated Cost :')
Updated_cost = update_cost(H, Conditions, weight=1)
print('*'*75)
print('Shortest Path :\n',shortest_path('A', Updated_cost,H))
```

Output:

```
Updated Cost :
D : {'OR': ['J']} >>> {'J': 1}
C : {'OR': ['G'], 'AND': ['H', 'I']} >>> {'H AND I': 2, 'G': 4}
B : {'OR': ['E', 'F']} >>> {'E OR F': 8}
A : {'OR': ['B'], 'AND': ['C', 'D']} >>> {'C AND D': 5, 'B': 9}
***************************************************************************
Shortest Path :
 A<--(C AND D) [C<--(H AND I) [H + I] + D<--J]
```

3. FOR A GIVEN SET OF TRAINING DATA EXAMPLES STORED IN A .CSV FILE, IMPLEMENT AND DEMONSTRATE THE CANDIDATE-ELIMINATION ALGORITHMTO OUTPUT A DESCRIPTION OF THE SET OF ALL HYPOTHESES CONSISTENT WITH THE TRAINING EXAMPLES.

```
import numpy as np
import pandas as pd

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

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

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

def learn(concepts, target):



    # 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("\nInitialization of specific_h and general_h")
    print(specific_h)
    #h=["#" for i in range(0,5)]
    #print(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("\nSteps 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("\nFinal Specific_h:", s_final, sep="\n")
print("\nFinal General_h:", g_final, sep="\n")
```

OUTPUT

```
     sky airTemp humidity    wind water forecast enjoySport
0  Sunny    Warm   Normal  Strong  Warm     Same       Yes
1  Sunny    Warm     High  Strong  Warm     Same       Yes
2  Rainy    Cold     High  Strong  Warm   Change        No
3  Sunny    Warm     High  Strong  Cool   Change       Yes
[['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']
 ['Sunny' 'Warm' 'High' 'Strong' 'Warm' 'Same']
 ['Rainy' 'Cold' 'High' 'Strong' 'Warm' 'Change']
 ['Sunny' 'Warm' 'High' 'Strong' 'Cool' 'Change']]
['Yes' 'Yes' 'No' 'Yes']

Initialization of specific_h and general_h
['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?',
'?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Steps of Candidate Elimination Algorithm 1
['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?',
'?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Steps of Candidate Elimination Algorithm 2
['Sunny' 'Warm' '?' 'Strong' 'Warm' 'Same']
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?',
'?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Steps of Candidate Elimination Algorithm 3
```

['Sunny' 'Warm' '?' 'Strong' 'Warm' 'Same']
[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', 'Same']]

Steps of Candidate Elimination Algorithm 4
['Sunny' 'Warm' '?' 'Strong' '?' '?']
[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

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

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

```
import numpy as np
import math
import csv

def read_data(filename):
    with open(filename, 'r') as csvfile:
        datareader = csv.reader(csvfile, delimiter=',')
        headers = next(datareader)
        metadata = []
        traindata = []
        for name in headers:
            metadata.append(name)
        for row in datareader:
            traindata.append(row)

    return (metadata, traindata)

class Node:
    def __init__(self, attribute):
        self.attribute = attribute
        self.children = []
        self.answer = ""

    def __str__(self):
        return self.attribute

def subtables(data, col, delete):
    dict = {}
    items = np.unique(data[:, col])
    count = np.zeros((items.shape[0], 1), dtype=np.int32)

    for x in range(items.shape[0]):
        for y in range(data.shape[0]):
            if data[y, col] == items[x]:
                count[x] += 1

    for x in range(items.shape[0]):
        dict[items[x]] = np.empty((int(count[x]), data.shape[1]),
dtype="|S32")
        pos = 0
        for y in range(data.shape[0]):
            if data[y, col] == items[x]:
                dict[items[x]][pos] = data[y]
                pos += 1
        if delete:
            dict[items[x]] = np.delete(dict[items[x]], col, 1)

    return items, dict

def entropy(S):
    items = np.unique(S)

