THE OXFORD COLLEGE OF ENGINEERING

BOMMANAHALLI, HOSUR ROAD, BENGALURU-560068.

(Affiliated to Visvesvaraya Technological University, Belgaum)

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

LAB MANUAL

Subject Name : Artificial Intelligence and Machine Learning Lab

Subject Code : 18CSL76

Semester : VII

Academic Year: 2023-24

Prepared by

Ms.VISALINI S

Ms.C A BINDYASHREE

Table of Contents

Children’s Education Society ®

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

THE OXFORD COLLEGE OF ENGINEERING

Hosur Road, Bommanahalli, Bengaluru-560 068

Website:www.theoxford.edu Email : enghodcse@theoxford.edu

(Approved by AICTE, New Delhi, Accredited by NBA, NAAC, New Delhi & Affiliated to VTU, Belgaum)

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING LABORATORY (Effective from the academic year 2018 -2019) SEMESTER – VII			
Course Code	**18CSL76**	**CIE Marks**	40
Number of Contact Hours/Week	0:0:2	**SEE Marks**	60
Total Number of Lab Contact Hours	36	**Exam Hours**	03
Credits – 2			

Course Learning Objectives: This course (18CSL76) will enable students to:

- Implement and evaluate AI and ML algorithms in and Python programming language.

Descriptions (if any):

Installation procedure of the required software must be demonstrated, carried out in groups and documented in the journal.

Programs List:

1.	Implement A* Search algorithm.
2.	Implement AO* Search algorithm.
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.
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.
5.	Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets.
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.
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.
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.
9.	Implement the non-parametric Locally Weighted Regressionalgorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs

Laboratory Outcomes: The student should be able to:

- Implement and demonstrate AI and ML algorithms.
- Evaluate different algorithms.

Conduct of Practical Examination:

- Experiment distribution
    - For laboratories having only one part: Students are allowed to pick one experiment from the lot with equal opportunity.
    - For laboratories having PART A and PART B: Students are allowed to pick one experiment from PART A and one experiment from PART B, with equal opportunity.
- Change of experiment is allowed only once and marks allotted for procedure to be made zero of the changed part only.
- Marks Distribution *(Courseed to change in accoradance with university regulations)*

  q) For laboratories having only one part – Procedure + Execution + Viva-Voce: 15+70+15 = 100 Marks

  r) For laboratories having PART A and PART B

     i. Part A – Procedure + Execution + Viva = 6 + 28 + 6 = 40 Marks

     ii. Part B – Procedure + Execution + Viva = 9 + 42 + 9 = 60 Marks

1. Implement A* Search algorithm

Algorithm:

1. Start with OPEN containing only the initial state (node). Set that node's g value 0 its h' value to whatever it is and its f' value h'+ 0 or h'. Set CLOSED to the empty list.
2. Until a goal node is found repeat the following procedure: If there are no nodes on OPEN, report failure. Otherwise pick the node on OPEN with lowest f' value. CALL it BESTNODE. Remove from OPEN. Place it on CLOSED. If BESTNODE is the goal node, exit and report a solution. Otherwise, generate the successors of BESTNODE. For each successor, do the following
    a) Set successors to point back to BESTNODE. These backwards links will make possible to recover the path once a solution is found.
    b) Compute
    c) If successor is already existed in OPEN call that node as OLD and we must decide whether OLD' s parent link should reset to point to BESTNODE (graphs exist in this case). If OLD is cheaper then we need do nothing. If successor is cheaper then reset OLD's parent link to point to BESTNODE. Record the new cheaper path in g(OLD) and update f'(OLD).
    d) If SUCCESSOR was not on OPEN, see if it is on CLOSED. If so, call node on CLOSED OLD and add OLD to the list of BESTNODE successors. Calculate all the g, f' and h' values for successors of that node which is better then move that. So, to propagate the new cost downward, do a depth first traversal of the tree starting at OLD, changing each nodes value (and thus also its f' value), terminating each branch when you reach either a node with no successor or a node which an equivalent or better path has already been found.
    e) If successor was not already on either OPEN or CLOSED, then put it on OPEN and add it to the list of BESTNODE successors. Compute

$$f'(successor) = g(successor) + h'(successor)$$

In [4]:

```python
def aStarAlgo(start_node , stop_node):
    open_set = set(start_node)
    closed_set = set()
    g = {}
    parents = {}
    g[start_node] = 0
    parents[start_node] = start_node
    while len(open_set)>0:
        n = None
        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_neighbours(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 not found')
            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 doesn't exist")
    return None

def get_neighbours(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None

def heuristic(n):
    H_dist = {
        'A' : 11,
        'B' : 6,
        'C' : 99,
        'D' : 1,
        'E' : 7,
        'G' : 0
        }
```

```
    return H_dist[n]

Graph_nodes = {
    'A' : [('B',2),('E',3)] ,
    'B' : [('C',1),('G',9)] ,
    'C' : None ,
    'E' : [('D',6)] ,
    'D' : [('G',1)]
}

aStarAlgo('A','G')
```

```
Path found : ['A', 'E', 'D', 'G']
```

Out[4]:

```
['A', 'E', 'D', 'G']
```

2. Implement AO* Search algorithm

Algorithm:

- **Input:** Weighted Directed Graph (G) with Heuristics(h) pre-computed, Start node.
- **Output:** Optimal path and cost in the graph

Step-1: Create an initial graph with a single node (start node).

Step-2: Transverse the graph following the current path, accumulating node that has not yet been expanded or solved.

Step-3: Select any of these nodes and explore it. If it has no successors then call this value- FUTILITY else calculate f'(n) for each of the successors.

Step-4: If **f'(n)=0**, then mark the node as **SOLVED**.

Step-5: Change the value of f'(n) for the newly created node to reflect its successors by backpropagation.

Step-6: Whenever possible use the most promising routes, If a node is marked as SOLVED then mark the parent node as SOLVED.

Step-7: If the starting node is SOLVED or value is greater than **FUTILITY** then stop else repeat from Step-2.

In [2]:

```
def recAOStar(n):
    global finalPath
    print('Expanding node:',n)
    and_nodes = []
    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 len(and_nodes)==0 and len(or_nodes)==0:
        return
    solvable = False
    marked = {}
    while not solvable:
        if len(marked)==len(and_nodes)+len(or_nodes):
            min_cost_least, min_cost_group_least = least_cost_grop(and_nodes, or_nodes, {})
            solvable = True
            change_heuristic(n, min_cost_least)
            optimal_child_group[n] = min_cost_group_least
            continue
        min_cost, min_cost_group = least_cost_group(and_nodes, or_nodes,marked)
        is_expanded = False
        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 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
        else:
            solvable = True
            change_heuristic(n, min_cost)
            optimal_child_group[n] = min_cost_group
    marked[min_cost_group] = 1
    return heuristic(n)

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
```

```python
    min_cost = 999999
    min_cost_group = None
    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]

def heuristic(n):
    return H_dist[n]

def change_heuristic(n,cost):
    H_dist[n] = cost
    return

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 give optimal cost are:')
print_path('A')
print("\nOptimal Cost is : ",optimal_cost)
```

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

In [ ]:

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.

Algorithm

Initialize G to the set of maximally general hypotheses in H Initialize S to the set of maximally specific hypotheses in H For each training example d, do

- If d is a positive example
    - Remove from G any hypothesis inconsistent with d
    - For each hypothesis s in S that is not consistent with d
        - Remove s from S
        - Add to S all minimal generalizations h of s such that
            - h is consistent with d, and some member of G is more general than h
        - Remove from S any hypothesis that is more general than another hypothesis in S
- If d is a negative example
    - Remove from S any hypothesis inconsistent with d
    - For each hypothesis g in G that is not consistent with d
        - Remove g from G
        - Add to G all minimal specializations h of g such that
            - h is consistent with d, and some member of S is more specific than h
        - Remove from G any hypothesis that is less general than another hypothesis in G

In [8]:

```
import numpy as np
import pandas as pd

data = pd.DataFrame(data=pd.read_csv('Training.csv'))
print(data)

concepts = np.array(data.iloc[:,0:-1])
target = np.array(data.iloc[:,-1])


def learn(concepts, target):
    specific_h = concepts[0].copy()
    print("\nInitialization of specific_h and general_h")
    print("\n",specific_h)
    general_h = [["?" for i in range(len(specific_h))] for i in range(len(specific_h))]
    print("\n",general_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] = '?'
                    general_h[x][x] = '?'
        if target[i] == "No":
            for x in range(len(specific_h)):
                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("\nSpecific_h ",i+1,"\n ")
        print(specific_h)
        print("\ngeneral_h ", i+1, "\n ")
        print(general_h)
    indices = [i for i, val in enumerate(general_h) if val == ['?', '?', '?', '?', '?', '?'
    for i in indices:
        general_h.remove(['?', '?', '?', '?', '?', '?'])
    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")
```

```
      Sky AirTemp Humidity    Wind Water Forecast EnjoySport
0    sunny    warm   normal  strong  warm     same        Yes
1    sunny    warm     high  strong  warm     same        Yes
2   cloudy    cold     high  strong  warm   change         No
3    sunny    warm     high  strong  cool   change        Yes

Initialization of specific_h and general_h

 ['sunny' 'warm' 'normal' 'strong' 'warm' 'same']

 [['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?',
'?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?',
'?', '?'], ['?', '?', '?', '?', '?', '?']]
```

```
steps of Candidate Elimination Algorithm 1

Specific_h 1

['sunny' 'warm' 'normal' 'strong' 'warm' 'same']

general_h 1

[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?',
'?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?',
'?', '?'], ['?', '?', '?', '?', '?', '?']]

steps of Candidate Elimination Algorithm 2

Specific_h 2

['sunny' 'warm' '?' 'strong' 'warm' 'same']

general_h 2

[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?',
'?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?',
'?', '?'], ['?', '?', '?', '?', '?', '?']]

steps of Candidate Elimination Algorithm 3

Specific_h 3

['sunny' 'warm' '?' 'strong' 'warm' 'same']

general_h 3

[['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?'],
['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?',
'?', '?', '?', '?'], ['?', '?', '?', '?', '?', 'same']]

steps of Candidate Elimination Algorithm 4

Specific_h 4

['sunny' 'warm' '?' 'strong' '?' '?']

general_h 4

[['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?'],
['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?',
'?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Final Specific_h:
['sunny' 'warm' '?' 'strong' '?' '?']

Final General_h:
[['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 to classify a new sample.

Algorithm

ID3(Examples, Target_attribute, Attributes)

Examples are the training examples.

Target_attribute is the attribute whose value is to be predicted by the tree.

Attributes is a list of other attributes that may be tested by the learned decision tree.

Returns a decision tree that correctly classifies the given Examples.

- Create a Root node for the tree
- If all Examples are positive, Return the single-node tree Root, with label = +
- If all Examples are negative, Return the single-node tree Root, with label = -
- If Attributes is empty, Return the single-node tree Root, with label = most common value of Target_attribute in Examples
- Otherwise Begin

    A ← the attribute from Attributes that best* classifies Examples

    The decision attribute for Root ← A

    For each possible value, v_i, of A,

    Add a new tree branch below *Root*, corresponding to the test A = v_i

    Let *Examples* $_{vi}$, be the subset of Examples that have value v_i for *A*

    If *Examples* $_{vi}$, is empty

    Then below this new branch add a leaf node with label = most common value of Target_attribute in Examples

    Else below this new branch add the subtree ID3(*Examples* $_{vi}$, Targe_tattribute, Attributes – {A}))

  End

  Return Root

In [11]:

```
import math

def dataset_split(data, arc, val):
    newData = []
    for rec in data:
         if rec[arc] == val:
            reducedSet = list(rec[:arc])
            reducedSet.extend(rec[arc+1:])
            newData.append(reducedSet)
    return newData

def calc_entropy(data):
    entries = len(data)
    labels = {}
    for rec in data:
        label = rec[-1]
        if label not in labels.keys():
            labels[label] = 0
        labels[label] += 1
    entropy = 0.0
    for key in labels:
        prob = float(labels[key])/entries
        entropy -= prob * math.log(prob, 2)
    return entropy

def attribute_selection(data):
    features = len(data[0]) - 1
    baseEntropy = calc_entropy(data)
    max_InfoGain = 0.0
    bestAttr = -1

    for i in range(features):
        AttrList = [rec[i] for rec in data]
        uniqueVals = set(AttrList)
        newEntropy = 0.0
        attrEntropy = 0.0
        for value in uniqueVals:
            newData = dataset_split(data, i, value)
            prob = len(newData)/float(len(data))
            newEntropy = prob * calc_entropy(newData)
            attrEntropy += newEntropy
        infoGain = baseEntropy - attrEntropy
        if infoGain > max_InfoGain:
            max_InfoGain = infoGain
            bestAttr = i
    return bestAttr

def decision_tree(data, labels):
    classList = [rec[-1] for rec in data]
    if classList.count(classList[0]) == len(classList):
        return classList[0]
    maxGainNode = attribute_selection(data)
    treeLabel = labels[maxGainNode]
    theTree = {treeLabel: {}}
    del(labels[maxGainNode])
    nodeValues = [rec[maxGainNode] for rec in data]
    uniqueVals = set(nodeValues)
    for value in uniqueVals:
        subLabels = labels[:]
```

```
        theTree[treeLabel][value] = decision_tree(dataset_split(data, maxGainNode, value),
    return theTree

def print_tree(tree, level):
    if tree == 'yes' or tree == 'no':
        print(' '*level, 'd=', tree)
        return
    for key,value in tree.items():
        print(' ' *level, key)
        print_tree(value, level*2)

with open('tennis.csv', 'r') as csvfile:
    fdata = [line.strip() for line in csvfile]
    metadata = fdata[0].split(',')
    train_data = [x.split(',') for x in fdata[1:]]

tree = decision_tree(train_data, metadata)
print_tree(tree, 1)
print(tree)
```

```
  Outlook
   overcast
     d= yes
   rain
     Wind
         weak
                 d= yes
         strong
                 d= no
   sunny

     Humidity
         high
                 d= no
         normal
                 d= yes
{'Outlook': {'overcast': 'yes', 'rain': {'Wind': {'weak': 'yes', 'strong':
'no'}}, 'sunny': {'Humidity': {'high': 'no', 'normal': 'yes'}}}}
```

In [ ]:

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

Algorithm

- Create a feed-forward network with n_i inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers
- Until the termination condition is met, Do
  - For each (x, t), in training examples, Do
    - Propagate the input forward through the network:

      1. Input the instance x, to the network and compute the output o_u of every unit u in the network.
    - Propagate the errors backward through the network

      2. For each network unit k, calculate its error term δ_k

      $$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

      3. For each network unit h, calculate its error term δ_h

      $$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{h,k}\delta_k$$

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

Where

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

In [2]:

```
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)

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def der_sigmoid(x):
    return x * (1 - x)

epoch = 5000
lr = 0.01

neurons_i = 2
neurons_h = 3
neurons_o = 1

weight_h = np.random.uniform(size=(neurons_i, neurons_h))
bias_h = np.random.uniform(size=(1, neurons_h))
weight_o = np.random.uniform(size=(neurons_h, neurons_o))
bias_o = np.random.uniform(size=(1, neurons_o))

for i in range(epoch):
    inp_h = np.dot(X, weight_h) + bias_h
    out_h = sigmoid(inp_h)
    inp_o = np.dot(out_h, weight_o) + bias_o
    out_o = sigmoid(inp_o)
    err_o = y - out_o
    grad_o = der_sigmoid(out_o)
    delta_o = err_o * grad_o
    err_h = delta_o.dot(weight_o.T)
    grad_h = der_sigmoid(out_h)
    delta_h = err_h * grad_h
    weight_o += out_h.T.dot(delta_o) * lr
    weight_h += X.T.dot(delta_h) * lr

print('Input: ', X)
print('Actual: ', y)
print('Predicted: ', out_o)
```

```
Input:  [[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual:  [[0.92]
 [0.86]
 [0.89]]
Predicted:  [[0.89077146]
 [0.8744389 ]
 [0.89555458]]
```

In [ ]:

```
```

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.

Algorithm:

Naive_Bayes_Learn($examples$)

For each target value v_j

$\hat{P}(v_j) \leftarrow$ estimate $P(v_j)$

For each attribute value a_i of each attribute a

$\hat{P}(a_i|v_j) \leftarrow$ estimate $P(a_i|v_j)$

Classify_New_Instance(x)

$$v_{NB} = \underset{v_j \in V}{\operatorname{argmax}} \hat{P}(v_j) \prod_{a_i \in x} \hat{P}(a_i|v_j)$$

In [2]:

```
import pandas as pd
import numpy as np

mush = pd.read_csv('mushrooms.csv')
mush = mush.replace('?',np.nan)
mush.dropna(axis=1,inplace=True)
target = 'class'
features = mush.columns[mush.columns!=target]
target_classes=mush[target].unique()
test = mush.sample(frac = .3)
mush = mush.drop(test.index)
cond_probs = {}
target_class_prob = {}

for t in target_classes:
    mush_t = mush[mush[target]==t][features]
    target_class_prob[t] = float(len(mush_t)/len(mush))
    class_prob = {}
    for col in mush_t.columns:
        col_prob = {}
        for val,cnt in mush_t[col].value_counts().iteritems():
            pr = cnt/len(mush_t)
            col_prob[val] = pr
        class_prob[col] = col_prob
    cond_probs[t] = class_prob

def calc_probs(x):
    probs = {}
    for t in target_classes:
        p = target_class_prob[t]
        for col,val in x.iteritems():
            try:
                p *= cond_probs[t][col][val]
            except:
                p = 0
        probs[t] = p
    return probs

def classify(x):
    probs = calc_probs(x)
    max = 0
    max_class = ' '
    for cl,pr in probs.items():
        if pr>max:
            max = pr
            max_class = cl
    return max_class

b = []
for i in mush.index:
    b.append(classify(mush.loc[i,features]) == mush.loc[i,target])
print(sum(b)," correct of ",len(mush))
print('Accuracy : ',sum(b)/len(mush))

b = []
for i in test.index:
    b.append(classify(test.loc[i,features]) == test.loc[i,target])
print(sum(b)," correct of ",len(test))
print('Accuracy : ',sum(b)/len(test))
```

```
5669  correct of  5687
Accuracy :  0.9968348865834359
2433  correct of  2437
Accuracy :  0.9983586376692655
```

In [ ]:

```

```

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.

Algorithm

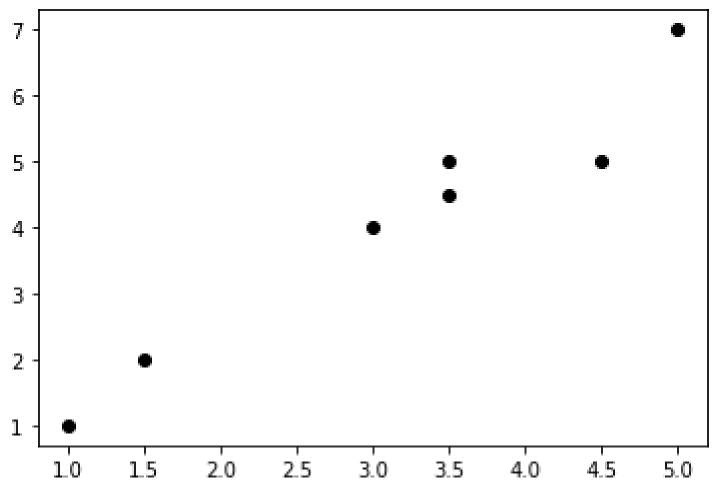
Step 1: Calculate the expected value $E[z_{ij}]$ of each hidden variable z_{ij}, assuming the current hypothesis $h = \langle \mu_1, \mu_2 \rangle$ holds.

Step 2: Calculate a new maximum likelihood hypothesis $h' = \langle \mu'_1, \mu'_2 \rangle$, assuming the value taken on by each hidden variable z_{ij} is its expected value $E[z_{ij}]$ calculated in Step 1. Then replace the hypothesis $h = \langle \mu_1, \mu_2 \rangle$ by the new hypothesis $h' = \langle \mu'_1, \mu'_2 \rangle$ and iterate.

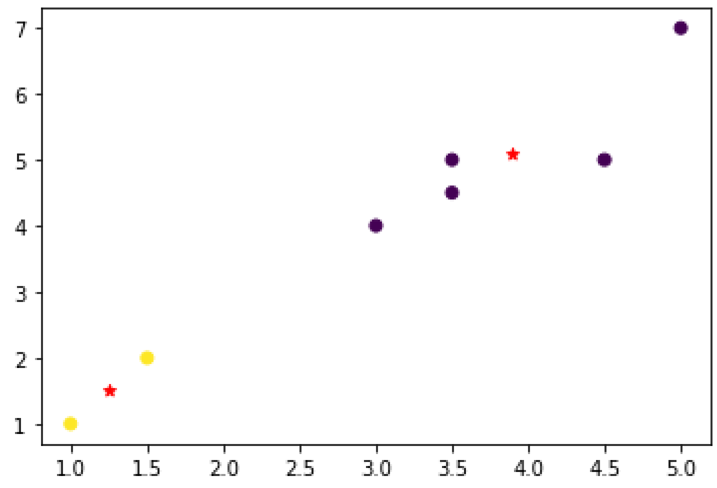
In [4]:

```python
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.mixture import GaussianMixture
from sklearn.cluster import KMeans
data = pd.read_csv('ex.csv')
f1 = data['V1'].values
f2 = data['V2'].values
X = np.array(list(zip(f1,f2)))
print("x: ",X)
print("Graph for whole dataset")
plt.scatter(f1,f2,c='black')
plt.show()
KMeans =KMeans(2)
labels = KMeans.fit(X).predict(X)
print("labels for KMeans:",labels)
print('Graph using KMeans Algorithm')
plt.scatter(f1,f2,c = labels)
centroids = KMeans.cluster_centers_
print("centroids: ",centroids)
plt.scatter(centroids[:,0],centroids[:,1],marker ='*',c='red')
plt.show()
gmm=GaussianMixture(2)
Labels=gmm.fit(X).predict(X)
print("Labels for GMM: ",labels)
print('Graph using EM Algorithm')
plt.scatter(f1,f2,c=labels)
plt.show()
```

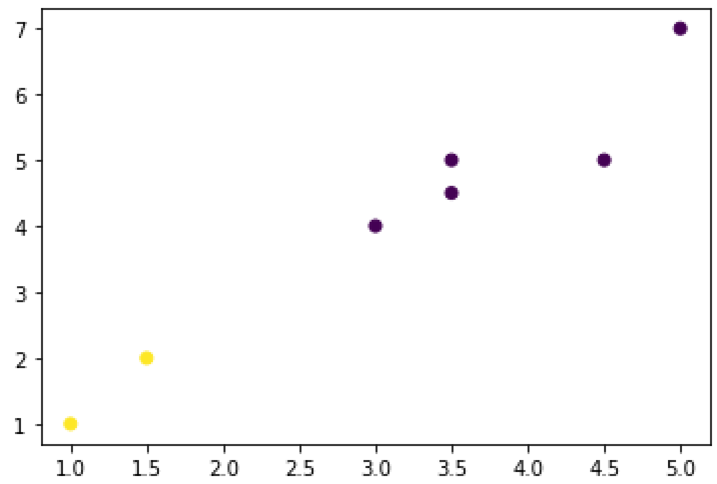
```
x:  [[1.  1. ]
 [1.5 2. ]
 [3.  4. ]
 [5.  7. ]
 [3.5 5. ]
 [4.5 5. ]
 [3.5 4.5]]
Graph for whole dataset
```




```
labels for KMeans: [1 1 0 0 0 0 0]
Graph using KMeans Algorithm
centroids: [[3.9 5.1 ]
 [1.25 1.5 ]]
```




```
Labels for GMM: [1 1 0 0 0 0 0]
Graph using EM Algorithm
```



In [ ]:

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.

Algorithm

- ***Step 1:*** First, Find the distance.
- ***Step 2:*** Find the rank
- ***Step 3:*** Find the nearest neighbor.

In [2]:

```
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()
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['targ
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]],predict
print("\nAccuracy:",kn.score(X_test,Y_test))
```

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

Accuracy: 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 draw graphs.

Algorithm

- Locally weighted linear regression is a supervised learning algorithm.
- It a non-parametric algorithm.
- There exists No training phase. All the work is done during the testing phase/while making predictions.

ALGORITHM:

1. Read the Given data Sample to X and the curve (linear or non linear) to Y
2. Set the value for Smoothening parameter or free parameter say τ
3. Set the bias /Point of interest set X0 which is a subset of X
4. Determine the weight matrix using:

$$w(x, x_o) = e^{-\frac{(x-x_o)^2}{2\tau^2}}$$

5. Determine the value of model term parameter β using :

$$\hat{\beta}(x_o) = (X^T W X)^{-1} X^T W y$$

6. Prediction = x0*β

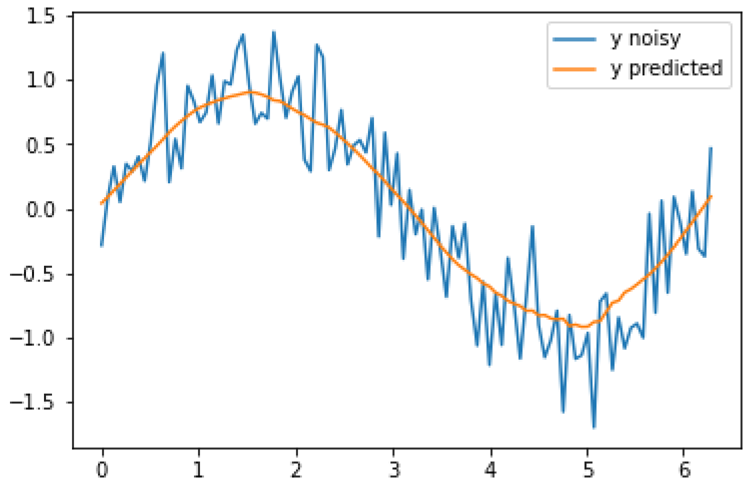
In [5]:

```
from math import ceil
import numpy as np
from scipy import linalg

def lowess(x, y, f=2./3., iter=3):
    n = len(x)
    r = int(ceil(f*n))
    h = [np.sort(np.abs(x-x[i]))[r] for i in range(n)]
    w = np.clip(np.abs((x[:,None]-x[None,:])/h) , 0.0 , 1.0)
    w = (1- w**3) ** 3
    yest = np.zeros(n)
    delta = np.ones(n)
    for iteration in range(iter):
        for i in range(n):
            weights = delta*w[:,i]
            b = np.array([np.sum(weights*y) , np.sum(weights*y*x)])
            A = np.array([[np.sum(weights) , np.sum(weights*x)],
                          [np.sum(weights*x),np.sum(weights*x*x)]])
            beta = linalg.solve(A,b)
            yest[i] = beta[0] + beta[1]*x[i]
            residuals = y - yest
            s = np.median(np.abs(residuals))
            delta = np.clip(residuals/(6.0*s),-1,1)
            delta = (1 - delta**2) ** 2
        return yest

if __name__=='__main__':
    import math
    n = 100
    x = np.linspace(0 , 2*math.pi , n)
    y = np.sin(x) + 0.3 * np.random.randn(n)
    f = 0.25
    yest = lowess(x,y,f,3)

import pylab as pl
pl.clf()
pl.plot(x,y,label='y noisy')
pl.plot(x,yest,label='y predicted')
pl.legend()
pl.show()
```



VIVA Questions

1. What is Artificial Intelligence?
2. Explain A* Search algorithm.
3. Explain AO* Search algorithm.
4. How is AO* search different from A* search algorithm.
5. What is machine learning?
6. Define supervised learning
7. Define unsupervised learning
8. Define semi supervised learning
9. Define reinforcement learning
10. What do you mean by hypotheses?
11. What is classification?
12. What is clustering?
13. Define precision, accuracy and recall
14. Define entropy
15. Define regression
16. How KNN is different from K-Means clustering
17. What is concept learning?
18. Define specific boundary and general boundary
19. Define target function
20. Define decision tree
21. What is ANN
22. Explain gradient descent approximation
23. State Bayes theorem
24. Define Bayesian belief networks
25. Differentiate hard and soft clustering
26. Define variance
27. What is inductive machine learning?
28. Why K Nearest Neighbor algorithm is lazy learning algorithm
29. Why naïve Bayes is naïve
30. Mention classification algorithms
31. Define pruning
32. Differentiate Clustering and classification
33. Mention clustering algorithms
34. Define Bias
35. What is learning rate? Why it is needed