**Program 1**

**Implement A\* Algorithm**

**Find the most cost effective path to read from start state A to final state J using A\* Algorithm.**

**Step 1: Place the starting node into OPEN and find its f (n) value.**

**Step 2: Remove the node from OPEN, having smallest f (n) value. If it is a goal node**

**then stop and return success.**

**Step 3: Else remove the node from OPEN, find all its successors.**

**Step 4: Find the f (n) value of all successors; place them into OPEN and place the removed node into CLOSE.**

**Step 5: Go to Step-2.**

**Step 6: Exit.**

****

****

#!/usr/bin/env python

# coding: utf-8

# In[1]:

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')

**Output**

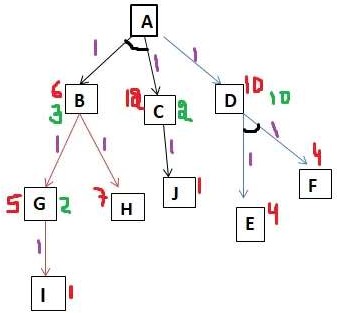
Path found: ['A', 'F', 'G', 'I', 'J']

**Program 2**

**Implement AO\* Algorithm**

**Algorithm**

1. It is an informed search and works as Best First Search.
2. AO\* algorithm is based on problem decomposition.
3. It represents an AND-OR graph algorithm that is used to find more than one solution.
4. It is an efficient method to explore a solution path.



# 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

**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']}

------------------------------------------------------------

HEURISTIC VALUES : {'A': 1, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {}

PROCESSING NODE : A

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {}

PROCESSING NODE : D

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {}

PROCESSING NODE : A

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {}

PROCESSING NODE : E

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 0, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {'E': []}

PROCESSING NODE : D

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 6, 'E': 0, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {'E': []}

PROCESSING NODE : A

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 7, 'B': 6, 'C': 12, 'D': 6, 'E': 0, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {'E': []}

PROCESSING NODE : F

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 7, 'B': 6, 'C': 12, 'D': 6, 'E': 0, 'F': 0, 'G': 5, 'H': 7}

SOLUTION GRAPH : {'E': [], 'F': []}

PROCESSING NODE : D

-----------------------------------------------------------------------------------------

HEURISTIC VALUES : {'A': 7, 'B': 6, 'C': 12, 'D': 2, 'E': 0, 'F': 0, 'G': 5, 'H': 7}

SOLUTION GRAPH : {'E': [], 'F': [], 'D': ['E', 'F']}

PROCESSING NODE : A

-----------------------------------------------------------------------------------------

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

------------------------------------------------------------

{'E': [], 'F': [], 'D': ['E', 'F'], 'A': ['D']}

------------------------------------------------------------

​

**Program 3**

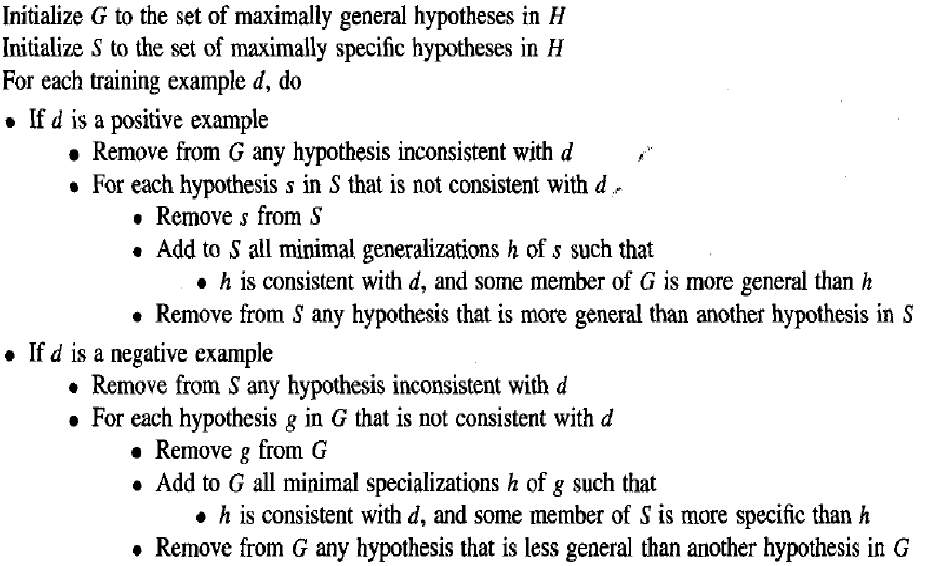
**For a given set of training data examples stored in a .CSV file, implement and demonstrate the Candidate-Elimination algorithm to output a description of the set of all hypotheses consistent with the training examples.**

