Artificial Intelligence And Machine Learning Laboratory Manual
(18CSL76)

Program-1

Implement A* Search Algorithm

Algorithm:

1. Place the starting node list in “OPEN” list.
2. Check if the OPEN list is empty or not

   If list is empty then,

      return failure and stop
3. Select the node from “OPEN” set which has the smallest value of evaluation f.

   If node n is goal node then,

      Return success and stop
4. Expand node n and generate all of its successors and put n to the CLOSED list

➢ For each successor ‘n’ check whether its already there in OPEN and CLOSED list.

➢ If not compute f value and place into OPEN

Else if node n is already there in OPEN and CLOSE then it should be attached to back pointer which should reflect lowest g(n) value.

5. Return Step 2.

Program:

```
#!/usr/bin/env python

#coding utf-8

#ln[1]:


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_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 doesnot exist")
   return None
def get_neighbors(v):
   if v in Graph_nodes:
      return Graph_nodes[v]
   else:
      return None
def heuristic(n):
   H_dist={
      'S':7,
      'A':6,
      'B':2,
      'C':1,
      'D':0

   }
   return H_dist[n]

Graph_nodes={
   'S':[('A',1),('B',4)],
   'A':[('B',2),('C',5),('D',12)],
   'B':[('C',2)],
   'C':[('D',3)],
```

```
}
aStarAlgo('S','D')
```

Output:

```
path found:['S', 'A', 'B', 'C', 'D']
['S', 'A', 'B', 'C', 'D']
```

Program 2

Implement AO* Algorithm

Algorithm:

1. Let G be a graph with only starting node INIT.
2. Repeat the following until INIT is labeled as SOLVED or h(INIT)>FUTILITY.

- Select an unexpanded node from the most promising path from INIT (call it NODE).
- Generate SUCCESSORS of NODE.

  If ther are none,

  Set h(NODE)=FUTILITY(i.e NODE is unsolvable)

  Otherwise,

  For each SUCCESSORS that is not an ancestor of NODE

  Do the following:

  1. Add SUCCESSOR toG.
  2. If SUCCESSOR is a terminal node,

     label it SOLVED and set h(SUCCESSOR)=0
  3. If SUCCESSOR is not a terminal node,

     Compute its h value
- Propagate the newly discovered information up the graph by doing the following:

  1. Let S be set of SOLVED node or nodes whose h values have been changed and need to have values propagated back to their parents.
  2. Initialize S to node, until S is empty.
  3. Repeat the following:

     Remove a node from S and call it CURRENT.

Program:

```
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):        # 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()
```

