1.Implement A* Search algorithm.

```
from collections import deque

class Graph:
    def __init__(self, adjac_lis):
        self.adjac_lis = adjac_lis

    def get_neighbors(self, v):
        return self.adjac_lis[v]

    # This is heuristic function which is having equal values for all nodes
    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }

        return H[n]

    def a_star_algorithm(self, start, stop):
        # In this open_lst is a lisy of nodes which have been visited, but
        who's
        # neighbours haven't all been always inspected, It starts off with
        the start
  #node
        # And closed_lst is a list of nodes which have been visited
        # and who's neighbors have been always inspected
        open_lst = set([start])
        closed_lst = set([])

        # poo has present distances from start to all other nodes
        # the default value is +infinity
        poo = {}
        poo[start] = 0

        # par contains an adjac mapping of all nodes
        par = {}
        par[start] = start

        while len(open_lst) > 0:
            n = None

            # it will find a node with the lowest value of f() -
            for v in open_lst:
                if n == None or poo[v] + self.h(v) < poo[n] + self.h(n):
                    #print(poo[v] + self.h(v))
                    #print(poo[n] + self.h(n))
                    n = v;
                    #print(v,n)
            if n == None:
                print('Path does not exist!')
                return None

            # if the current node is the stop
            # then we start again from start
            if n == stop:
                reconst_path = []
```

```
                while par[n] != n:
                    reconst_path.append(n)
                    n = par[n]

                reconst_path.append(start)

                reconst_path.reverse()

                print('Path found: {}'.format(reconst_path))
                return reconst_path

            # for all the neighbors of the current node do
            for (m, weight) in self.get_neighbors(n):
                #print(m)
              # if the current node is not presentin both open_lst and
        closed_lst
                # add it to open_lst and note n as it's par
                if m not in open_lst and m not in closed_lst:
                    open_lst.add(m)
                    par[m] = n

                    poo[m] = poo[n] + weight

                # otherwise, check if it's quicker to first visit n, then m
                # and if it is, update par data and poo data
                # and if the node was in the closed_lst, move it to
        open_lst

                else:
                    if poo[m] > poo[n] + weight:
                        poo[m] = poo[n] + weight
                        par[m] = n

                        if m in closed_lst:
                            closed_lst.remove(m)
                            open_lst.add(m)

            # remove n from the open_lst, and add it to closed_lst
            # because all of his neighbors were inspected
            open_lst.remove(n)
            closed_lst.add(n)

        print('Path does not 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* Search algorithm.

```
def recAOStar(n):
    global finalPath
    print("Expanding Node : ", n)
    and_nodes = []
    or_nodes = []
#Segregation of AND and OR nodes
    if (n in allNodes):
        if 'AND' in allNodes[n]:
            and_nodes = allNodes[n]['AND']
        if 'OR' in allNodes[n]:
            or_nodes = allNodes[n]['OR']
# If leaf node then return
    if len(and_nodes) == 0 and len(or_nodes) == 0:
        return
    solvable = False
    marked = {}
    while not solvable:
# If all the child nodes are visited and expanded, take the least cost of all the child nodes
        if len(marked) == len(and_nodes) + len(or_nodes):
            min_cost_least, min_cost_group_least = least_cost_group(and_nodes, or_nodes, {})
            solvable = True
            change_heuristic(n, min_cost_least)
            optimal_child_group[n] = min_cost_group_least
            continue
# Least cost of the unmarked child nodes
    min_cost, min_cost_group = least_cost_group(and_nodes, or_nodes, marked)
    is_expanded = False
    # If the child nodes have sub trees then recursively visit them to recalculate the heuristic of
the child node
    if len(min_cost_group) > 1:
        if (min_cost_group[0] in allNodes):
            is_expanded = True
            recAOStar(min_cost_group[0])
```

```
        if (min_cost_group[1] in allNodes):
            is_expanded = True
            recAOStar(min_cost_group[1])
        else:
            if (min_cost_group in allNodes):
                is_expanded = True
                recAOStar(min_cost_group)
                # If the child node had any subtree and expanded, verify if the new heuristic value
is still the least among all nodes
        if is_expanded:
            min_cost_verify, min_cost_group_verify = least_cost_group(and_nodes, or_nodes,
{})
    if min_cost_group == min_cost_group_verify:
        solvable = True
        change_heuristic(n, min_cost_verify)
        optimal_child_group[n] = min_cost_group
        # If the child node does not have any subtrees then no change in heuristic, so update the
min cost of the current node
    else:
        solvable = True
        change_heuristic(n, min_cost)
        optimal_child_group[n] = min_cost_group
        #Mark the child node which was expanded
    marked[min_cost_group] = 1
    return heuristic(n)
# Function to calculate the min cost among all the child nodes
def least_cost_group(and_nodes, or_nodes, marked):
    node_wise_cost = {}
    for node_pair in and_nodes:
        if not node_pair[0] + node_pair[1] in marked:
            cost = 0
            cost = cost + heuristic(node_pair[0]) + heuristic(node_pair[1]) + 2
            node_wise_cost[node_pair[0] + node_pair[1]] = cost
    for node in or_nodes:
```

```
        if not node in marked:
            cost = 0
            cost = cost + heuristic(node) + 1
            node_wise_cost[node] = cost
    min_cost = 999999
    min_cost_group = None
    # Calculates the min heuristic
    for costKey in node_wise_cost:
        if node_wise_cost[costKey] < min_cost:
            min_cost = node_wise_cost[costKey]
            min_cost_group = costKey
    return [min_cost, min_cost_group]
    # Returns heuristic of a node
def heuristic(n):
    return H_dist[n]
    # Updates the heuristic of a node
def change_heuristic(n, cost):
    H_dist[n] = cost
    return
    # Function to print the optimal cost nodes
def print_path(node):
    print(optimal_child_group[node], end="")
    node = optimal_child_group[node]
    if len(node) > 1:
        if node[0] in optimal_child_group:
            print("->", end="")
            print_path(node[0])
        if node[1] in optimal_child_group:
            print("->", end="")
            print_path(node[1])
    else:
        if node in optimal_child_group:
            print("->", end="")
            print_path(node)
```

```
H_dist = { 'A': -1,'B': 4, 'C': 2, 'D': 3, 'E': 6,'F': 8, 'G': 2,'H': 0, 'I': 0, 'J': 0}
allNodes = { 'A': {'AND': [('C', 'D')], 'OR': ['B']},
        'B': {'OR': ['E', 'F']}, 'C': {'OR': ['G'], 'AND': [('H', 'I')]}, 'D': {'OR': ['J']} }
optimal_child_group = {}
optimal_cost = recAOStar('A')
print('Nodes which gives optimal cost are')
print_path('A')
print('\nOptimal Cost is :: ', optimal_cost)
```

Output:

```
Expanding Node : A
Expanding Node : B
Expanding Node : C
Expanding Node : D
Nodes which gives optimal cost are
CD->HI->J
Optimal Cost is :: 5
```

3.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.

```
import numpy as np
import pandas as pd
data = pd.read_csv('data4.csv')
concepts = np.array(data.iloc[:,0:-1])
target = np.array(data.iloc[:,-1])
def learn(concepts,target):
    count = 0
    first = ['?','?','?','?','?','?']
    for i, val in enumerate(target):
        if val == 'Yes':
            #print(specific_h)
            break
    specific_h = concepts[i].copy()
    generic_h = [["?" for i in range(len(specific_h))] for i in range (len(specific_h))]
    for i,h in enumerate(concepts):
        if target[i] == "Yes":
            for x in range(len(specific_h)):
                if h[x] != specific_h[x]:
                    specific_h[x] = "?"
                    generic_h[x][x] = "?"
        if target[i] == "No":
            for x in range(len(specific_h)):
                if h[x] != specific_h[x]:
                    generic_h[x][x] = specific_h[x]
                else:
                    generic_h[x][x] == "?"
        if generic_h[x][x] == "?":
            print(f'S{count} : {specific_h}')
            print(f'G{count} : {first}')
            count+=1
        else:
            print(f'S{count} : {specific_h}')
            print(f'G{count} : {generic_h}')
            for x in range(len(generic_h)):
                first[x] = generic_h[x]
            count += 1
    indices=[i for i, val in enumerate(generic_h)if val == ['?','?','?','?','?','?']]
    for i in indices:
        generic_h.remove(['?','?','?','?','?','?'])
    return specific_h, generic_h

s_final,g_final = learn(concepts,target)
print("final s:",s_final,sep="\n")
print("final g:",g_final,sep="\n")
```

dataset:data4.csv

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

Output

S0 : ['sunny' 'warm' 'high' 'strong' 'warm' 'same']
G0 : ['?', '?', '?', '?', '?', '?']
S1 : ['sunny' 'warm' 'high' 'strong' 'warm' 'same']
G1 : [['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', 'same']]
S2 : ['sunny' 'warm' 'high' 'strong' '?' '?']
G2 : [['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]
final s:
['sunny' 'warm' 'high' 'strong' '?' '?']
final g:
[['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?']]

4.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 toclassify a new sample.

```
import pandas as pd
import math
import numpy as np

data = pd.read_csv("play.csv")
features = [feat for feat in data]
features.remove("classification")

class Node:
    def __init__(self):
        self.children = []
        self.value = ""
        self.isLeaf = False
        self.pred = ""

def entropy(examples):
    pos = 0.0
    neg = 0.0
    for _, row in examples.iterrows():
        if row["classification"] == "Yes":
            pos += 1
        else:
            neg += 1
    if pos == 0.0 or neg == 0.0:
        return 0.0
    else:
        p = pos / (pos + neg)
        n = neg / (pos + neg)
        return -(p * math.log(p, 2) + n * math.log(n, 2))

def info_gain(examples, attr):
    uniq = np.unique(examples[attr])
    #print ("\n",uniq)
    gain = entropy(examples)
    #print ("\n",gain)
    for u in uniq:
        subdata = examples[examples[attr] == u]
        #print ("\n",subdata)
        sub_e = entropy(subdata)
        gain -= (float(len(subdata)) / float(len(examples))) * sub_e
        #print ("\n",gain)
    return gain

def ID3(examples, attrs):
    root = Node()

    max_gain = 0
```

```
    max_feat = ""
    for feature in attrs:
        #print ("\n",examples)
        gain = info_gain(examples, feature)
        if gain > max_gain:
            max_gain = gain
            max_feat = feature
    root.value = max_feat
    #print ("\nMax feature attr",max_feat)
    uniq = np.unique(examples[max_feat])
    #print ("\n",uniq)
    for u in uniq:
        #print ("\n",u)
        subdata = examples[examples[max_feat] == u]
        #print ("\n",subdata)
        if entropy(subdata) == 0.0:
            newNode = Node()
            newNode.isLeaf = True
            newNode.value = u
            newNode.pred = np.unique(subdata["classification"])
            root.children.append(newNode)
        else:
            dummyNode = Node()
            dummyNode.value = u
            new_attrs = attrs.copy()
            new_attrs.remove(max_feat)
            child = ID3(subdata, new_attrs)
            dummyNode.children.append(child)
            root.children.append(dummyNode)
    return root

def printTree(root: Node, depth=0):
    for i in range(depth):
        print("\t", end="")
    print(root.value, end="")
    if root.isLeaf:
        print(" -> ", root.pred)
    print()
    for child in root.children:
        printTree(child, depth + 1)

root = ID3(data, features)
printTree(root)
```

dataset :
play.csv

A1,A2,A3,classification
True,Hot,High,No
True,Hot,High,No
False,Hot,High,Yes
False,Cool,Normal,Yes
False,Cool,Normal,Yes
True,Cool,High,No
True,Hot,High,No
True,Hot,Normal,Yes
False,Cool,Normal,Yes
False,Cool,High,No

output

```
A3
        High
                A1
                        False
                                A2
                                        Cool ->  ['No']

                                        Hot ->  ['Yes']

                        True ->  ['No']

        Normal ->  ['Yes']
```

5. 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]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float)
X = X/np.amax(X,axis=0) # maximum of X array longitudinally y = y/100
#Sigmoid Function
def sigmoid (x):
   return (1/(1 + np.exp(-x)))
#Derivative of Sigmoid Function
def derivatives_sigmoid(x):
   return x * (1 - x)
                                        #Variable initialization

epoch=7000                          #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 at output layer
                                        #weight and bias initialization

wh=np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))
bh=np.random.uniform(size=(1,hiddenlayer_neurons))
wout=np.random.uniform(size=(hiddenlayer_neurons,output_neurons))

bout=np.random.uniform(size=(1,output_neurons))
        # draws a random range of numbers uniformly of dim x*y
#Forward Propagation
for i in range(epoch):
   hinp1=np.dot(X,wh)
   hinp=hinp1 + bh
   hlayer_act = sigmoid(hinp)
   outinp1=np.dot(hlayer_act,wout)
   outinp= outinp1+ bout
   output = sigmoid(outinp)
#Backpropagation
   EO = y-output
   outgrad = derivatives_sigmoid(output)
   d_output = EO* outgrad
   EH = d_output.dot(wout.T)
   hiddengrad = derivatives_sigmoid(hlayer_act)
```

```
#how much hidden layer wts contributed to error
    d_hiddenlayer = EH * hiddengrad
    wout += hlayer_act.T.dot(d_output) *lr
# dotproduct of nextlayererror and currentlayerop

    bout += np.sum(d_output, axis=0,keepdims=True) *lr
    wh += X.T.dot(d_hiddenlayer) *lr
#bh += np.sum(d_hiddenlayer, axis=0,keepdims=True) *lr
print("Input: \n" + str(X))
print("Actual Output: \n" + str(y))
print("Predicted Output: \n" ,output)
```

output

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

Actual Output:
[[ 0.92]
 [ 0.86]
 [ 0.89]]
Predicted Output:
 [[ 0.89559591]
 [ 0.88142069]
 [ 0.8928407 ]]
```

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 csv
import random
import math

def loadCsv(filename):
  lines = csv.reader(open(filename, "r"));
  dataset = list(lines)
  for i in range(len(dataset)):
                                        #converting strings into numbers for processing
        dataset[i] = [float(x) for x in dataset[i]]

  return dataset

def splitDataset(dataset, splitRatio):                    #67% training size
  trainSize = int(len(dataset) * splitRatio);
  trainSet = [ ]
  copy = list(dataset);
  while len(trainSet) < trainSize:

        #generate indices for the dataset list randomly to pick ele for
                                                                training
        data index = random.randrange(len(copy));
        trainSet.append(copy.pop(index))
  return [trainSet, copy]

def separateByClass(dataset):
  separated = {}

#creates a dictionary of classes 1 and 0 where the values are the instances belonging
to # each class
  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 = {}

                                #summaries is a dic of tuples(mean,std) for each class value
  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 = {}

                                      #class and attribute information as mean and sd
   for classValue, classSummaries in summaries.items():
        probabilities[classValue] = 1
        for i in range(len(classSummaries)):
              mean, stdev = classSummaries[i]
                                            #take mean and sd of every attribute
        for class 0 and 1 seperaely
              x = inputVector[i]                          #testvector's first
              attribute
              probabilities[classValue]  *=  calculateProbability(x,  mean,  stdev);
                                                              #use normal dist
     return probabilities

def predict(summaries, inputVector):
  probabilities = calculateClassProbabilities(summaries, inputVector)
  bestLabel, bestProb = None, -1


  for classValue, probability in probabilities.items():
                                        #assigns that class which has he highest prob
        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


def main():
  filename = 'diabetesdata.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 of the classifier is : {0}%'.format(accuracy))

main()
```

Output

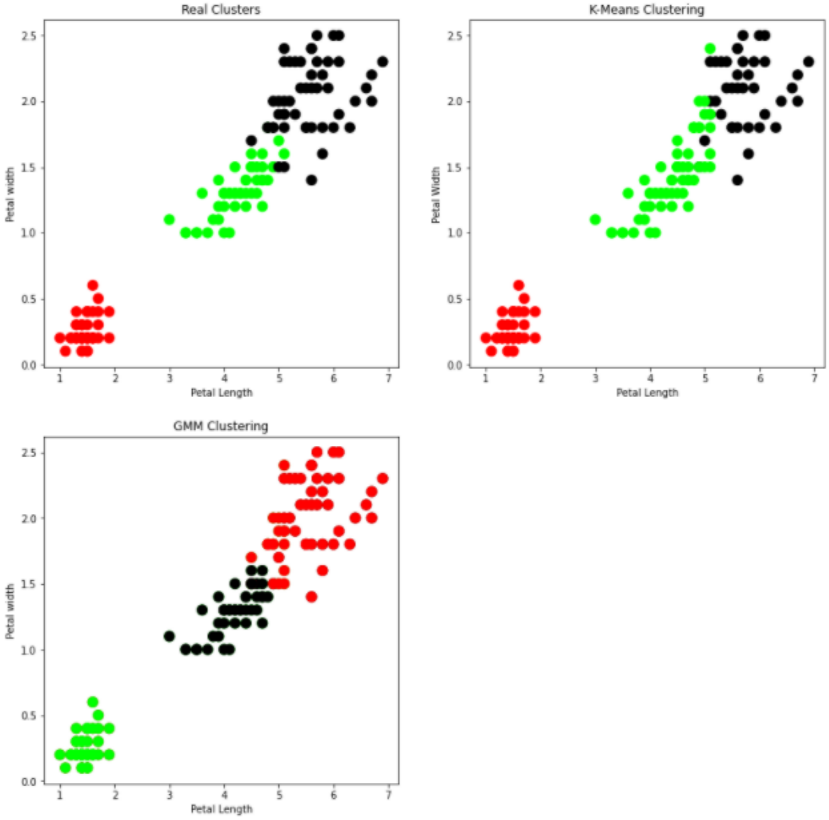
```
confusion matrix is as follows
[[17 0  0]
 [ 0 17  0]
 [ 0  0 11]]
Accuracy metrics
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        17
           1       1.00      1.00      1.00        17
           2       1.00      1.00      1.00        11

avg / total       1.00      1.00      1.00        45
```

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.

```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import KMeans
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=(14,14))
colormap=np.array(['red','lime','black'])
plt.subplot(2,2,1)
plt.scatter(X.Petal_Length,X.Petal_Width,c=colormap[y.Targets],s=40)
plt.title('Real Clusters')
plt.xlabel('Petal Length')
plt.ylabel('Petal width')
plt.subplot(2,2,2)
plt.scatter(X.Petal_Length,X.Petal_Width,c=colormap[model.labels_],s=40)
plt.title('K-Means Clustering')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
plt.subplot(2,2,2)
plt.scatter(X.Petal_Length,X.Petal_Width,c=colormap[model.labels_],s=40)
plt.title('K-Means Clustering')
plt.ylabel('Petal Width')
from sklearn import preprocessing
scaler=preprocessing.StandardScaler()
scaler.fit(X)
xsa=scaler.transform(X)
xs=pd.DataFrame(xsa,columns=X.columns)
from sklearn.mixture import GaussianMixture
gmm=GaussianMixture(n_components=3)
gmm.fit(xs)
gmm_y=gmm.predict(xs)
plt.subplot(2,2,3)
plt.scatter(X.Petal_Length,X.Petal_Width,c=colormap[gmm_y],s=40)
plt.title('GMM Clustering')
plt.xlabel('Petal Length')
plt.ylabel('Petal width')
print('Observation:The GMM using EM algo based clustering matched the true labels more
closely than KMeans.')
```

output

```
Observation:The GMM using EM algo based clustering matched the true labels more closely than KMeans.
```



8. 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.

```
from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier
import numpy as np
from sklearn.model_selection import train_test_split
iris_dataset = load_iris()
#print(iris_dataset)
targets = iris_dataset.target_names
print("Class : number")
for i in range(len(targets)):
    print(targets[i], ':', i)
X_train, X_test, y_train, y_test = train_test_split(iris_dataset["data"], iris_dataset["target"])
kn = KNeighborsClassifier(1)
kn.fit(X_train, y_train)
for i in range(len(X_test)):
    x_new = np.array([X_test[i]])
    prediction = kn.predict(x_new)
    print("Actual:[{0}] [{1}],Predicted:{2} {3}".format(y_test[i], targets[y_test[i]], prediction,
targets[prediction]))
print("\nAccuracy:",kn.score(X_test,y_test))
```

Output:

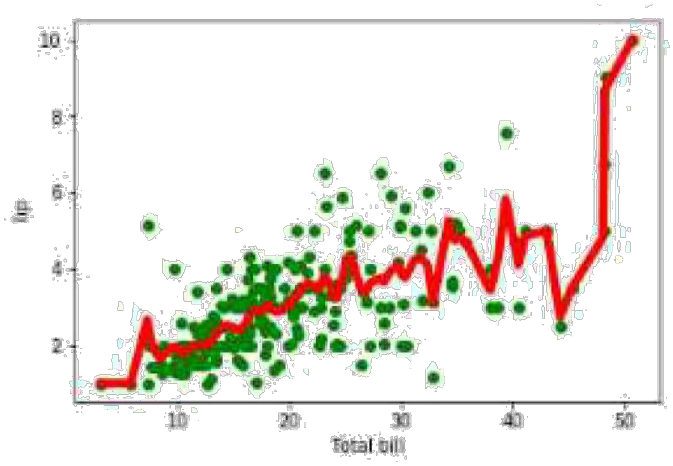
```
Class : number
setosa : 0
versicolor : 1
virginica : 2
Actual:[0] [setosa],Predicted:[0] ['setosa']
Actual:[0] [setosa],Predicted:[0] ['setosa']
Actual:[0] [setosa],Predicted:[0] ['setosa']
Actual:[0] [setosa],Predicted:[0] ['setosa']
Actual:[1] [versicolor],Predicted:[1] ['versicolor']
Actual:[1] [versicolor],Predicted:[1] ['versicolor']
Actual:[2] [virginica],Predicted:[2] ['virginica']
Actual:[0] [setosa],Predicted:[0] ['setosa']
Actual:[2] [virginica],Predicted:[2] ['virginica']
Actual:[1] [versicolor],Predicted:[1] ['versicolor']
Actual:[1] [versicolor],Predicted:[1] ['versicolor']
Actual:[1] [versicolor],Predicted:[1] ['versicolor']
Actual:[2] [virginica],Predicted:[2] ['virginica']
Actual:[1] [versicolor],Predicted:[1] ['versicolor']
Actual:[2] [virginica],Predicted:[2] ['virginica']
Actual:[2] [virginica],Predicted:[2] ['virginica']
Actual:[0] [setosa],Predicted:[0] ['setosa']
Actual:[0] [setosa],Predicted:[0] ['setosa']
Actual:[2] [virginica],Predicted:[2] ['virginica']
Actual:[1] [versicolor],Predicted:[1] ['versicolor']
Actual:[1] [versicolor],Predicted:[1] ['versicolor']
Actual:[1] [versicolor],Predicted:[1] ['versicolor']
Actual:[2] [virginica],Predicted:[2] ['virginica']
Actual:[0] [setosa],Predicted:[0] ['setosa']
Actual:[1] [versicolor],Predicted:[1] ['versicolor']
Actual:[0] [setosa],Predicted:[0] ['setosa']
Actual:[2] [virginica],Predicted:[2] ['virginica']
Actual:[2] [virginica],Predicted:[2] ['virginica']
Actual:[0] [setosa],Predicted:[0] ['setosa']
Actual:[0] [setosa],Predicted:[0] ['setosa']
Actual:[0] [setosa],Predicted:[0] ['setosa']
Actual:[0] [setosa],Predicted:[0] ['setosa']
Actual:[0] [setosa],Predicted:[0] ['setosa']
Actual:[2] [virginica],Predicted:[2] ['virginica']
Actual:[2] [virginica],Predicted:[2] ['virginica']
Actual:[2] [virginica],Predicted:[2] ['virginica']
Actual:[2] [virginica],Predicted:[2] ['virginica']
Actual:[1] [versicolor],Predicted:[1] ['versicolor']

Accuracy:  1.0
```

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

```
from numpy import *
import operator
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('data10.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)

m= np1.shape(mbill)[1]
```

```
one = np1.mat(np1.ones(m))
X= np1.hstack((one.T,mbill.T))                    #set k here
ypred = localWeightRegression(X,mtip,2)
SortIndex = X[:,1].argsort(0)
xsort = X[SortIndex][:,0]
```

Output



Questions

1. What is machine learning?
2. Define supervised learning
3. Define unsupervised learning
4. Define semi supervised learning
5. Define reinforcement learning
6. What do you mean by hypotheses?
7. What is classification?
8. What is clustering?
9. Define precision, accuracy and recall
10. Define entropy
11. Define regression
12. How Knn is different from k-means clustering?
13. What is concept learning
14. Define specific boundary and general boundary
15. Define target function
16. Define decision tree
17. What is ANN
18. Explain gradient descent approximation
19. State Bayes theorem
20. Define Bayesian belief network
21. Differentiate hard and soft clustering
22. Define variance
23. What is inductive machine learning?
24. Why K nearest neighbour algorithm is lazy learning algorithm?
25. Why naïve Bayes is naïve?
26. Mention classification algorithms
27. Define pruning
28. Differentiate Clustering and classification
29. Mention clustering algorithms
30. Define Bias