```
#LAB3 CANDIDATE ELIMINATION
import csv
with open("trainingexamples.csv") as f:
  csv_file = csv.reader(f)
  data = list(csv_file)
  specific = data[0][:-1]
  general = [['?' for i in range(len(specific))] for j in
range(len(specific))]
  step=1 #for printing purpose
  for i in data:
    if i[-1] == "Y":
      for j in range(len(specific)):
        if i[j] != specific[j]:
          specific[j] = "?"
          general[j][j] = "?"
    elif i[-1] == "N":
      for j in range(len(specific)):
        if i[j] != specific[j]:
          general[j][j] = specific[j]
        else:
          general[j][j] = "?"
    print("\nStep {} of candidate elimination algo".format(step))
    step+=1
    print(specific)
    print(general)
  gh = [] # gh = general Hypothesis
  for i in general:
    for j in i:
      if j != '?':
        gh.append(i)
        break
  print("\nFinal Specific hypothesis:\n", specific)
  print("\nFinal General hypothesis:\n", gh)
#A*
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
  g[start_node] = 0
  parents[start_node] = start_node
  while len(open_set) > 0:
    n = None # n is node refering
    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):
        if m not in open_set and m not in closed_set:
          open_set.add(m)
          parents[m] = n
          g[m] = g[n] + weight
        else:
          if g[m] > g[n] + weight:
            g[m] = g[n] + weight
            parents[m] = n
          if m in closed_set:
            closed_set.remove(m)
            open_set.add(m)
    if n == None:
      print('Path does not exist!')
      return None
    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
    open_set.remove(n)
    closed_set.add(n)
  print('Path does not exist!')
  return None
```

```
#Naive-bayesian
from sklearn.naive_bayes import GaussianNB
from sklearn.datasets import load_iris
from sklearn import metrics
from sklearn.model_selection import train_test_split
dataset = load_iris()
X = dataset. data
y = dataset. target
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=1)
gnb = GaussianNB()
classifier = gnb.fit(X_train, y_train)
y_pred = classifier.predict(X_test)
print( 'Accuracy metrics:', metrics.classification_report(y_test,
y_pred))
print( 'Accuracy of the classifier is:',
metrics.accuracy_score(y_test, y_pred))
print("Confusion matrix:")
print(metrics.confusion_matrix(y_test, y_pred))
#AO*
class Graph:
  def __init__(self, graph, heuristicNodeList, startNode):
    self.graph = graph
    self.H = heuristicNodeList
    self.start = startNode
    self.parent = {}
    self.status = {}
    self.solutionGraph = {}
  def applyAOStar(self):
    self.aoStar(self.start, False)
  def getNeighbors(self, v):
    return self.graph.get(v, '')
  def getStatus(self, v):
    return self.status.get(v, 0)
  def setStatus(self, v, val):
    self.status[v] = val
  def getHeuristicNodeValue(self, n):
    return self.H.get(n, 0)
  def setHeuristicNodeValue(self, n, value):
    self.H[n] = value
  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):
    minimumCost = 0
    costToChildNodeListDict = {}
    costToChildNodeListDict[minimumCost] = []
    flag = True
    for nodeInfoTupleList in self.getNeighbors(v):
      cost = 0
      nodeList = []
      for c, weight in nodeInfoTupleList:
        cost = cost + self.getHeuristicNodeValue(c) + weight
        nodeList.append(c)
      if flag == True:
        minimumCost = cost
        costToChildNodeListDict[minimumCost] = nodeList
        flag = False
      else:
        if minimumCost > cost:
          minimumCost = cost
          costToChildNodeListDict[minimumCost] = nodeList
    return minimumCost,
costToChildNodeListDict[minimumCost]
  def aoStar(self, v, backTracking):
    print("HEURISTIC VALUES :", self.H)
    print("SOLUTION GRAPH :", self.solutionGraph)
    print("PROCESSING NODE :", v)
    print("-----------------------------------------------------------------------
-----------------")
    if self.getStatus(v) >= 0:
      minimumCost, childNodeList =
self.computeMinimumCostChildNodes(v)
      self.setHeuristicNodeValue(v, minimumCost)
```

```
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None
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]
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')
#knn
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn import datasets
iris=datasets.load_iris()
print("Iris Data set loaded...")
x_train, x_test, y_train, y_test =
train_test_split(iris.data,iris.target,test_size=0.1)
print("Dataset is split into training and testing...")
print("Size of trainng data and its
label",x_train.shape,y_train.shape)
print("Size of trainng data and its label",x_test.shape,
y_test.shape)
for i in range(len(iris.target_names)):
    print("Label", i , "-",str(iris.target_names[i]))
classifier = KNeighborsClassifier(n_neighbors=1)
classifier.fit(x_train, y_train)
y_pred=classifier.predict(x_test)
print("Results of Classification using K-nn with K=1 ")
for r in range(0,len(x_test)):
    print(" Sample:", str(x_test[r]), " Actual-label:", str(y_test[r]), "
Predicted-label:",str(y_pred[r]))
print("Classification Accuracy :" , classifier.score(x_test,y_test));
from sklearn.metrics import classification_report,
confusion_matrix
print('Confusion Matrix')
print(confusion_matrix(y_test,y_pred))
print('Accuracy Metrics')
print(classification_report(y_test,y_pred))
#k-means
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn import datasets
iris=datasets.load_iris()
print("Iris Data set loaded...")
x_train, x_test, y_train, y_test =
train_test_split(iris.data,iris.target,test_size=0.1)
print("Dataset is split into training and testing...")
print("Size of trainng data and its
label",x_train.shape,y_train.shape)
```

```
            self.setStatus(v, len(childNodeList))
            solved = True
            for childNode in childNodeList:
                self.parent[childNode] = v
                if self.getStatus(childNode) != -1:
                    solved = solved & False
            if solved == True:
                self.setStatus(v, -1)
                self.solutionGraph[v] = childNodeList
            if v != self.start:
                self.aoStar(self.parent[v], True)
            if backTracking == False:
                for childNode in childNodeList:
                    self.setStatus(childNode, 0)
                    self.aoStar(childNode, False)
h2 = {'A': 1, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}
graph2 = {
    'A': [[('B', 1), ('C', 1)], [('D', 1)]],
    'B': [[('G', 1)], [('H', 1)]],
    'D': [[('E', 1), ('F', 1)]]
}
G2 = Graph(graph2, h2, 'A')
G2.applyAOStar()
G2.printSolution()
#Ann
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**2)))
    beta = (np.linalg.pinv(xw @ X)) @ (xw @ Y)
    return (beta @ x0)
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)#evenly spaced numbers over [-
3,3] totally num(1000) numbers
domain = X
Y = np.log(np.abs((X ** 2) - 1) + .5)#just creating y values...chosing
this to get W shaped curve
draw(10)
draw(0.1)
draw(0.01)
draw(0.001)
#ID3
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):  #col is basically a column header
    dic={}
    coldata=[row[col] for row in data]
    attr=list(set(coldata))  #set returns only unique values in
coldata
    counts=[0]*len(attr)     #create empty list for every unique
value
    r=len(data)     #no of rows
    c=len(data[0])  #no of columns in each row
```

```
print("Size of trainng data and its label",x_test.shape,
y_test.shape)
for i in range(len(iris.target_names)):
    print("Label", i , "-",str(iris.target_names[i]))
classifier = KNeighborsClassifier(n_neighbors=1)
classifier.fit(x_train, y_train)
y_pred=classifier.predict(x_test)
print("Results of Classification using K-nn with K=1 ")
for r in range(0,len(x_test)):
    print(" Sample:", str(x_test[r]), " Actual-label:", str(y_test[r]), "
Predicted-label:",str(y_pred[r]))
print("Classification Accuracy :" , classifier.score(x_test,y_test));
from sklearn.metrics import classification_report,
confusion_matrix
print('Confusion Matrix')
print(confusion_matrix(y_test,y_pred))
print('Accuracy Metrics')
print(classification_report(y_test,y_pred))
#localWeightRegression
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**2)))
    beta = (np.linalg.pinv(xw @ X)) @ (xw @ Y)
    return (beta @ x0)
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)#evenly spaced numbers over [-
3,3] totally num(1000) numbers
domain = X
Y = np.log(np.abs((X ** 2) - 1) + .5)#just creating y values...chosing
this to get W shaped curve
draw(10)
draw(0.1)
draw(0.01)
draw(0.001)
```

```
    for x in range(len(attr)):   #no of unique values in the "col"
column
        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])]
#initialing the dictionary items
        pos=0
        for y in range(r):
            if data[y][col]==attr[x]:
                if delete:
                    del data[y][col]   #removing tat particular column
(upper in the tree/parent)
                dic[attr[x]][pos]=data[y] #all rows for each unique value
                pos+=1
    return attr,dic   #attr is a list, dic is a set

def entropy(S):
    attr=list(set(S))  #S will basically have last column data(not
necessarily of all rows)
    if len(attr)==1:
        return 0    #if there is either only yes/ only no =>entrop is 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) #find no
of yes and no of no
    sums=0
    for cnt in counts:
        sums+=-1*cnt*math.log(cnt,2)  #base 2(second parameter)
    return sums
def compute_gain(data,col): #col  is column-header
    attr,dic = subtables(data,col,delete=False) #here no deletion,
we just calculate gain
    total_size=len(data)  # |S| value in formula
    entropies=[0]*len(attr) #entropies of each value
    ratio=[0]*len(attr)   # to maintain |Sv|/|S| values
    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) #len of dic=> |Sv|
value
        entropies[x]=entropy([row[-1] for row in dic[attr[x]]])
        total_entropy-=ratio[x]*entropies[x] #acc to formula
    return total_entropy
def build_tree(data,features):
    lastcol=[row[-1] for row in data]
    if(len(set(lastcol)))==1:  #if last column contains either only
"yes" or only "no"
        node=Node("")         #we are not building the tree further(so
no attribute)
        node.answer=lastcol[0] #it'll be either yes/no
        return node
    n=len(data[0])-1   #-1 boz we dont need the last column values
    gains=[0]*n       # gain is initialized to be 0 for all attributes
    for col in range(n):
        gains[col]=compute_gain(data,col)  #compute gain of each
attribute
    split=gains.index(max(gains))         # split will have the index of
attribute with "highest gain"
    node=Node(features[split])           # features list will have
attribute headings(col names)
    fea = features[:split]+features[split+1:]
    attr,dic=subtables(data,split,delete=True)
                                    #dic will have all rows for all those
attributes(key: values)
    for x in range(len(attr)):
        child=build_tree(dic[attr[x]],fea)     #for each value of the
attribute
        node.children.append((attr[x],child))  #again build the tree
(but fea exclude the one already taken)
    return node
```

```
def print_tree(node,level):
   if node.answer!="":           #if its a leaf node
      print(" "*level,node.answer) #just print "level" no of spaces, followed by answer (yes/no)
      return
   print(" "*level,node.attribute)  #attribute in the node
   for value,n in node.children:
      print(" "*(level+1),value)
      print_tree(n,level+2)       # recursive call to the next node (child)
def classify(node,x_test,features): #features: column headers
   if node.answer!="":      #this will be true only for leaf nodes(answer: yes/no)
      print(node.answer)
      return
   pos=features.index(node.attribute) #node.attribute will have the col header
   for value, n in node.children:   #for every value of that attribute
      if x_test[pos]==value:      # for that particular value go along that value
         classify(n,x_test,features) #go deeper in the tree
dataset,features=load_csv("traintennis.csv")
#lastcol=[row[-1] for row in dataset]
node1=build_tree(dataset,features)
print("The decision tree for the dataset using ID3 algorithm is")
print_tree(node1,0)
testdata,features=load_csv("testtennis.csv")
for xtest in testdata:   #xtest is each row in testdata
   print("The test instance:",xtest)
   print("The label for test instance:",end=" ")
   classify(node1,xtest,features)
```