**Task: The CANDIDATE-ELIMINATION algorithm computes the version space containing all hypotheses from H that are consistent with an observed sequence of training examples.**

**Dataset: Enjoy Sports Training Examples:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Example** | **Sky** | **AirTemp** | **Humidity** | **Wind** | **Water** | **Forecast** | **EnjoySport** |
| 1 | Sunny | Warm | Normal | Strong | Warm | Same | Yes |
| 2 | Sunny | Warm | High | Strong | Warm | Same | Yes |
| 3 | Rainy | Cold | High | Strong | Warm | Change | No |
| 4 | Sunny | Warm | High | Strong | Cool | Change | Yes |

**Candidate Elimination Algorithm:**

****

**Candidate Elimination Program:**

import numpy as np

import pandas as pd

# Loading Data from a CSV File

data = pd.DataFrame(data=pd.read\_csv('finds.csv'))

# Separating concept features from Target

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

# Isolating target into a separate DataFrame

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

def learn(concepts, target):

# learn() function implements the learning method of the Candidate elimination algorithm.

#Arguments:

#concepts - a data frame with all the features ,target - a data frame with corresponding output values

# Initialise S0 with the first instance from concepts

# .copy() makes sure a new list is created instead of just pointing to the same memory location

specific\_h = concepts[0].copy()

general\_h = [["?" for i in range(len(specific\_h))] for i in range(len(specific\_h))]

# The learning iterations

for i, h in enumerate(concepts):

# Checking if the hypothesis has a positive target

if target[i] == "Yes":

for x in range(len(specific\_h)):

# Change values in S & G only if values change

if h[x] != specific\_h[x]:

specific\_h[x] = '?'

general\_h[x][x] = '?'

# Checking if the hypothesis has a positive target

if target[i] == "No":

for x in range(len(specific\_h)):

# For negative hyposthesis change values only in G

if h[x] != specific\_h[x]:

general\_h[x][x] = specific\_h[x]

else:

general\_h[x][x] = '?'

# find indices where we have empty rows, meaning those that are unchanged

indices = [i for i,val in enumerate(general\_h) if val == ['?', '?', '?', '?', '?', '?']]

for i in indices:

# remove those rows from general\_h

general\_h.remove(['?', '?', '?', '?', '?', '?'])

# Return final values

return specific\_h, general\_h

s\_final, g\_final = learn(concepts, target)

print("Final S:", s\_final, sep="\n")

print("Final G:", g\_final, sep="\n")

**out put**

Final S:

['Sunny' 'Warm' '?' 'Strong' '?' '?']

Final G:

[['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.**

**Task: ID3 determines the information gain for each candidate attribute, then selects the one with highest information gain as the root node of the tree. The information gain values for all four attributes are calculated using the following formula:**

**Entropy(S)=∑- P(I).log2P(I)**

**Gain(S,A)=Entropy(S)-∑[P(S/A).Entropy(S/A)]**

**Dataset: pima-indians-diabetes.csv**

**ID3 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*

o Otherwise Begin

* A ← the attribute from *Attributes* that best\* classifies *Examples*

## The decision attribute for *Root* ←A

* For each possible value, υi, of A,
* Add a new tree branch below *Root*, corresponding to the test A = υi
* Let *Examplesυi* ,be the subset of *Examples* that have value υi for A
* If *Examplesυi*, 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(*Examplesυi,Target\_attribute, Attributes–{A}*))