    if items.size == 1:
        return 0

    counts = np.zeros((items.shape[0], 1))
    sums = 0
```

```
    for x in range(items.shape[0]):
        counts[x] = sum(S == items[x]) / (S.size * 1.0)

    for count in counts:
        sums += -1 * count * math.log(count, 2)
    return sums

def gain_ratio(data, col):
    items, dict = subtables(data, col, delete=False)

    total_size = data.shape[0]
    entropies = np.zeros((items.shape[0], 1))
    intrinsic = np.zeros((items.shape[0], 1))

    for x in range(items.shape[0]):
        ratio = dict[items[x]].shape[0]/(total_size * 1.0)
        entropies[x] = ratio * entropy(dict[items[x]][:, -1])
        intrinsic[x] = ratio * math.log(ratio, 2)

    total_entropy = entropy(data[:, -1])
    iv = -1 * sum(intrinsic)

    for x in range(entropies.shape[0]):
        total_entropy -= entropies[x]

    return total_entropy / iv

def create_node(data, metadata):
    if (np.unique(data[:, -1])).shape[0] == 1:
        node = Node("")
        node.answer = np.unique(data[:, -1])[0]
        return node

    gains = np.zeros((data.shape[1] - 1, 1))

    for col in range(data.shape[1] - 1):
        gains[col] = gain_ratio(data, col)

    split = np.argmax(gains)

    node = Node(metadata[split])
    metadata = np.delete(metadata, split, 0)

    items, dict = subtables(data, split, delete=True)

    for x in range(items.shape[0]):
        child = create_node(dict[items[x]], metadata)
        node.children.append((items[x], child))

    return node

def empty(size):
    s = ""
    for x in range(size):
        s += "   "
    return s
```

```
def print_tree(node, level):
    if node.answer != "":
        print(empty(level), node.answer)
        return
    print(empty(level), node.attribute)
    for value, n in node.children:
        print(empty(level + 1), value)
        print_tree(n, level + 2)

metadata, traindata = read_data("tennisdata.csv")
data = np.array(traindata)
node = create_node(data, metadata)
print_tree(node, 0)
```

Program 5

```
import numpy as np

X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)     # X = (hours sleeping, hours studying)

y = np.array(([92], [86], [89]), dtype=float)          # y = score on test


# scale units

X = X/np.amax(X, axis=0)       # maximum of X array

y = y/100                    # max test score is 100


class Neural_Network(object):

   def __init__(self):

                     # Parameters

      self.inputSize = 2

      self.outputSize = 1

      self.hiddenSize = 3

                     # Weights

      self.W1 = np.random.randn(self.inputSize, self.hiddenSize)       # (3x2) weight matrix from input
to hidden layer

      self.W2 = np.random.randn(self.hiddenSize, self.outputSize)      # (3x1) weight matrix from
hidden to output layer


   def forward(self, X):

                     #forward propagation through our network

      self.z = np.dot(X, self.W1)            # dot product of X (input) and first set of 3x2 weights

      self.z2 = self.sigmoid(self.z)          # activation function

      self.z3 = np.dot(self.z2, self.W2)       # dot product of hidden layer (z2) and second set of 3x1
weights

      o = self.sigmoid(self.z3)              # final activation function

      return o
```

```
    def sigmoid(self, s):
        return 1/(1+np.exp(-s))     # activation function

    def sigmoidPrime(self, s):
        return s * (1 - s)          # derivative of sigmoid

    def backward(self, X, y, o):
                                  # backward propgate through the network
        self.o_error = y - o        # error in output
        self.o_delta = self.o_error*self.sigmoidPrime(o) # applying derivative of sigmoid to
        self.z2_error = self.o_delta.dot(self.W2.T)    # z2 error: how much our hidden layer weights
contributed to output error
        self.z2_delta = self.z2_error*self.sigmoidPrime(self.z2) # applying derivative of sigmoid to z2
error
        self.W1 += X.T.dot(self.z2_delta)       # adjusting first set (input --> hidden) weights
        self.W2 += self.z2.T.dot(self.o_delta)  # adjusting second set (hidden --> output) weights

    def train (self, X, y):
        o = self.forward(X)
        self.backward(X, y, o)

NN = Neural_Network()
print ("\nInput: \n" + str(X))
print ("\nActual Output: \n" + str(y))
print ("\nPredicted Output: \n" + str(NN.forward(X)))
print ("\nLoss: \n" + str(np.mean(np.square(y - NN.forward(X)))))     # mean sum squared loss)
NN.train(X, y)
```

Input:

```
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
```

Actual Output:

```
[[0.92]
 [0.86]
 [0.89]]
```

Predicted Output:

```
[[0.37569264]
 [0.37059885]
 [0.36376607]]
```

Loss:

0.2709020442986832

6.**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 necessary libarities
import pandas as pd
from sklearn import tree
from sklearn.preprocessing import LabelEncoder
from sklearn.naive_bayes import GaussianNB

# load data from CSV
data = pd.read_csv('tennisdata.csv')
print("THe first 5 values of data is :\n",data.head())

# obtain Train data and Train output
X = data.iloc[:,:-1]
print("\nThe First 5 values of train data is\n",X.head())

y = data.iloc[:,-1]
print("\nThe first 5 values of Train output is\n",y.head())

# Convert then in numbers
le_outlook = LabelEncoder()
X.Outlook = le_outlook.fit_transform(X.Outlook)

le_Temperature = LabelEncoder()
X.Temperature = le_Temperature.fit_transform(X.Temperature)

le_Humidity = LabelEncoder()
X.Humidity = le_Humidity.fit_transform(X.Humidity)

le_Windy = LabelEncoder()
X.Windy = le_Windy.fit_transform(X.Windy)

print("\nNow the Train data is :\n",X.head())

le_PlayTennis = LabelEncoder()
y = le_PlayTennis.fit_transform(y)
print("\nNow the Train output is\n",y)

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)

classifier = GaussianNB()
classifier.fit(X_train,y_train)

from sklearn.metrics import accuracy_score
print("Accuracy is:",accuracy_score(classifier.predict(X_test),y_test))
```

OUTPUT

```
THe first 5 values of data is :
    Outlook Temperature Humidity  Windy PlayTennis
0    Sunny        Hot     High  False         No
1    Sunny        Hot     High   True         No
2  Overcast       Hot     High  False        Yes
3    Rainy       Mild     High  False        Yes
4    Rainy       Cool   Normal  False        Yes

The First 5 values of train data is
    Outlook Temperature Humidity  Windy
0    Sunny        Hot     High  False
1    Sunny        Hot     High   True
2  Overcast       Hot     High  False
3    Rainy       Mild     High  False
4    Rainy       Cool   Normal  False

The first 5 values of Train output is
 0     No
1     No
2    Yes
3    Yes
4    Yes
Name: PlayTennis, dtype: object

Now the Train data is :
   Outlook  Temperature  Humidity  Windy
0        2            1         0      0
1        2            1         0      1
2        0            1         0      0
3        1            2         0      0
4        1            0         1      0

Now the Train output is
 [0 0 1 1 1 0 1 0 1 1 1 1 1 0]
Accuracy is: 0.3333333333333333
```

7.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.

```
from sklearn.cluster import KMeans

from sklearn import preprocessing

from sklearn.mixture import GaussianMixture

from sklearn.datasets import load_iris

import sklearn.metrics as sm

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt


dataset=load_iris()

# print(dataset)

X=pd.DataFrame(dataset.data)

X.columns=['Sepal_Length','Sepal_Width','Petal_Length','Petal_Width']

y=pd.DataFrame(dataset.target)

y.columns=['Targets']

# print(X)

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

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

# REAL PLOT

plt.subplot(1,3,1)

plt.scatter(X.Petal_Length,X.Petal_Width,c=colormap[y.Targets],s=40)

plt.title('Real')
```

