CONTENTS

Ex.No: 1. IMPLEMENTATION OF UNINFORMED SEARCH ALGORITHMS

(BFS, DFS)

AIM: To implement the Uninformed Search strategies such as Breadth first and Depth first search Algorithms.

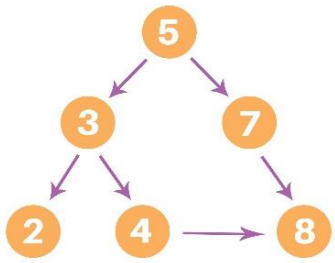
(i) Depth-First Search algorithm.

Algorithm:

The recursive method of the Depth-First Search algorithm is implemented using stack. A standard Depth-First Search implementation puts every vertex of the graph into one in all 2 categories: 1) Visited 2) Not Visited. The only purpose of this algorithm is to visit all the vertex of the graph avoiding cycles.

The DSF algorithm follows as:

1. We will start by putting any one of the graph's vertex on top of the stack.
2. After that take the top item of the stack and add it to the visited list of the vertex.
3. Next, create a list of that adjacent node of the vertex. Add the ones which aren't in the visited list of vertexes to the top of the stack.
4. Lastly, keep repeating steps 2 and 3 until the stack is empty.



PROGRAM:

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}
visited = set() # Set to keep track of visited nodes of graph.
def dfs(visited, graph, node):  #function for dfs
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

Explanation:

In the above code, first, we will create the graph for which we will use the depth-first search. After creation, we will create a set for storing the value of the visited nodes to keep track of the visited nodes of the graph.

After the above process, we will declare a function with the parameters as visited nodes, the graph itself and the node respectively. And inside the function, we will check whether any node of the graph is visited or not using the “if” condition. If not, then we will print the node and add it to the visited set of nodes.

Then we will go to the neighboring node of the graph and again call the DFS function to use the neighbor parameter.

At last, we will run the driver code which prints the final result of DFS by calling the DFS the first time with the starting vertex of the graph.

(ii) BFS Algorithm

As breadth-first search is the process of traversing each node of the graph, a standard BFS algorithm traverses each vertex of the graph into two parts: 1) Visited 2) Not Visited. So, the purpose of the algorithm is to visit all the vertex while avoiding cycles.

BFS starts from a node, then it checks all the nodes at distance one from the beginning node, then it checks all the nodes at distance two, and so on. So as to recollect the nodes to be visited, BFS uses a queue.

The steps of the algorithm work as follow:

1. Start by putting any one of the graph's vertices at the back of the queue.
2. Now take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add those which are not within the visited list to the rear of the queue.
4. Keep continuing steps two and three till the queue is empty.

Many times, a graph may contain two different disconnected parts and therefore to make sure that we have visited every vertex, we can also run the BFS algorithm at every node.