End

Return Root

**ID3 Program:**

import csv

import math

import random

# Majority Function which tells which class has more entries in given data-set

def majorClass(attributes, data, target):

freq = {}

index = attributes.index(target)

for tuple in data:

if tuple[index] in freq:

freq[tuple[index]] += 1

else:

freq[tuple[index]] = 1

max = 0

major = ""

for key in freq.keys():

if freq[key]>max:

max = freq[key]

major = key

return major

# Calculates the entropy of the data given the target attribute

def entropy(attributes, data, targetAttr):

freq = {}

dataEntropy = 0.0

i = 0

for entry in attributes:

if (targetAttr == entry):

break

i = i + 1

i = i - 1

for entry in data:

if entry[i] in freq:

freq[entry[i]] += 1.0

else:

freq[entry[i]] = 1.0

for freq in freq.values():

dataEntropy += (-freq/len(data)) \* math.log(freq/len(data), 2)

return dataEntropy

# Calculates the information gain (reduction in entropy) in the data when a particular attribute is chosen for splitting the data.

def info\_gain(attributes, data, attr, targetAttr):

freq = {}

subsetEntropy = 0.0

i = attributes.index(attr)

for entry in data:

if entry[i] in freq:

freq[entry[i]] += 1.0

else:

freq[entry[i]] = 1.0

for val in freq.keys():

valProb = freq[val] / sum(freq.values())

dataSubset = [entry for entry in data if entry[i] == val]

subsetEntropy += valProb \* entropy(attributes, dataSubset, targetAttr)

return (entropy(attributes, data, targetAttr) - subsetEntropy)

# This function chooses the attribute among the remaining attributes which has the maximum information gain.

def attr\_choose(data, attributes, target):

best = attributes[0]

maxGain = 0;

for attr in attributes:

newGain = info\_gain(attributes, data, attr, target)

if newGain>maxGain:

maxGain = newGain

best = attr

return best

# This function will get unique values for that particular attribute from the given data

def get\_values(data, attributes, attr):

index = attributes.index(attr)

values = []

for entry in data:

if entry[index] not in values:

values.append(entry[index])

return values

# This function will get all the rows of the data where the chosen "best" attribute has a value "val"

def get\_data(data, attributes, best, val):

new\_data = [[]]

index = attributes.index(best)

for entry in data:

if (entry[index] == val):

newEntry = []

for i in range(0,len(entry)):

if(i != index):

newEntry.append(entry[i])

new\_data.append(newEntry)

new\_data.remove([])

return new\_data

# This function is used to build the decision tree using the given data, attributes and the target attributes. It returns the decision tree in the end.

def build\_tree(data, attributes, target):

data = data[:]

vals = [record[attributes.index(target)] for record in data]

default = majorClass(attributes, data, target)

if not data or (len(attributes) - 1) <= 0:

return default

elif vals.count(vals[0]) == len(vals):

return vals[0]

else:

best = attr\_choose(data, attributes, target)

tree = {best:{}}

for val in get\_values(data, attributes, best):

new\_data = get\_data(data, attributes, best, val)

newAttr = attributes[:]

newAttr.remove(best)

subtree = build\_tree(new\_data, newAttr, target)

tree[best][val] = subtree

return tree

#Main function

def execute\_decision\_tree():

data = []

#load file

with open("weather.csv") as tsv:

for line in csv.reader(tsv):

data.append(tuple(line))

print("Number of records:",len(data))

#set attributes

attributes=['outlook','temperature','humidity','wind','play']

target = attributes[-1]

#set training data

acc = []

training\_set = [x for i, x in enumerate(data)]

tree = build\_tree( training\_set, attributes, target )

print(tree)