Output:

```
HEURISTIC VALUES  : {'A': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, 'T':
 3}
SOLUTION GRAPH    : {}
PROCESSING NODE   : A
-----------------------------------------------------------------------------------------
HEURISTIC VALUES  : {'A': 10, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, '
T': 3}
SOLUTION GRAPH    : {}
PROCESSING NODE   : B
-----------------------------------------------------------------------------------------
HEURISTIC VALUES  : {'A': 10, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, '
T': 3}
SOLUTION GRAPH    : {}
PROCESSING NODE   : A
-----------------------------------------------------------------------------------------
HEURISTIC VALUES  : {'A': 10, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, '
T': 3}
SOLUTION GRAPH    : {}
PROCESSING NODE   : G
-----------------------------------------------------------------------------------------
```

```
HEURISTIC VALUES  : {'A': 10, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 8, 'H': 7, 'I': 7, 'J': 1, '
T': 3}
SOLUTION GRAPH    : {}
PROCESSING NODE   : B
--------------------------------------------------------------------------------------
HEURISTIC VALUES  : {'A': 10, 'B': 8, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 8, 'H': 7, 'I': 7, 'J': 1, '
T': 3}
SOLUTION GRAPH    : {}
PROCESSING NODE   : A
--------------------------------------------------------------------------------------
HEURISTIC VALUES  : {'A': 12, 'B': 8, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 8, 'H': 7, 'I': 7, 'J': 1, '
T': 3}
SOLUTION GRAPH    : {}
PROCESSING NODE   : I
--------------------------------------------------------------------------------------
HEURISTIC VALUES  : {'A': 12, 'B': 8, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 8, 'H': 7, 'I': 0, 'J': 1, '
T': 3}
SOLUTION GRAPH    : {'I': []}
PROCESSING NODE   : G
--------------------------------------------------------------------------------------
HEURISTIC VALUES  : {'A': 12, 'B': 8, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1, '
T': 3}
SOLUTION GRAPH    : {'I': [], 'G': ['I']}
PROCESSING NODE   : B
--------------------------------------------------------------------------------------
HEURISTIC VALUES  : {'A': 12, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1, '
T': 3}
SOLUTION GRAPH    : {'I': [], 'G': ['I'], 'B': ['G']}
PROCESSING NODE   : A
--------------------------------------------------------------------------------------
HEURISTIC VALUES  : {'A': 6, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1, 'T':
 3}
SOLUTION GRAPH    : {'I': [], 'G': ['I'], 'B': ['G']}
PROCESSING NODE   : C
--------------------------------------------------------------------------------------
HEURISTIC VALUES  : {'A': 6, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1, 'T':
 3}
SOLUTION GRAPH    : {'I': [], 'G': ['I'], 'B': ['G']}
PROCESSING NODE   : A
--------------------------------------------------------------------------------------
HEURISTIC VALUES  : {'A': 6, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1, 'T':
 3}
SOLUTION GRAPH    : {'I': [], 'G': ['I'], 'B': ['G']}
PROCESSING NODE   : J
--------------------------------------------------------------------------------------
HEURISTIC VALUES  : {'A': 6, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 0, 'T':
 3}
```

SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G'], 'J': []}
PROCESSING NODE : C

HEURISTIC VALUES : {'A': 6, 'B': 2, 'C': 1, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 0, 'T': 3}
SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G'], 'J': [], 'C': ['J']}
PROCESSING NODE : A

FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE: A

{'I': [], 'G': ['I'], 'B': ['G'], 'J': [], 'C': ['J'], 'A': ['B', 'C']}

Program 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:

1. Initialize G to the set of normally general hypothesis in H.
2. Initialize S to the set of maximally specific hypothesis in H.
3. For each training example d, do:
   a. If d is positive example
      - Remove from G any hypothesis h inconsistent with d
      - For each hypothesis s in S not consistent with d:
        - Remove s from S
        - Add to S all minimal generalizations of S consistent with d & having a generalization in G
        - Remove from S any hypothesis with a more specific h i n S
   b. If d is negative example
      - Remove from S any hypothesis h inconsistent with d
      - For each hypothesis g in G not consistent with d:
        - Remove g from G
        - Add to G all minimal specializations of g consistent with d &having a specialization in S
        - Remove from G any hypothesis having a more general hypothesis in G

File: Enjoysport.csv

sky	airtemp	humidity	wind	water	forcast	enjoysport
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

Program:

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

```
data=pd.read_csv("enjoysport.csv")

concepts=np.array(data.iloc[:,0:-1])
print("\nInstances are;\n",concepts)
target=np.array(data.iloc[:,-1])
print("\nTarget value are:",target)

def learn(concepts,target):
    specific_h=concepts[0].copy()
    print("\n Initialization of specific_h and general_h")
    print("\n Specific Boundary:",specific_h)
    general_h=[["?" for i in range(len(specific_h))] for i in range(len(specific_h))]
    print("\n Generic Boundary:",general_h)

    for i,h in enumerate(concepts):
        print("\n instance ",i+1,"is",h)
        if target[i]=="yes":
            print("\n Instance is Positive")
            for x in range(len(specific_h)):
                if h[x]!=specific_h[x]:
                    specific_h[x]='?'
                    general_h[x][x]='?'
        if target[i]=="no":
            print("Instance is negative")
            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("Specific Boundary after",i+1,"Instance is",specific_h)
        print("Generic Boundary after ",i+1,"Instance is",general_h)
        print("\n")
    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("Final Specific_h:",s_final,sep="\n")
print("Final General_h:",g_final,sep="\n")
```

