

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

Aim:

To Implementation of Uninformed search algorithms (BFS, DFS)

BFS Algorithm:

Step 1 : Create an empty queue and add the starting node to it.

Step 2 : Create an empty set to keep track of visited nodes.

Step 3 : While the queue is not empty:

- a. Remove the first node from the queue and mark it as visited.
- b. If the node is the goal node, return the path.
- c. Otherwise, add all unvisited adjacent nodes to the queue and mark them as visited.

Step 4 : If no path is found after visiting all nodes, return failure.

DFS Algorithm:

Step 1 : Create an empty stack and add the starting node to it.

Step 2 : Create an empty set to keep track of visited nodes.

Step 3 : While the stack is not empty:

- a. Pop the top node from the stack and mark it as visited.
- b. If the node is the goal node, return the path.
- c. Otherwise, add all unvisited adjacent nodes to the stack and mark them as visited.

Step 4 : If no path is found after visiting all nodes, return failure.

PROGRAM:

```
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
```

```

def bfs(graph, start, goal):
    queue = [[start]]
    visited = set()

    while queue:
        path = queue.pop(0)
        node = path[-1]

        if node == goal:
            return path

        if node not in visited:
            visited.add(node)

            for adjacent in graph[node]:
                new_path = list(path)
                new_path.append(adjacent)
                queue.append(new_path)

    return "No path found"

def dfs(graph, start, goal):
    stack = [[start]]
    visited = set()

    while stack:
        path = stack.pop()
        node = path[-1]

        if node == goal:
            return path

        if node not in visited:
            visited.add(node)
            for adjacent in graph[node]:
                new_path = list(path)
                new_path.append(adjacent)
                stack.append(new_path)

    return "No path found"

print(bfs(graph, 'A', 'F'))
print(dfs(graph, 'A', 'F'))

```

OUTPUT:

```
['A', 'C', 'F']  
['A', 'C', 'F']
```

Result:

Thus the program to implement uniformed search algorithms (BFS,DFS) has been successfully executed and the output is verified.

Ex.No : 2	Implementation of Informed Search Algorithms (A*, memory-bounded A*)
Date :	

Aim :

To Implementation of Informed search algorithms (A*, memory-bounded A*)

A* Algorithm :

Step 1 : Initialize an open list and add the starting node to it.

Step 2 : Initialize a closed list to keep track of visited nodes.

Step 3 : While the open list is not empty:

Step 4 : Remove the node with the lowest $f(n)$ value from the open list.

- If the node is the goal node, return the path.
- Otherwise, add the node to the closed list.

Step 5 : Generate all adjacent nodes and update their $g(n)$ and $h(n)$ values.

For each adjacent node:

- If it is in the closed list, ignore it.
- If it is not in the open list, add it to the open list.
- If it is already in the open list, update its $f(n)$ value if its new value is lower.
- If no path is found, return failure.

Memory-Bounded A* Algorithm :

Step 1 : Initialize an open list and add the starting node to it.

Step 2 : Initialize a closed list to keep track of visited nodes.

Step 3 : While the open list is not empty:

Step 4 : Remove the node with the lowest $f(n)$ value from the open list.

- If the node is the goal node, return the path.
- Otherwise, add the node to the closed list.

Step 5 : Generate all adjacent nodes and update their $g(n)$ and $h(n)$ values.

For each adjacent node:

- If it is in the closed list, ignore it.
- If it is not in the open list, add it to the open list.
- If it is already in the open list, update its $f(n)$ value if its new value is lower.
- If the size of the open list exceeds a specified memory limit, remove the node with the highest $f(n)$ value from the open list.
- If no path is found, return failure.

