`

JYOTHY INSTITUTE OF TECHNOLOGY

AFFILIATED TO VTU, BELAGAVI

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ACCREDITED BY NBA, NEW DELHI

LAB MANUAL

FOR

ARTIFICIAL INTELLIGENCE & MACHINE LEARNING LABORATORY

(18CSL76)

`

Course Details

Course Name	**:**	**Artificial Intelligence & Machine Learning Lab**
Course Code	**:**	**18CSL76**
Course prerequisite	**:**	**Basic Knowledge of Python Programming**

Course Objectives

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

Course Outcomes

• Implement and demonstrate AI and ML algorithms

• Evaluate different algorithms

Conduction of Practical Examination:

• Experiment distribution

For laboratories having only one part: Students are allowed to pick one experiment from the lot with equal opportunity. o

For laboratories having PART A and PART B: Students are allowed to pick one experiment from PART A and one experiment from PART B, with equal opportunity.

• Change of experiment is allowed only once and marks allotted for procedure to be made zero of the changed part only.

• Marks Distribution (Courseed to change in accoradance with university regulations)

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

r) For laboratories having PART A and PART B i. Part A – Procedure + Execution + Viva = 6 + 28 + 6 = 40 Marks ii. Part B – Procedure + Execution + Viva = 9 + 42 + 9 = 60 Marks

`

LAB EXPERIMENTS

1	Implement A* Search Algorithm
2	Implement AO* Algorithm
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.
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.
5	Build an Artificial Neural Network by implementing the Back propagation Algorithm and test the same using appropriate data sets.
6	Write a program to implement the naïve Bayesian classifier for a sample training data set stored as a .CSV file. Compute the accuracy of the classifier, considering few test data sets.
7	Apply EM algorithm to cluster a set of data stored in a .CSV file. Use the same data set for clustering using k-Means algorithm. Compare the results of these two algorithms and comment on the quality of clustering. You can add Java/Python ML library classes/API in the program.
8	Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions. Java/Python ML library classes can be used for this problem.
9	Implement the non-parametric Locally Weighted Regression algorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs.

`

Program 1 : Implement A* Search Algorithm

Source Code:

```
def A_star(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
                    if m not in open_set and m not in closed_set:
                        open_set.add(m)
                        #n is set its parent
                        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:    # if better cost found, then update the existing cost 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('Optimal Path :', 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 = {
        'S': 8,
        'A': 8,
        'B': 4,
        'C': 3,
        'D': 1000,
        'E': 1000,
        'G': 0,

    }
```

```
`
    return H_dist[n]

#Describe your graph here
Graph_nodes = {'S': [['A', 1], ['B', 5], ['C', 8]],
        'A': [['D', 3], ['E', 7], ['G', 9]],
        'B': [['G', 4]],
        'C': [['G', 5]],
        'D': None,
        'E': None}

A_star('S', 'G')
```

Output:

Optimal Path : ['S', 'B', 'G']

`

Program 2 : Implement AO Star Search Algorithm

Source Code:

```
def recAOStar(n):
    global finalPath
    print("Expanding Node : ", n)
    and_nodes = []
    or_nodes = []

    #Segregation of AND and OR nodes
    if (n in allNodes):
        if 'AND' in allNodes[n]:
            and_nodes = allNodes[n]['AND']
        if 'OR' in allNodes[n]:
            or_nodes = allNodes[n]['OR']

    # If leaf node then return
    if len(and_nodes) == 0 and len(or_nodes) == 0:
        return

    solvable = False
    marked = {}

    while not solvable:
        # If all the child nodes are visited and expanded, take the least cost of all the child nodes
        if len(marked) == len(and_nodes) + len(or_nodes):
            min_cost_least, min_cost_group_least = least_cost_group(and_nodes, or_nodes, {})
            solvable = True
            change_heuristic(n, min_cost_least)
            optimal_child_group[n] = min_cost_group_least
            continue

        # Least cost of the unmarked child nodes
        min_cost, min_cost_group = least_cost_group(and_nodes, or_nodes, marked)

        is_expanded = False

        # If the child nodes have sub trees then recursively visit them to recalculate the heuristic of the child node
        if len(min_cost_group) > 1:
            if (min_cost_group[0] in allNodes):
                is_expanded = True
                recAOStar(min_cost_group[0])
            if (min_cost_group[1] in allNodes):
                is_expanded = True
                recAOStar(min_cost_group[1])
        else:
            if (min_cost_group in allNodes):
                is_expanded = True
                recAOStar(min_cost_group)


        # If the child node had any subtree and expanded, verify if the new heuristic value is still the least among all nodes
        if is_expanded:
            min_cost_verify, min_cost_group_verify = least_cost_group(and_nodes, or_nodes, {})

            if min_cost_group == min_cost_group_verify:
                solvable = True
```

`
```
            change_heuristic(n, min_cost_verify)
            optimal_child_group[n] = min_cost_group

    # If the child node does not have any subtrees then no change in heuristic, so update the min cost of the current node
    else:
        solvable = True
        change_heuristic(n, min_cost)
        optimal_child_group[n] = min_cost_group

    #Mark the child node which was expanded
    marked[min_cost_group] = 1
  return heuristic(n)


# Function to calculate the min cost among all the child nodes
def least_cost_group(and_nodes, or_nodes, marked):
  node_wise_cost = {}

  for node_pair in and_nodes:
    if not node_pair[0] + node_pair[1] in marked:
      cost = 0
      cost = cost + heuristic(node_pair[0]) + heuristic(node_pair[1]) + 2
      node_wise_cost[node_pair[0] + node_pair[1]] = cost

  for node in or_nodes:
    if not node in marked:
      cost = 0
      cost = cost + heuristic(node) + 1
      node_wise_cost[node] = cost

  min_cost = 999999
  min_cost_group = None

  # Calculates the min heuristic
  for costKey in node_wise_cost:
    if node_wise_cost[costKey] < min_cost:
      min_cost = node_wise_cost[costKey]
      min_cost_group = costKey

  return [min_cost, min_cost_group]


# Returns heuristic of a node
def heuristic(n):
  return H_dist[n]

# Updates the heuristic of a node
def change_heuristic(n, cost):
  H_dist[n] = cost
  return

# Function to print the optimal cost nodes
def print_path(node):
  print(optimal_child_group[node], end="")
  node = optimal_child_group[node]

  if len(node) > 1:
    if node[0] in optimal_child_group:
      print("->", end="")
```

```
`
            print_path(node[0])
        if node[1] in optimal_child_group:
            print("->", end="")
            print_path(node[1])
    else:
        if node in optimal_child_group:
            print("->", end="")
            print_path(node)


#Describe the heuristic here
H_dist = {
    'A': -1,
    'B': 4,
    'C': 2,
    'D': 3,
    'E': 6,
    'F': 8,
    'G': 2,
    'H': 0,
    'I': 0,
    'J': 0
}


#Describe your graph here
allNodes = {
    'A': {'AND': [('C', 'D')], 'OR': ['B']},
    'B': {'OR': ['E', 'F']},
    'C': {'OR': ['G'], 'AND': [('H', 'I')]},
    'D': {'OR': ['J']}
}

optimal_child_group = {}
optimal_cost = recAOStar('A')

print('Nodes which gives optimal cost are')
print_path('A')
print('\nOptimal Cost is  :: ', optimal_cost)
```

`

Output:

Expanding Node : A

Expanding Node : B

Expanding Node : C

Expanding Node : D

Nodes which gives optimal cost are

CD->HI->J

Optimal Cost is :: 5

`

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.

Source Code:

```
import numpy as np
import pandas as pd

# Loading Data from a CSV File
data1= pd.read_csv('Ex02.csv')
print(data1)
#data = pd.DataFrame(data=data1)
# Separating concept features from Target
concepts = np.array(data1.iloc[:,0:-1])
# Isolating target into a separate DataFrame
#copying last column to target array
target = np.array(data1.iloc[:,-1])

print('concepts')
print(concepts)

print('target')
print(target)

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()
    print("initialization of specific_h and general_h")
    print("\n specific_h :")
    print(specific_h)

    general_h = [["?" for i in range(len(specific_h))]
            for i in range(len(specific_h))]
    print("\n general_h:")
    print(general_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 negative 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] = '?'

        print(" \n ------ Candidate Elimination Algorithm Step : ",i+1)
        print("\n specific_h:")
        print(specific_h)
        print("\n general_h:")
        print(general_h)
    # 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("\n Final Specific_h:", s_final)
print("\n Final General_h:", g_final)
```