#execute algorithm on test data

results = []

test\_set = [('rainy','mild','high','strong')]

for entry in test\_set:

tempDict = tree.copy()

result = ""

while(isinstance(tempDict, dict)):

child=[]

nodeVal=next(iter(tempDict))

child=tempDict[next(iter(tempDict))].keys()

tempDict = tempDict[next(iter(tempDict))]

index = attributes.index(nodeVal)

value = entry[index]

if(value in tempDict.keys()):

result = tempDict[value]

tempDict = tempDict[value]

else:

result = "Null"

break

if result != "Null":

results.append(result == entry[-1])

print(result)

if \_\_name\_\_ == "\_\_main\_\_":

execute\_decision\_tree()

**Input:**

Weather.csv

**Output:**

Number of records: 15

{'wind': {'wind': 'play', 'weak': {'humidity': {'high': {'temperature': {'hot': {'outlook': {'sunny': 'no', 'overcast': 'yes'}}, 'mild': {'outlook': {'rainy': 'yes', 'sunny': 'no'}}}}, 'normal': 'yes'}}, 'strong': {'humidity': {'high': {'outlook': {'sunny': 'no', 'overcast': 'yes', 'rainy': 'no'}}, 'normal': {'outlook': {'rainy': 'no', 'overcast': 'yes', 'sunny': 'yes'}}}}}}

no

**Program 5:**

**Build an Artificial Neural Network by implementing the Back propagation algorithm and test the same using appropriate data set**

The stochastic gradient descent version of the BACKPROPAGATION algorithm for feed forward networks containing two layers of sigmoid units.

**Step 1**: begins by constructing a network with the desired number of hidden and output units and initializing all network weights to small random values. . For each training example, it applies the network to the example, calculates the error of the network output for this example, computes the gradient with respect to the error on this example, then updates all weights in the network. This gradient descent step is iterated (often thousands of times, using the same training examples multiple times) until the network performs acceptably well.

**Step 2:** The gradient descent weight-update rule is similar to the delta training rule The only difference is that the error (t - o) in the delta rule is replaced by a more complex error term aj.

**Step 3:** updates weights incrementally, following the Presentation of each training example. This corresponds to a stochastic approximation to gradient descent. To obtain the true gradient of E one would sum the Sj, xji values over all training examples before altering weight values.

**Step 4:** The weight-update loop in BACKPROPAGATION may be iterated thousands of times in a typical application. A variety of termination conditions can be used to halt the procedure.

One may choose to halt after a fixed number of iterations through the loop, or once the error on the training examples falls below some threshold.

**Dataset:**

**ANN Program:**

from math import exp

from random import seed

from random import random

# Initialize a network

def initialize\_network(n\_inputs, n\_hidden, n\_outputs):

network = list()

hidden\_layer = [{'weights':[random() for i in range(n\_inputs + 1)]} for i in range(n\_hidden)]

network.append(hidden\_layer)

output\_layer = [{'weights':[random() for i in range(n\_hidden + 1)]} for i in range(n\_outputs)]

network.append(output\_layer)

return network

# Calculate neuron activation for an input

def activate(weights, inputs):

activation = weights[-1]

for i in range(len(weights)-1):

activation += weights[i] \* inputs[i]

return activation

# Transfer neuron activation

def transfer(activation):

return 1.0 / (1.0 + exp(-activation))

# Forward propagate input to a network output

def forward\_propagate(network, row):

inputs = row

for layer in network:

new\_inputs = []

for neuron in layer:

activation = activate(neuron['weights'], inputs)

neuron['output'] = transfer(activation)

new\_inputs.append(neuron['output'])

inputs = new\_inputs

return inputs

# Calculate the derivative of an neuron output

def transfer\_derivative(output):

return output \* (1.0 - output)

# Backpropagate error and store in neurons

def backward\_propagate\_error(network, expected):

for i in reversed(range(len(network))):

layer = network[i]

errors = list()

if i != len(network)-1:

for j in range(len(layer)):

error = 0.0

for neuron in network[i + 1]:

error += (neuron['weights'][j] \* neuron['delta'])

errors.append(error)

else:

for j in range(len(layer)):

neuron = layer[j]

errors.append(expected[j] - neuron['output'])

for j in range(len(layer)):

neuron = layer[j]

neuron['delta'] = errors[j] \* transfer\_derivative(neuron['output'])

# Update network weights with error

def update\_weights(network, row, l\_rate):

for i in range(len(network)):

inputs = row[:-1]

if i != 0:

inputs = [neuron['output'] for neuron in network[i - 1]]

for neuron in network[i]:

for j in range(len(inputs)):

neuron['weights'][j] += l\_rate \* neuron['delta'] \* inputs[j]

neuron['weights'][-1] += l\_rate \* neuron['delta']

# Train a network for a fixed number of epochs

def train\_network(network, train, l\_rate, n\_epoch, n\_outputs):

for epoch in range(n\_epoch):

sum\_error = 0

for row in train:

outputs = forward\_propagate(network, row)

expected = [0 for i in range(n\_outputs)]

expected[row[-1]] = 1

sum\_error += sum([(expected[i]-outputs[i])\*\*2 for i in range(len(expected))])

backward\_propagate\_error(network, expected)

update\_weights(network, row, l\_rate)

print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l\_rate, sum\_error))

# Test training backprop algorithm

seed(1)

dataset = [[2.7810836,2.550537003,0],

[1.465489372,2.362125076,0],

[3.396561688,4.400293529,0],

[1.38807019,1.850220317,0],

[3.06407232,3.005305973,0],

[7.627531214,2.759262235,1],

[5.332441248,2.088626775,1],

[6.922596716,1.77106367,1],

[8.675418651,-0.242068655,1],

[7.673756466,3.508563011,1]]

n\_inputs = len(dataset[0]) - 1

n\_outputs = len(set([row[-1] for row in dataset]))

network = initialize\_network(n\_inputs, 2, n\_outputs)

print(network)

train\_network(network, dataset, 0.5, 20, n\_outputs)

for layer in network:

print(layer)

**Output**

>epoch=0, lrate=0.500, error=6.350

>epoch=1, lrate=0.500, error=5.531

>epoch=2, lrate=0.500, error=5.22147

>epoch=3, lrate=0.500, error=4.951

>epoch=4, lrate=0.500, error=4.519

>epoch=5, lrate=0.500, error=4.173

>epoch=6, lrate=0.500, error=3.835

>epoch=7, lrate=0.500, error=3.506

>epoch=8, lrate=0.500, error=3.192

>epoch=9, lrate=0.500, error=2.898

>epoch=10, lrate=0.500, error=2.626

>epoch=11, lrate=0.500, error=2.377

>epoch=12, lrate=0.500, error=2.153

>epoch=13, lrate=0.500, error=1.953

>epoch=14, lrate=0.500, error=1.774

>epoch=15, lrate=0.500, error=1.614

>epoch=16, lrate=0.500, error=1.472

>epoch=17, lrate=0.500, error=1.346

>epoch=18, lrate=0.500, error=1.233

>epoch=19, lrate=0.500, error=1.132

[{'weights': [-1.4688375095432327, 1.850887325439514, 1.0858178629550297], 'output':

0.029980305604426185, 'delta': -0.0059546604162323625}, {'weights': [0.37711098142462157, -0.0625909894552989, 0.2765123702642716], 'output': 0.9456229000211323, 'delta':

0.0026279652850863837}]

[{'weights': [2.515394649397849, -0.3391927502445985, -0.9671565426390275], 'output':

0.23648794202357587, 'delta': -0.04270059278364587}, {'weights': [-2.5584149848484263,

1.0036422106209202, 0.42383086467582715], 'output': 0.7790535202438367, 'delta':

0.03803132596437354}]

**Program 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.**

**Task**: It is a classification technique based on [Bayes’ Theorem](https://en.wikipedia.org/wiki/Bayes%27_theorem) with an assumption of independence among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. For example, a fruit may be considered to be an apple if it is red, round, and about 3 inches in diameter. Even if these features depend on each other or upon the existence of the other features, all of these properties independently contribute to the probability that this fruit is an apple and that is why it is known as ‘Naive’.

Dataset : Pima-indians-diabetes.csv

It is a classification technique based on Bayes‟ Theorem with an assumption of independence among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. For example, a fruit may be considered to be an apple if it is red, round, and about 3 inches in diameter. Even if these features depend on each other or upon the existence of the other features, all of these properties independently contribute to the probability that this fruit is an apple and that is why it is known as „Naive‟.

Bayes theorem provides a way of calculating posterior probability P(c|x) from P(c), P(x) and P(x|c). Look at the equation below:

**1) Handling Of Data:**

* Load the data from the CSV file and split in to training and test data set.
* Training data set can be used to by Naïve Bayes to make predictions.
* And Test data set can be used to evaluate the accuracy of the model.

**2) Summarize Data:**

The summary of the training data collected involves the mean and the standard deviation for each attribute, by class value.

* + - These are required when making predictions to calculate the probability of specific attribute values belonging to each class value.
    - Summary data can be break down into the following sub-tasks:
* **Separate Data By Class**: The first task is to separate the training dataset instances by class value so that we can calculate statistics for each class. We can do that by creating a map of each class value to a list of instances that belong to that class and sort the entire dataset of instances into the appropriate lists.
* **Calculate Mean**:We need to calculate the mean of each attribute for a class value. 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.
* **Calculate Standard Deviation**: We also need to calculate the standard deviation of each attribute for a class value. 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.
* **Summarize Dataset**: For a given list of instances (for a class value) we can calculate the mean and the standard deviation for each attribute.
* 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.
* **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.

**3) Make Predictions:**

* Making predictions involves calculating the probability that a given data instance belongs to each class,
* then selecting the class with the largest probability as the prediction.
* Finally, estimation of the accuracy of the model by making predictions for each data instance in the test dataset.

**4) Evaluate Accuracy**: The predictions can be compared to the class values in the test dataset and a classification\ accuracy can be calculated as an accuracy ratio between 0& and 100%.

**Naïve Bayes Program:**

import csv

import random

import math

def safe\_div(x,y):

if y == 0:

return 0

return x / y

def loadCsv(filename):

lines = csv.reader(open(filename, "r"))

dataset = list(lines)

for i in range(len(dataset)):

dataset[i] = [float(x) for x in dataset[i]]

return dataset

def splitDataset(dataset, splitRatio):

trainSize = int(len(dataset) \* splitRatio)

trainSet = []

copy = list(dataset)

while len(trainSet) < trainSize:

index = random.randrange(len(copy))

trainSet.append(copy.pop(index))

return [trainSet, copy]

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

def mean(numbers):

return sum(numbers)/float(len(numbers))

def stdev(numbers):

avg = mean(numbers)

variance = sum([pow(x-avg,2) for x in numbers])/float(len(numbers)-1)

return math.sqrt(variance)

def summarize(dataset):

summaries = [(mean(attribute), stdev(attribute)) for attribute in zip(\*dataset)]

del summaries[-1]

return summaries

def summarizeByClass(dataset):

separated = separateByClass(dataset)

summaries = {}

for classValue, instances in separated.items():

summaries[classValue] = summarize(instances)

return summaries

def calculateProbability(x, mean, stdev):

exponent = math.exp(-(math.pow(x-mean,2)/(2\*math.pow(stdev,2))))

return (1 / (math.sqrt(2\*math.pi) \* stdev)) \* exponent

def calculateClassProbabilities(summaries, inputVector):

probabilities = {}

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

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

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)):

#print(testSet[i][-1]," ",predictions[i])

if testSet[i][-1] == predictions[i]:

correct += 1

return (correct/float(len(testSet))) \* 100.0

def main():

filename = 'pima-indians-diabetes.data.csv'

splitRatio = 0.67

dataset = loadCsv(filename)

trainingSet,testSet=splitDataset(dataset, splitRatio) #dividing into training and test data

#trainingSet = dataset #passing entire dataset as training data

#testSet=[[8.0,183.0,64.0,0.0,0.0,23.3,0.672,32.0]]

print('Split {0} rows into train={1} and test={2} rows'.format(len(dataset), len(trainingSet), len(testSet)))

# prepare model

summaries = summarizeByClass(trainingSet)

# test model

predictions = getPredictions(summaries, testSet)

accuracy = getAccuracy(testSet, predictions)

print('Accuracy: {0}%'.format(accuracy))

main()

**Input:**

Pima-indians-diabetes.csv

**OUTPUT:**

Split 768 rows into train=576 and test=192 rows

Accuracy: 77.604 %

2257

1502

['alt.atheism', 'comp.graphics', 'sci.med', 'soc.religion.christian']

From: sd345@city.ac.uk (Michael Collier)

Subject: Converting images to HP LaserJet III?

Nntp-Posting-Host: hampton

Organization: The City University

Lines: 14

Does anyone know of a good way (standard PC application/PD utility) to

convert tif/img/tga files into LaserJet III format. We would also like to

do the same, converting to HPGL (HP plotter) files.

Please email any response.

Is this the correct group?

Thanks in advance. Michael.

--

Michael Collier (Programmer) The Computer Unit,

Email: M.P.Collier@uk.ac.city The City University,

Tel: 071 477-8000 x3769 London,

Fax: 071 477-8565 EC1V 0HB.

1

Accuracy: 0.8348868175765646

precision recall f1-score support

alt.atheism 0.97 0.60 0.74 319

comp.graphics 0.96 0.89 0.92 389

sci.med 0.97 0.81 0.88 396

soc.religion.christian 0.65 0.99 0.78 398

avg / total 0.88 0.83 0.84 1502

confusion matrix is

[[192 2 6 119]

[ 2 347 4 36]

[ 2 11 322 61]

[ 2 2 1 393]]

**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.**

**Introduction to Expectation-Maximization (EM)**

The EM algorithm tends to get stuck less than K-means algorithm. The idea is to assign data points partially to different clusters instead of assigning to only one cluster. To do this partial assignment, we model each cluster using a probabilistic distribution So a data point associates with a cluster with certain probability and it belongs to the cluster with the highest probability in the final assignment

**Expectation-Maximization (EM) algorithm**

**Step 1**: An initial guess is made for the model’s parameters and a probability

distribution is created. This is sometimes called the “E-Step” for the

“Expected” distribution.

**Step 2**: Newly observed data is fed into the model.

**Step 3:** The probability distribution from the E-step is drawn to include the new

data. This is sometimes called the “M-step.”

**Step 4:** Steps 2 through 4 are repeated until stability.

**Dataset:**

**EM algorithm Programs:**

import matplotlib.pyplot as plt

from sklearn import datasets

from sklearn.cluster import KMeans

import pandas as pd

import numpy as np

# import some data to play with

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']

# Build the K Means Model

model = KMeans(n\_clusters=3)

model.fit(X) # model.labels\_ : Gives cluster no for which samples belongs to

# # Visualise the clustering results

plt.figure(figsize=(14,14))

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

# Plot the Original Classifications using Petal features

plt.subplot(2, 2, 1)

plt.scatter(X.Petal\_Length, X.Petal\_Width, c=colormap[y.Targets], s=40)

plt.title('Real Clusters')

plt.xlabel('Petal Length')

plt.ylabel('Petal Width')

# Plot the Models Classifications

plt.subplot(2, 2, 2)

plt.scatter(X.Petal\_Length, X.Petal\_Width, c=colormap[model.labels\_], s=40)

plt.title('K-Means Clustering')

plt.xlabel('Petal Length')

plt.ylabel('Petal Width')

# General EM for GMM

from sklearn import preprocessing

# transform your data such that its distribution will have a

# mean value 0 and standard deviation of 1.

scaler = preprocessing.StandardScaler()

scaler.fit(X)

xsa = scaler.transform(X)

xs = pd.DataFrame(xsa, columns = X.columns)

from sklearn.mixture import GaussianMixture

gmm = GaussianMixture(n\_components=3)

gmm.fit(xs)

gmm\_y = gmm.predict(xs)

plt.subplot(2, 2, 3)

plt.scatter(X.Petal\_Length, X.Petal\_Width, c=colormap[gmm\_y], s=40)

plt.title('GMM Clustering')

plt.xlabel('Petal Length')

plt.ylabel('Petal Width')

print('Observation: The GMM using EM algorithm based clustering matched the true labels more closely than the Kmeans.')

