VISVESVARAYA TECHNOLOGICAL UNIVERSITY
Belgaum, Karnataka-590 014

LABORATORY MANUAL

Artificial Intelligence and Machine Learning Laboratory

(18CSL76)

Compiled by

Prof. Ancy Thomas,
Prof. Reshma.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

ACHARYA INSTITUTE OF TECHNOLOGY

(Affiliated to Visvesvaraya Technological University, Belgaum)

Soldevanahalli, Bengaluru-560107

2021-2022

TABLE OF CONTENTS

1. Vision, Mission of the Institute

- **Vision of Institute**

Acharya Institute of Technology, committed to the cause of value-based education in all disciplines, envisions itself as a fountainhead of innovative human enterprise, with inspirational initiatives for Academic Excellence.

- **Mission of the institute**

Acharya Institute of Technology strives to provide excellent academic ambiance to the students for achieving global standards of technical education, foster intellectual and personal development, meaningful research and ethical service to sustainable societal needs.

2. Vision, Mission of the Department

- **Vision of the Department**

Envisions to be recognized for quality education and research in the field of Computing, leading to creation of competent engineers, who are innovative and adaptable to the changing demands of industry and society

- **Mission of the Department:**

    - Act as a nurturing ground for young computing aspirants to attain the excellence by imparting quality education and professional ethics
    - Collaborate with industries and provide exposure to latest tools/ technologies.
    - Create an environment conducive for research and continuous learning

3. Program Educational Objectives (PEOs)

Students shall

- Have a successful career in academia, R&D organizations, IT industry or pursue higher studies in specialized fields of Computer Science and Engineering and allied disciplines.
- Be competent, creative and a valued professional in the chosen field
- Engage in life-long learning, professional development and adapt to the working environment quickly Become effective collaborators and exhibit a high level of professionalism by leading or participating in addressing technical, business, environmental and societal challenges.

4. Program Specific Outcomes:

PSO	Statement
	Students Shall
PSO-1	Apply the knowledge of hardware, system software, algorithms, networking and databases
PSO-2	Design, analyze and develop efficient, Secure algorithms using appropriate data structures, databases for processing of data.
PSO-3	Be Capable of developing stand alone, embedded and web-based solutions having easy to operate interface using Software Engineering practices and contemporary computer programming languages.

5. Program Outcomes

Engineering Graduates will be able to:

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

6. Course Outcome

After the course completion Students will be able to

1. Implement artificial intelligence and machine learning algorithms without using Python machine learning libraries.
2. Apply the machine learning algorithms on appropriate datasets using Python machine learning libraries
3. Calculate the target values, accuracy of different machine learning algorithms.
4. Communicate effectively through written records and viva-voce.

PROGRAM-1

Implement A* Search algorithm.

```
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 doesn't 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--- doesn't exist")
    return None



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


def heuristic(n):
    H_dist = {
        'A':10,
        'B':8,
        'C':5,
        'D':7,
        'E':3,
        'F':6,
        'G':5,
        'H':3,
        'I':1,
        'J':0
    }

    return H_dist[n]
```

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

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

OUTPUT:

```
         Path found: ['A', 'F', 'G', 'I', 'J']
Out[4]:  ['A', 'F', 'G', 'I', 'J']
```

PROGRAM-2

Implement AO* Search algorithm.

```
class Graph:

    def __init__(self, graph, heuristicNodeList,startnode):
        self.graph = graph
        self.H = heuristicNodeList
        self.start = startnode
        self.parent = {}
        self.status = {}
        self.solutionGraph={}

    def applyAOStar(self):
        self.aoStar(self.start,False)

    def getNeighbors(self,v):
        return self.graph.get(v,'')

    def getStatus(self,v):
        return self.status.get(v,0)

    def setStatus(self,v,val):
        self.status[v]=val

    def getHeuristicNodeValue(self,n):
        return self.H.get(n,0)

    def setHeuristicNodeValue(self,n,value):
        self.H[n]=value

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

print("------------------------------------------------------------")
        print(self.solutionGraph)
```

```
print("-----------------------------------------------------------")

    def computeMinimumCostChildNodes(self,v):
        minimumCost=0
        costToChildNodeListDict={}
        costToChildNodeListDict[minimumCost]=[]
        flag=True
        for nodeInfoTupleList in self.getNeighbors(v):
            cost=0
            nodeList=[]
            for c, weight in nodeInfoTupleList:
                cost=cost+self.getHeuristicNodeValue(c)+weight
                nodeList.append(c)
            if flag == True:
                minimumCost=cost
                costToChildNodeListDict[minimumCost]=nodeList
                flag=False
            else:
                if minimumCost > cost:
                    minimumCost = cost
                    costToChildNodeListDict[minimumCost]=nodeList

        return minimumCost,costToChildNodeListDict[minimumCost]

    def aoStar(self,v,backTracking):
        print("HEURISTIC VALUES:",self.H)
        print("SOLUTION GRAPH:",self.solutionGraph)
        print("PROCESSING NODE:", v)
        print("-----------------------------------")
        if self.getStatus(v) >= 0:

minimumCost,childNodeList,=self.computeMinimumCostChildNodes(v)
            self.setHeuristicNodeValue(v,minimumCost)
            self.setStatus(v,len(childNodeList))
            solved=True

            for childNode in childNodeList:
                self.parent[childNode]=v
                if self.getStatus(childNode) != -1:
                    solved= solved & False
            if solved == True:
```

```
                    self.setStatus(v,-1)
                    self.solutionGraph[v]=childNodeList
                if v != self.start:
                    self.aoStar(self.parent[v],True)
                if backTracking == False:
                    for childNode in childNodeList:
                        self.setStatus(childNode,0)
                        self.aoStar(childNode,False)


h1 ={'A':1, 'B':6, 'C':12, 'D':10, 'E':4, 'F':4, 'G':5, 'H':7}
graph1 = {
    'A' : [[('B',1),('C',1)],[('D',1)]],
    'B' : [[('G',1)],[('H',1)]],
    'D' : [[('E',1),('F',1)]]
}

G1=Graph(graph1,h1,'A')
G1.applyAOStar()
G1.printSolution()
```