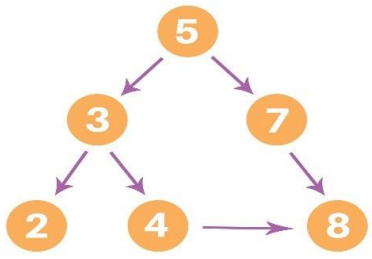


FIGURE 0

PROGRAM

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}
visited = []
queue = []
def bfs(visited, graph, node):
  visited.append(node)
  queue.append(node)
  while queue:
    m = queue.pop(0)
    print (m, end = " ")
    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)
print("Following is the Breadth-First Search")
bfs(visited, graph, '5')
```

Explanation:

In the above code, first, we will create the graph for which we will use the breadth-first search. After creation, we will create two lists, one to store the visited node of the graph and another one for storing the nodes in the queue.

After the above process, we will declare a function with the parameters as visited nodes, the graph itself and the node respectively. And inside a function, we will keep appending the visited and queue lists.

Then we will run the while loop for the queue for visiting the nodes and then will remove the same node and print it as it is visited.

At last, we will run the for loop to check the not visited nodes and then append the same from the visited and queue list.

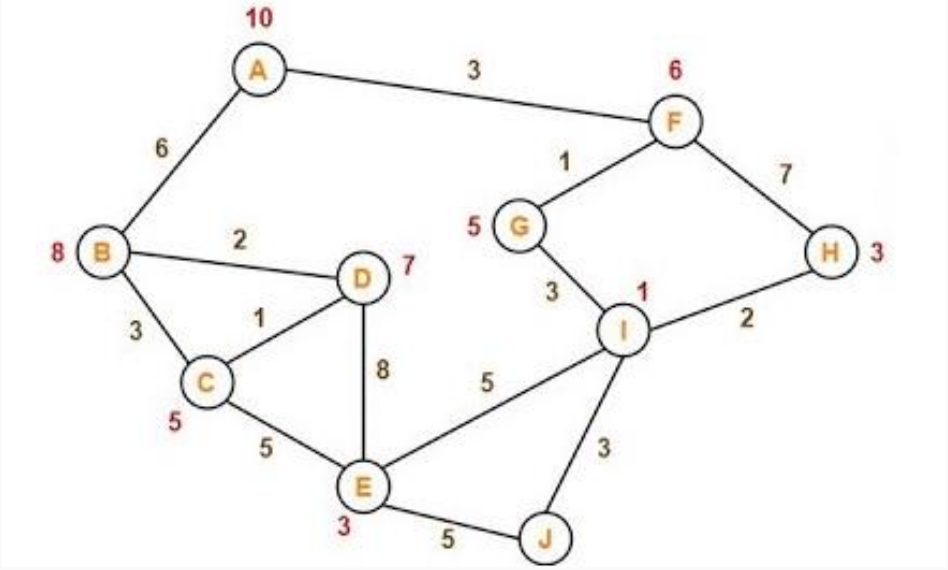
As the driver code, we will call the user to define the bfs function with the first node we wish to visit.

Ex.No :2 IMPLEMENTATION OF A* ALGORITHM

Aim: To Implement Heuristic Search Strategy A* Algorithm to find the Shortest solution path in decision tree.

Algorithm:

1. Initialize the open list
2. Initialize the closed list put the starting node on the open list (you can leave its f at zero)
3. while the open list is not empty
   a) find the node with the least f on the open list, call it "q"
   b) pop q off the open list
   c) generate q's 8 successors and set their parents to q
   d) for each successor
   i) if successor is the goal, stop search
   ii) else, compute both g and h for successor
   successor.g = q.g + distance between successor and q
   successor.h = distance from goal to successor (This can be done using many ways, we will discuss three heuristics- Manhattan, Diagonal and Euclidean Heuristics)
   successor.f = successor.g + successor.h
   iii) if a node with the same position as successor is in the OPEN list which has a lower f than successor, skip this successor
   iv) if a node with the same position as successor is in the CLOSED list which has
   a lower f than successor, skip this successor otherwise, add the node to the open list
   end (for loop)
   e) push q on the closed list end (while loop)



PROGRAM

```
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

    #distance 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': 11,
        'B': 6,
        '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': [('A', 6), ('C', 3), ('D', 2)],
    'C': [('B', 3), ('D', 1), ('E', 5)],
```

```
    'D': [('B', 2), ('C', 1), ('E', 8)],

    'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],

    'F': [('A', 3), ('G', 1), ('H', 7)],

    'G': [('F', 1), ('I', 3)],

    'H': [('F', 7), ('I', 2)],

    'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],

}

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

Output:

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

Ex No: 3 IMPLEMENT NAÏVE BAYES MODELS

Aim: To implement Naïve bayes models of Machine Learning.

Algorithm:

Step 1: Calculate the prior probability for given class labels

Step 2: Find Likelihood probability with each attribute for each class

Step 3: Put these value in Bayes Formula and calculate posterior probability.

Step 4: See which class has a higher probability, given the input belongs to the higher probability class.

Program:

```
#Import scikit-learn dataset library

from sklearn import datasets

#Load dataset

wine = datasets.load_wine()

# print the names of the 13 features

print("Features: ", wine.feature_names)

# print the label type of wine(class_0, class_1, class_2)

print("Labels: ", wine.target_names)

# print data(feature)shape

wine.data.shape

# print the wine data features (top 5 records)

print(wine.data [0:5])

# print the wine labels (0: Class,0, 1:class_2, 2:class_2)

print(wine.target)
```

```
# Import train_test_split function

from sklearn.model_selection import train_test_split

# Split dataset into training set and test set

X_train, X_test, y_train, y_test = train_test_split (wine.data, wine.target, test_size=0.3, random_state=109)

# 70% training and 30% test

#Import Gaussian Naive Bayes model

from sklearn.naive_bayes import GaussianNB

#Create a Gaussian Classifier

gnb = GaussianNB()

#Train the model using the training sets

gnb.fit(X_train, y_train)

#Predict the response for test dataset

y_pred = gnb.predict(X_test)

# Evaluating model

#Import scikit-learn metrics module for accuracy calculation

from sklearn import metrics

# Model Accuracy

print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

Output:

Display features and labels in the dataset:

Features: ['alcohol', 'malic_acid', 'ash', 'alcalinity_of_ash', 'magnesium', 'total_phenols', 'flavanoids', 'nonflavanoid_phenols', 'proanthocyanins', 'color_intensity', 'hue', 'od280/od315_of_diluted_wines', 'proline']

Labels: ['class_0' 'class_1' 'class_2']

Display the shape of the dataset:

(178, 13)

Display the top 5 records in the dataset:

```
[[1.423e+01 1.710e+00 2.430e+00 1.560e+01 1.270e+02 2.800e+00 3.060e+00
2.800e-01 2.290e+00 5.640e+00 1.040e+00 3.920e+00 1.065e+03]
[1.320e+01 1.780e+00 2.140e+00 1.120e+01 1.000e+02 2.650e+00 2.760e+00
2.600e-01 1.280e+00 4.380e+00 1.050e+00 3.400e+00 1.050e+03]
[1.316e+01 2.360e+00 2.670e+00 1.860e+01 1.010e+02 2.800e+00 3.240e+00
3.000e-01 2.810e+00 5.680e+00 1.030e+00 3.170e+00 1.185e+03]
[1.437e+01 1.950e+00 2.500e+00 1.680e+01 1.130e+02 3.850e+00 3.490e+00
2.400e-01 2.180e+00 7.800e+00 8.600e-01 3.450e+00 1.480e+03]
[1.324e+01 2.590e+00 2.870e+00 2.100e+01 1.180e+02 2.800e+00 2.690e+00
3.900e-01 1.820e+00 4.320e+00 1.040e+00 2.930e+00 7.350e+02]]
```

Display the labels in the dataset:

```
[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 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 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]
```

Model Accuracy:

Accuracy: 0.9074074074074074

Ex.No : 4 IMPLEMENTATION OF BAYESIAN NETWORK

Aim:

To Implement Bayesian network.

Algorithm:

1. Identify which are the main variable in the problem to solve. ...
2. Define structure of the network, that is, the causal relationships between all the variables (nodes).
3. Define the probability rules governing the relationships between the variables.

Program

Install Packages

```
pip install pgmpy

pip install networkx
```

Program

```
from pgmpy.models import BayesianNetwork

from pgmpy.factors.discrete import TabularCPD

import networkx as nx

import pylab as plt

# Defining Bayesian Structure

model = BayesianNetwork([('Guest', 'Host'), ('Price', 'Host')])

# Defining the CPDs:

cpd_guest = TabularCPD('Guest', 3, [[0.33], [0.33], [0.33]])

cpd_price = TabularCPD('Price', 3, [[0.33], [0.33], [0.33]])
```

```
cpd_host = TabularCPD('Host', 3, [[0, 0, 0, 0, 0.5, 1, 0, 1, 0.5], [0.5, 0, 1, 0, 0, 0, 1, 0, 0.5], [0.5, 1, 0, 1, 0.5, 0, 0, 0, 0]], evidence=['Guest', 'Price'], evidence_card=[3, 3])

# Associating the CPDs with the network structure.

model.add_cpds(cpd_guest, cpd_price, cpd_host)

model.check_model()
```

Output: True

Program

```
# Infering the posterior probability

from pgmpy.inference import VariableElimination

infer = VariableElimination(model)

posterior_p = infer.query(['Host'], evidence={'Guest': 2, 'Price': 2})

print(posterior_p)
```

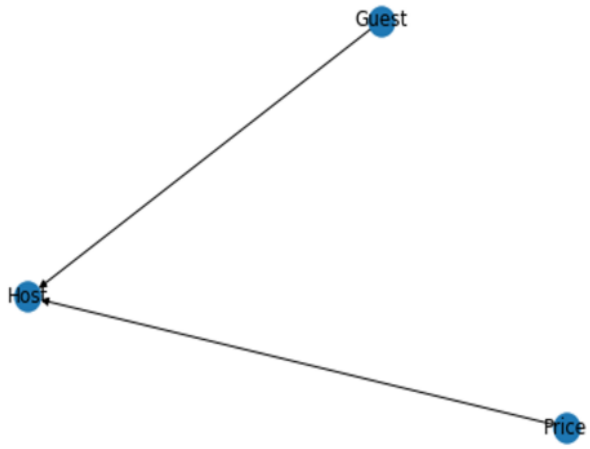
Output

```
+---------+-------------+
| Host    | phi(Host)   |
+=========+=============+
| Host(0) | 0.5000      |
+---------+-------------+
| Host(1) | 0.5000      |
+---------+-------------+
| Host(2) | 0.0000      |
+---------+-------------+
```

Program

```
nx.draw(model, with_labels=True)

plt.savefig('model.png')
```

```
plt.close()
```



Ex No: 5 IMPLEMENTATION OF LOGISTIC REGRESSION MODELS

Aim: To Implement Logistic Regression Models

Algorithm:

1. Import packages, functions, and classes
2. Get data to work with and, if appropriate, transform it
3. Create a classification model and train (or fit) it with your existing data
4. Evaluate your model to see if its performance is satisfactory

Program:

```
# importing libraries

import statsmodels.api as sm

import pandas as pd

# loading the training dataset

data = pd.read_csv('pima_diabetes.csv', index_col = 0)

# defining the dependent and independent variables

Xtrain = data[['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI', 'DiabetesPedigreeFunction','Age']]

ytrain = data[['Outcome']]

# building the model and fitting the data

log_reg = sm.Logit(ytrain, Xtrain).fit()

# printing the summary table

print(log_reg.summary())
```

Output:

```
Optimization terminated successfully.
         Current function value: 0.622121
         Iterations 5
                           Logit Regression Results
==============================================================================
Dep. Variable:                Outcome   No. Observations:                  768
Model:                          Logit   Df Residuals:                      761
Method:                           MLE   Df Model:                            6
Date:                Mon, 17 Oct 2022   Pseudo R-squ.:                 0.03815
Time:                        19:32:45   Log-Likelihood:                -477.79
converged:                       True   LL-Null:                       -496.74
Covariance Type:            nonrobust   LLR p-value:                 1.172e-06
==============================================================================================
                                 coef    std err          z      P>|z|      [0.025      0.975]
----------------------------------------------------------------------------------------------
Glucose                        0.0122      0.003      4.579      0.000       0.007       0.017
BloodPressure                 -0.0298      0.005     -6.404      0.000      -0.039      -0.021
SkinThickness               1.809e-05      0.006      0.003      0.998      -0.012       0.012
Insulin                        0.0006      0.001      0.772      0.440      -0.001       0.002
BMI                           -0.0059      0.011     -0.562      0.574      -0.027       0.015
DiabetesPedigreeFunction       0.2486      0.237      1.051      0.293      -0.215       0.712
Age                            0.0040      0.007      0.573      0.567      -0.010       0.018
==============================================================================================
```

Ex No: 6.a IMPLEMENTATION OF DECISION TREES

Aim: To Implement Decision Tree in Machine Learning

Algorithm:

Step-1: Begin the tree with the root node, says S, which contains the complete dataset.

Step-2: Find the best attribute in the dataset using Attribute Selection Measure (ASM).

Step-3: Divide the S into subsets that contains possible values for the best attributes.

Step-4: Generate the decision tree node, which contains the best attribute.

Step-5: Recursively make new decision trees using the subsets of the dataset created in step -3. Continue this process until a stage is reached where you cannot further classify the nodes and called the final node as a leaf node.

Program:

```
import pandas

from sklearn import tree

from sklearn.tree import DecisionTreeClassifier

df = pandas.read_csv("data.csv")

print("Input:")

print(df.head(5))

d = {'UK':0,'USA':1,'N':2}

df['Nationality'] = df['Nationality'].map(d)

d = {'YES':1, 'NO':0}

df['Go'] = df['Go'].map(d)

print("Transformed Data:")

print(df.head(5))
```

```
features = ['Age','Experience','Rank','Nationality']

X = df[features]

y = df['Go']

dtree = DecisionTreeClassifier()

dtree = dtree.fit(X,y)

print(dtree.predict([[40,10,6,1]]))

print("[1]means 'Go'")

print("[0]means 'NO'")
```

DATA SET : (data.csv)

Age	Experience	Rank	Nationality	Go
36	10	9	UK	NO
42	12	4	USA	NO
23	4	6	N	NO
52	4	4	USA	NO
43	21	8	USA	YES

Output:

```
Transformed Data:
   Age  Experience  Rank  Nationality  Go
0   36          10     9            0   0
1   42          12     4            1   0
2   23           4     6            2   0
3   52           4     4            1   0
4   43          21     8            1   1
[0]
[1]means 'Go'
[0]means 'NO'
```

Ex No: 6.b IMPLEMENTATION OF RANDOM FORESTS

Aim: To Implement Random Forest Algorithm of Machine Learning

Algorithm:

Step-1: Select random K data points from the training set.

Step-2: Build the decision trees associated with the selected data points (Subsets).

Step-3: Choose the number N for decision trees that you want to build.

Step-4: Repeat Step 1 & 2.

Step-5: For new data points, find the predictions of each decision tree, and assign the new data points to the category that wins the majority votes.

Program:

```
# Pandas is used for data manipulation

import pandas as pd

# Read in data and display first 5 rows

features = pd.read_csv('temps.csv')

features.head(5)

print('The shape of our features is:', features.shape)

# Descriptive statistics for each column

features.describe()

# One-hot encode the data using pandas get_dummies

features = pd.get_dummies(features)

# Display the first 5 rows of the last 12 columns

features.iloc[:,5:].head(5)

import numpy as np
```

```
# Labels are the values we want to predict
labels = np.array(features['actual'])
# Remove the labels from the features
# axis 1 refers to the columns
features= features.drop('actual', axis = 1)
# Saving feature names for later use
feature_list = list(features.columns)
# Convert to numpy array
features = np.array(features)
# Using Skicit-learn to split data into training and testing sets
from sklearn.model_selection import train_test_split
# Split the data into training and testing sets
train_features, test_features, train_labels, test_labels = train_test_split(features, labels, test_size =
0.25, random_state = 42)
print('Training Features Shape:', train_features.shape)
print('Training Labels Shape:', train_labels.shape)
print('Testing Features Shape:', test_features.shape)
print('Testing Labels Shape:', test_labels.shape)
# Import the model we are using
from sklearn.ensemble import RandomForestRegressor
# Limit depth of tree to 3 levels
rf_small = RandomForestRegressor(n_estimators=10, max_depth = 3)
# Train the model on training data
```

```
rf_small.fit(train_features, train_labels)

# Extract the small tree

tree_small = rf_small.estimators_[5]

# Save the tree as a png image

export_graphviz(tree_small, out_file = 'small_tree.dot', feature_names = feature_list, rounded =

True, precision = 1)

(graph, ) = pydot.graph_from_dot_file('small_tree.dot')

graph.write_png('small_tree.png');

# Use the forest's predict method on the test data

predictions = rf_small.predict(test_features)

# Calculate the absolute errors

errors = abs(predictions - test_labels)

# Print out the mean absolute error (mae)

print('Mean Absolute Error:', round(np.mean(errors), 2), 'degrees.')

# Calculate mean absolute percentage error (MAPE)

mape = 100 * (errors / test_labels)

# Calculate and display accuracy

accuracy = 100 - np.mean(mape)

print('Accuracy:', round(accuracy, 2), '%.')
```

Output:

	year	month	day	week	temp_2	temp_1	average	actual	forecast_noaa	forecast_acc	forecast_under	friend
0	2016	1	1	Fri	45	45	45.6	45	43	50	44	29
1	2016	1	2	Sat	44	45	45.7	44	41	50	44	61
2	2016	1	3	Sun	45	44	45.8	41	43	46	47	56
3	2016	1	4	Mon	44	41	45.9	40	44	48	46	53
4	2016	1	5	Tues	41	40	46.0	44	46	46	46	41

The shape of our features is: (348, 12)

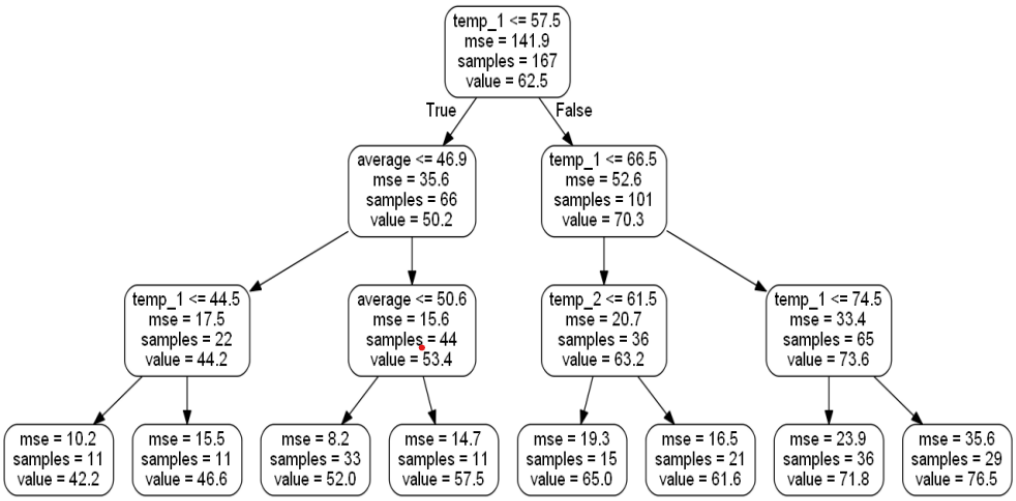
Training Features Shape: (261, 17)

Training Labels Shape: (261,)

Testing Features Shape: (87, 17)

Testing Labels Shape: (87,)

RandomForestRegressor(max_depth=3, n_estimators=10)



Mean Absolute Error: 4.0 degrees.

Accuracy: 93.73 %.

Ex No: 7 IMPLEMENMTATION OF SVM MODELS

Aim: To Implement Support Vector Machine Model of Machine Learning

Algorithm:

Step 1: Load the important libraries

Step 2: Import dataset and extract the X variables and Y separately.

Step 3: Divide the dataset into train and test

Step 4: Initializing the SVM classifier model

Step 5: Fitting the SVM classifier model

Step 6: Coming up with predictions

Step 7: Evaluating model's performance

Program:

```
import pandas

from sklearn.model_selection import train_test_split

from sklearn.svm import SVC

from sklearn.metrics import confusion_matrix

data = pandas.read_csv("vector.csv")

print("Input: ")

print(data.head(10))

training_set, test_set = train_test_split(data, test_size = 0.3, random_state=1)

x_train = training_set.iloc[:,0:2].values

y_train = training_set.iloc[:,2].values

x_test = test_set.iloc[:,0:2].values
```

```
y_test = test_set.iloc[:,2].values

classifier = SVC(kernel='linear', random_state=1)

classifier.fit(x_train, y_train)

y_pred = classifier.predict(x_test)

test_set["prediction"] = y_pred

print("Output")

print(test_set)

cm = confusion_matrix(y_test, y_pred)

accuracy = float(cm.diagonal().sum()/len(y_test))

print("\nAccuracy of SVM for the given dataset: ", accuracy)
```

Dataset

	weight	SIZE	class
0	69	4.39	orange
1	69	4.21	orange
2	65	4.09	orange
3	72	5.85	apple
4	67	4.70	orange
5	73	5.68	apple
6	70	5.56	apple
7	75	5.11	apple
8	74	5.36	apple
9	65	4.27	orange

Output:

```
Output
   weight  SIZE   class prediction
2      65  4.09  orange     orange
9      65  4.27  orange     orange
6      70  5.56   apple     orange

Accuracy of SVM for the given dataset:  0.6666666666666666
```

Ex.No: 8 IMPLEMENTATION OF ENSEMBLING TECHNIQUES

Aim: To implement Ensemble Learning Techniques

Algorithm:

1. Split the train dataset into n parts
2. A base model (say linear regression) is fitted on n-1 parts and predictions are made for the nth part. This is done for each one of the n part of the train set.
3. The base model is then fitted on the whole train dataset.
4. This model is used to predict the test dataset.
5. The Steps 2 to 4 are repeated for another base model which results in another set of predictions for the train and test dataset.
6. The predictions on train data set are used as a feature to build the new model.
7. This final model is used to make the predictions on test dataset

Program:

```
#Implement VotingClassifier

#Importing necessary libraries:

from sklearn.model_selection import train_test_split

from sklearn.datasets import make_moons

from sklearn.linear_model import LogisticRegression

from sklearn.svm import SVC

from sklearn.ensemble import RandomForestClassifier

from sklearn.ensemble import VotingClassifier

from sklearn.metrics import accuracy_score

#Creating dataset:

X, y = make_moons(n_samples=500, noise=0.30)

X_train, X_test, y_train, y_test = train_test_split(X, y)
```

```
#Initializing the models:
log = LogisticRegression()
rnd = RandomForestClassifier(n_estimators=100)
svm = SVC()
voting = VotingClassifier(
 estimators=[('logistics_regression', log), ('random_forest', rnd), ('support_vector_machine', svm)],
 voting='hard')
#Fitting training data:
voting.fit(X_train, y_train)
#prediction using test data
for clf in (log, rnd, svm, voting):
 clf.fit(X_train, y_train)
 y_pred = clf.predict(X_test)
 print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
#Implement BaggingClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
bagging_clf = BaggingClassifier(
 DecisionTreeClassifier(), n_estimators=250,
 max_samples=100, bootstrap=True, random_state=101)
#Fitting training data:
```

```
bagging_clf.fit(X_train, y_train)
#prediction using test data
y_pred = bagging_clf.predict(X_test)
print(accuracy_score(y_test, y_pred))
#Implement AdaBoostClassifier
from sklearn.ensemble import AdaBoostClassifier
adaboost_clf = AdaBoostClassifier(
 DecisionTreeClassifier(max_depth=1), n_estimators=200,
 algorithm="SAMME.R", learning_rate=0.5, random_state=42)
#Fitting training data:
adaboost_clf.fit(X_train, y_train)
#prediction using test data
y_pred = adaboost_clf.predict(X_test)
accuracy_score(y_test, y_pred)
```

Output:

```
#For VotingClassifier
LogisticRegression 0.848
RandomForestClassifier 0.88
SVC 0.896
VotingClassifier 0.896
#For BaggingClassifier
0.888
#For AdaBoostClassifier      0.864
```

Ex No: 9 IMPLEMENTATION OF CLUSTERING ALGORITHMS

Aim: To implement Clustering Algorithms in Machine Learning

Algorithm:

Step 1: Prepare Data. As with any ML problem, It must be normalize, scale, and transform feature data.

Step 2: Create Similarity Metric. Before a clustering algorithm can group data, it needs to know how similar pairs of examples are.

Step 3: Run Clustering Algorithm.

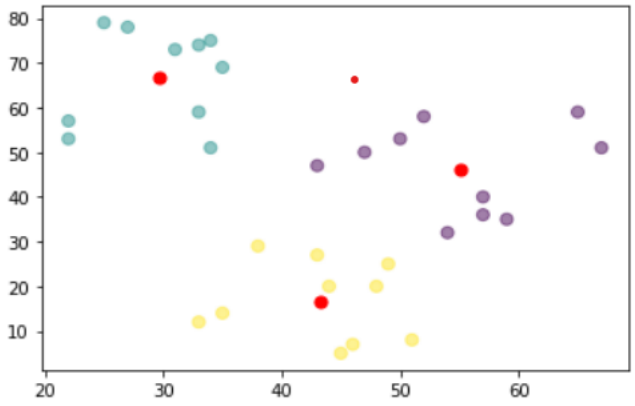
Step 4: Interpret Results and Adjust.

Program:

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
data = {'x':
[25,34,22,27,33,33,31,22,35,34,67,54,57,43,50,57,59,52,65,47,49,48,35,33,44,45,38,43,51,4
6],
 'y': [79,51,53,78,59,74,73,57,69,75,51,32,40,47,53,36,35,58,59,50,25,20,14,12,20,5,29,27,8,7] }
 df = pd.DataFrame(data, columns=['x', 'y'])
kmeans = KMeans(n_clusters=3).fit(df)
centroids = kmeans.cluster_centers_
print(centroids)
plt.scatter(df['x'], df['y'], c= kmeans.labels_.astype(float), s=50, alpha=0.5)
plt.scatter(centroids[:, 0], centroids[:, 1], c='red', s=50)
plt.show()
```

Output:

```
[[55.1 46.1]
 [29.6 66.8]
 [43.2 16.7]]
```



Ex No: 10 **IMPLEMENT GMM ALGORITHMS**

Aim:

To implement Gaussian Mixture Model (GMM) algorithm of Machine Learning

Algorithm:

1. Decide the number of clusters (to decide this, we can use domain knowledge or other methods such as BIC/AIC) for the given dataset. Assume that we have 1000 data points, and we set the number of groups as 2.
2. Initiate mean, covariance, and weight parameter per cluster. (we will explore more about this in a later section)
3. Use the Expectation Maximization algorithm to do the following,

- Expectation Step (E step): Calculate the probability of each data point belonging to each distribution, then evaluate the likelihood function using the current estimate for the parameters
- Maximization step (M step): Update the previous mean, covariance, and weight parameters to maximize the expected likelihood found in the E step
- Repeat these steps until the model converges.

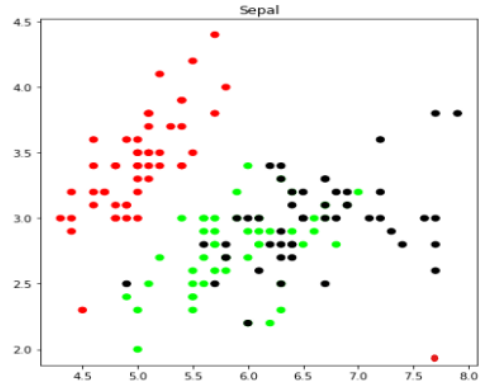
Program:

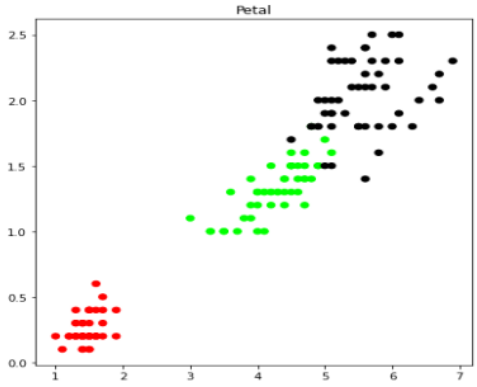
```
import matplotlib.pyplot as plt

from sklearn import datasets

import sklearn.metrics as sm

import pandas as pd

import numpy as np

%matplotlib inline

# import some data to play with
```