**Output**



**Program 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.**

**TASK:** The task of this program is to classify the IRIS data set examples by using the k-Nearest Neighbour algorithm. The new instance has to be classified based on its k nearest neighbors.

**Dataset: iris.csv**

**ALGORITHM**

Let m be the number of training data samples. Let p be an unknown point.

1. Store the training samples in an array of data points arr[]. This means each element of this array represents a tuple (x, y).
2. for i=0 to m:

Calculate Euclidean distance d(arr[i], p).

1. Make set S of K smallest distances obtained. Each of these distances correspond to an already classified data point.
2. Return the majority label among S.

**KNN Program**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

# ### Step 2 : Load the inbuilt data or the csv/excel file into pandas dataframe and clean the data # In[66]:

from sklearn.datasets import load\_iris

data = load\_iris()

df = pd.DataFrame(data.data, columns=data.feature\_names)

df['Class'] = data.target\_names[data.target]

df.head()

x = df.iloc[:, :-1].values

y = df.Class.values

print(x[:5])

print(y[:5])

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x, y, test\_size = 0.2)

from sklearn.neighbors import KNeighborsClassifier

knn\_classifier = KNeighborsClassifier(n\_neighbors=5)

knn\_classifier.fit(x\_train, y\_train)

predictions = knn\_classifier.predict(x\_test)

print(predictions)

from sklearn.metrics import accuracy\_score, confusion\_matrix

print("Training accuracy Score is : ", accuracy\_score(y\_train, knn\_classifier.predict(x\_train)))

print("Testing accuracy Score is : ", accuracy\_score(y\_test, knn\_classifier.predict(x\_test)))

print("Training Confusion Matrix is : \n", confusion\_matrix(y\_train, knn\_classifier.predict(x\_train)))

print("Testing Confusion Matrix is : \n", confusion\_matrix(y\_test, knn\_classifier.predict(x\_test)))

**Input:**

Iris.csv

**Output:**

0.9666666666666667

**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.**

### Locally Weighted Regression –

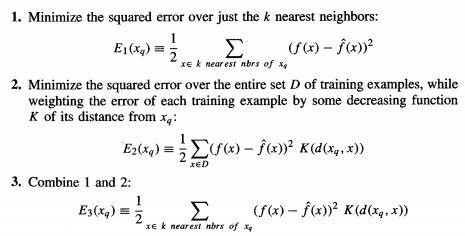
1. Nonparametric regression is a category of regression analysis in which the predictor does not take a predetermined form but is constructed according to information derived from the data(training examples).
2. Nonparametric regression requires larger sample sizes than regression based on parametric models. Because larger the data available ,accuracy will be high.

### Locally Weighted Linear Regression –

Locally weighted regression is called local because the function is approximated based a only on data near the query point, weighted because the contribution of each training example is weighted by its distance from the query point.

Query point is nothing but the point nearer to the target function , which will help in finding the actual position of the target function.

Let us consider the case of locally weighted regression in which the target function f is approximated near x, using a linear function of the form



**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))) # eye - identity matrix

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

def graphPlot(X,ypred):

sortindex = X[:,1].argsort(0) #argsort - index of the smallest

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();

# load data points

data = pd.read\_csv('10data\_tips.csv')

bill = np.array(data.total\_bill) # We use only Bill amount and Tips data

tip = np.array(data.tip)

mbill = np.mat(bill) # .mat will convert nd array is converted in 2D array

mtip = np.mat(tip)

m= np.shape(mbill)[1]

one = np.mat(np.ones(m))

X = np.hstack((one.T,mbill.T)) # 244 rows, 2 cols

ypred = localWeightRegression(X,mtip,8) # increase k to get smooth curves

graphPlot(X,ypred)

**Output**