OUTPUT:

```
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 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.

```
import random
import csv

def g_0(n):
    return ("?",)*n

def s_0(n):
    return ('Φ',)*n



def more_general(h1, h2):
    more_general_parts = []
    for x,y in zip(h1,h2):
        mg = x =="?" or (x!= 'Φ' and (x == y or y == 'Φ'))
        more_general_parts.append(mg)
    return all(more_general_parts)



def fulfills(example , hypothesis):
    return more_general(hypothesis, example)

def min_generalizations(h ,x):
    h_new = list(h)
    for i in range(len(h)):
        if not fulfills(x[i:i+1], h[i:i+1]):
            h_new[i] = '?' if h[i] != 'Φ' else x[i]
    return [tuple(h_new)]



def min_specializations(h, domains, x):
    results = []
    for i in range(len(h)):
        if h[i] == "?":
            for val in domains[i]:
```

```
                if x[i] != val:
                    h_new = h[:i] + (val,) + h[i+1:]
                    results.append(h_new)
        elif h[i] != 'Φ':
            h_new = h[:i] + ('Φ',) + h[i+1:]
            results.append(h_new)
    return results

with open('weather.csv') as csvFile:
    examples = [tuple(line) for line in csv.reader(csvFile)]



def get_domains(examples):
    d = [set() for i in examples[0]]
    for x in examples:
        for i, xi in enumerate(x):
            d[i].add(xi)
    return [list(sorted(x)) for x in d]

get_domains(examples)


def candidate_elimination(examples):
    domains = get_domains(examples)[:-1]

    G = set([g_0(len(domains))])
    S = set([s_0(len(domains))])
    i=0

    print("\n G[{0}]:".format(i), G)
    print("\n S[{0}]:".format(i), S)

    for xcx in examples:
        i +=1
        x, cx = xcx[:-1], xcx[-1] # Splitting data into attributes
and decisions
        if cx == 'Yes':
            G = {g for g in G if fulfills(x, g)}
            S = generalize_S(x, G, S)
        else:
```

```
            S = {s for s in S if not fulfills(x, S)}
            G = specialize_G(x, domains, G, S)

        print("\n G[{0}]:".format(i), G)
        print("\n S[{0}]:".format(i), S)
    return

def generalize_S(x, G, S):
    S_prev = list(S)
    for s in S_prev:
        if s not in S:
            continue
        if not fulfills(x, s):
            S.remove(s)
            Splus = min_generalizations(s, x)
            # Keep only generalizations that have a counterpart in G
            S.update([h for h in Splus if any([more_general(g ,h)
for g in G])])

            # Remove the hypotheses less specific than any other in
S
           S.difference_update([h for h in S if any([more_general(h,
h1) for h1 in S if h!= h1])])
    return S




def specialize_G(x, domains, G, S):
    G_prev = list(G)
    for g in G_prev:
        if g not in G:
            continue
        if fulfills(x, g):
            G.remove(g)
            Gminus = min_specializations(g, domains, x)
            # Keep only specializations that have a counterpart in S
            G.update([h for h in Gminus if any([more_general(h, s)
for s in S])])
```

```
            # Remove the hypotheses less general than any other in G
            G.difference_update([h for h in G if
any([more_general(g1, h) for g1 in G if h != g1])])
    return G

candidate_elimination(examples)
```

DATASET: weather.csv

SKY	SKYTEMP	HUMID	WIND	WATER	FORECAST	ENJOYSPORT
Sunny	Warm	Normal	Strong	Warm	Same	Yes
Sunny	Warm	High	Strong	Warm	Same	Yes
Rainy	Warm	High	Strong	Warm	Change	No
Sunny	Warm	High	Strong	Cool	Change	Yes

OUTPUT:

```
G[0]: {('?', '?', '?', '?', '?', '?')}

S[0]: {('Φ', 'Φ', 'Φ', 'Φ', 'Φ', 'Φ')}

G[1]: {('?', '?', '?', '?', '?', '?')}

S[1]: {('Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same')}

G[2]: {('?', '?', '?', '?', '?', '?')}

S[2]: {('Sunny', 'Warm', '?', 'Strong', 'Warm', 'Same')}

G[3]: {('?', '?', '?', '?', '?', 'Same'), ('Sunny', '?', '?', '?', '?', '?')}

S[3]: {('Sunny', 'Warm', '?', 'Strong', 'Warm', 'Same')}

G[4]: {('Sunny', '?', '?', '?', '?', '?')}

S[4]: {('Sunny', 'Warm', '?', 'Strong', '?', '?')}
```

PROGRAM-4

Write a program to demonstrate the working of the decision tree based ID3 algorithm. Use an appropriate dataset for building the decision tree and apply this knowledge to classify a new sample.

```
import pandas as pd
import warnings
from pandas import DataFrame

# To remove warning SettingWithCopy warning
from pandas.core.common import SettingWithCopyWarning
warnings.simplefilter(action="ignore",
category=SettingWithCopyWarning)


df_tennis = pd.read_csv('tennis.csv')

#Function to calculate the entropy of probability of observations
# -p*log2*p


def entropy(probs):
    import math
    return sum( [-prob*math.log(prob, 2) for prob in probs] )


#Function to calculate the entropy of the given Data Sets/List with
respect to target attributes
def entropy_of_list(a_list):
    from collections import Counter
    cnt = Counter(x for x in a_list)   # Counter calculates the
proportion of class
    num_instances = len(a_list)*1.0   # = 14
    probs = [x / num_instances for x in cnt.values()]  # x means no
of YES/NO

    return entropy(probs) # Call Entropy :
```

```
total_entropy = entropy_of_list(df_tennis['PlayTennis'])
print("\n Total Entropy of PlayTennis Data Set:",total_entropy,
"\n\n")


def information_gain(df, split_attribute_name, target_attribute_name,
trace=0):
    '''
    Takes a DataFrame of attributes, and quantifies the entropy of a
target
    attribute after performing a split along the values of another
attribute.
    '''
    # Split Data by Possible Values of Attribute:
    df_split = df.groupby(split_attribute_name)

    # Calculate Entropy for Target Attribute, as well as
    # Proportion of Obs in Each Data-Split
    nobs = len(df.index) * 1.0
    df_agg_ent = df_split.agg({target_attribute_name :
[entropy_of_list, lambda x: len(x)/nobs] })[target_attribute_name]
    df_agg_ent.columns = ['Entropy', 'PropObservations']
    if trace: # helps understand what func. is doing:
        print(df_agg_ent)

    # Calculate Information Gain:
    new_entropy = sum( df_agg_ent['Entropy'] *
df_agg_ent['PropObservations'] )
    old_entropy = entropy_of_list(df[target_attribute_name])
    return old_entropy - new_entropy


print('\n Info-gain for Outlook is :'+str(
information_gain(df_tennis, 'Outlook', 'PlayTennis')),"\n")
print('\n Info-gain for Humidity is: ' + str(
information_gain(df_tennis, 'Humidity', 'PlayTennis')),"\n")
print('\n Info-gain for Wind is:' + str( information_gain(df_tennis,
'Wind', 'PlayTennis')),"\n")
print('\n Info-gain for Temperature is:' + str(
information_gain(df_tennis, 'Temperature','PlayTennis')),"\n")
```

```
def id3(df, target_attribute_name, attribute_names,
default_class=None):

    ## Tally target attribute:
    from collections import Counter
    cnt = Counter(x for x in df[target_attribute_name])# class of YES
/NO

    ## First check: Is this split of the dataset homogeneous?
    if len(cnt) == 1:
        return next(iter(cnt))  # next input data set, or raises
StopIteration when EOF is hit.

    ## Second check: Is this split of the dataset empty?
    # if yes, return a default value
    elif df.empty or (not attribute_names):
        return default_class  # Return None for Empty Data Set

    ## Otherwise: This dataset is ready to be divided up!
    else:
        # Get Default Value for next recursive call of this function:
        default_class = max(cnt.keys()) #No of YES and NO Class
        # Compute the Information Gain of the attributes:
        gainz = [information_gain(df, attr, target_attribute_name)
for attr in attribute_names] #
        index_of_max = gainz.index(max(gainz)) # Index of Best
Attribute
        # Choose Best Attribute to split on:
        best_attr = attribute_names[index_of_max]

        # Create an empty tree, to be populated in a moment
        tree = {best_attr:{}} # Initiate the tree with best attribute
as a node
        remaining_attribute_names = [i for i in attribute_names if i
!= best_attr]

        # Split dataset
        # On each split, recursively call this algorithm.
```

```
        # populate the empty tree with subtrees, which
        # are the result of the recursive call
        for attr_val, data_subset in df.groupby(best_attr):
            subtree = id3(data_subset,
                        target_attribute_name,
                        remaining_attribute_names,
                        default_class)
            tree[best_attr][attr_val] = subtree
        return tree