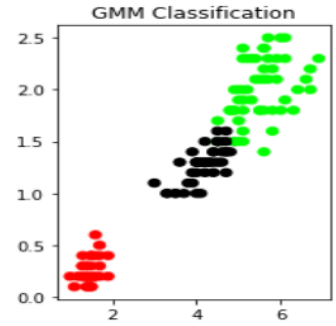
```
iris = datasets.load_iris()

#print("\n IRIS DATA :",iris.data);

#print("\n IRIS FEATURES :\n",iris.feature_names)

#print("\n IRIS TARGET :\n",iris.target)

#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']

#print(X.columns)

#print("X:",x)

#print("Y:",y)

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')
# GMM
from sklearn import preprocessing
scaler = preprocessing.StandardScaler()
scaler.fit(X)
xsa = scaler.transform(X)
xs = pd.DataFrame(xsa, columns = X.columns)
xs.sample(5)
from sklearn.mixture import GaussianMixture
gmm = GaussianMixture(n_components=3)
gmm.fit(xs)
y_cluster_gmm = gmm.predict(xs)
y_cluster_gmm
plt.subplot(1, 2, 1)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y_cluster_gmm], s=40)
plt.title('GMM Classification')
# Accuracy
sm.accuracy_score(y, y_cluster_gmm)
# Confusion Matrix
sm.confusion_matrix(y, y_cluster_gmm)
```

Output:







array([[50, 0, 0],

[0, 5, 45],

[0, 50, 0]], dtype=int64)

Ex No: 11 BUILD A SIMPLE NEURAL NETWORKS MODELS

Aim:

To Implement a simple neural network models of Machine Learning.

Algorithm:

1. Import the libraries. For example: import numpy as np
2. Define/create input data. For example, use numpy to create a dataset and an array of data values.
3. Add weights and bias (if applicable) to input features. These are learnable parameters, meaning that they can be adjusted during training.
    - Weights = input parameters that influences output
    - Bias = an extra threshold value added to the output
4. Train the network against known, good data in order to find the correct values for the weights and biases.
5. Test the Network against a set of test data to see how it performs.
6. Fit the model with hyperparameters (parameters whose values are used to control the learning process), calculate accuracy, and make a prediction.

Program:

```
# Import python libraries required in this example:

from keras.models import Sequential

from keras.layers import Dense, Activation

import numpy as np

# Use numpy arrays to store inputs (x) and outputs (y):

x = np.array([[0,0], [0,1], [1,0], [1,1]])

y = np.array([[0], [1], [1], [0]])
```

```
# Define the network model and its arguments.
# Set the number of neurons/nodes for each layer:
model = Sequential()
model.add(Dense(2, input_shape=(2,)))
model.add(Activation('sigmoid'))
model.add(Dense(1))
model.add(Activation('sigmoid'))
# Compile the model and calculate its accuracy:
model.compile(loss='mean_squared_error', optimizer='sgd', metrics=['accuracy'])
# Print a summary of the Keras model:
model.summary()
```

Output:

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 2)	6
activation (Activation)	(None, 2)	0
dense_1 (Dense)	(None, 1)	3
activation_1 (Activation)	(None, 1)	0

Total params: 9
Trainable params: 9
Non-trainable params: 0

Ex No: 12 BUILD A DEEP LEARNING NEURAL NETWORKS MODELS

Aim:

To implement a Deep Learning Neural networks models

Algorithm:

Step 1: Import all the required library

Step 2: Define Vector Variables for Input and Output

Step 3: Define Weight Variable

Step 4: Define placeholders for Input and Output

Step 5: Calculate Output and Activation Function

Step 6: Calculate the Cost or Error

Step 7: Minimize Error

Step 8: Initialize all the variables

Step 9: Training Perceptron in Iterations

Program:

```
import tensorflow as tf

from tensorflow import keras

fashiondata=tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test)=fashiondata.load_data()

x_test.shape

x_train.shape

x_train, x_test=x_train/255, x_test/255

model = tf.keras.models.Sequential([tf.keras.layers.Flatten(input_shape=(28,28)),

 tf.keras.layers.Dense(128,activation='relu'),

 tf.keras.layers.Dropout(0.2),
```

```
 tf.keras.layers.Dense(10,activation='softmax')])

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)

model.evaluate(x_test, y_test)
```

Output:(10000, 28, 28)
(60000, 28, 28)

```
Epoch 1/5
1875/1875 [==============================] - 7s 3ms/step - loss: 0.0672 - accuracy: 0.97
93
Epoch 2/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0578 - accuracy: 0.98
11
Epoch 3/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0528 - accuracy: 0.98
25
Epoch 4/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0500 - accuracy: 0.98
33
Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0453 - accuracy: 0.98
44
313/313 [==============================] - 1s 3ms/step - loss: 0.0697 - accuracy: 0.9797
[0.06965507566928864, 0.9797000288963318]
```