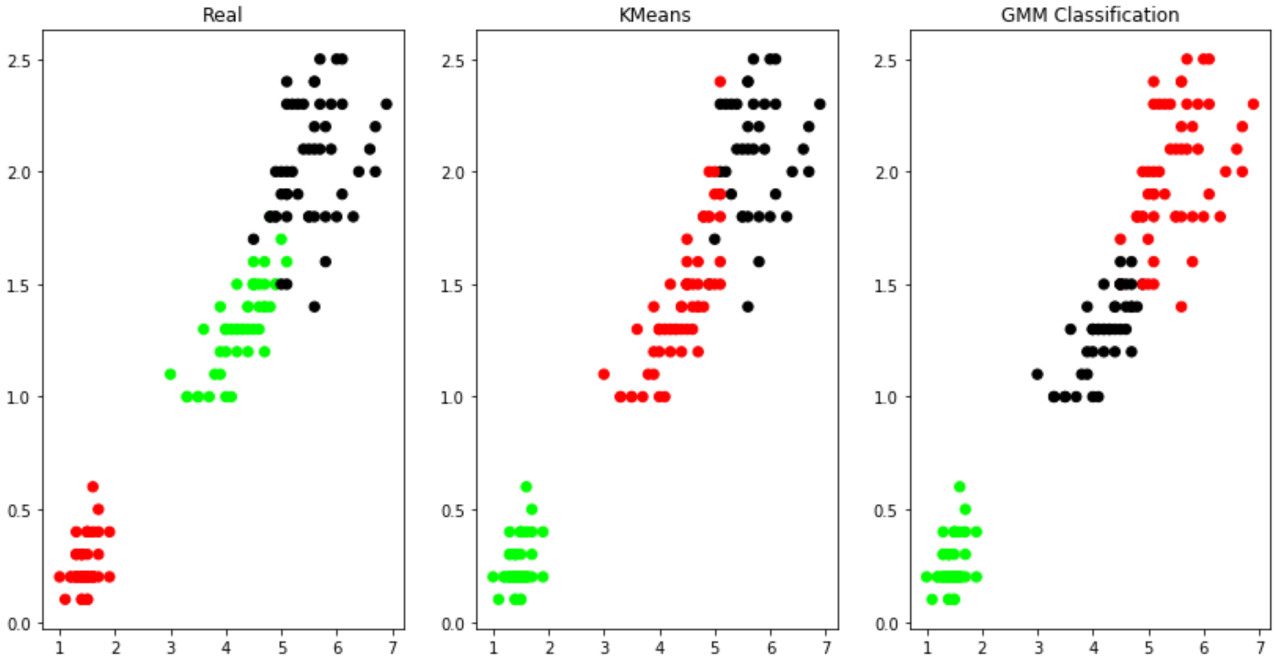
 # K-PLOT

```
plt.subplot(1,3,2)

model=KMeans(n_clusters=3)

model.fit(X)

predY=np.choose(model.labels_,[0,1,2]).astype(np.int64)

plt.scatter(X.Petal_Length,X.Petal_Width,c=colormap[predY],s=40)

plt.title('KMeans')
```

 # GMM PLOT

```
scaler=preprocessing.StandardScaler()

scaler.fit(X)

xsa=scaler.transform(X)

xs=pd.DataFrame(xsa,columns=X.columns)

gmm=GaussianMixture(n_components=3)

gmm.fit(xs)

y_cluster_gmm=gmm.predict(xs)

plt.subplot(1,3,3)

plt.scatter(X.Petal_Length,X.Petal_Width,c=colormap[y_cluster_gmm],s=40)

plt.title('GMM Classification')
```

OUTPUT



Real
KMeans
GMM Classification

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

```
from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
import numpy as np

dataset=load_iris()
#print(dataset)
X_train,X_test,y_train,y_test=train_test_split(dataset["data"],dataset["target"],random_state=0)

kn=KNeighborsClassifier(n_neighbors=1)
kn.fit(X_train,y_train)

for i in range(len(X_test)):
    x=X_test[i]
    x_new=np.array([x])
    prediction=kn.predict(x_new)

print("TARGET=",y_test[i],dataset["target_names"][y_test[i]],"PREDICTED=",prediction,dataset["
target_names"][prediction])
print(kn.score(X_test,y_test))
```

OUTPUT

```
TARGET= 2 virginica PREDICTED= [2] ['virginica']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 0 setosa PREDICTED= [0] ['setosa']
TARGET= 2 virginica PREDICTED= [2] ['virginica']
TARGET= 0 setosa PREDICTED= [0] ['setosa']
TARGET= 2 virginica PREDICTED= [2] ['virginica']
TARGET= 0 setosa PREDICTED= [0] ['setosa']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 2 virginica PREDICTED= [2] ['virginica']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 0 setosa PREDICTED= [0] ['setosa']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 0 setosa PREDICTED= [0] ['setosa']
TARGET= 0 setosa PREDICTED= [0] ['setosa']
TARGET= 2 virginica PREDICTED= [2] ['virginica']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 0 setosa PREDICTED= [0] ['setosa']
```

TARGET= 0 setosa PREDICTED= [0] ['setosa']
TARGET= 2 virginica PREDICTED= [2] ['virginica']
TARGET= 0 setosa PREDICTED= [0] ['setosa']
TARGET= 0 setosa PREDICTED= [0] ['setosa']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 0 setosa PREDICTED= [0] ['setosa']
TARGET= 2 virginica PREDICTED= [2] ['virginica']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 0 setosa PREDICTED= [0] ['setosa']
TARGET= 2 virginica PREDICTED= [2] ['virginica']
TARGET= 2 virginica PREDICTED= [2] ['virginica']
TARGET= 1 versicolor PREDICTED= [1] ['versicolor']
TARGET= 0 setosa PREDICTED= [0] ['setosa']
TARGET= 1 versicolor PREDICTED= [2] ['virginica']

0.9736842105263158

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

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


# Bokeh version is in alternatives folder


def radial_kernel(x0, X, tau):
    return np.exp(np.sum((X - x0) ** 2, axis=1) / (-2 * tau * tau))  # Weight or Radial Kernel Bias
Function


def local_regression(x0, X, Y, tau):
    # add bias term
    x0 = np.r_[1, x0]  # Add one to avoid the loss in information
    X = np.c_[np.ones(len(X)), X]

    # fit model: normal equations with kernel
    xw = X.T * radial_kernel(x0, X, tau)  # XTranspose * W

    beta = np.linalg.pinv(xw @ X) @ xw @ Y  # @ Matrix Multiplication or Dot Product

    # predict value
    return x0 @ beta  # @ Matrix Multiplication or Dot Product for prediction


n = 1000
# Generate dataset
X = np.linspace(-3, 3, num=n)
print("The Data Set ( 10 Samples) X:\n", X[1:10])
Y = np.log(np.abs(X ** 2 - 1) + .5)
print("The Fitting Curve Data Set (10 Samples) Y:\n", Y[1:10])
# Jitter X
X += np.random.normal(scale=.1, size=n)
print("Jitter (10 Samples) X :\n", X[1:10])

domain = np.linspace(-3, 3, num=300)
print(" Xo Domain Space(10 Samples):\n", domain[1:10])


def plot_lwr(tau):
    # Prediction through regression
    predictions = [local_regression(x0, X, Y, tau) for x0 in domain]
    plt.scatter(X, Y, color='blue', alpha=0.3, s=20)
    plt.plot(domain, predictions, color='red', linewidth=3)
    plt.show()
```

```
# Plotting the curves with different tau
plot_lwr(10.)
plot_lwr(1.)
plot_lwr(0.1)
plot_lwr(0.01)
```

OUTPUT
The Data Set (10 Samples) X:
[-2.99399399 -2.98798799 -2.98198198 -2.97597598 -2.96996997 -2.96396396 -2.95795796 -2.95195195 -2.94594595]
The Fitting Curve Data Set (10 Samples) Y:
[2.13582188 2.13156806 2.12730467 2.12303166 2.11874898 2.11445659 2.11015444 2.10584249 2.10152068]
Jitter (10 Samples) X :
[-3.00550309 -2.9692418 -3.10678549 -3.00803474 -3.04121224 -2.80934575 -2.97409936 -2.99156208 -2.93666494]
Xo Domain Space(10 Samples):
[-2.97993311 -2.95986622 -2.93979933 -2.91973244 -2.89966555 -2.87959866 -2.85953177 -2.83946488 -2.81939799]