Input csv file:

Ex02.csv

Output:

```
Sky  Temp Humidity    Wind Water Forecast Play
0  Sunny  Warm   Normal  Strong  Warm    Same  Yes
1  Sunny  Warm    High  Strong  Warm    Same  Yes
2  Rainy  Cold    High  Strong  Warm   Change   No
3  Sunny  Warm    High  Strong  Cool   Change  Yes
concepts
[['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']
 ['Sunny' 'Warm' 'High' 'Strong' 'Warm' 'Same']
 ['Rainy' 'Cold' 'High' 'Strong' 'Warm' 'Change']
 ['Sunny' 'Warm' 'High' 'Strong' 'Cool' 'Change']]
target
['Yes' 'Yes' 'No' 'Yes']
initialization of specific_h and general_h

 specific_h :
['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']

 general_h:
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?',
'?']]

 ------ Candidate Elimination Algorithm Step :  1

 specific_h:
['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']

 general_h:
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?',
'?']]

 ------ Candidate Elimination Algorithm Step :  2
```

`

specific_h:
['Sunny' 'Warm' '?' 'Strong' 'Warm' 'Same']

general_h:
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

------ Candidate Elimination Algorithm Step : 3

specific_h:
['Sunny' 'Warm' '?' 'Strong' 'Warm' 'Same']

general_h:
[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', 'Same']]

------ Candidate Elimination Algorithm Step : 4

specific_h:
['Sunny' 'Warm' '?' 'Strong' '?' '?']

general_h:
[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Final Specific_h: ['Sunny' 'Warm' '?' 'Strong' '?' '?']

Final General_h: [['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?']]

`

Program 4 : Write a program to demonstrate the working of the decision tree based ID3 algorithm. Use an appropriate data set for building the decision tree and apply this knowledge to classify a new sample.

Source Code:

```
import pandas as pd
import numpy as np
#Import the dataset and define the feature as well as the target datasets / columns#
dataset = pd.read_csv('playtennis.csv',
                      names=['outlook','temperature','humidity','wind','class',])
#Import all columns omitting the fist which consists the names of the animals
#We drop the animal names since this is not a good feature to split the data on
attributes =('Outlook','Temperature','Humidity','Wind','PlayTennis')
def entropy(target_col):
    """
    Calculate the entropy of a dataset.
    The only parameter of this function is the target_col parameter which specifies
    the target column
    """
    elements,counts = np.unique(target_col,return_counts = True)
    total_count = np.sum(counts)
    entropy = np.sum([(-counts[i]/total_count)*np.log2(counts[i]/total_count) for i in range(len(elements))])
    #print('Entropy =', entropy)
    return entropy
def InfoGain(data,split_attribute_name,target_name="class"):
    #Calculate the entropy of the total dataset
    total_entropy = entropy(data[target_name])

    ##Calculate the entropy of the dataset

    #Calculate the values and the corresponding counts for the split attribute
    vals,counts= np.unique(data[split_attribute_name],return_counts=True)

    #Calculate the weighted entropy
    Weighted_Entropy =
np.sum([(counts[i]/np.sum(counts))*entropy(data.where(data[split_attribute_name]==vals[i]).dropna()[targe
t_name]) for i in range(len(vals))])

    #Calculate the information gain
    Information_Gain = total_entropy - Weighted_Entropy
    return Information_Gain

def ID3(data,originaldata,features,target_attribute_name="class",parent_node_class = None):
    #Define the stopping criteria --> If one of this is satisfied, we want to return a leaf node#

    #If all target_values have the same value, return this value

    if len(np.unique(data[target_attribute_name])) <= 1:
        return np.unique(data[target_attribute_name])[0]
```

`

```
    #If the dataset is empty, return the mode target feature value in the original dataset
    elif len(data)==0:
        return
np.unique(originaldata[target_attribute_name])[np.argmax(np.unique(originaldata[target_attribute_name],ret
urn_counts=True)[1])]

    elif len(features) ==0:
        #return parent_node_class
        return
np.unique(originaldata[target_attribute_name])[np.argmax(np.unique(originaldata[target_attribute_name],ret
urn_counts=True)[1])]


    #If none of the above holds true, grow the tree!

    else:
        #Set the default value for this node --> The mode target feature value of the current node
        parent_node_class =
np.unique(data[target_attribute_name])[np.argmax(np.unique(data[target_attribute_name],return_counts=Tr
ue)[1])]

        #Select the feature which best splits the dataset
        item_values = [InfoGain(data,feature,target_attribute_name) for feature in features] #Return the
information gain values for the features in the dataset
        best_feature_index = np.argmax(item_values)
        best_feature = features[best_feature_index]

        #Create the tree structure. The root gets the name of the feature (best_feature) with the maximum
information
        #gain in the first run
        tree = {best_feature:{}}


        #Remove the feature with the best inforamtion gain from the feature space
        features = [i for i in features if i != best_feature]

        #Grow a branch under the root node for each possible value of the root node feature

        for value in np.unique(data[best_feature]):
            value = value
            #Split the dataset along the value of the feature with the largest information gain and therwith create
sub_datasets
            sub_data = data.where(data[best_feature] == value).dropna()

            #Call the ID3 algorithm for each of those sub_datasets with the new parameters --> Here the
recursion comes in!
            subtree = ID3(sub_data,dataset,features,target_attribute_name,parent_node_class)

            #Add the sub tree, grown from the sub_dataset to the tree under the root node
            tree[best_feature][value] = subtree

        return(tree)
```

`

```
def predict(query,tree,default = 1):

    #1.
    for key in list(query.keys()):
        if key in list(tree.keys()):
            #2.
            try:
                result = tree[key][query[key]]
            except:
                return default

            #3.
            result = tree[key][query[key]]
            #4.
            if isinstance(result,dict):
                return predict(query,result)
            else:
                return result

def train_test_split(dataset):
    training_data = dataset.iloc[:14].reset_index(drop=True)
    #We drop the index respectively relabel the index
    #starting form 0, because we do not want to run into errors regarding the row labels / indexes
    #testing_data = dataset.iloc[10:].reset_index(drop=True)
    return training_data  #,testing_data
def test(data,tree):
    #Create new query instances by simply removing the target feature column from the original dataset and
    #convert it to a dictionary
    queries = data.iloc[:,:-1].to_dict(orient = "records")

    #Create a empty DataFrame in whose columns the prediction of the tree are stored
    predicted = pd.DataFrame(columns=["predicted"])

    #Calculate the prediction accuracy
    for i in range(len(data)):
        predicted.loc[i,"predicted"] = predict(queries[i],tree,1.0)

    print('\n The prediction accuracy is: ',(np.sum(predicted["predicted"] ==
data["class"])/len(data))*100,'%')
"""
Train the tree, Print the tree and predict the accuracy
"""
XX = train_test_split(dataset)
training_data=XX
#elements,counts = np.unique(training_data["class"],return_counts = True)

"""
i=0
for value in np.unique(training_data["outlook"]):
        value = value
        sub_data = training_data.where(training_data["outlook"] == value).dropna()
```

```
`
        print(i+1, "Subdata for value=", value, "is:\n", sub_data)
        i+=1
"""

#testing_data=XX[1]
tree = ID3(training_data,training_data,training_data.columns[:-1])
print(' \n Display Tree',tree)
print('\n len of training data =',len(training_data))
test(training_data,tree)
```

Input csv file:
tennis.csv

Output:

Display Tree {'outlook': {'Overcast': 'Yes', 'Rain': {'wind': {'Weak': 'Yes', 'Strong': 'No'}}, 'Sunny': {'humidity': {'High': 'No', 'Normal': 'Yes'}}}}

len of training data = 14

The prediction accuracy is: 100.0 %

`

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

Source Code:

```
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)
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=5.221

>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

[{'delta': -0.0059546604162323625, 'output': 0.029980305604426185, 'weights': [-1.4688375095432327, 1.850887325439514, 1.0858178629550297]}, {'delta':

`

0.0026279652850863837, 'output': 0.9456229000211323, 'weights': [0.37711098142462157, -0.0625909894552989, 0.2765123702642716]}]

[{'delta': -0.04270059278364587, 'output': 0.23648794202357587, 'weights': [2.515394649397849, -0.3391927502445985, -0.9671565426390275]}, {'delta': 0.03803132596437354, 'output': 0.7790535202438367, 'weights': [-2.5584149848484263, 1.0036422106209202, 0.42383086467582715]}]

`

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.

Source Code:

```
import csv
import random
import math
#1.Load Data
def loadCsv(filename):
    filename="diabetes1.csv"
    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):
    #splitRatio = 0.7
    trainSize = int(len(dataset) * splitRatio)
    trainSet = []
    copy = list(dataset)
    while len(trainSet) < trainSize:
        #Using randrange() to generate numbers 0 to len(copy)=length of dataset
        index = random.randrange(len(copy))
        # pop: removes and returns the element at
        #the given index (passed as an argument) from the list,
        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 = {}
    # dictionary.items returns a copy of the
    #dictionary's list of (key, value) pairs
    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)
        print(i+1,': ', testSet[i],"--",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 = 'diabetes1.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))
```

Input csv file:

diabetes1.csv

Output:

Split 768 rows into train=537 and test=231 rows

2

{0.0: [(3.4066852367688023, 3.0428154943228063), (108.72701949860725, 26.69385968724664), (68.0891364902507, 17.863162966051384), (19.657381615598887, 14.792310581609273), (68.25069637883009, 96.91759785845593), (30.44428969359332, 7.411610586860471), (0.42533704735376027, 0.2978357042937455), (31.200557103064067, 11.35972989889816)], 1.0: [(4.842696629213483, 3.6727700914615315), (139.11797752808988, 32.623907429973684), (69.61797752808988, 22.059757372227057), (21.837078651685392, 18.588356005524794), (97.21348314606742, 144.7597247036402), (35.04213483146067, 7.791743416799809), (0.5590337078651684, 0.38155395763400757), (36.79213483146067, 10.525543046973642)]}

Accuracy: 75.32467532467533%

`

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.

Source Code:

```
import numpy as np
import math
import matplotlib.pyplot as plt
import csv
def get_binomial_log_likelihood(obs,probs):
    """ Return the (log)likelihood of obs, given the probs"""
    # Binomial Distribution Log PDF
    # ln (pdf)      = Binomial Coeff * product of probabilities
    # ln[f(x|n, p)] =   comb(N,k)    * num_heads*ln(pH) + (N-num_heads) * ln(1-pH)
    N = sum(obs);#number of trials
    k = obs[0] # number of heads
    binomial_coeff = math.factorial(N) / (math.factorial(N-k) * math.factorial(k))
    prod_probs = obs[0]*math.log(probs[0]) + obs[1]*math.log(1-probs[0])
    log_lik = binomial_coeff + prod_probs
    return log_lik
# 1st:  Coin B, {HTTTHHTHTH}, 5H,5T
# 2nd:  Coin A, {HHHHTHHHHH}, 9H,1T
# 3rd:  Coin A, {HTHHHHHTHH}, 8H,2T
# 4th:  Coin B, {HTHTTTHHTT}, 4H,6T
# 5th:  Coin A, {THHHTHHHTH}, 7H,3T
# so, from MLE: pA(heads) = 0.80 and pB(heads)=0.45
data=[]
with open("cluster.csv") as tsv:
    for line in csv.reader(tsv):
        data=[int(i) for i in line]

# represent the experiments
head_counts = np.array(data)
tail_counts = 10-head_counts
experiments = list(zip(head_counts,tail_counts))
# initialise the pA(heads) and pB(heads)
pA_heads = np.zeros(100); pA_heads[0] = 0.60
pB_heads = np.zeros(100); pB_heads[0] = 0.50
# E-M begins!
delta = 0.001
j = 0 # iteration counter
improvement = float('inf')
while (improvement>delta):
    expectation_A = np.zeros((len(experiments),2), dtype=float)
    expectation_B = np.zeros((len(experiments),2), dtype=float)
    for i in range(0,len(experiments)):
        e = experiments[i] # i'th experiment
          # loglikelihood of e given coin A:
        ll_A = get_binomial_log_likelihood(e,np.array([pA_heads[j],1-pA_heads[j]]))
```

`

```
        # loglikelihood of e given coin B
        ll_B = get_binomial_log_likelihood(e,np.array([pB_heads[j],1-pB_heads[j]]))
        # corresponding weight of A proportional to likelihood of A , ex. .45
        weightA = math.exp(ll_A) / ( math.exp(ll_A) + math.exp(ll_B) )
        # corresponding weight of B proportional to likelihood of B , ex. .55
        weightB = math.exp(ll_B) / ( math.exp(ll_A) + math.exp(ll_B) )
        expectation_A[i] = np.dot(weightA, e) #multiply weightA * e .45xNo. of heads and 45xNo. of tails for
coin A
        expectation_B[i] = np.dot(weightB, e) #multiply weightB * e .45xNo. of heads and 45xNo. of Tails for
coin B
    pA_heads[j+1] = sum(expectation_A)[0] / sum(sum(expectation_A)); #summing up the data no. of heads
and tails for coin A
    pB_heads[j+1] = sum(expectation_B)[0] / sum(sum(expectation_B)); #summing up the data no. of heads
and tails for coin B
    #checking the improvement to maximise the accuracy.
    improvement = ( max( abs(np.array([pA_heads[j+1],pB_heads[j+1]]) -
                np.array([pA_heads[j],pB_heads[j]]) )) )
    print(np.array([pA_heads[j+1],pB_heads[j+1]]) -
                np.array([pA_heads[j],pB_heads[j]]) )
    j = j+1
plt.figure();
plt.plot(range(0,j),pA_heads[0:j])#for plotting the graph coin A
plt.plot(range(0,j),pB_heads[0:j])#for plotting the graph coin B
plt.show()
```

Output:

```
[ 0.00796672 -0.09125939]

[ 0.04620638 -0.05680878]

[ 0.02203957 -0.02519619]

[ 0.00533685 -0.00675812]

[ 0.00090446 -0.00162885]

[ 6.34794565e-05 -4.42987679e-04]
```

`

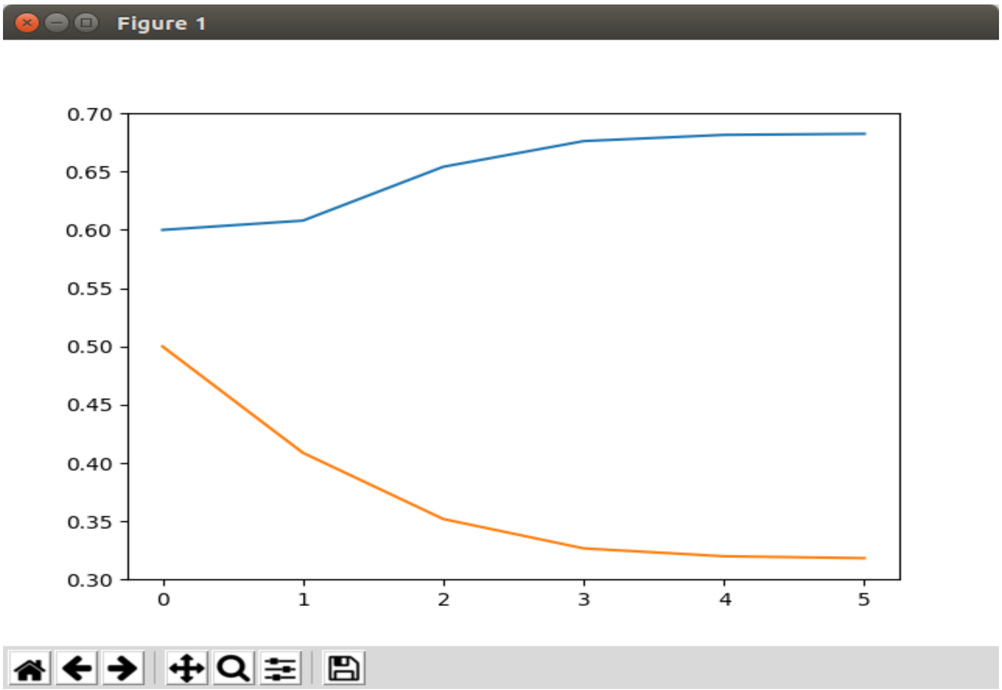


Figure 1
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0
1
2
3
4
5

`

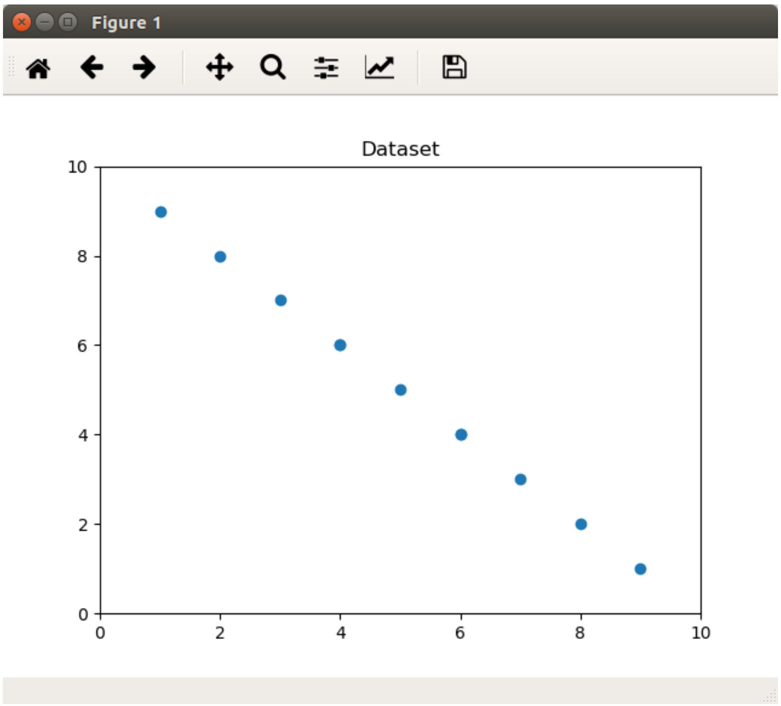
K-Means:

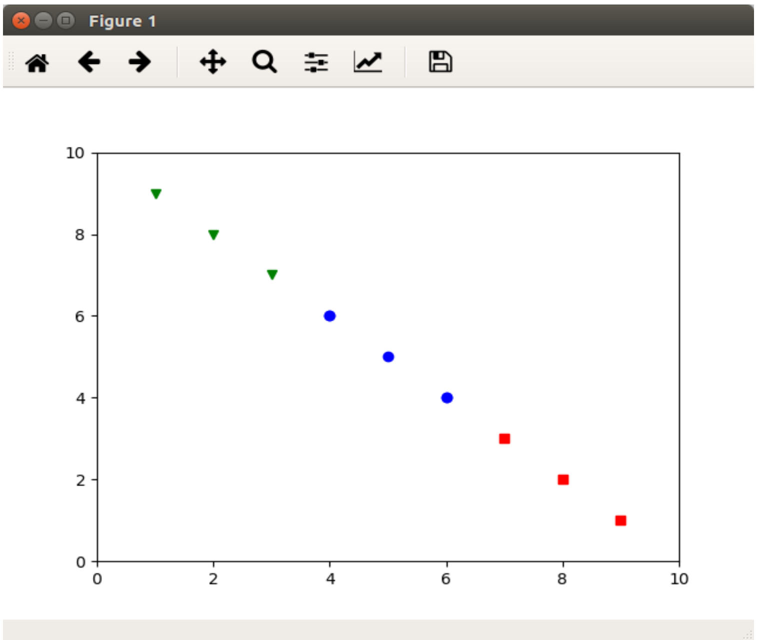
 # clustering dataset

```
from sklearn.cluster import KMeans
from sklearn import metrics
import numpy as np
import matplotlib.pyplot as plt
import csv
data=[]
ydata=[]
with open("cluster.csv") as tsv:
    for line in csv.reader(tsv):
        data=[int(i) for i in line]
        ydata=[10-int(i) for i in line]

x1 = np.array(data)#np.array([3, 1, 1, 2, 1, 6, 6, 6, 5, 6, 7, 8, 9, 8, 9, 9, 8])
x2 = np.array(ydata)#np.array([5, 4, 6, 6, 5, 8, 6, 7, 6, 7, 1, 2, 1, 2, 3, 2, 3])
print(x1)
plt.plot()
plt.xlim([0, 10])
plt.ylim([0, 10])
plt.title('Dataset')
plt.scatter(x1, x2)
plt.show()
# create new plot and data
plt.plot()
X = np.array(list(zip(x1, x2))).reshape(len(x1), 2)
colors = ['b', 'g', 'r']
markers = ['o', 'v', 's']
# KMeans algorithm
K = 3
kmeans_model = KMeans(n_clusters=K).fit(X)
plt.plot()
for i, l in enumerate(kmeans_model.labels_):
    plt.plot(x1[i], x2[i], color=colors[l], marker=markers[l],ls='None')
    plt.xlim([0, 10])
    plt.ylim([0, 10])
plt.show()
```

Output:





`

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.

Source Code:

```
#import numpy as np
import pandas as pd
# Importing the dataset
dataset = pd.read_csv('iris.csv')
#dataset.groupby('species').size()
#Dividing data into features and labels
feature_columns = ['sepal_length', 'sepal_width', 'petal_length','petal_width']
X = dataset[feature_columns].values
y = dataset['species'].values
"""
KNeighborsClassifier does not accept string labels.
We need to use LabelEncoder to transform them into numbers.
Iris-setosa correspond to 0,
Iris-versicolor correspond to 1 and
Iris-virginica correspond to 2.
"""
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
y = le.fit_transform(y)
#Spliting dataset into training set and test set
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)
# Fitting K-NN to the Training set
from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors = 3)
# Fitting the model
classifier.fit(X_train, y_train)
# Predicting the Test set results
y_pred = classifier.predict(X_test)
print("y_pred    y_test")
for i in range(len(y_pred)):
    print(y_pred[i], "    ", y_test[i])
# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(cm)
from sklearn.metrics import accuracy_score
accuracy = accuracy_score(y_test, y_pred)*100
print('Accuracy of our model is equal ' + str(round(accuracy, 2)) + ' %.')
```

Input csv file:

iris_data.csv

`

Output:

y_pred	*y_test*
2	2
1	1
0	0
2	2
0	0
2	2
0	0
1	1
1	1
1	1
2	2
1	1
1	1
1	1
2	1
0	0
1	1
1	1
0	0
0	0
2	2
1	1
0	0

`

0	*0*
2	*2*
0	*0*
0	*0*
1	*1*
1	*1*
0	*0*

Confusion Matrix:

[[11 0 0]

[0 12 1]

[0 0 6]]

Accuracy of our model is equal 96.67 %.

`

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.

Source Code:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
def kernel(point,xmat,k):
    m,n = np.shape(xmat)
    weights = np.mat(np.eye(m))
    for j in range(m):
        diff = point - X[j]
        weights[j,j] = np.exp(diff*diff.T/(-2.0*k**2))
    return weights
def localWeight(point,xmat,ymat,k):
    wei = kernel(point,xmat,k)
    W = (X.T*(wei*X)).I*(X.T*(wei*ymat.T))
    return W
def localWeightRegression(xmat,ymat,k):
    m,n = np.shape(xmat)
    ypred = np.zeros(m)
    for i in range(m):
        ypred[i] = xmat[i]*localWeight(xmat[i],xmat,ymat,k)
    return ypred
data = pd.read_csv('LR.csv')
colA = np.array(data.colA)
colB = np.array(data.colB)
mcolA = np.mat(colA)
mcolB = np.mat(colB)
m = np.shape(mcolA)[1]
one = np.ones((1,m), dtype=int)
X = np.hstack((one.T,mcolA.T))
print(X.shape)
ypred = localWeightRegression(X,mcolB,0.5)
SortIndex = X[:,1].argsort(0)
xsort = X[SortIndex][:,0]
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.scatter(colA,colB, color='green')
ax.plot(xsort[:,1],ypred[SortIndex], color = 'red', linewidth=5)
plt.xlabel('colA')
plt.ylabel('colB')
```

Input csv file:

LR.csv

`

Output: