12/9/2022

MY AIML Lab Manual

Author: Nitish K 092

1. Implement A* Search algorithm.

Soln)

```
def aStarAlgo(start_node, stop_node):


        open_set = set(start_node)

        closed_set = set()

        g = {} #store distance from starting node

        parents = {}# parents contains an adjacency map of all nodes


        #ditance of starting node from itself is zero

        g[start_node] = 0

        #start_node is root node i.e it has no parent nodes

        #so start_node is set to its own parent node

        parents[start_node] = start_node




        while len(open_set) > 0:

            n = None


            #node with lowest f() is found

            for v in open_set:

                if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):

                    n = v




            if n == stop_node or Graph_nodes[n] == None:

                pass

            else:
```

```
    for (m, weight) in get_neighbors(n):
      #nodes 'm' not in first and last set are added to first
      #n is set its parent
      if m not in open_set and m not in closed_set:
          open_set.add(m)
          parents[m] = n
          g[m] = g[n] + weight



      #for each node m,compare its distance from start i.e g(m) to the
      #from start through n node
      else:
          if g[m] > g[n] + weight:
              #update g(m)
              g[m] = g[n] + weight
              #change parent of m to n
              parents[m] = n

              #if m in closed set,remove and add to open
              if m in closed_set:
                  closed_set.remove(m)
                  open_set.add(m)

if n == None:
    print('Path does not exist!')
    return None

# if the current node is the stop_node
# then we begin reconstructin the path from it to the start_node
```

```
        if n == stop_node:
            path = []

            while parents[n] != n:
                path.append(n)
                n = parents[n]

            path.append(start_node)

            path.reverse()

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


        # remove n from the open_list, and add it to closed_list
        # because all of his neighbors were inspected
        open_set.remove(n)
        closed_set.add(n)

    print('Path does not exist!')
    return None

#define fuction to return neighbor and its distance
#from the passed node
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
```

```
        return None
    #for simplicity we ll consider heuristic distances given
    #and this function returns heuristic distance for all nodes
    def heuristic(n):
        H_dist = {
            'A': 10,
            'B': 8,
            'C': 5,
            'D': 7,
            'E': 3,
            'F': 6,
            'G': 5,
            'H': 3,
            'I': 1,
            'J': 0
        }

        return H_dist[n]

    #Describe your graph here
    Graph_nodes = {
        'A': [('B', 6), ('F', 3)],
        'B': [('C', 3), ('D', 2)],
        'C': [('D', 1), ('E', 5)],
        'D': [('C', 1), ('E', 8)],
        'E': [('I', 5), ('J', 5)],
        'F': [('G', 1),('H', 7)] ,
        'G': [('I', 3)],
        'H': [('I', 2)],
```

```
    'I': [('E', 5), ('J', 3)],

}
aStarAlgo('A', 'J')
```

2. Implement AO* algorithm

Soln)

```
# Recursive implementation of AO* aglorithm by Dr. K PARAMESHA, Professor, VVCE, Mysuru, INDIA

class Graph:
    def __init__(self, graph, heuristicNodeList, startNode):  #instantiate graph object with graph
topology, heuristic values, start node

        self.graph = graph
        self.H=heuristicNodeList
        self.start=startNode
        self.parent={}
        self.status={}
        self.solutionGraph={}

    def applyAOStar(self):        # starts a recursive AO* algorithm
        self.aoStar(self.start, False)

    def getNeighbors(self, v):     # gets the Neighbors of a given node
        return self.graph.get(v,'')

    def getStatus(self,v):        # return the status of a given node
        return self.status.get(v,0)

    def setStatus(self,v, val):    # set the status of a given node
        self.status[v]=val

    def getHeuristicNodeValue(self, n):
        return self.H.get(n,0)     # always return the heuristic value of a given node

    def setHeuristicNodeValue(self, n, value):
        self.H[n]=value            # set the revised heuristic value of a given node


    def printSolution(self):
        print("FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE:",self.start)
        print("------------------------------------------------------------")
        print(self.solutionGraph)
        print("------------------------------------------------------------")

    def computeMinimumCostChildNodes(self, v):  # Computes the Minimum Cost of child nodes of a
given node v
        minimumCost=0
        costToChildNodeListDict={}
        costToChildNodeListDict[minimumCost]=[]
        flag=True
```

```
        for nodeInfoTupleList in self.getNeighbors(v):  # iterate over all the set of child node/s
            cost=0
            nodeList=[]
            for c, weight in nodeInfoTupleList:
                cost=cost+self.getHeuristicNodeValue(c)+weight
                nodeList.append(c)

            if flag==True:                      # initialize Minimum Cost with the cost of first set of child node/s
                minimumCost=cost
                costToChildNodeListDict[minimumCost]=nodeList      # set the Minimum Cost child node/s
                flag=False
            else:                               # checking the Minimum Cost nodes with the current Minimum Cost
                if minimumCost>cost:
                    minimumCost=cost
                    costToChildNodeListDict[minimumCost]=nodeList  # set the Minimum Cost child node/s


        return minimumCost, costToChildNodeListDict[minimumCost]   # return Minimum Cost and
Minimum Cost child node/s


    def aoStar(self, v, backTracking):     # AO* algorithm for a start node and backTracking status flag

        print("HEURISTIC VALUES  :", self.H)
        print("SOLUTION GRAPH    :", self.solutionGraph)
        print("PROCESSING NODE   :", v)
        print("-----------------------------------------------------------------------------------------")

        if self.getStatus(v) >= 0:        # if status node v >= 0, compute Minimum Cost nodes of v
            minimumCost, childNodeList = self.computeMinimumCostChildNodes(v)
            self.setHeuristicNodeValue(v, minimumCost)
            self.setStatus(v,len(childNodeList))

            solved=True                   # check the Minimum Cost nodes of v are solved
            for childNode in childNodeList:
                self.parent[childNode]=v
                if self.getStatus(childNode)!=-1:
                    solved=solved & False

            if solved==True:             # if the Minimum Cost nodes of v are solved, set the current node
status as solved(-1)
                self.setStatus(v,-1)
                self.solutionGraph[v]=childNodeList # update the solution graph with the solved nodes
which may be a part of solution


            if v!=self.start:           # check the current node is the start node for backtracking the current
node value
```

```
            self.aoStar(self.parent[v], True)   # backtracking the current node value with backtracking
status set to true

        if backTracking==False:     # check the current call is not for backtracking
            for childNode in childNodeList:   # for each Minimum Cost child node
                self.setStatus(childNode,0)   # set the status of child node to 0(needs exploration)
                self.aoStar(childNode, False) # Minimum Cost child node is further explored with
backtracking status as false



h1 = {'A': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, 'T': 3}
graph1 = {
    'A': [[('B', 1), ('C', 1)], [('D', 1)]],
    'B': [[('G', 1)], [('H', 1)]],
    'C': [[('J', 1)]],
    'D': [[('E', 1), ('F', 1)]],
    'G': [[('I', 1)]]
}
G1= Graph(graph1, h1, 'A')
G1.applyAOStar()
G1.printSolution()

h2 = {'A': 1, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}  # Heuristic values of Nodes
graph2 = {                                      # Graph of Nodes and Edges
    'A': [[('B', 1), ('C', 1)], [('D', 1)]],      # Neighbors of Node 'A', B, C & D with repective weights
    'B': [[('G', 1)], [('H', 1)]],               # Neighbors are included in a list of lists
    'D': [[('E', 1), ('F', 1)]]                  # Each sublist indicate a "OR" node or "AND" nodes
}

G2 = Graph(graph2, h2, 'A')                      # Instantiate Graph object with graph, heuristic values and
start Node
G2.applyAOStar()                                 # Run the AO* algorithm
G2.printSolution()                               # Print the solution graph as output of the AO* algorithm search
```

3. Implement candidate Elimination algorithm.

Soln)

```
import csv
pd = open('datasets/trainingexamples.csv')
with pd  as csvFile:
    data = [tuple(line) for line in csv.reader(csvFile)]
data

def Domain(): #All possible unique values an attribute/field can hold.
    D =[]
    for i in range(len(data[0])):
        D.append(list(set([ele[i] for ele in data])))
    return D

D = Domain()
D

def consistant(h1, h2):
    for x, y in zip(h1, h2):
        if not (x == "?" or (x != "ϕ" and (x == y or y == "ϕ"))):
            return False
    return True

def candidate_elimination():
    G = {('?',)*(len(data[0]) - 1),}
    S = ['ϕ']*(len(data[0]) - 1)
    no = 0
    print("\n G[{0}]:".format(no), G)
    print("\n S[{0}]:".format(no), S)
    for item in data:
        no += 1
        inp , res = item[:-1] , item[-1]

        if res in "Yy":
            i = 0               #Remove from G any inconsistancy
            G = {g for g in G if consistant(g,inp)}
            for s,x in zip(S,inp):                  # similar to find-s
                if not s==x:
                    S[i] = '?' if s != 'ϕ' else x
                i += 1
        else:
            S = S                           #unaffected for this eg.
            Gprev = G.copy()
            for g in Gprev:                         #for each hypothesis
                if g not in G:                      # if g gets removed.
                    continue
```

```
        for i in range(len(g)):                    #for every fiels/atribute
            if g[i] == "?":                        #if it can be more generalized.
                for val in D[i]:              # for each possible values in domain.
                    if inp[i] != val and val == S[i]:                    # check if this possible value in domain
is applicable.
                        g_new = g[:i] + (val,) + g[i+1:]
                        G.add(g_new)
            else:
                G.add(g)                                     # difference_update() used to remove the items
from the set which is passed to it.
        G.difference_update([h for h in G if
                        any([consistant(h, g1) for g1 in G if h != g1])])
    print("\n G[{0}]:".format(no), G)
    print("\n S[{0}]:".format(no), S)
candidate_elimination()
```

4. Implement ID3 algorithm.

Soln)

```
import math
import csv

def load_csv(filename):
    lines=csv.reader(open(filename,"r"));
    dataset = list(lines)
    headers = dataset.pop(0)
    return dataset,headers

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

def subtables(data,col,delete):
    dic={}
    coldata=[row[col] for row in data]
    attr=list(set(coldata))
    counts=[0]*len(attr)
    r=len(data)
    c=len(data[0])
    for x in range(len(attr)):
        for y in range(r):
            if data[y][col]==attr[x]:
                counts[x]+=1
    for x in range(len(attr)):
        dic[attr[x]]=[[0 for i in range(c)] for j in range(counts[x])]
        pos=0
        for y in range(r):
            if data[y][col]==attr[x]:
                if delete:
                    del data[y][col]
                dic[attr[x]][pos]=data[y]
                pos+=1
    return attr,dic

def entropy(S):
    attr=list(set(S))
    if len(attr)==1:
        return 0
    counts=[0,0]
    for i in range(2):
        counts[i]=sum([1 for x in S if attr[i]==x])/(len(S)*1.0)
```

```
    sums=0
    for cnt in counts:
        sums+=-1*cnt*math.log(cnt,2)
    return sums

def compute_gain(data,col):
    attr,dic = subtables(data,col,delete=False)
    total_size=len(data)
    entropies=[0]*len(attr)
    ratio=[0]*len(attr)
    total_entropy=entropy([row[-1] for row in data])
    for x in range(len(attr)):
        ratio[x]=len(dic[attr[x]])/(total_size*1.0)
        entropies[x]=entropy([row[-1] for row in dic[attr[x]]])

        total_entropy-=ratio[x]*entropies[x]
    return total_entropy

def build_tree(data,features):
    lastcol=[row[-1] for row in data]
    if(len(set(lastcol)))==1:
        node=Node("")
        node.answer=lastcol[0]
        return node
    n=len(data[0])-1
    gains=[0]*n
    for col in range(n):
        gains[col]=compute_gain(data,col)
    split=gains.index(max(gains))
    node=Node(features[split])
    fea = features[:split]+features[split+1:]
    attr,dic=subtables(data,split,delete=True)
    for x in range(len(attr)):
        child=build_tree(dic[attr[x]],fea)
        node.children.append((attr[x],child))
    return node

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

def classify(node,x_test,features):
    if node.answer!="":
```

```
        print(node.answer)
        return
    pos=features.index(node.attribute)
    for value, n in node.children:
        if x_test[pos]==value:
            classify(n,x_test,features)


dataset,features=load_csv("datasets/traintennis.csv")
node1=build_tree(dataset,features)
print("The decision tree for the dataset using ID3 algorithm is")
print_tree(node1,0)
testdata,features=load_csv("datasets/testtennis.csv")
for xtest in testdata:
    print("The test instance:",xtest)
    print("The label for test instance:",end=" ")
    classify(node1,xtest,features)
```

5. Implement Backpropagation

Soln)

```
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=5000            #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
for i in range(epoch):

#Forward Propogation
   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)
```

```
#how much hidden layer wts contributed to error
    hiddengrad = derivatives_sigmoid(hlayer_act)
    d_hiddenlayer = EH * hiddengrad

# dotproduct of nextlayererror and currentlayerop
wout += hlayer_act.T.dot(d_output) *lr
wh += X.T.dot(d_hiddenlayer) *lr

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

6. Naive Bayesian Classifier Algorithm.

```
print("\nNaive Bayes Classifier for concept learning problem")
import csv
import random
import math
import operator

def safe_div(x,y):
 if y == 0:
  return 0
 return x/y

# 1.Data Handling
# 1.1 Loading the Data from csv file of ConceptLearning dataset.
def loadCsv(filename):
  lines = csv.reader(open(filename))
  dataset = list(lines)
  for i in range(len(dataset)):
    dataset[i] = [float(x) for x in dataset[i]]
  return dataset


#1.2 Splitting the Data set into Training Set
def splitDataset(dataset, splitRatio):
  trainSize = int(len(dataset) * splitRatio)
  trainSet = []
  copy = list(dataset)
  i=0
  while len(trainSet) < trainSize:
  #index = random.randrange(len(copy))
    trainSet.append(copy.pop(i))
  return [trainSet, copy]


#2.Summarize Data
#The naive bayes model is comprised of a
#summary of the data in the training dataset.
#This summary is then used when making predictions.
#involves the mean and the standard deviation for each attribute, by class value

#2.1: Separate Data By Class
#Function to categorize the dataset in terms of classes
#The function assumes that the last attribute (-1) is the class value.
#The function returns a map of class values to lists of data instances.

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

#The mean is the central middle or central tendency of the data,
# and we will use it as the middle of our gaussian distribution
# when calculating probabilities

#2.2 : Calculate Mean
def mean(numbers):
  return safe_div(sum(numbers),float(len(numbers)))

#The standard deviation describes the variation of spread of the data,
#and we will use it to characterize the expected spread of each attribute
#in our Gaussian distribution when calculating probabilities.

#2.3 : Calculate Standard Deviation
def stdev(numbers):
  avg = mean(numbers)
  variance = safe_div(sum([pow(x-avg,2) for x in numbers]),float(len(numbers)-1))
  return math.sqrt(variance)

#2.4 : Summarize Dataset
#Summarize Data Set for a list of instances (for a class value)
#The zip function groups the values for each attribute across our data instances
#into their own lists so that we can compute the mean and standard deviation values
#for the attribute.

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

#2.5 : Summarize Attributes By Class
#We can pull it all together by first separating our training dataset into
#instances grouped by class.Then calculate the summaries for each attribute.

def summarizeByClass(dataset):
  separated = separateByClass(dataset)
  summaries = {}
  for classValue, instances in separated.items():
    summaries[classValue] = summarize(instances)
  print("Summarize Attributes By Class")
  print(summaries)
  print(" ")
```

```
  return summaries

#3.Make Prediction
#3.1 Calculate Probaility Density Function
def calculateProbability(x, mean, stdev):
  exponent = math.exp(-safe_div(math.pow(x-mean,2),(2*math.pow(stdev,2))))
  final = safe_div(1 , (math.sqrt(2*math.pi) * stdev)) * exponent
  return final

#3.2 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

#3.3 Prediction : look for the largest probability and return the associated class
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

#4.Make Predictions
# Function which return predictions for list of predictions
# For each instance
def getPredictions(summaries, testSet):
  predictions = []
  for i in range(len(testSet)):
    result = predict(summaries, testSet[i])
    predictions.append(result)
  return predictions

#5. Computing Accuracy
def getAccuracy(testSet, predictions):
  correct = 0
  for i in range(len(testSet)):
    if testSet[i][-1] == predictions[i]:
      correct += 1
  accuracy = safe_div(correct,float(len(testSet))) * 100.0
  return accuracy
```

```
def main():
  filename = 'diabetes2.csv'
  splitRatio = 0.9
  dataset = loadCsv(filename)
  trainingSet, testSet = splitDataset(dataset, splitRatio)
  print('Split {0} rows into'.format(len(dataset)))
  print('Number of Training data: ' + (repr(len(trainingSet))))
  print('Number of Test Data: ' + (repr(len(testSet))))
  print("\nThe values assumed for the concept learning attributes are\n")
  print("OUTLOOK=> Sunny=1 Overcast=2 Rain=3\nTEMPERATURE=> Hot=1 Mild=2
Cool=3\nHUMIDITY=> High=1 Normal=2\nWIND=> Weak=1 Strong=2")
  print("TARGET CONCEPT:PLAY TENNIS=> Yes=10 No=5")
  print("\nThe Training set are:")
  for x in trainingSet:
    print(x)
    print("\nThe Test data set are:")
  for x in testSet:
    print(x)
  print("\n")

# prepare model
  summaries = summarizeByClass(trainingSet)

# test model
  predictions = getPredictions(summaries, testSet)
  actual = []
  for i in range(len(testSet)):
   vector = testSet[i]
  actual.append(vector[-1])

# Since there are five attribute values, each attribute constitutes to 20% accuracy. So if all attributes
#match with predictions then 100% accuracy
  print('Actual values: {0}%'.format(actual))
  print('Predictions: {0}%'.format(predictions))
  accuracy = getAccuracy(testSet, predictions)
  print('Accuracy: {0}%'.format(accuracy))

main()
```

7. EM Algorithm

Soln)

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


l1 = [0,1,2]
def rename(s):
    l2 = []
    for i in s:
        if i not in l2:
            l2.append(i)

    for i in range(len(s)):
        pos = l2.index(s[i])
        s[i] = l1[pos]

    return s

# import some data to play with
iris = datasets.load_iris()

print("\n IRIS DATA :",iris.data);
print("\n IRIS FEATURES :\n",iris.feature_names)
print("\n IRIS TARGET  :\n",iris.target)
print("\n IRIS TARGET NAMES:\n",iris.target_names)


# Store the inputs as a Pandas Dataframe and set the column names
X = pd.DataFrame(iris.data)

#print(X)
X.columns = ['Sepal_Length','Sepal_Width','Petal_Length','Petal_Width']

#print(X.columns) #print("X:",x)
#print("Y:",y)
y = pd.DataFrame(iris.target)
y.columns = ['Targets']

# Set the size of the plot
plt.figure(figsize=(14,7))
```

```
# Create a colormap
colormap = np.array(['red', 'lime', 'black'])

# Plot Sepal
plt.subplot(1,2,1)
plt.scatter(X.Sepal_Length,X.Sepal_Width, c=colormap[y.Targets], s=40)
plt.title('Sepal')

plt.subplot(1,2,2)
plt.scatter(X.Petal_Length,X.Petal_Width, c=colormap[y.Targets], s=40)
plt.title('Petal')
plt.show()

print("Actual Target is:\n", iris.target)

# K Means Cluster
model = KMeans(n_clusters=3)
model.fit(X)

# Set the size of the plot
plt.figure(figsize=(14,7))

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

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

# Plot the Models Classifications
plt.subplot(1,2,2)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[model.labels_], s=40)
plt.title('K Mean Classification')
plt.show()

km = rename(model.labels_)
print("\nWhat KMeans thought: \n", km)
print("Accuracy of KMeans is ",sm.accuracy_score(y, km))
print("Confusion Matrix for KMeans is \n",sm.confusion_matrix(y, km))

#The GaussianMixture scikit-learn class can be used to model this problem
#and estimate the parameters of the distributions using the expectation-maximization algorithm.

from sklearn import preprocessing
scaler = preprocessing.StandardScaler()
scaler.fit(X)
xsa = scaler.transform(X)
```

```
xs = pd.DataFrame(xsa, columns = X.columns)
print("\n",xs.sample(5))

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

y_cluster_gmm = gmm.predict(xs)

plt.subplot(1, 2, 1)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y_cluster_gmm], s=40)
plt.title('GMM Classification')
plt.show()

em = rename(y_cluster_gmm)
print("\nWhat EM thought: \n", em)
print("Accuracy of EM is ",sm.accuracy_score(y, em))
print("Confusion Matrix for EM is \n", sm.confusion_matrix(y, em))
```

8. KNN Algorithm.

Soln)

```
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()

#display the iris dataset
print("\n IRIS FEATURES \ TARGET NAMES: \n ", iris_dataset.target_names)
for i in range(len(iris_dataset.target_names)):
    print("\n[{0}]:[{1}]".format(i,iris_dataset.target_names[i]))

print("\n IRIS DATA :\n",iris_dataset["data"])

#split the data into training and testing data
X_train, X_test, y_train, y_test = train_test_split(iris_dataset["data"], iris_dataset["target"],
random_state=0)

print("\n Target :\n",iris_dataset["target"])
print("\n X TRAIN \n", X_train)
print("\n X TEST \n", X_test)
print("\n Y TRAIN \n", y_train)
print("\n Y TEST \n", y_test)

#train and fit the model
kn = KNeighborsClassifier(n_neighbors=5)
kn.fit(X_train, y_train)

#predicting from model
x_new = np.array([[5, 2.9, 1, 0.2]])
print("\n XNEW \n",x_new)
prediction = kn.predict(x_new)
print("\n Predicted target value: {}\n".format(prediction))
print("\n Predicted feature name: {}\n".format(iris_dataset["target_names"][prediction]))

i=1
x= X_test[i]
x_new = np.array([x])
print("\n XNEW \n",x_new)

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

```
  print("\n Actual : {0} {1}, Predicted
:{2}{3}".format(y_test[i],iris_dataset["target_names"][y_test[i]],prediction,iris_dataset["target_nam
es"][ prediction]))
print("\n TEST SCORE[ACCURACY]: {:.2f}\n".format(kn.score(X_test, y_test)))
```

9. Regression Algorithm

Soln)

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

def local_regression(x0, X, Y, tau):
    x0 = [1, x0]
    X = [[1, i] for i in X]
    X = np.asarray(X)
    xw = (X.T) * np.exp(np.sum((X - x0) ** 2, axis=1) / (-2 * tau))
    beta = np.linalg.pinv(xw @ X) @ xw @ Y @ x0
    return beta

def draw(tau):
    prediction = [local_regression(x0, X, Y, tau) for x0 in domain]
    plt.plot(X, Y, 'o', color='black')
    plt.plot(domain, prediction, color='red')
    plt.show()

X = np.linspace(-3, 3, num=1000)
domain = X
Y = np.log(np.abs(X ** 2 - 1) + .5)


draw(10)
draw(0.1)
draw(0.01)
draw(0.001)
```