# Get Predictor Names (all but 'class')
attribute_names = list(df_tennis.columns)
attribute_names.remove('PlayTennis') #Remove the class attribute
# Run Algorithm:
from pprint import pprint
tree = id3(df_tennis,'PlayTennis',attribute_names)
print("\n\nThe Resultant Decision Tree is :\n")
pprint(tree)
attribute = next(iter(tree))


def classify(instance, tree, default=None): # Instance of Play Tennis
with Predicted

    attribute = next(iter(tree)) # Outlook/Humidity/Wind



    if instance[attribute] in tree[attribute].keys(): # Value of the
attributes in  set of Tree keys
        result = tree[attribute][instance[attribute]]

        if isinstance(result, dict): # this is a tree, delve deeper
            return classify(instance, result)
        else:
            return result # this is a label
    else:
        return default
```

```
df_tennis['actual'] = df_tennis.apply(classify, axis=1,
args=(tree,'No') )
    # classify func allows for a default arg: when tree doesn't have
answer for a particular
    # combination of attribute-values, we can use 'no' as the default
guess

df_tennis[['PlayTennis', 'actual']]
training_data = df_tennis.iloc[1:-4] # all but last four instances
test_data  = df_tennis.iloc[-4:] # just the last four
train_tree = id3(training_data, 'PlayTennis', attribute_names)

test_data['predicted'] = test_data.apply(
# <---- test_data source
                                          classify,
                                          axis=1,
                                          args=(train_tree,'Yes') ) #
<---- train_data tree


print("\n\n", test_data)
print ('\n\n Accuracy is : ' + str(
sum(test_data['PlayTennis']==test_data['predicted'] ) /
(1.0*len(test_data.index)) ))
```

OUTPUT:

```
 Total Entropy of PlayTennis Data Set: 0.9402859586706309


 Info-gain for Outlook is :0.2467498197744391


 Info-gain for Humidity is: 0.15183550136234136


 Info-gain for Wind is:0.04812703040826927


 Info-gain for Temperature is:0.029222565658954647


The Resultant Decision Tree is :

{'Outlook': {'Overcast': 'Yes',
             'Rain': {'Wind': {'Strong': 'No', 'Weak': 'Yes'}},
             'Sunny': {'Humidity': {'High': 'No', 'Normal': 'Yes'}}}}


   PlayTennis   Outlook Temperature Humidity    Wind actual predicted
10        Yes     Sunny        Mild   Normal  Strong    Yes        No
11        Yes  Overcast        Mild     High  Strong    Yes       Yes
12        Yes  Overcast         Hot   Normal    Weak    Yes       Yes
13         No      Rain        Mild     High  Strong     No        No


 Accuracy is : 0.75
```

PROGRAM-5

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

```
import numpy as np
X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float)
X = X/np.amax(X,axis=0) # maximum of X array longitudinally
y = y/100

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

#Derivative of Sigmoid Function
def derivatives_sigmoid(x):
      return x * (1 - x)

#Variable initialization
epoch=5000     #Setting training iterations
lr=0.1      #Setting learning rate
inputlayer_neurons = 2      #number of features in data set
hiddenlayer_neurons = 3     #number of hidden layers neurons
output_neurons = 1          #number of neurons at output layer

#weight and bias initialization
wh=np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))
bh=np.random.uniform(size=(1,hiddenlayer_neurons))
wout=np.random.uniform(size=(hiddenlayer_neurons,output_neurons))
bout=np.random.uniform(size=(1,output_neurons))


#draws a random range of numbers uniformly of dim x*y
for i in range(epoch):

#Forward Propagation
      hinp1=np.dot(X,wh)
      hinp=hinp1 + bh
      hlayer_act = sigmoid(hinp)
```

```
        outinp1=np.dot(hlayer_act,wout)
        outinp= outinp1+ bout
        output = sigmoid(outinp)

#Backpropagation
        EO = y-output
        outgrad = derivatives_sigmoid(output)
        d_output = EO* outgrad
        EH = d_output.dot(wout.T)

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

# dot product of nextlayer error and current layer op
wout += hlayer_act.T.dot(d_output) *lr
wh += X.T.dot(d_hiddenlayer) *lr

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

OUTPUT:

```
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.87101847]
 [0.85945811]
 [0.86890533]]
```

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.

```
print("Naive Bayes Classifier for Concept Learning \n")

import csv
import random
import math
import operator

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

def loadCsv(filename):
    lines = csv.reader(open(filename))
    dataset = list(lines)
#     print(dataset)
    for i in range(len(dataset)):
#         print(dataset[i])
        dataset[i] = [float(x) for x in dataset[i]]
    return dataset

def splitDataset(dataset, splitRatio):
    trainSize = int(len(dataset) * splitRatio)
    trainSet = []
    copy = list(dataset)
    i=0
    while len(trainSet) < trainSize:
        trainSet.append(copy.pop(i))
    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 safe_div(sum(numbers), float(len(numbers)))


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


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)
    print("Summarize Attribute By Class")
    print(summaries)
    print(" ")
    return summaries
```

```
def calculateProbability(x, mean, stdev):
    exponent = math.exp(-safe_div(math.pow(x-mean, 2),
(2*math.pow(stdev, 2))))
    final = safe_div(1, (math.sqrt(2*math.pi) * stdev)) * exponent
    return final


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)):
        if testSet[i][-1] == predictions[i]:
            correct += 1
    accuracy = safe_div(correct, float(len(testSet))) * 100.0
    return accuracy

def main():
    filename = 'tennis_NB.csv'
    splitRatio = 0.9
    dataset = loadCsv(filename)
    trainingSet, testSet = splitDataset(dataset, splitRatio)
    print("Split {0} rows into".format(len(dataset)))
    print("Number of Training Data: "+ (repr(len(trainingSet))))
    print("Number of Test Data: "+ (repr(len(testSet))))

    print("\nThe Values assumed for the concept learing attiributes
are: \n")
    print("OUTLOOK=> Sunny=1 Overcast=2 Rain=3\nTEMPERATURE=> Hot=1
Mild=2 Cool=3\nHUMIDITY=> High=1 Normal=2\nWIND=> Weak=1 Strong=2")

    print("TARGET CONCEPT: PLAY TENNIS=> Yes=10 No=5")

    print('\n')

    summaries = summarizeByClass(trainingSet)

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

    print('Actual values: {0}'.format(actual))
    print('Predictions: {0}'.format(predictions))
    accuracy = getAccuracy(testSet, predictions)
    print('Accuracy: {0}%'.format(accuracy))

main()
```

DATASET: tennis_NB.csv

PlayTennis	Outlook	Temperature	Humidity	Wind
No	Sunny	Hot	High	Weak
No	Sunny	Hot	High	Strong
Yes	Overcast	Hot	High	Weak
Yes	Rain	Mild	High	Weak
Yes	Rain	Cool	Normal	Weak
No	Rain	Cool	Normal	Strong
Yes	Overcast	Cool	Normal	Strong
No	Sunny	Mild	High	Weak
Yes	Sunny	Cool	Normal	Weak
Yes	Rain	Mild	Normal	Weak
Yes	Sunny	Mild	Normal	Strong
Yes	Overcast	Mild	High	Strong
Yes	Overcast	Hot	Normal	Weak
No	Rain	Mild	High	Strong

The above dataset is converted using the below instructions:

OUTLOOK=> Sunny=1 Overcast=2 Rain=3
TEMPERATURE=> Hot=1 Mild=2 Cool=3
HUMIDITY=> High=1 Normal=2
WIND=> Weak=1 Strong=2
TARGET CONCEPT: PLAY TENNIS=> Yes=10 No=5

5	**1**	**1**	**1**	**1**
5	**1**	**1**	**1**	**2**
10	**2**	**1**	**1**	**1**
10	**3**	**2**	**1**	**1**
10	**3**	**3**	**2**	**1**
10	**3**	**3**	**2**	**2**
5	**2**	**3**	**2**	**2**
10	**1**	**2**	**1**	**1**
5	**1**	**3**	**2**	**1**
10	**3**	**2**	**2**	**1**
10	**1**	**2**	**2**	**2**
10	**2**	**2**	**1**	**2**
10	**2**	**1**	**2**	**1**
5	**3**	**2**	**1**	**2**

OUTPUT:

```
Naive Bayes Classifier for Concept Learning

Split 14 rows into
Number of Training Data: 12
Number of Test Data: 2

The Values assumed for the concept learing attiributes are:

OUTLOOK=> Sunny=1 Overcast=2 Rain=3
TEMPERATURE=> Hot=1 Mild=2 Cool=3
HUMIDITY=> High=1 Normal=2
WIND=> Weak=1 Strong=2
TARGET CONCEPT: PLAY TENNIS=> Yes=10 No=5
```

```
Summarize Attribute By Class
{1.0: [(8.571428571428571, 2.439750182371333), (2.0, 1.0), (2.0,
0.816496580927726), (1.4285714285714286, 0.5345224838248488)], 2.0: [(8.0,
```

```
2.7386127875258306), (1.8, 0.8366600265340756), (2.2, 0.8366600265340756),
(1.6, 0.5477225575051661)]}
```

```
Actual values: [2.0]
Predictions: [1.0, 1.0]
Accuracy: 50.0%
[[10.0, 2.0, 1.0, 2.0, 1.0], [5.0, 3.0, 2.0, 1.0, 2.0]]
```

Naïve Bayesian classifier for a Diabetes Dataset

```
import csv
import random
import math

#1.Load Data
def loadCsv(filename):
    lines = csv.reader(open(filename, "rt"))
    dataset = list(lines)
    for i in range(len(dataset)):
        dataset[i] = [float(x) for x in dataset[i]]
    return dataset


#Split the data into Training and Testing  randomly
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]


#Seperatedata by Class
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


#Calculate Mean
def mean(numbers):
    return sum(numbers)/float(len(numbers))


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


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


#Summarize Attributes by Class
def summarizeByClass(dataset):
    separated = separateByClass(dataset)
    print(len(separated))
    summaries = {}
    for classValue, instances in separated.items():
            summaries[classValue] = summarize(instances)
    print(summaries)
    return summaries


#Calculate Gaussian Probability Density Function
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
```

```
#Calculate Class Probabilities
def calculateClassProbabilities(summaries, inputVector):
    probabilities = {}
    for classValue, classSummaries in summaries.items():
        probabilities[classValue] = 1
        for i in range(len(classSummaries)):
            mean, stdev = classSummaries[i]
            x = inputVector[i]
            probabilities[classValue] *= calculateProbability(x,
mean, stdev)
    return probabilities


#Make a Prediction
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


#return a list of predictions for each test instance.
def getPredictions(summaries, testSet):
    predictions = []
    for i in range(len(testSet)):
        result = predict(summaries, testSet[i])
        predictions.append(result)
    return predictions


#calculate accuracy ratio.
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


filename = 'DBetes.csv'
splitRatio = 0.70
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)


# test model
predictions = getPredictions(summaries, testSet)
accuracy = getAccuracy(testSet, predictions)
print('Accuracy: {0}%'.format(accuracy))
```

OUTPUT:

```
Split 250 rows into train=175 and test=75 rows
2
{1.0:   [(4.813559322033898,   3.7759427974151385),   (141.54237288135593,
31.497849972546458),       (74.62711864406779,       16.117989847591424),
(19.440677966101696,       15.886744839689554),       (95.69491525423729,
151.64889738190598)],   0.0:   [(3.396551724137931,   2.852548243984293 3),
(108.27586206896552,       30.127435483552063),       (67.11206896551724,
18.854486976855657),       (18.844827586206897,       13.871837875961914),
(70.26724137931035, 124.17703755796266)]}
Accuracy: 72.0%
```

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

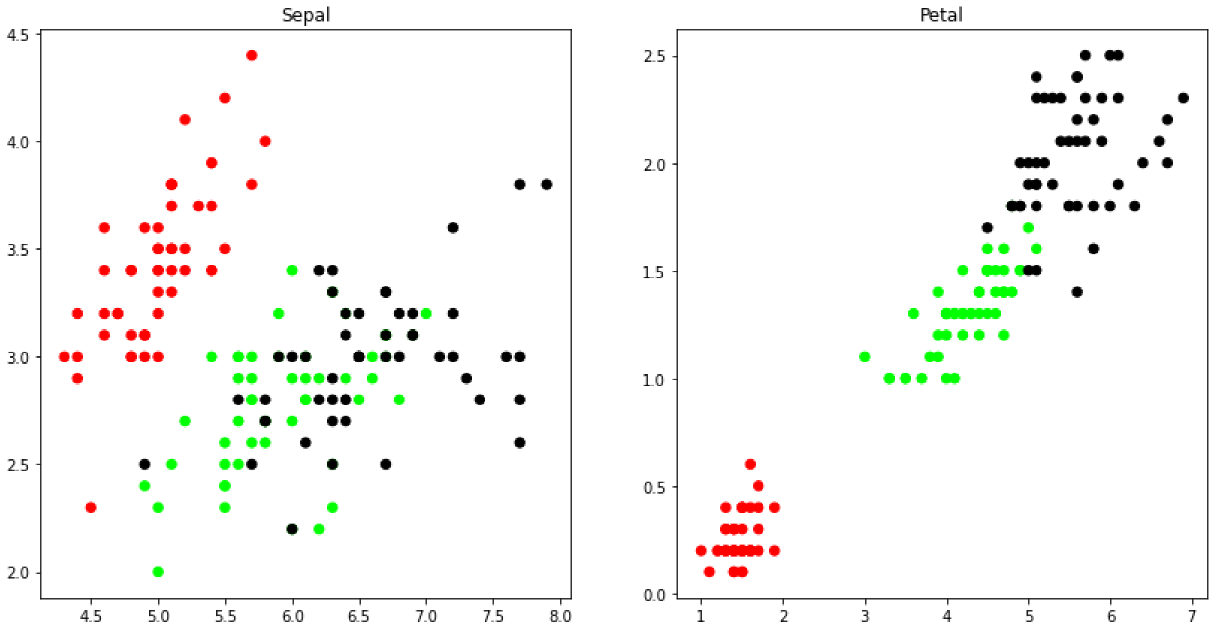
```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import KMeans
import sklearn.metrics as sm
import pandas as pd
import numpy as np

l1 = [0,1,2]

def rename(s):
    l2 = []
    for i in s:
        if i not in l2:
            l2.append(i)
    for i in range(len(s)):
        pos = l2.index(s[i])
        s[i] = l1[pos]
    return s


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

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

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

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

```
y = pd.DataFrame(iris.target)
y.columns = ['Targets']

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

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

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

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

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

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

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

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

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

# Plot the Models Classifications
```

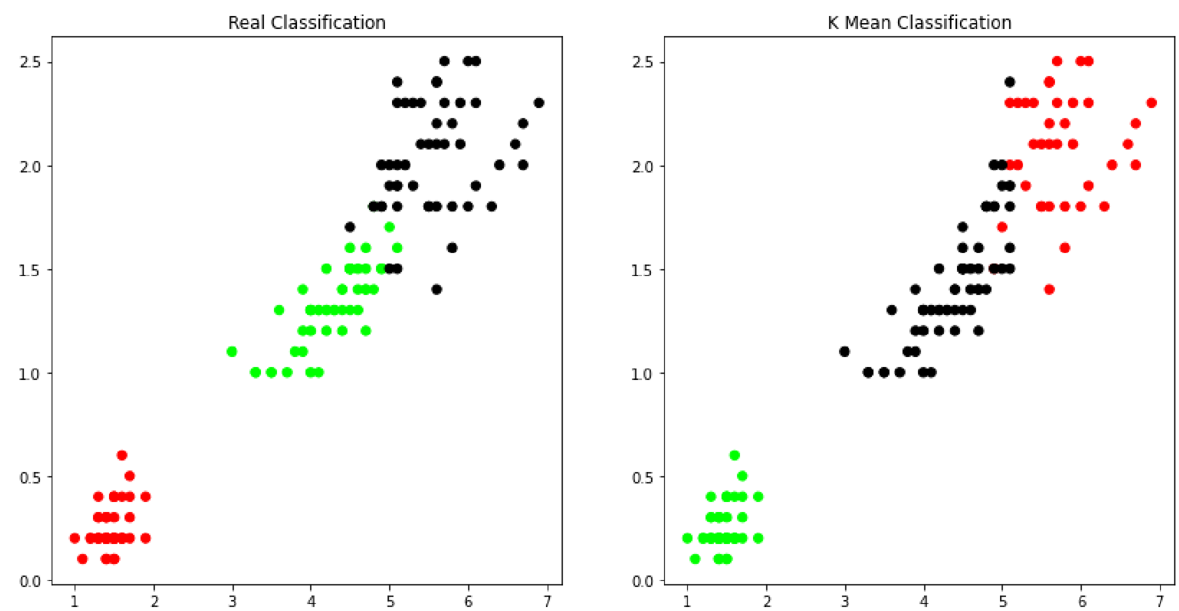
```
plt.subplot(1,2,2)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[model.labels_],
s=40)
plt.title('K Mean Classification')
plt.show()

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

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

from sklearn import preprocessing
scaler = preprocessing.StandardScaler()
scaler.fit(X)
xsa = scaler.transform(X)
xs = pd.DataFrame(xsa, columns = X.columns)

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

y_cluster_gmm = gmm.predict(xs)

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

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

OUTPUT:

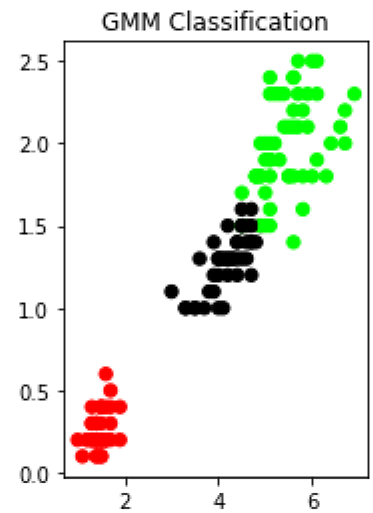
```
IRIS FEATURES :
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']

IRIS TARGET NAMES:
['setosa' 'versicolor' 'virginica']
```




```
Actual Target 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 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]
```



```
What KMeans thought:
 [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 2 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 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 2 2 2 2 1 2 2 2 2
 2 2 1 1 2 2 2 2 1 2 1 2 1 2 2 1 1 2 2 2 2 2 1 2 2 2 2 1 2 2 2 1 2 2 2 1 2
 2 1]
Accuracy of KMeans is  0.8933333333333333
Confusion Matrix for KMeans is
 [[50  0  0]
 [ 0 48  2]
 [ 0 14 36]]
```




```
What EM thought:
 [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 2 1 2 1 2 1
 1 1 1 2 1 1 1 1 1 2 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]
Accuracy of EM is  0.9666666666666667
Confusion Matrix for EM is
 [[50  0  0]
 [ 0 45  5]
 [ 0  0 50]]
```

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.

```
from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier
import numpy as np
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
iris_dataset=load_iris()

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

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

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

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

for i in range(len(X_test)):
  x = X_test[i]
  x_new = np.array([x])
  prediction = kn.predict(x_new)
  print("\n Actual : {0} {1}, Predicted
:{2}{3}".format(y_test[i],iris_dataset["target_names"][y_test[i]],pre
diction,iris_dataset["target_names"][ prediction]))
print("\n TEST SCORE[ACCURACY]: {:.2f}\n".format(kn.score(X_test,
y_test)))
```

OUTPUT:

```
IRIS FEATURES \ TARGET NAMES:
  ['setosa' 'versicolor' 'virginica']

[0]:[setosa]

[1]:[versicolor]

[2]:[virginica]

 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 : 0 setosa, Predicted :[0]['setosa']

 Actual : 2 virginica, Predicted :[2]['virginica']

 Actual : 0 setosa, Predicted :[0]['setosa']

 Actual : 1 versicolor, Predicted :[1]['versicolor']

 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 : 1 versicolor, Predicted :[1]['versicolor']

 Actual : 1 versicolor, Predicted :[1]['versicolor']

 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 : 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 : 0 setosa, Predicted :[0]['setosa']

Actual : 2 virginica, Predicted :[2]['virginica']

Actual : 0 setosa, Predicted :[0]['setosa']

Actual : 0 setosa, Predicted :[0]['setosa']

Actual : 1 versicolor, Predicted :[1]['versicolor']

Actual : 1 versicolor, Predicted :[1]['versicolor']

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 : 0 setosa, Predicted :[0]['setosa']

Actual : 1 versicolor, Predicted :[2]['virginica']

TEST SCORE[ACCURACY]: 0.97
```

PROGRAM-9

Implement the non-parametric Locally Weighted Regression algorithm in order to fit data points. Select the appropriate data set for your experiment and draw graphs.

<u>Locally Weighted Regression</u>

- **Locally weighted regression is a very powerful non-parametric model used in statistical learning .**
-
- **Given a dataset X,Y , We attempt to find a model parameter β(x) that minimizes residual sum of weighted squared errors. The weights are given by a kernel function (k or W) which can be chosen arbitrarily.**

ALGORITHM:

1. Read the given data sample to **X** and the curve(linear or non-linear) to **Y**
2. Set the value for smoothing 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:

$$w(x,\ x_0) \ = \ e^{\frac{-(x-x_0)}{2\tau^2}}$$

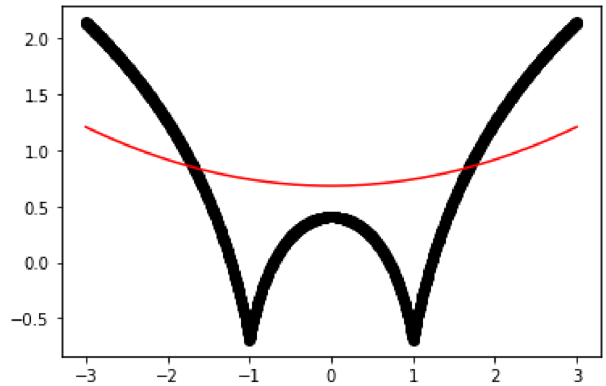
5. Determine the value of model term parameter **β** using:

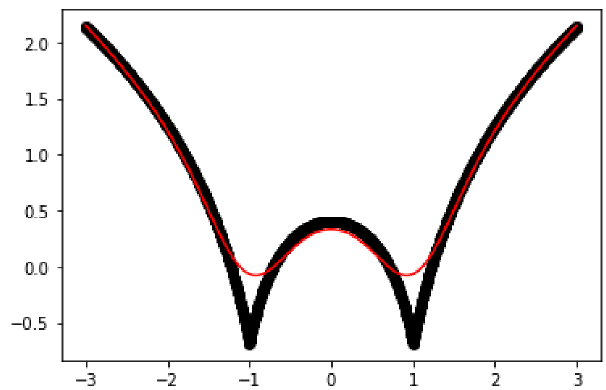
$$\beta(x_0) \ = \ \left(X^T W X\right)^{-1} X^T W y$$

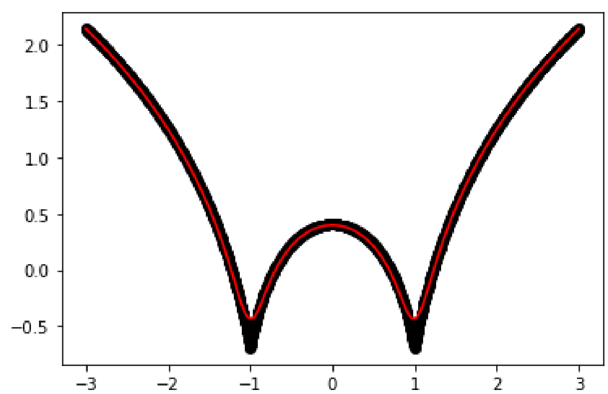
6. Prediction = X_0*$\boldsymbol{\beta}$

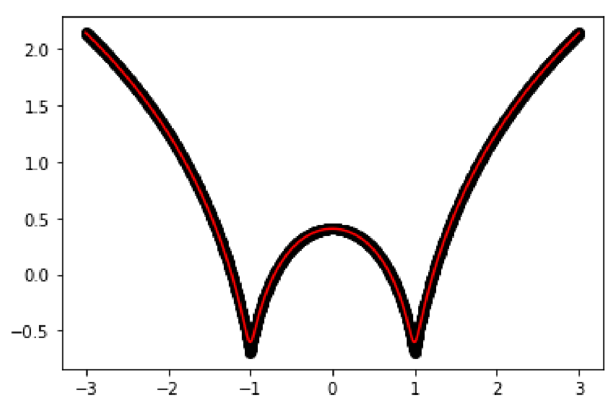
```python
import numpy as np
import matplotlib.pyplot as plt

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

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

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

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

OUTPUT:


2.0
1.5
1.0
0.5
0.0
−0.5
−3
−2
−1
0
1
2
3


2.0
1.5
1.0
0.5
0.0
−0.5
−3
−2
−1
0
1
2
3


2.0
1.5
1.0
0.5
0.0
−0.5
−3
−2
−1
0
1
2
3


2.0
1.5
1.0
0.5
0.0
−0.5
−3
−2
−1
0
1
2
3