Output:

```
Instances are;
 [['sunny' 'warm' 'normal' 'strong' 'warm' 'same']
 ['sunny' 'warm' 'high' 'strong' 'warm' 'same']
```

 ['rainy' 'cold' 'high' 'strong' 'warm' 'change']
 ['sunny' 'warm' 'high' 'strong' 'cool' 'change']]

Target value are: ['yes' 'yes' 'no' 'yes']

Initialization of specific_h and general_h

Specific Boundary: ['sunny' 'warm' 'normal' 'strong' 'warm' 'same']

Generic Boundary: [['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

instance 1 is ['sunny' 'warm' 'normal' 'strong' 'warm' 'same']

Instance is Positive
Specific Boundary after 1 Instance is ['sunny' 'warm' 'normal' 'strong' 'warm' 'same']
Generic Boundary after 1 Instance is [['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

instance 2 is ['sunny' 'warm' 'high' 'strong' 'warm' 'same']

Instance is Positive
Specific Boundary after 2 Instance is ['sunny' 'warm' '?' 'strong' 'warm' 'same']
Generic Boundary after 2 Instance is [['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

instance 3 is ['rainy' 'cold' 'high' 'strong' 'warm' 'change']
Instance is negative
Specific Boundary after 3 Instance is ['sunny' 'warm' '?' 'strong' 'warm' 'same']
Generic Boundary after 3 Instance is [['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', 'same']]

instance 4 is ['sunny' 'warm' 'high' 'strong' 'cool' 'change']

Instance is Positive
Specific Boundary after 4 Instance is ['sunny' 'warm' '?' 'strong' '?' '?']
Generic Boundary after 4 Instance is [['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Final Specific_h:

['sunny' 'warm' '?' 'strong' '?' '?']
Final General_h:
[['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?']]

Program 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.

1.Create a Root node for the tree
2.If all Examples are positive,
    Return the single-node tree Root,
        with label = +
3.If all Examples are negative,
    Return the single-node tree Root,
        with label = -
4.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, vi, of A,
    - Add a new tree branch below Root, corresponding to the test A = vi
    - Let Examples vi, be the subset of Examples that have value vi 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,Attribute – {A}))

5.End
6.Return Root

Files:

id3_test_1.csv

outlook	temparture	humidity	wind
rain	cool	normal	strong
sunny	mild	normal	strong

id.csv

Outlook	Temperature	Humidity	Wind	Answer
sunny	hot	high	weak	no
sunny	hot	high	strong	no
overcast	hot	high	weak	yes
rain	mild	high	weak	yes
rain	cool	normal	weak	yes
rain	cool	normal	strong	no
overcast	cool	normal	strong	yes
sunny	mild	high	weak	no
sunny	cool	normal	weak	yes
rain	mild	normal	weak	yes
sunny	mild	normal	strong	yes
overcast	mild	high	strong	yes
overcast	hot	normal	weak	yes
rain	mild	high	strong	no

Program:

```
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))
    for k in attr:
        dic[k]=[]
    for y in range(len(data)):
        key=data[y][col]
        if delete:
            del data[y][col]
        dic[key].append(data[y])
    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):
    attValues,dic=subtables(data,col,delete=False)
    total_entropy=entropy([row[-1] for row in data])
    for x in range(len(attValues)):
        ratio=len(dic[attValues[x]])/(len(data)*1.0)
        entro=entropy([row[-1] for row in dic[attValues[x]]])
        total_entropy-=ratio*entro
    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=[compute_gain(data,col) for col in range(n)]
    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.answer)
    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("id3.csv")
node = build_tree(dataset, features) # Build decision tree
print ("The decision tree for the dataset using ID3 algorithm is ")
print_tree(node, 0)
testdata, features =load_csv("id3_test_1.csv")
for xtest in testdata:
    print("The test instance xtest")
    print("The predicted labe1 ", end="" )
    classify(node, xtest,features)
```

Output:

The decision tree for the dataset using ID3 algorithm is

```
 overcast
   yes
 sunny

 normal
     yes
 high
     no
```

rain

weak
  yes
strong
  no
The test instance xtest
The predicted labe1 no
The test instance xtest
The predicted labe1 yes

Program 5

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

Algorithm:

BACKPROPAGATION (*training_example, η, n_{in}, n_{out}, n_{hidden}*)

Each training example is a pair of the form $(\vec{x}, \vec{t})$, where $(\vec{x})$ is the vector of network input values, $(\vec{t})$ and is the vector of target network output values.
η is the learning rate (e.g., .05). n_i, is the number of network inputs, n_{hidden} the number of units in the hidden layer, and n_{out} the number of output units.
The input from unit i into unit j is denoted x_{ji}, and the weight from unit i to unit j is denoted w_{ji}

- 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 $(\vec{x}, \vec{t})$, in training examples, Do

      Propagate the input forward through the network:
      1. Input the instance $\vec{x}$, 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 output unit k, calculate its error term δ_k

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

3. For each hidden 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

$$\Delta w_{ji} = \eta \delta_j x_{i,j}$$

Program:

```
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)
y=y/100

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

def sigmoid_grad(x):
   return x*(1-x)
epoch=1000
eta=0.2
input_neurons=2
hidden_neurons=3
output_neurons=1
wh=np.random.uniform(size=(input_neurons,hidden_neurons))
bh=np.random.uniform(size=(1,hidden_neurons))
wout=np.random.uniform(size=(hidden_neurons,output_neurons))
bout=np.random.uniform(size=(1,output_neurons))
for i in range(epoch):
   h_ip=np.dot(X,wh)+bh
   h_act=sigmoid(h_ip)
   o_ip=np.dot(h_act,wout)+bout
   output=sigmoid(o_ip)
   Eo=y-output
```

```
    outgrad=sigmoid_grad(output)
    d_output=Eo*outgrad
    Eh=d_output.dot(wout.T)
    hiddengrad=sigmoid_grad(h_act)
    d_hidden=Eh*hiddengrad
    wout+=h_act.T.dot(d_output)*eta
    why=X.T.dot(d_hidden)*eta
print("Normalized input:\n"+str(X))
print("Actual output:\n"+str(y))
print("Predicted Output:\n",output)
```

Output:

```
Normalized input:
[[2. 9.]
 [1. 5.]
 [3. 6.]]
Actual output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.89224518]
 [0.88542877]
 [0.89224316]]
```

Program 6

Write a program to implement the navier Bayesian classifier for a sample training data set stored as a .csv file. Compute the accuracy of the classifier,considering few test data sets.

Explanation:

Bayes' Theorem is stated as:

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

Where,

P(h|D) is the probability of hypothesis h given the data D. This is called the **posterior probability**.

P(D|h) is the probability of data d given that the hypothesis h was true.

P(h) is the probability of hypothesis h being true. This is called the **prior probability of h.**
P(D) is the probability of the data. This is called the **prior probability of D**

After calculating the posterior probability for a number of different hypotheses h, and is interested in finding the most probable hypothesis h ∈ H given the observed data D. Any such maximally probable hypothesis is called a ***maximum a posteriori (MAP) hypothesis***.

Bayes theorem to calculate the posterior probability of each candidate hypothesis is ***hMAP*** is a MAP hypothesis provided.

$$h_{MAP} = \arg\max_{h \in H} P(h|D)$$

$$= \arg\max_{h \in H} \frac{P(D|h)P(h)}{P(D)}$$

$$= \arg\max_{h \in H} P(D|h)P(h)$$

(Ignoring P(D) since it is a constant)

Gaussian Naive Bayes

A Gaussian Naive Bayes algorithm is a special type of Naïve Bayes algorithm. It's specifically used when the features have continuous values. It's also assumed that all the features are following a Gaussian distribution i.e., normal distribution

Representation for Gaussian Naive Bayes

We calculate the probabilities for input values for each class using a frequency. With real-valued inputs, we can calculate the mean and standard deviation of input values (x) for each class to summarize the distribution.

This means that in addition to the probabilities for each class, we must also store the mean and standard deviations for each input variable for each class.

Gaussian Naive Bayes Model from Data

The probability density function for the normal distribution is defined by two parameters (mean and standard deviation) and calculating the mean and standard deviation values of each input variable (x) for each class value.

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$ Mean

$$\sigma = \left[\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu)^2\right]^{0.5}$$ Standard deviation

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$ Normal distribution

Files:

naviedata.csv:

6	148	72	35	0	33.6	0.627	50	1
1	85	66	29	0	26.6	0.351	31	0
8	183	64	0	0	23.3	0.672	32	1
1	89	66	23	94	28.1	0.167	21	0
0	137	40	35	168	43.1	2.288	33	1
…	…	…	…	…	…	…	…	…

Program:

```
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 = {} #dictionary of classes 1 and 0
#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): #creates a dictionary of classes
        summaries = [(mean(attribute), stdev(attribute)) for attribute in zip(*dataset)];
        del summaries[-1] #excluding labels +ve or -ve
        return summaries

def summarizebyclass(dataset):
        separated = separatebyclass(dataset);
   #print(separated)
        summaries = {}
        for classvalue, instances in separated.items():
#for key,value in dic.items()
#summaries is a dic of tuples(mean,std) for each class value
            summaries[classvalue] = summarize(instances) #summarize is used to cal to
mean and std
        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 = {} # probabilities contains the all prob of all class of test data
	for classvalue, classsummaries in summaries.items():#class and attribute information as mean and sd
		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): #training and test data is passed
	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 = 'naivedata.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);
    #print(summaries)
  # test model
    predictions = getpredictions(summaries, testset) #find the predictions of test data with
the training data
    accuracy = getaccuracy(testset, predictions)
    print('Accuracy of the classifier is : {0}%'.format(accuracy))
main()
```

Output:

Split 768 rows into train=514 and test=254 rows

Accuracy of the classifier is : 73.62204724409449%

Program 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

Program:

```
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

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=(14,7))
colormap=np.array(['red','lime','black'])
plt.subplot(1,3,1)
plt.scatter(X.Petal_Length,X.Petal_Width,c=colormap[y.Targets],s=40)
plt.title('Real Clasification')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
```

```
plt.subplot(1,3,2)
plt.scatter(X.Petal_Length,X.Petal_Width,c=colormap[model.labels_],s=40)
plt.title('K Mean Classification')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
print('The accuracy score of K-Mean:',sm.accuracy_score(y,model.labels_))
print('The Confusion matrix of K-Mean:',sm.confusion_matrix(y,model.labels_))

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)

y_gmm=gmm.predict(xs)

plt.subplot(1,3,3)
plt.scatter(X.Petal_Length,X.Petal_Width,c=colormap[y_gmm],s=40)
plt.title('GMM Classification')
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')

print('The accuracy score of EM:',sm.accuracy_score(y,y_gmm)*100)
print('The confusion matrix of EM:',sm.confusion_matrix(y,y_gmm))
```

Output:

The accuracy score of K-Mean: 0.24

The Confusion matrix of K-Mean: [[0 50 0]

[2 0 48]

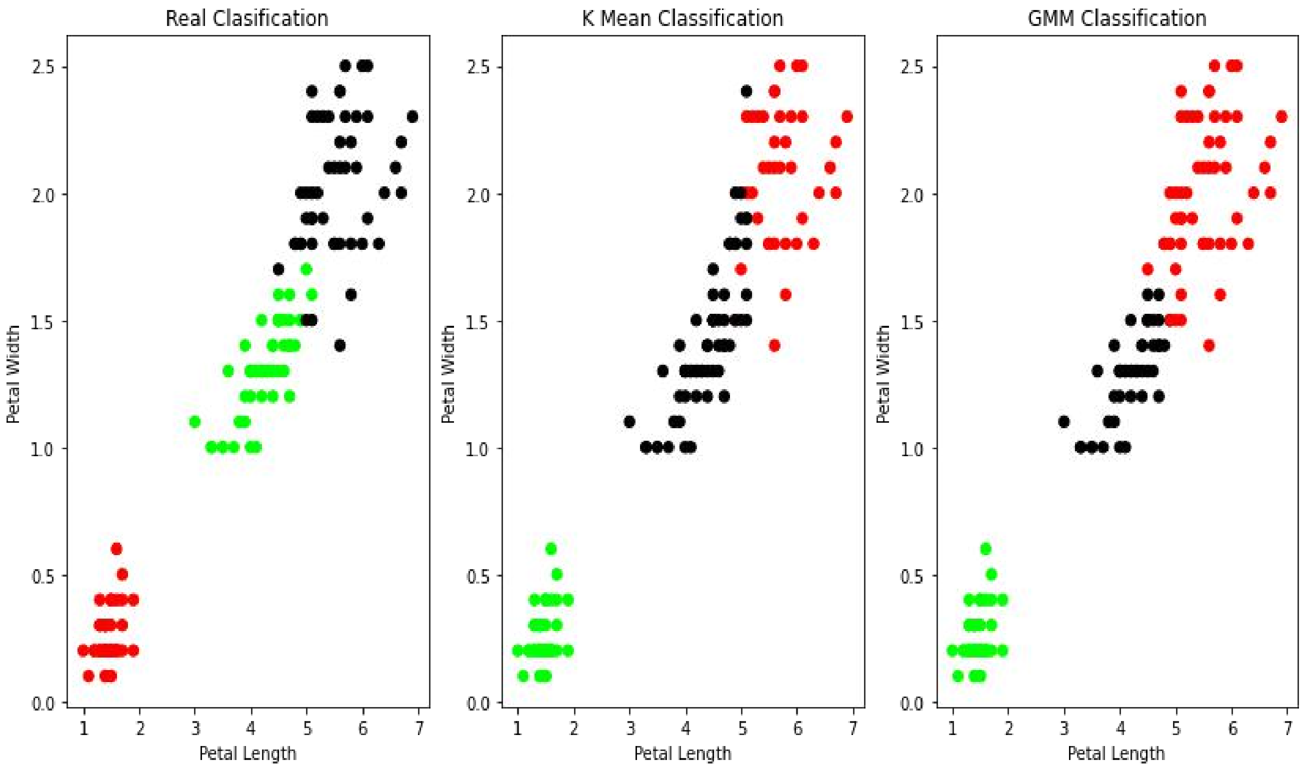
[36 0 14]]

The accuracy score of EM:30.0

The confusion matrix of EM: [[50 0 0]

[0 5 45]

[0 50 0]]



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.

Algorithm:

K-Nearest Neighbor Algorithm

Training algorithm:

- For each training example (x, f (x)), add the example to the list training examples

Classification algorithm:

- Given a query instance x_q to be classified,
    - Let $x_1 \ldots x_k$ denote the k instances from training examples that are nearest to x_q
    - Return

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} f(x_i)}{k}$$

- Where, $f(x_i)$ function to calculate the mean value of the k nearest training examples.

Program:

```
from sklearn.datasets import load_iris

iris=load_iris()

print("Features Names:",iris.feature_names,"Iris Data:",iris.data,"Target
Names:",iris.target_names,"Target:",iris.target)

from sklearn.model_selection import train_test_split

X_train,X_test,y_train,y_test=train_test_split(iris.data,iris.target,test_size=.25)


from sklearn.neighbors import KNeighborsClassifier

clf=KNeighborsClassifier()
```

```
clf.fit(X_train,y_train)
print("Accuracy=",clf.score(X_test,y_test))
print("Predicted Data")
print(clf.predict(X_test))
prediction=clf.predict(X_test)
print("Test Data:")
print(y_test)
diff=prediction-y_test
print("Result is")
print(diff)
print("Total no of samples misclassified=",sum(abs(diff)))
```

Output:

```
Features Names: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
Iris Data: [[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]
 [5.4 3.9 1.7 0.4]
 [4.6 3.4 1.4 0.3]
 [5.  3.4 1.5 0.2]
 [4.4 2.9 1.4 0.2]
 [4.9 3.1 1.5 0.1]
 [5.4 3.7 1.5 0.2]
 [4.8 3.4 1.6 0.2]
 [4.8 3.  1.4 0.1]
 [4.3 3.  1.1 0.1]
 [5.8 4.  1.2 0.2]
 [5.7 4.4 1.5 0.4]
 [5.4 3.9 1.3 0.4]
 [5.1 3.5 1.4 0.3]
```

[5.7 3.8 1.7 0.3]

[5.1 3.8 1.5 0.3]

[5.4 3.4 1.7 0.2]

[5.1 3.7 1.5 0.4]

[4.6 3.6 1. 0.2]

[5.1 3.3 1.7 0.5]

[4.8 3.4 1.9 0.2]

[5. 3. 1.6 0.2]

[5. 3.4 1.6 0.4]

[5.2 3.5 1.5 0.2]

[5.2 3.4 1.4 0.2]

[4.7 3.2 1.6 0.2]

[4.8 3.1 1.6 0.2]

[5.4 3.4 1.5 0.4]

[5.2 4.1 1.5 0.1]

[5.5 4.2 1.4 0.2]

[4.9 3.1 1.5 0.2]

[5. 3.2 1.2 0.2]

[5.5 3.5 1.3 0.2]

[4.9 3.6 1.4 0.1]

[4.4 3. 1.3 0.2]

[5.1 3.4 1.5 0.2]

[5. 3.5 1.3 0.3]

[4.5 2.3 1.3 0.3]

[4.4 3.2 1.3 0.2]

[5. 3.5 1.6 0.6]

[5.1 3.8 1.9 0.4]

[4.8 3. 1.4 0.3]

[5.1 3.8 1.6 0.2]

[4.6 3.2 1.4 0.2]

```
[5.3 3.7 1.5 0.2]
[5.  3.3 1.4 0.2]
[7.  3.2 4.7 1.4]
[6.4 3.2 4.5 1.5]
[6.9 3.1 4.9 1.5]
[5.5 2.3 4.  1.3]
[6.5 2.8 4.6 1.5]
[5.7 2.8 4.5 1.3]
[6.3 3.3 4.7 1.6]
[4.9 2.4 3.3 1. ]
[6.6 2.9 4.6 1.3]
[5.2 2.7 3.9 1.4]
[5.  2.  3.5 1. ]
[5.9 3.  4.2 1.5]
[6.  2.2 4.  1. ]
[6.1 2.9 4.7 1.4]
[5.6 2.9 3.6 1.3]
[6.7 3.1 4.4 1.4]
[5.6 3.  4.5 1.5]
[5.8 2.7 4.1 1. ]
[6.2 2.2 4.5 1.5]
[5.6 2.5 3.9 1.1]
[5.9 3.2 4.8 1.8]
[6.1 2.8 4.  1.3]
[6.3 2.5 4.9 1.5]
[6.1 2.8 4.7 1.2]
[6.4 2.9 4.3 1.3]
[6.6 3.  4.4 1.4]
[6.8 2.8 4.8 1.4]
[6.7 3.  5.  1.7]
```

[6. 2.9 4.5 1.5]

[5.7 2.6 3.5 1.]

[5.5 2.4 3.8 1.1]

[5.5 2.4 3.7 1.]

[5.8 2.7 3.9 1.2]

[6. 2.7 5.1 1.6]

[5.4 3. 4.5 1.5]

[6. 3.4 4.5 1.6]

[6.7 3.1 4.7 1.5]

[6.3 2.3 4.4 1.3]

[5.6 3. 4.1 1.3]

[5.5 2.5 4. 1.3]

[5.5 2.6 4.4 1.2]

[6.1 3. 4.6 1.4]

[5.8 2.6 4. 1.2]

[5. 2.3 3.3 1.]

[5.6 2.7 4.2 1.3]

[5.7 3. 4.2 1.2]

[5.7 2.9 4.2 1.3]

[6.2 2.9 4.3 1.3]

[5.1 2.5 3. 1.1]

[5.7 2.8 4.1 1.3]

[6.3 3.3 6. 2.5]

[5.8 2.7 5.1 1.9]

[7.1 3. 5.9 2.1]

[6.3 2.9 5.6 1.8]

[6.5 3. 5.8 2.2]

[7.6 3. 6.6 2.1]

[4.9 2.5 4.5 1.7]

[7.3 2.9 6.3 1.8]

[6.7 2.5 5.8 1.8]

[7.2 3.6 6.1 2.5]

[6.5 3.2 5.1 2.]

[6.4 2.7 5.3 1.9]

[6.8 3. 5.5 2.1]

[5.7 2.5 5. 2.]

[5.8 2.8 5.1 2.4]

[6.4 3.2 5.3 2.3]

[6.5 3. 5.5 1.8]

[7.7 3.8 6.7 2.2]

[7.7 2.6 6.9 2.3]

[6. 2.2 5. 1.5]

[6.9 3.2 5.7 2.3]

[5.6 2.8 4.9 2.]

[7.7 2.8 6.7 2.]

[6.3 2.7 4.9 1.8]

[6.7 3.3 5.7 2.1]

[7.2 3.2 6. 1.8]

[6.2 2.8 4.8 1.8]

[6.1 3. 4.9 1.8]

[6.4 2.8 5.6 2.1]

[7.2 3. 5.8 1.6]

[7.4 2.8 6.1 1.9]

[7.9 3.8 6.4 2.]

[6.4 2.8 5.6 2.2]

[6.3 2.8 5.1 1.5]

[6.1 2.6 5.6 1.4]

[7.7 3. 6.1 2.3]

[6.3 3.4 5.6 2.4]

[6.4 3.1 5.5 1.8]

[6. 3. 4.8 1.8]

[6.9 3.1 5.4 2.1]

[6.7 3.1 5.6 2.4]

[6.9 3.1 5.1 2.3]

[5.8 2.7 5.1 1.9]

[6.8 3.2 5.9 2.3]

[6.7 3.3 5.7 2.5]

[6.7 3. 5.2 2.3]

[6.3 2.5 5. 1.9]

[6.5 3. 5.2 2.]

[6.2 3.4 5.4 2.3]

[5.9 3. 5.1 1.8]] Target Names: ['setosa' 'versicolor' 'virginica'] Target: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2

2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2

2 2]

Accuracy= 1.0

Predicted Data

[2 2 1 2 2 1 2 0 0 2 0 0 2 2 0 2 1 2 1 1 0 0 2 2 1 2 2 0 1 0 2 1 2 1 1 2 0

1]

Test Data:

[2 2 1 2 2 1 2 0 0 2 0 0 2 2 0 2 1 2 1 1 0 0 2 2 1 2 2 0 1 0 2 1 2 1 1 2 0

1]

Result is

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0]

Total no of samples misclassified= 0

Program 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 Regression 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 :

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

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

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

5. Prediction = x0*β

CSV Files:

tips.csv

total_bill	tip	sex	smoker	day	time	size
16.99	1.01	Female	No	Sun	Dinner	2
10.34	1.66	Male	No	Sun	Dinner	3
21.01	3.5	Male	No	Sun	Dinner	3
23.68	3.31	Male	No	Sun	Dinner	2
…	…	…	…	…	…	…

Program:

```
import matplotlib.pyplot as plt

import pandas as pd
```

```
import numpy as np

def kernel(point, xmat, k):
    m,n = np.shape(xmat)
    weights = np.mat(np.eye((m)))
    for j in range(m):
        diff = point - X[j]
        weights[j,j] = np.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 = np.shape(xmat)
    ypred = np.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('tips.csv')
bill = np.array(data.total_bill)
tip = np.array(data.tip)
#preparing and add 1 in bill
mbill = np.mat(bill)
mtip = np.mat(tip)
m= np.shape(mbill)[1]
```

```
one = np.mat(np.ones(m))
X = np.hstack((one.T,mbill.T))

#set k here
ypred = localWeightRegression(X,mtip,0.5)
SortIndex = X[:,1].argsort(0)
xsort = X[SortIndex][:,0]
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.scatter(bill,tip, color='green')
ax.plot(xsort[:,1],ypred[SortIndex], color = 'red', linewidth=5)
plt.xlabel('Total bill')
plt.ylabel('Tip')
plt.show();
```

Output: