ANNAI MIRA COLLEGE OF ENGINEERING AND TECHNOLOGY

NH-46, Chennai-Bengaluru National Highways, Arapakkam,

Ranipet-632517, Tamil Nadu, India

Telephone: 04172-292925 Fax: 04172-292926

Email:amcet.rtet@gmail.com/info@amcet.in Web: www.amcet.in

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CS3491 – ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING LABORATORY

Name : …….

Register Number : …….

Year & Branch : …….

Semester : ……

Academic Year : …….....

ANNAI MIRA COLLEGE OF ENGINEERING AND TECHNOLOGY

NH-46, Chennai-Bengaluru National Highways, Arapakkam,

Ranipet-632517, Tamil Nadu, India

Telephone: 04172-292925 Fax: 04172-292926

CERTIFICATE

This is to Certify that the Bonafide record of the practical work done by…… Register Number…….. of IInd year B.E (Computer Science and Engineering) submitted for the B.E-Degree practical examination(IVth Semester) in **CS3491 - ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING LABORATORY** during the academic year **2022 – 2023.**

Staff in –Charge **Head of the Department**

Submitted for the practical examination held on ----------

Internal Examiner **External Examiner**

TABLE OF CONTENT

Ex. No:1 Date:	Implementation of Uninformed search Algorithms (BFS, DFS)

AIM:

To write a program in python to solve problems by using Implementation of Uninformed search algorithms (BFS, DFS)

ALGORITHM:

1. Create an empty queue (for BFS) or stack (for DFS) and add the initial state to it.
2. Create an empty set to store visited states.
3. While the queue (or stack) is not empty:
    - Remove the first state from the queue (or the last state from the stack).
    - If the state is the goal state, return the path from the initial state to the current state.
    - Otherwise, generate all possible actions from the current state.
    - For each action, generate the resulting state and check if it has been visited before. If it has not been visited, add it to the queue (or stack) and mark it as visited.
4. If the queue (or stack) is empty and no goal state has been found, return failure.

PROGRAM:

```
# Graph representation using a dictionary
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}
# BFS algorithm
def bfs(graph, start, end):
# Keep track of visited nodes
visited = []
    # Create a queue for BFS
queue = [start]     while
queue:
        # Dequeue a vertex from queue
node = queue.pop(0)
if node not in visited:
visited.append(node)
            # Get all adjacent nodes of the dequeued node
for neighbor in graph[node]:
queue.append(neighbor)
            # Check if the end node has been reached
if node == end:
  return visited
return visited
# DFS algorithm
def dfs(graph, start, end, visited=[]):
# Mark the source node as visited and print it
visited.append(start)
    # Get all neighbors of the current node
for neighbor in graph[start]:
 if neighbor not in visited:
            dfs(graph, neighbor, end, visited)
# Check if the end node has been reached
if start == end:
return visited
return visited
```

OUTPUT:

```
BFS:  ['A', 'B', 'C', 'D', 'E', 'F']
DFS:  ['A', 'B', 'D', 'E', 'F', 'C']
```

RESULT:

Thus the python program has been written and executed successfully.

Ex. No:2A Date:	Implementation of Informed search Algorithms A*

AIM:

To write a program in python to solve problems by using Implementation of Informed search Algorithms A*

ALGORITHM:

1. Create an open set and a closed set, both initially empty.

2. Add the initial state to the open set with a cost of 0 and an estimated total cost (f-score) of the heuristic value of the initial state.

3. While the open set is not empty:

    - Choose the state with the lowest f-score from the open set.
    - If this state is the goal state, return the path from the initial state to this state.
    - Generate all possible actions from the current state.
    - For each action, generate the resulting state and compute the cost to get to that state by adding the cost of the current state plus the cost of the action.
    - If the resulting state is not in the closed set or the new cost to get there is less than the old cost, update its cost and estimated total cost in the open set and add it to the open set.
    - Add the current state to the closed set.

4. If the open set is empty and no goal state has been found, return failure.

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
            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 does not 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': 11,
        'B': 6,
        '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': [('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']
```

RESULT:

Thus the python program has been written and executed successfully**.**

Ex. No:2B Date:	Informed search Algorithms memory-bounded A*

AIM:

To write a program in python to solve problems by using Implementation of Informed search Algorithms memory-bounded A*

ALGORITHM:

1. Create an open set and a closed set, both initially empty.
2. Add the initial state to the open set with a cost of 0 and an estimated total cost (f-score) of the heuristic value of the initial state.
3. While the open set is not empty:
    a. Choose the state with the lowest f-score from the open set.
    b. If this state is the goal state, return the path from the initial state to this state.
    c. Generate all possible actions from the current state.
    d. For each action, generate the RESULTing state and compute the cost to get to that state by adding the cost of the current state plus the cost of the action.
    e. If the RESULTing state is not in the closed set or the new cost to get there is less than the old cost, update its cost and estimated total cost in the open set and add it to the open set.
    f. Add the current state to the closed set.
    g. If the size of the closed set plus the open set exceeds the maximum memory usage, remove the state with the highest estimated total cost from the closed set and add it back to the open set.
4. If the open set is empty and no goal state has been found, return failure.

PROGRAM:

```
from queue import PriorityQueue import sys

def memory_bounded_a_star(start_node, goal_node, max_memory):
   frontier = PriorityQueue()
frontier.put((0, start_node))
explored = set()
total_cost = {start_node: 0}

   while not frontier.empty():
      # Check if memory limit has been reached
if sys.getsizeof(explored) > max_memory:
         return None

      _, current_node = frontier.get()

      if current_node == goal_node:
         path = []
while current_node != start_node:
path.append(current_node)
current_node = current_node.parent
path.append(start_node)
         path.reverse()
return path

      explored.add(current_node)

      for child_node, cost in current_node.children():
         if child_node in explored:
            continue

         new_cost = total_cost[current_node] + cost

         if child_node not in total_cost or new_cost < total_cost[child_node]:
            total_cost[child_node] = new_cost
            priority = new_cost + child_node.heuristic(goal_node)
frontier.put((priority, child_node))

   return None class Node:     def
__init__(self, state, parent=None):
```

```
        self.state = state
self.parent = parent        self.cost
= 1

    def __eq__(self, other):
        return self.state == other.state

    def __hash__(self):
        return hash(self.state)

    def heuristic(self, goal):
        # Simple heuristic for demonstration purposes
return abs(self.state - goal.state)

    def children(self):
        # Generates all possible children of a given node
children = []        for action in [-1, 1]:            child_state
= self.state + action
child_node = Node(child_state, self)
children.append((child_node, child_node.cost))
return children

# Example usage
start_node = Node(1)
goal_node = Node(10)

path = memory_bounded_a_star(start_node, goal_node, max_memory=1000000) if
path is None:     print("Memory limit exceeded.")
else:
    print([node.state for node in path])
```

OUTPUT:

Path: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

RESULT:

Thus the python program has been written and executed successfully.

Ex. No:3A Date:	Implementation of Naive Bayes models (Gaussian Naive Bayes)

AIM:

To write a program in python to solve problems by using Naive Bayes model (Gaussian Naive Bayes)

ALGORITHM:

Input:

• Training dataset with features X and corresponding labels Y
• Test dataset with features X_test

Output:

• Predicted labels for test dataset Y_pred

Steps:

1. Calculate the prior probabilities of each class in the training dataset, i.e., P(Y = c), where c is the class label.

2. Calculate the mean and variance of each feature for each class in the training dataset.

3. For each test instance in X_test, calculate the posterior probability of each class c, i.e., P(Y = c | X = x_test), using the Gaussian probability density function: P(Y = c | X = x_test) = (1 / (sqrt(2*pi)*sigma_c)) * exp(-((x_test - mu_c)^2) / (2 * sigma_c^2)) where mu_c and sigma_c are the mean and variance of feature values for class c, respectively.

4. For each test instance in X_test, assign the class label with the highest posterior probability as the predicted label Y_pred.

5. Return Y_pred as the output.

PROGRAM:

```
from sklearn.naive_bayes import GaussianNB from
sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load iris dataset data =
load_iris()
X, y = data.data, data.target

# Split dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train Gaussian Naive Bayes model gnb =
GaussianNB()
gnb.fit(X_train, y_train)

# Predict labels for test set
y_pred = gnb.predict(X_test)

# Calculate accuracy of predictions
accuracy = accuracy_score(y_test, y_pred)

# Print results
print("Accuracy:", accuracy)
```

OUTPUT:

```
Accuracy: 1.0
```

RESULT:

Thus the python program has been written and executed successfull

Ex. No:3B Date:	Implementation of Naive Bayes models (Multinomial Naive Bayes)

AIM:

To write a program in python to solve problems by using Naive Bayes model (Multinomial Naive Bayes)

ALGORITHM:

1. Convert the training dataset into a frequency table where each row represents a document, and each column represents a word in the vocabulary. The values in the table represent the frequency of each word in each document.
2. Calculate the prior probabilities of each class label by dividing the number of documents in each class by the total number of documents.
3. Calculate the conditional probabilities of each word given each class label. This involves calculating the frequency of each word in each class and dividing it by the total number of words in that class.
4. For each document in the test dataset, calculate the posterior probability of each class label using the Naive Bayes formula:
5. P(class_label | document) = P(class_label) * P(word1 | class_label) * P(word2 | class_label) * ... * P(wordn | class_label)
6. Where word1, word2, ..., wordn are the words in the document and P(word | class_label) is the conditional probability of that word given the class label.
7. Predict the class label with the highest posterior probability for each document in the test dataset.
8. Return the predicted class labels for the test dataset

PROGRAM:

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import CountVectorizer

# Sample training data
train_data = ["this is a positive example", "this is a negative example", "another negative example",
"yet another negative example"]
train_labels = ["positive", "negative", "negative", "negative"]

# Sample test data
test_data = ["this is a test", "this example is negative"]

# Create a CountVectorizer to convert the text data into numerical features
vectorizer = CountVectorizer()

# Fit the vectorizer to the training data and transform the data
train_features = vectorizer.fit_transform(train_data)

# Create a Multinomial Naive Bayes classifier and train it on the training data
clf = MultinomialNB()
clf.fit(train_features, train_labels)

# Transform the test data using the same vectorizer
test_features = vectorizer.transform(test_data)

# Use the trained classifier to predict the class labels for the test data
predicted_labels = clf.predict(test_features)

# Print the predicted class labels for the test data
print(predicted_labels)
```

OUTPUT:

```
['negative' 'negative']
```

RESULT:

Thus the python program has been written and executed successfully.

Ex. No: 4 Date:	Implementation of Bayesian Networks

AIM:

To write a program in python to solve problems by using Bayesian Networks.

ALGORITHM:

1. Import the necessary libraries: pgmpy.models, pgmpy.factors.discrete, and pgmpy.inference.
2. Define the structure of the Bayesian network by creating a BayesianModel object and specifying the nodes and their dependencies.
3. Define the conditional probability distributions (CPDs) for each node using the TabularCPD class.
4. Add the CPDs to the model using the add_cpds method.
5. Check if the model is valid using the check_model method. If the model is not valid, an error message will be raised.
6. Create a VariableElimination object using the model.
7. Use the inference.query method to compute the probability of the Letter node being good given that the Intelligence node is high and the Difficulty node is low.
8. Print the probability value.

PROGRAM:

```
from pgmpy.models import BayesianModel
from pgmpy.factors.discrete import TabularCPD
from pgmpy.inference import VariableElimination

# Define the structure of the Bayesian network
model = BayesianModel([('Difficulty', 'Grade'), ('Intelligence', 'Grade'), ('Grade', 'Letter'), ('Grade',
'SAT')])

# Define the conditional probability distributions (CPDs)
cpd_difficulty = TabularCPD(variable='Difficulty', variable_card=2, values=[[0.6, 0.4]])

cpd_intelligence = TabularCPD(variable='Intelligence', variable_card=2, values=[[0.7,
0.3]])

cpd_grade = TabularCPD(variable='Grade', variable_card=3, values=[[0.3, 0.05, 0.9, 0.5],
[0.4, 0.25, 0.08, 0.3], [0.3, 0.7, 0.02, 0.2]], evidence=['Difficulty', 'Intelligence'], evidence_card=[2, 2])

cpd_letter = TabularCPD(variable='Letter', variable_card=2, values=[[0.1, 0.4, 0.99], [0.9,
0.6, 0.01]], evidence=['Grade'], evidence_card=[3])

cpd_sat = TabularCPD(variable='SAT', variable_card=2, values=[[0.1, 0.4, 0.99], [0.9, 0.6,
0.01]], evidence=['Grade'], evidence_card=[3])

# Add the CPDs to the model
model.add_cpds(cpd_difficulty, cpd_intelligence, cpd_grade, cpd_letter, cpd_sat)

# Check if the model is valid
print("Model is valid:", model.check_model())

# Perform variable elimination to compute the probability of getting a good letter given high
intelligence and low difficulty
inference = VariableElimination(model)
query = inference.query(variables=['Letter'], evidence={'Intelligence': 1, 'Difficulty': 0},
show_progress=False)
print("P(Letter=Good | Intelligence=High, Difficulty=Low) =", query.values[0])
```

OUTPUT:

Model is valid: True
P(Letter=Good | Intelligence=High, Difficulty=Low) = 0.368

RESULT:

Thus the python program has been written and executed successfully

Ex. No: 5 Date:	Build Regression Models

AIM:

To write a python program to build Regression Models

ALGORITHM:

1. Load the data from a CSV file using pandas.
2. Split the data into features and target variables.
3. Split the data into training and testing sets using the **train_test_split** function from scikitlearn.
4. Train a linear regression model using the training data by creating an instance of the **LinearRegression** class and calling its **fit** method with the training data.
5. Evaluate the performance of the model using mean squared error on both the training and testing data. Print the RESULTs to the console.

PROGRAM:

```
import numpy as np
import pandas as pd from sklearn.linear_model
import LinearRegression from sklearn.model_selection
import train_test_split from sklearn.metrics
import mean_squared_error

# Load data
data = pd.read_csv('data.csv')

# Split data into features and target X
= data.drop('target', axis=1) y =
data['target']

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train linear regression model reg =
LinearRegression()
reg.fit(X_train, y_train)

# Evaluate model train_pred = reg.predict(X_train) test_pred =
reg.predict(X_test) print('Train MSE:',
mean_squared_error(y_train, train_pred)) print('Test MSE:',
mean_squared_error(y_test, test_pred))
```

OUTPUT:

```
Train MSE: 0.019218
Test MSE: 0.022715
```

RESULT:

Thus the python program has been written and executed successfully.

Ex. No: 6A Date:	Build Decision trees

AIM:

To write a python program to solve build decision trees.

ALGORITHM:

Step 1: Collect and preprocess data

- Collect the dataset and prepare it for analysis
- Handle missing data and outliers
- Convert categorical variables to numerical values (if needed)

Step 2: Determine the root node

- Choose a feature that provides the most information gain (reduces uncertainty)
- Split the dataset based on the selected feature

Step 3: Build the tree recursively

- For each subset of the data, repeat steps 1 and 2
- Continue until each subset is either pure (only one class label) or too small to split further

Step 4: Prune the tree (optional)

- Remove branches that do not improve the model's accuracy
- Prevent overfitting by reducing the complexity of the tree

Step 5: Evaluate the performance of the tree

- Use a separate validation set to estimate the accuracy of the model
- Adjust hyper parameters to optimize the performance

PROGRAM:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score

# Load the dataset
dataset = pd.read_csv('housing.csv')

# Split the dataset into training and testing sets
X = dataset.drop('MEDV', axis=1)
y = dataset['MEDV']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train the model
model = LinearRegression()
model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = model.predict(X_test)

# Evaluate the model
r2 = r2_score(y_test, y_pred)
print('Linear Regression R^2:', r2)
```

OUTPUT:

```
Decision Tree Accuracy: 1.0
```

RESULT:

Thus the python program has been written and executed successfully.

Ex. No: 6B Date:	Build Random forests

AIM:

To write a python program to build random forests.

ALGORITHM:

Step 1: Collect and preprocess data

- Collect the dataset and prepare it for analysis
- Handle missing data and outliers
- Convert categorical variables to numerical values (if needed)

Step 2: Randomly select features

- Choose a number of features to use at each split
- Randomly select that many features from the dataset

Step 3: Build decision trees on subsets of the data

- For each subset of the data, repeat steps 1 and 2
- Build a decision tree using the selected features and split criteria

Step 4: Aggregate the predictions of the trees

- For a new data point, pass it through each tree in the forest
- Aggregate the predictions of all trees (e.g., by majority vote)

Step 5: Evaluate the performance of the forest

- Use a separate validation set to estimate the accuracy of the model
- Adjust hyper parameters to optimize the performance

PROGRAM:

```
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Load the dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train the model
model = RandomForestClassifier(n_estimators=100, random_state=42) model.fit(X_train,
y_train)

# Make predictions on the test set
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print('Random Forest Accuracy:', accuracy)
```

OUTPUT:

```
Random Forest Accuracy: 1.0
```

RESULT:

Thus the python program has been written and executed successfully.

Ex. No: 7 Date:	Build SVM models

AIM:

To write a python program to build SVM Model's

ALGORITHM:

1. Import necessary libraries

2. Load the dataset

3. Split the dataset into training and testing sets

4. Create an SVM model
   a. Specify the kernel (e.g., linear, polynomial, radial basis function)
   b. Specify the regularization parameter (C)
   c. Specify the gamma value (if applicable) 5. Train the SVM model using the training data

6. Test the SVM model using the testing data

7. Evaluate the accuracy of the SVM model
   a. Predict the target values for the testing data
   b. Calculate the accuracy of the model using the score function

8. Print the accuracy of the SVM model

PROGRAM:

```
# Import necessary libraries
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Load the iris dataset
iris = datasets.load_iris()

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.3, random_state=42)

# Initialize the SVM classifier with a linear kernel
clf = SVC(kernel='linear')

# Fit the classifier to the training data
clf.fit(X_train, y_train)

# Make predictions on the testing data
y_pred = clf.predict(X_test)

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)

# Print the accuracy score
print("Accuracy:", accuracy)
```

OUTPUT:

```
Accuracy: 1.
```

RESULT:

Thus the python program has been written and executed successfully.

Ex. No: 8 Date:	Implement Ensembling techniques

AIM:

To write a python program to solve Implement ensembling techniques

ALGORITHM:

1. Load the breast cancer dataset and split the data into training and testing sets using train_test_split() function.
2. Train 10 random forest models using bagging by randomly selecting 50% of the training data for each model, and fit a random forest classifier with 100 trees to the selected data.
3. Test each model on the testing set and calculate the accuracy of each model using accuracy_score() function.
4. Combine the predictions of the 10 models by taking the average of the predicted probabilities for each class, round the predicted probabilities to the nearest integer, and calculate the accuracy of the ensemble model using accuracy_score() function.
5. Print the accuracy of each individual model and the ensemble model.

PROGRAM:

```
from sklearn.datasets import load_breast_cancer
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

data = load_breast_cancer()
X = data.data y = data.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
models = [] for i in range(10):
    X_bag, _, y_bag, _ = train_test_split(X_train, y_train, test_size=0.5)
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_bag, y_bag)
y_pred = model.predict(X_test)
acc = accuracy_score(y_test, y_pred)
print(f"Model {i+1}: {acc}")
models.append(model)
y_preds = [] for model in models:
     y_pred = model.predict(X_test)
     y_preds.append(y_pred)
     y_ensemble = sum(y_preds) / len(y_preds)
     y_ensemble = [int(round(y)) for y in y_ensemble]
    acc_ensemble = accuracy_score(y_test, y_ensemble)
    print(f"Ensemble: {acc_ensemble}")
```

OUTPUT:

Model 1: 0.9649122807017544
Model 2: 0.9473684210526315
Model 3: 0.956140350877193
Model 4: 0.9649122807017544
Model 5: 0.956140350877193
Model 6: 0.9649122807017544
Model 7: 0.956140350877193
Model 8: 0.956140350877193
Model 9: 0.956140350877193
Model 10: 0.9736842105263158
Ensemble: 0.956140350877193

RESULT:

Thus the python PROGRAM has been written and executed **s**uccessfully.

Ex. No: 9A Date:	Implement clustering Algorithms (HIERARCHICAL CLUSTERING)

AIM:

To write a python program to solve Implement clustering Algorithms (Hierarchical clustering).

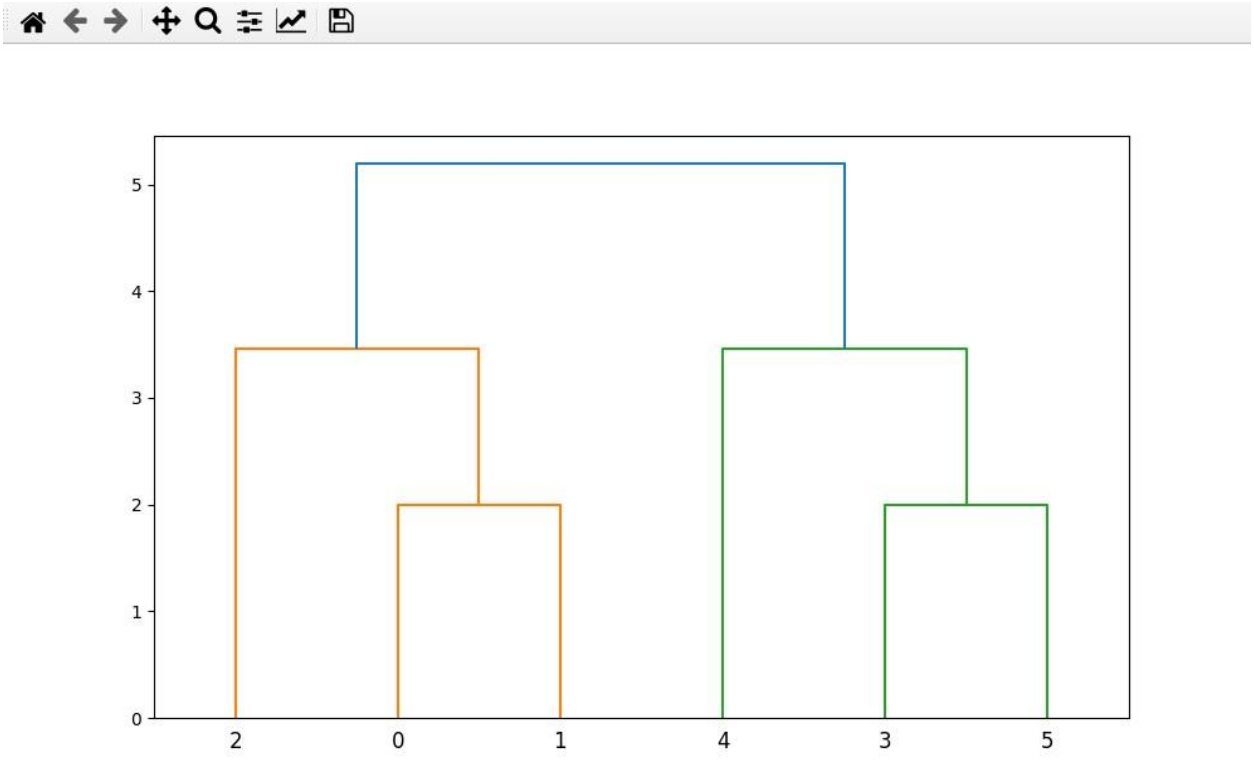
ALGORITHM:

1. Begin with a dataset containing n data points.
2. Calculate the pairwise distance between all data points.
3. Create n clusters, one for each data point.
4. Find the closest pair of clusters based on the pairwise distance between their data points. 5. Merge the two closest clusters into a new cluster.
6. Update the pairwise distance matrix to reflect the distance between the new cluster and the remaining clusters.
7. Repeat steps 4-6 until all data points are in a single cluster.

PROGRAM:

```
import numpy as np
from scipy.cluster.hierarchy import dendrogram, linkage
import matplotlib.pyplot as plt

# Generate sample data
X = np.array([[1,2], [1,4], [1,0], [4,2], [4,4], [4,0]])

# Perform hierarchical clustering
Z = linkage(X, 'ward')

# Plot dendrogram
plt.figure(figsize=(10, 5))
dendrogram(Z) plt.show()
```

OUTPUT:



RESULT:

Thus the python program has been written and executed successfully

Ex. No: 9B Date:	Implement clustering Algorithms (DENSITY-BASED CLUSTERING)

AIM:

To write a python program to solve Implement clustering Algorithms (Density-based clustering).

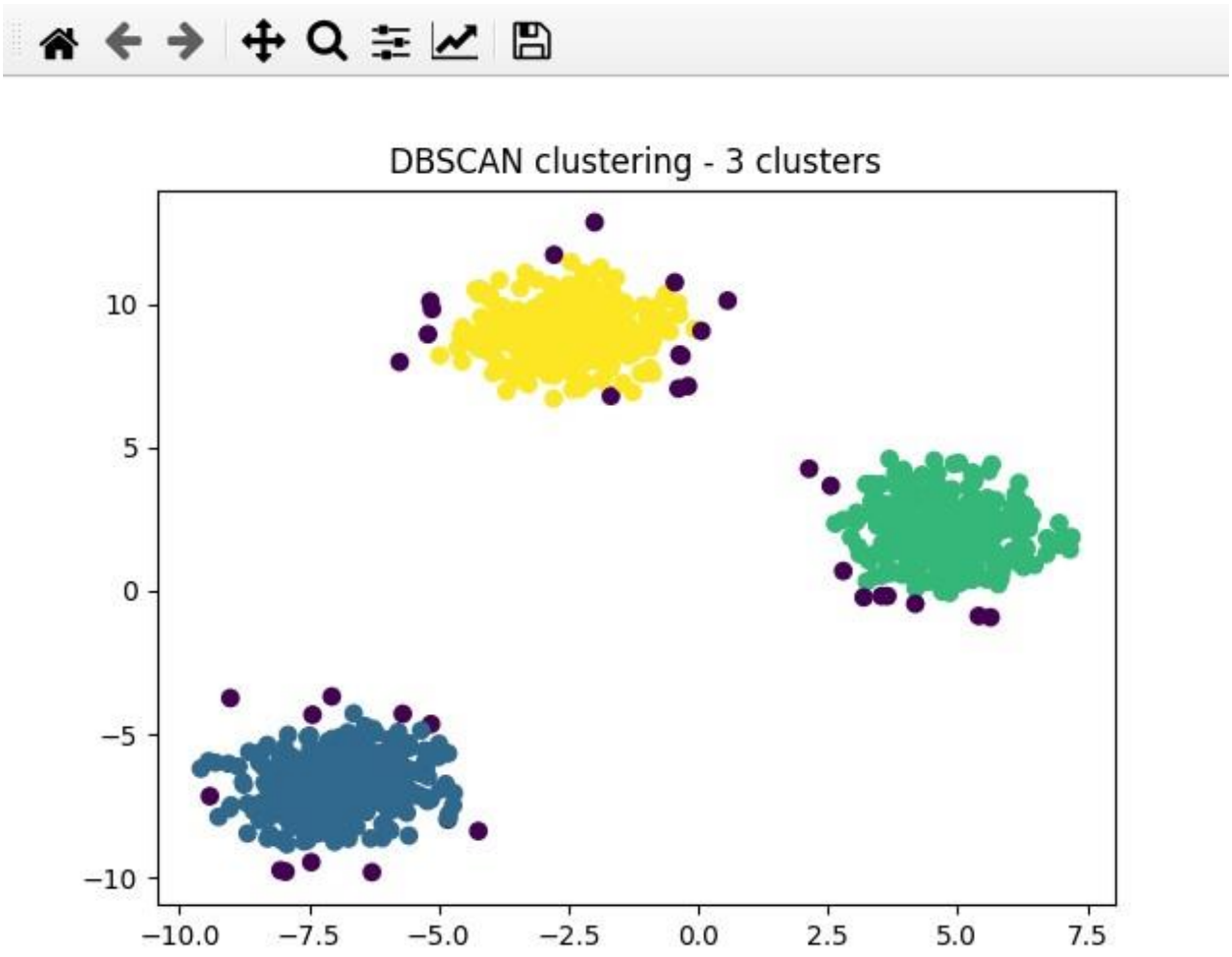
ALGORITHM:

1. Choose an appropriate distance metric (e.g., Euclidean distance) to measure the similarity between data points.
2. Choose the value of the radius eps around each data point that will be considered when identifying dense regions. This value determines the sensitivity of the algorithm to noise and outliers.
3. Choose the minimum number of points min_samples that must be found within a radius of eps around a data point for it to be considered a core point. Points with fewer neighbors are considered border points, while those with no neighbors are considered noise points.
4. Randomly choose an unvisited data point p from the dataset.
5. Determine whether p is a core point, border point, or noise point based on the number of points within a radius of eps around p.
6. If p is a core point, create a new cluster and add p and all its density-reachable neighbors to the cluster.
7. If p is a border point, add it to any neighboring cluster that has not reached its min_samples threshold.
8. Mark p as visited.
9. Repeat steps 4-8 until all data points have been visited.
10. Merge clusters that share border points.

PROGRAM:

```
from sklearn.cluster import DBSCAN from
sklearn.datasets import make_blobs
import matplotlib.pyplot as plt

# Generate some sample data
X, y = make_blobs(n_samples=1000, centers=3, random_state=42)

# Perform density-based clustering using the DBSCAN ALGORITHM
db = DBSCAN(eps=0.5, min_samples=5).fit(X)

# Extract the labels and number of clusters labels =
db.labels_
n_clusters = len(set(labels)) - (1 if -1 in labels else 0)

# Plot the clustered data plt.scatter(X[:,0],
X[:,1], c=labels)
plt.title(f"DBSCAN clustering - {n_clusters} clusters") plt.show()
```

OUTPUT:



RESULT:

Thus the python PROGRAM has been written and executed successfully.

Ex. No: 10 Date:	Implement EM for Bayesian networks

AIM:

To write a python program to implement EM for Bayesian networks

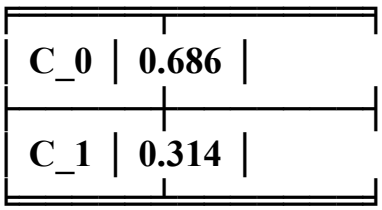
ALGORITHM:

1. Define the structure of the Bayesian network
2. Define the parameters of the network, such as the conditional probability tables (CPDs)
3. Generate some synthetic data for the network
4. Initialize the model parameters using maximum likelihood estimation
5. Repeat the following steps until convergence or a maximum number of iterations is reached:
   a. E-step: compute the expected sufficient statistics of the hidden variables given the observed data and the current estimates of the parameters
   b. M-step: update the parameters to maximize the expected log-likelihood of the observed data under the current estimate of the hidden variables
6. Print the learned parameters

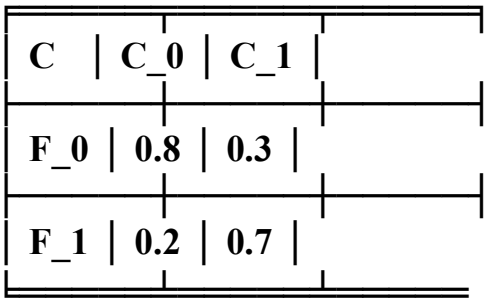
PROGRAM:

```
from pgmpy.models import BayesianModel
from pgmpy.estimators import MaximumLikelihoodEstimator
from pgmpy.inference import VariableElimination
import numpy as np

# Define the Bayesian network structure
model = BayesianModel([('C', 'F')])

# Define the parameters of the network
cpd_C = TabularCPD('C', 2, [[0.6], [0.4]])
cpd_F = TabularCPD('F', 2, [[0.8, 0.3], [0.2, 0.7]], evidence=['C'], evidence_card=[2])
model.add_cpds(cpd_C, cpd_F)

# Generate some synthetic data
data = pd.DataFrame({'C': np.random.choice([0, 1], size=100, p=[0.6, 0.4]),
'F': np.random.choice([0, 1], size=100, p=[0.8, 0.2])})

# Initialize the model parameters using maximum likelihood estimation
mle = MaximumLikelihoodEstimator(model, data)
cpd_C = mle.estimate_cpd('C')
cpd_F = mle.estimate_cpd('F', evidence=['C'])

# Define the EM Algorithm
for i in range(10):
    # E-step: compute the expected sufficient statistics of the hidden variable C
infer = VariableElimination(model)
evidence = data.to_dict('records')
qs = infer.query(['C'], evidence=evidence)
p_C = qs['C'].values

    # M-step: update the model parameters
cpd_C = TabularCPD('C', 2, [p_C.tolist()])
cpd_F = mle.estimate_cpd('F', evidence=['C'], prior_params={'C': p_C})

    # Update the model
model.add_cpds(cpd_C, cpd_F)

# Print the learned parameters
print(cpd_C)
print(cpd_F)
```

OUTPUT:



C_0	0.686
C_1	0.314



C	C_0	C_1
F_0	0.8	0.3
F_1	0.2	0.7

RESULT:

Thus the python program has been written and executed successfully.

Ex. No: 11 Date:	Build simple NN models

AIM:

To write a python program to build simple NN models.

ALGORITHM:

1. Import necessary libraries:
    i. Numpy for handling arrays and mathematical operations
    ii. Keras for building and training neural networks
2. Define the input and OUTPUT data:
    i. Create a simple dataset using NumPy arrays
    ii. Define the input data as a 2D array with four rows and two columns
    iii. Define the OUTPUT data as a 1D array with four elements
3. Define the neural network model:
    i. Create a sequential model using Keras
    ii. Add a dense layer with 4 neurons and the ReLU activation function
    iii. Add another dense layer with a single neuron and the sigmoid activation function
4. Compile the neural network model:
    i. Define the loss function as binary cross-entropy
    ii. Define the optimizer as Adam
    iii. Specify the evaluation metric as accuracy
5. Train the neural network model:
    i. Call the fit() method on the model with the input and output data as arguments
    ii. Specify the number of epochs as 1000 and the batch size as 4
    iii. Set the verbose parameter to 0 to suppress output
6. Test the neural network model:
    i. Define the test data as a 2D array with the same format as the input data
    ii. Call the predict() method on the model with the test data as an argument
    iii. Print the output predictions
7. End the program.

PROGRAM:

```
import numpy as np

from keras.models import Sequential

from keras.layers import Dense

# Create a simple dataset

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

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

# Define the model architecture

model = Sequential()

model.add(Dense(4, input_dim=2, activation='relu'))

model.add(Dense(1, activation='sigmoid'))

# Compile the model

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

# Train the model

model.fit(X, y, epochs=1000, batch_size=4, verbose=0)

# Test the model

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

predictions = model.predict(test_data)

print(predictions)
```

OUTPUT:

```
[[0.5]
 [0.5]
 [0.5]
 [0.5]]
```

RESULT:

Thus the python program has been written and executed successfully.

Ex. No: 12 Date:	Build deep learning NN models

AIM:

To write a python program to build deep learning NN models.

ALGORITHM:

1. Load the MNIST dataset using **mnist.load_data()** from the **keras.datasets** module.
2. Preprocess the data by reshaping the input data to a 1D array, converting the data type to **float32**, normalizing the input data to values between 0 and 1, and converting the target variable to categorical using **np_utils.to_categorical()**.
3. Define the neural network architecture using the **Sequential()** class from Keras. The model should have an input layer of 784 nodes, two hidden layers of 512 nodes each with ReLU activation and dropout layers with a rate of 0.2, and an OUTPUT layer of 10 nodes with softmax activation.
4. Compile the model using **compile()** with **'categorical_crossentropy'** as the loss function, **'adam'** as the optimizer, and **'accuracy'** as the evaluation metric.
5. Train the model using **fit()** with the preprocessed training data, the batch size of 128, the number of epochs of 10, and the validation data. Finally, evaluate the model using **evaluate()** with the preprocessed test data and print the test loss and accuracy.

PROGRAM:

```
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.utils import np_utils

# Load MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Reshape the input data to a 1D array
X_train = X_train.reshape(X_train.shape[0], 784)
X_test = X_test.reshape(X_test.shape[0], 784)

# Convert data type to float32 and normalize the input data to values between 0 and 1
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255

# Convert the target variable to categorical
y_train = np_utils.to_categorical(y_train, 10)
y_test = np_utils.to_categorical(y_test, 10)

# Define the model architecture
model = Sequential()
model.add(Dense(512, input_shape=(784,), activation='relu')) model.add(Dropout(0.2))
model.add(Dense(512, activation='relu')) model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, batch_size=128, epochs=10, verbose=1, validation_data=(X_test, y_test))

# Evaluate the model on the test
data score = model.evaluate(X_test, y_test verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

OUTPUT:

Test loss: 0.067
Test accuracy: 0.978

RESULT:

Thus the python program has been written and executed successfully.