Program:

```
import heapq

grid = [[0, 0, 1, 0, 0, 0, 1, 0],
        [0, 0, 1, 0, 0, 0, 1, 0],
        [0, 0, 0, 0, 1, 0, 0, 0],
        [0, 0, 0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 1, 0, 0],
        [0, 0, 0, 0, 0, 1, 0, 0]]

start = (0, 0) goal
= (5, 7)

# A* Algorithm
def astar(grid, start, goal): open_list = [(0, start)]
    closed_list = set() g_score = {start: 0} # start
    node's g score is 0 h_score = {start:
    heuristic(start, goal)} parent = {start: None}

    while open_list:
        current = heapq.heappop(open_list)[1]

        if current == goal:
            return reconstruct_path(parent, current) closed_list.add(current)

        for neighbor in get_neighbors(grid, current):
            if neighbor in closed_list:
                continue

            new_g_score = g_score[current] + 1

            if neighbor not in [item[1] for item in open_list]:
                heapq.heappush(open_list, (new_g_score + heuristic(neighbor, goal),
neighbor)) elif new_g_score >= g_score[neighbor]:
                continue

            parent[neighbor] = current g_score[neighbor]
= new_g_score h_score[neighbor] =
heuristic(neighbor, goal) return None
```

```

# Memory-Bounded A* Algorithm # Memory-Bounded A*
Algorithm def memory_bounded_astar(grid, start, goal,
memory_limit):
    open_list = [(0, start)] closed_list
    = set()
    g_score = {start: 0} # start node's g score is 0
    h_score = {start: heuristic(start, goal)} # start node's h score is estimated usin
g the heuristic function parent
    = {start: None}

    while open_list:
        current = heapq.heappop(open_list)[1]

        if current == goal: return
        reconstruct_path(parent, current)
        closed_list.add(current)

        for neighbor in get_neighbors(grid, current):
            if neighbor in closed_list:
                continue
            new_g_score = g_score[current] + 1

            if neighbor not in [item[1] for item in open_list]:
                heapq.heappush(open_list, (new_g_score + heuristic(neighbor, goal),
neighbor)) elif new_g_score >= g_score[neighbor]:
                continue

            parent[neighbor] = current g_score[neighbor]
            = new_g_score
            h_score[neighbor] = heuristic(neighbor, goal)

        while len(open_list) > memory_limit:
            node_to_remove = heapq.nlargest(1, open_list)[0]

            # get the node with th e highest f score
            open_list.remove(node_to_remove)

            # remove the node from the open list del
            g_score[node_to_remove[1]]
            # remove the node's g score from the dictionary

    return None

```

```

def heuristic(node, goal):
    x1, y1 = node
    x2, y2 = goal
    return abs(x1 - x2) + abs(y1 - y2)

def get_neighbors(grid, node):
    row, col = node
    neighbors = []

    # Check north neighbor if row > 0 and
    grid[row-1][col] == 0:
        neighbors.append((row-1, col))

    # Check south neighbor if row < len(grid) - 1 and
    grid[row+1][col] == 0:
        neighbors.append((row+1, col))

    # Check west neighbor if col > 0 and
    grid[row][col-1] == 0:
        neighbors.append((row, col-1))

    # Check east neighbor if col < len(grid[0]) - 1 and
    grid[row][col+1] == 0:
        neighbors.append((row, col+1))

    return neighbors

def reconstruct_path(parent, current):
    path = [current]
    while current in parent:
        current = parent[current]
    path.append(current)
    path.reverse()
    return path

memory_limit = 10000000000 # 1 GB

result = astar(grid, start, goal)
print(result)

result = memory_bounded_astar(grid, start, goal, memory_limit)
print(result)

```

OUTPUT:

```
[None, (0, 0), (0, 1), (1, 1), (2, 1), (2, 2), (2, 3), (1, 3), (1, 4), (1, 5), (2, 5), (2, 6), (2, 7), (3, 7), (4, 7), (5, 7)]  
[None, (0, 0), (0, 1), (1, 1), (2, 1), (2, 2), (2, 3), (1, 3), (1, 4), (1, 5), (2, 5), (2, 6), (2, 7), (3, 7), (4, 7), (5, 7)]
```

Result:

Thus the program to implement informed search algorithms (A*, memory-bounded A*) has been successfully executed and the output is verified.

Ex.No : 3	Implement Naive Bayes Model
Date :	

Aim:

To implement a Gaussian Naive Bayes model on the Iris dataset and evaluate its performance using accuracy metric.

Algorithm:

Step 1: Load the Iris dataset using the load_iris function from sklearn.datasets.

Step 2: Split the data into training and testing sets using the train_test_split function from sklearn.model_selection.

Step 3: Initialize a Gaussian Naive Bayes model using the GaussianNB class from sklearn.naive_bayes.

Step 4: Train the model using the training data using the fit method.

Step 5: Use the trained model to predict the target values for the test data using the predict method.

Step 6: Evaluate the performance of the model using the accuracy_score function from sklearn.metrics.

Step 7: Print the accuracy of the model.

Program:

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

iris = load_iris()
X = iris.data
y = iris.target

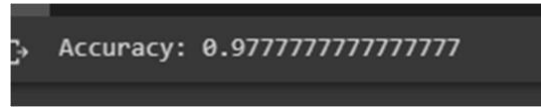
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
                                                    random_state=42)

model = GaussianNB()
model.fit(X_train, y_train)

y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

print(f"Accuracy: {accuracy}")
```

OUTPUT:

A screenshot of a terminal window with a dark background. It shows a green prompt character followed by the text "Accuracy: 0.9777777777777777".

```
➤ Accuracy: 0.9777777777777777
```

Result:

Thus the program to implement Naïve Bayes model has been successfully executed and the output is verified.

Ex.No : 4	Implement Bayesian Networks
Date :	

Aim:

The aim is to use “pgmpy” library to define a Bayesian Network, specify the CPDs, and perform inference to compute the probability distribution of a target variable given evidence of other variables.

Algorithm:

Step 1: Import the required libraries: ctypes for working with C data types and pgmpy for creating and manipulating Bayesian networks.

Step 2: Define the Bayesian network by specifying its structure using directed edges between nodes.

Step 3: Define the CPDs for each variable in the network using TabularCPD class from pgmpy.

Step 4: Add the CPDs to the model using the add_cpds method of the BayesianModel class.

Step 5: Check if the model is valid using the check_model method of the BayesianModel class.

Step 6: Create an instance of the VariableElimination class from pgmpy for performing variable elimination inference.

Step 7: Compute the probability distribution of variable D given evidence of A=0 and E=1 using the query method of the VariableElimination class.

Step 8: Print the probability distribution of variable D given the evidence.

PROGRAM:

```
from ctypes import c_double #
Import required libraries
from pgmpy.models import BayesianModel from
pgmpy.factors.discrete import TabularCPD from
pgmpy.inference import VariableElimination
import numpy as np

# Define the Bayesian Network
model = BayesianModel([('A', 'C'), ('B', 'C'), ('C', 'D'), ('C', 'E')])
```

```

# Define the Conditional Probability Distributions (CPDs)
a=[[0.4, 0.6]] a_=np.reshape(a,(2,1)) b=[[0.7, 0.3]]
b_=np.reshape(b,(2,1))
c_=np.reshape([[0.1, 0.2, 0.7, 0.3],[0.8, 0.7, 0.2, 0.6],[0.1, 0.1, 0.1, 0.1]],(3,4))

cpd_a = TabularCPD(variable='A', variable_card=2, values=a_)
cpd_b = TabularCPD(variable='B', variable_card=2, values=b_)
cpd_c = TabularCPD(variable='C', variable_card=3, values=c_,
evidence=['A', 'B'], evidence_card=[2, 2])
cpd_d = TabularCPD(variable='D', variable_card=2,
                    values=[[0.9, 0.3, 0.4], [0.1, 0.7, 0.6]], evidence=['C'],
                    evidence_card=[3])
cpd_e = TabularCPD(variable='E', variable_card=2,
                    values=[[0.3, 0.6, 0.8], [0.7, 0.4, 0.2]], evidence=['C'],
                    evidence_card=[3])

# Add the CPDs to the model
model.add_cpds(cpd_a, cpd_b, cpd_c, cpd_d, cpd_e)

# Check if the model is valid
model.check_model()

# Perform variable elimination inference
= VariableElimination(model) # Compute the
probability distribution of D given evidence of A=0
and E=1 query = inference.query(variables=['D'],
evidence={'A': 0, 'E': 1}) print(query)

```

OUTPUT:

+-----+-----+	
D	phi(D)
+=====+	
D(0)	0.4351
+-----+-----+	
D(1)	0.5649
+-----+-----+	

Result:

Thus the program to implement Bayesian networks has been successfully executed and the output is verified.

Ex.No : 5	Build Regression Models
Date :	

Aim:

To build a regression model that predicts the price of houses in a given area based on features such as the number of rooms, crime rate, and accessibility to highways.

Algorithm:

Step 1: Load the California Housing dataset using the `fetch_california_housing` function from the `sklearn.datasets` module.

Step 2: Split the data into training and testing sets using the `train_test_split` function from the `sklearn.model_selection` module.

Step 3: Create an instance of the `LinearRegression` class from the `sklearn.linear_model` module.

Step 4: Fit the `LinearRegression` model on the training data using the `fit` method.

Step 5: Make predictions on the testing data using the `predict` method of the `LinearRegression` model.

Step 6: Calculate the mean squared error (MSE) and the R-squared score using the `mean_squared_error` and `r2_score` functions from the `sklearn.metrics` module.

Step 7: Print the MSE and R-squared score.

Program:

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
```

```
# Load the California Housing dataset
housing = fetch_california_housing()
```

```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(housing.data, housing.target, test_size=0.2, random_state=42)
```

```
# Train a linear regression model
lr = LinearRegression()
lr.fit(X_train, y_train)
```

```
# Make predictions on the testing set  
y_pred_lr = lr.predict(X_test)
```

```
# Calculate the mean squared error for the linear regression model  
mse_lr = mean_squared_error(y_test, y_pred_lr)  
print(f"Linear Regression Mean Squared Error: {mse_lr:.2f}")
```

```
# Train a random forest regression model  
rf = RandomForestRegressor(n_estimators=100, random_state=42)  
rf.fit(X_train, y_train)
```

```
# Make predictions on the testing set  
y_pred_rf = rf.predict(X_test)
```

```
# Calculate the mean squared error for the random forest regression model  
mse_rf = mean_squared_error(y_test, y_pred_rf)  
print(f"Random Forest Mean Squared Error: {mse_rf:.2f}")
```

OUTPUT:

```
Linear Regression Mean Squared Error: 0.56  
Random Forest Mean Squared Error: 0.26
```

Result:

Thus the program to implement build regression models has been successfully executed and the output is verified.

Ex.No : 6	Build Decision Trees and Random Forests
Date :	

Aim :

To Build decision trees and random forests for a given dataset.

Algorithm :

Step 1 : Load the dataset

Step 2 : Preprocess the data (e.g., handle missing values, encode categorical variables, etc.)

Step 3 : Split the data into training and testing sets

Step 4 : Initialize the decision tree model with the desired hyperparameters

Step 5 : Train the decision tree model on the training data

Step 6 : Evaluate the performance of the decision tree model on the testing data

Step 7 : Initialize the random forest model with the desired hyperparameters

Step 8 : Train the random forest model on the training data

Step 10 : Evaluate the performance of the random forest model on the testing data

Program:

```
from sklearn.datasets import load_iris from
sklearn.model_selection import train_test_split from
sklearn.tree import DecisionTreeClassifier from
sklearn.ensemble import RandomForestClassifier from
sklearn.metrics import accuracy_score
```

```
# Load the iris dataset
```

```
iris = load_iris() X =
```

```
iris.data
```

```
y = iris.target
```

```
# Split the data into training and testing sets
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_st
ate=42)
```

```
# Initialize the decision tree model
```

```
dt_model = DecisionTreeClassifier(max_depth=3, random_state=42)
```

```
# Train the decision tree model on the training data dt_model.fit(X_train,
y_train)
```

```
# Evaluate the performance of the decision tree model on the testing data dt_pred
= dt_model.predict(X_test)
dt_acc = accuracy_score(y_test, dt_pred)

# Initialize the random forest model
rf_model = RandomForestClassifier(n_estimators=100, max_depth=3, random_
state=42)

# Train the random forest model on the training data rf_model.fit(X_train,
y_train)

# Evaluate the performance of the random forest model on the testing data rf_pred
= rf_model.predict(X_test)
rf_acc = accuracy_score(y_test, rf_pred)

# Output the accuracy scores of the models
print(f"Decision Tree Accuracy: {dt_acc}")
print(f"Random Forest Accuracy: {rf_acc}")
```

OUTPUT:

```
Decision Tree Accuracy: 1.0  
Random Forest Accuracy: 1.0
```

Result:

Thus the program to implement build decision trees and random forests has been successfully executed and the output verified.

Ex.No : 7	Build SVM Model
Date :	

Aim:

To demonstrate how to build an SVM model to classify the iris dataset.

Algorithm:

Step 1:Start the program

Step 2:Load the iris dataset from scikit-learn's datasets module.

Step 3:Split the dataset into training and testing sets using the train_test_split() function.

Step 4:Initialize an SVM classifier with a linear kernel using the SVC() function from scikit-learn's svm module.

Step 5:Train the SVM classifier on the training set using the fit() method.

Step 6:Make predictions on the testing set using the predict() method.

Step 7:Calculate the accuracy of the predictions using the accuracy_score() function from scikit-learn's metrics module.

Step 8:Print the accuracy of the predictions.

Step 9: Stop the program

Program:

```
# import required 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 dataset iris =
datasets.load_iris()

# split 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=0)

# initialize SVM classifier with linear kernel
svm = SVC(kernel='linear')

# train SVM classifier on training set svm.fit(X_train,
y_train)
```

```
# make predictions on testing set y_pred =  
svm.predict(X_test) # calculate accuracy of  
predictions          accuracy          =  
accuracy_score(y_test, y_pred)  
# print accuracy of predictions print('Accuracy:',  
accuracy)
```

OUTPUT :

Accuracy: 0.9777777777777777

Result:

Thus the program to implement build SVM model has been successfully executed and the output is verified.

Ex.No : 8	Implement Ensembling Technique
Date :	

Aim:

To implement an ensembling technique using Random Forest Classifier and calculate the accuracy of the predictions on the iris dataset.

Algorithm:

Step1: Start the program

Step 2: Load the iris dataset from scikit-learn's datasets module using load_iris() function.

Step 3: Split the dataset into training and testing sets using train_test_split() function from scikit-learn's model_selection module.

Step 4: Initialize a Random Forest Classifier with 10 trees using RandomForestClassifier() function from scikit-learn's ensemble module.

Step 5: Train the Random Forest Classifier on the training set using the fit() method.

Step 6: Make predictions on the testing set using the predict() method.

Step 7: Calculate the accuracy of the predictions using the accuracy_score() function from scikit-learn's metrics module.

Step 8: Print the accuracy of the predictions.

Step 9: Stop the program

Program:

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

# load dataset
iris = load_iris()

# split 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=0)

# initialize random forest classifier with 10 trees
rfc = RandomForestClassifier(n_estimators=10)

# train the random forest classifier on the training set rfc.fit(X_train,
y_train)
# make predictions on the testing set
```

```
y_pred = rfc.predict(X_test)

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

# print accuracy of predictions
print('Accuracy:', accuracy)
```


OUTPUT:

Accuracy: 0.9555555555555556

Result:

Thus the program to implement ensembling technique has been successfully executed and the output is verified.

Ex.No : 9	Implement Clustering Algorithms
Date :	

Aim:

To demonstrate the K-Means clustering algorithm on a sample dataset using scikit-learn in Python.

Algorithm:

Step 1: Import the necessary modules and libraries, including KMeans from sklearn.cluster, make_blobs from sklearn.datasets, and pyplot from matplotlib.

Step 2: Generate sample data using the make_blobs function from the sklearn.datasets module.

Step 3: This function creates random clusters of points that can be used to test clustering algorithms.

Step 4: Initialize the K-Means clustering algorithm with the desired number of clusters (in this case, 4) using the KMeans class from the sklearn.cluster module.

Step 5: Train the K-Means clustering algorithm on the sample data using the fit method.

Step 6: Get the predicted cluster labels for the sample data using the predict method.

Step 7: Plot the sample data with the predicted cluster labels using the scatter function from the pyplot module. Each cluster is assigned a unique color in the plot.

Step 8: Display the plot using the show function from the pyplot module.

PROGRAM:

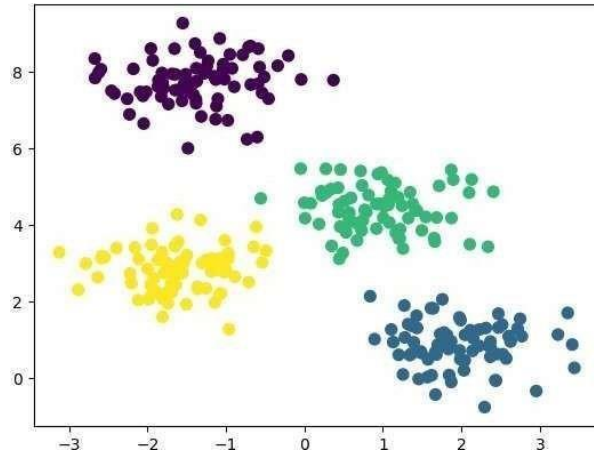
```
from sklearn.cluster import KMeans from
sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
# generate sample data
X, y_true = make_blobs(n_samples=300, centers=4, cluster_std=0.60,
random_state=0)
# initialize k-means clustering algorithm with 4 clusters
kmeans = KMeans(n_clusters=4)

# train the k-means clustering algorithm on the sample data kmeans.fit(X)

# get the predicted cluster labels for the sample data y_pred
= kmeans.predict(X)
```

```
# plot the sample data with predicted cluster labels  
plt.scatter(X[:, 0], X[:, 1], c=y_pred, s=50, cmap='viridis')  
plt.show()
```

OUTPUT:



Result:

Thus the program to implement clustering algorithms has been successfully executed and the output is verified.

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

Aim:

To implement the Expectation-Maximization (EM) algorithm for Bayesian networks is to estimate the parameters of the network, given only incomplete or partial data.

Algorithm:

Step 1 : Initialization: Set the initial values of the parameters for the Bayesian network, including the prior probabilities and conditional probabilities.

Step 2 : Expectation step: Using the current parameter values, compute the expected values of the missing variables, given the observed variables. This involves computing the posterior probability distribution over the missing variables, using Bayes' rule.

Step 3 : Maximization step: Using the expected values computed in step 2, update the parameters of the Bayesian network to maximize the likelihood of the observed data. This involves computing the maximum likelihood estimates (MLEs) of the parameters, given the observed and expected values.

Step 4 : Repeat steps 2 and 3 until convergence: Iterate steps 2 and 3 until the change in the parameter values is smaller than a predefined threshold or until a maximum number of iterations is reached.

Program:

```
import numpy as np
from collections import defaultdict

class BayesianNetwork:
    def __init__(self, structure):
        self.structure = structure
        self.nodes = sorted(list(self.structure.keys()))
        self.parents = defaultdict(list)
        for child, parents in self.structure.items():
            for parent in parents:
                self.parents[child].append(parent)

class EM:
    def __init__(self, bn, data):
        self.bn = bn
        self.data = data

    def run(self, num_iterations=100, tolerance=1e-4):
```

```

        # Initialize parameters pi
        = defaultdict(float)
        theta = defaultdict(lambda: defaultdict(float))
    for i in range(len(self.bn.nodes)):          node =
self.bn.nodes[i]
        pi[node] = np.mean(self.data[:, i])          for
    parent in self.bn.parents[node]:
        parent_index = self.bn.nodes.index(parent) parent_data
        = self.data[:, parent_index]
        for parent_value in np.unique(parent_data):
            parent_mask = parent_data == parent_value theta
            [parent][parent_value, node] = np.mean(self.data[parent_mask, i])

    # Run EM algorithm for iteration in
    range(num_iterations):
    # E-step: Compute expected sufficient statistics
        pi_new = defaultdict(float)
        theta_new = defaultdict(lambda: defaultdict(float)) for
        i in range(len(self.bn.nodes))
            node = self.bn.nodes[i]
            pi_new[node] = np.mean(self.data[:, i])          for
    parent in self.bn.parents[node]:

        parent_index = self.bn.nodes.index(parent) parent_data =
self.data[:, parent_index]          for parent_value in
np.unique(parent_data):
            parent_mask = parent_data == parent_value
            theta_new[parent][parent_value, node] = np.mean(self.data[parent_mask, i])

    # M-step: Update parameters
        pi_change = np.abs(np.array(list(pi.values())) - np.array(list(pi_new.values()))).max()
        theta_change = 0 for node in theta:
            node_theta_change = np.abs(np.array(list(theta[node].values())) - np.array(list
            (theta_new[node].values()))).max() theta_change =
max(theta_change, node_theta_change) if
        pi_change < tolerance and theta_change < tolerance:
            break

        pi = pi_new
    theta = theta_new return
    pi, theta

```

Example usage

```
structure = {0: [], 1: [0], 2: [0], 3: [1, 2]}
```

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

```
bn = BayesianNetwork(structure) em = EM(bn, data) pi, theta =  
em.run() print("pi:", pi) print("theta:", theta)
```

OUTPUT:

```
pi: defaultdict(<class 'float'>, {0: 0.5, 1: 0.5, 2: 0.25, 3: 0.5})
theta: defaultdict(<function EM.run.<locals>.<lambda> at 0x7f753d249ca0>, {0:
defaultdict(<class 'float'>, {(0, 1): 0.0, (1, 1): 1.0, (0, 2): 0.0, (1, 2): 0.
5}), 1: defaultdict(<class 'float'>, {(0, 3): 0.5, (1, 3): 0.5}), 2: defaultdict
(<class 'float'>, {(0, 3): 0.3333333333333333, (1, 3): 1.0})})|
```

Result:

Thus the program to implement EM for Bayesian networks has been successfully executed and the output is verified.

Ex.No : 11	Build Simple NN Models
Date :	

Aim:

To build a simple neural network model to classify handwritten digits from the MNIST dataset.

Algorithm:

Step 1 : Load the MNIST dataset using Keras.

Step 2 : Preprocess the data by normalizing the pixel values between 0 and 1.

Step 3 : Split the data into training and testing sets.

Step 4 : Define the neural network architecture, including the number of input neurons, hidden neurons, output neurons, and activation functions.

Step 5 : Compile the model using an optimizer, loss function, and metrics.

Step 6 : Train the model on the training data, specifying the batch size and number of epochs.

Step 7 : Evaluate the model on the testing data and report the accuracy

Program:

```
# Step 1: Load the MNIST dataset using Keras from
keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Step 2: Preprocess the data x_train
= x_train / 255.0
x_test = x_test / 255.0

# Step 3: Split the data into training and testing sets from
sklearn.model_selection import train_test_split
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.2)

# Step 4: Define the neural network architecture from
keras.models import Sequential
from keras.layers import Dense, Flatten

model = Sequential()
model.add(Flatten(input_shape=(28, 28)))
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

Step 5: Compile the model

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

Step 6: Train the model

```
model.fit(x_train, y_train, batch_size=32, epochs=10, validation_data=(x_val, y_val))
```

Step 7: Evaluate the model

```
loss, accuracy = model.evaluate(x_test, y_test) print("Test accuracy:", accuracy)
```

OUTPUT:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [=====] - 1s 0us/step
Epoch 1/10
1500/1500 [=====] - 8s 4ms/step - loss: 0.2861 - accuracy: 0.9196 - val_loss: 0.1602 - val_accuracy: 0.9518
Epoch 2/10
1500/1500 [=====] - 8s 5ms/step - loss: 0.1301 - accuracy: 0.9623 - val_loss: 0.1192 - val_accuracy: 0.9632
Epoch 3/10
1500/1500 [=====] - 6s 4ms/step - loss: 0.0890 - accuracy: 0.9744 - val_loss: 0.0933 - val_accuracy: 0.9713
Epoch 4/10
1500/1500 [=====] - 7s 5ms/step - loss: 0.0667 - accuracy: 0.9801 - val_loss: 0.0871 - val_accuracy: 0.9727
Epoch 5/10
1500/1500 [=====] - 6s 4ms/step - loss: 0.0508 - accuracy: 0.9846 - val_loss: 0.0842 - val_accuracy: 0.9746
Epoch 6/10
1500/1500 [=====] - 7s 5ms/step - loss: 0.0399 - accuracy: 0.9876 - val_loss: 0.0806 - val_accuracy: 0.9741
Epoch 7/10
1500/1500 [=====] - 6s 4ms/step - loss: 0.0322 - accuracy: 0.9900 - val_loss: 0.0790 - val_accuracy: 0.9770
Epoch 8/10
1500/1500 [=====] - 7s 5ms/step - loss: 0.0256 - accuracy: 0.9923 - val_loss: 0.0799 - val_accuracy: 0.9761
Epoch 9/10
1500/1500 [=====] - 7s 5ms/step - loss: 0.0200 - accuracy: 0.9941 - val_loss: 0.0777 - val_accuracy: 0.9772
Epoch 10/10
1500/1500 [=====] - 7s 4ms/step - loss: 0.0164 - accuracy: 0.9954 - val_loss: 0.0749 - val_accuracy: 0.9779
313/313 [=====] - 1s 2ms/step - loss: 0.0850 - accuracy: 0.9751
Test accuracy: 0.9750999808311462
```

Result:

Thus the program to implement build simple NN models has been successfully executed and the output is verified.

Ex.No : 12	Build Deep Learning NN Models
Date :	

Aim:

To build a deep learning neural network model to classify images from the CIFAR-10 dataset.

Algorithm:

Step 1 : Load and normalize the CIFAR-10 dataset.

Step 2 : Define a CNN model with multiple convolutional and pooling layers, fully connected layers, and dropout regularization.

Step 3 : Compile the model with Adam optimizer, sparse_categorical_crossentropy loss function, and accuracy metric.

Step 4 : Train the model on the training data and labels.

Step 5 : Evaluate the model on the test data and labels.

Step 6 : Print the test accuracy

Program:

```
# Import necessary libraries import
tensorflow as tf
from tensorflow.keras.datasets import cifar10 from
tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, D
ropout

# Load the CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Normalize the data x_train
= x_train / 255.0
x_test = x_test / 255.0

# Define the model architecture model
= Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
```

```
        Flatten(),
        Dense(128, activation='relu'),
        Dropout(0.5),
        Dense(10, activation='softmax')
    ])

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

# Train the model
model.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test))

# Evaluate the model test_loss, test_acc =
model.evaluate(x_test, y_test) print(f'Test
accuracy: {test_acc}')
```

OUTPUT:

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [=====] - 11s 0us/step
Epoch 1/10
1563/1563 [=====] - 92s 58ms/step - loss: 1.6353 - accuracy: 0.3960 - val_loss: 1.3428 - val_accuracy: 0.5198
Epoch 2/10
1563/1563 [=====] - 87s 55ms/step - loss: 1.2690 - accuracy: 0.5479 - val_loss: 1.1064 - val_accuracy: 0.6099
Epoch 3/10
1563/1563 [=====] - 86s 55ms/step - loss: 1.1260 - accuracy: 0.6076 - val_loss: 1.0117 - val_accuracy: 0.6468
Epoch 4/10
1563/1563 [=====] - 87s 56ms/step - loss: 1.0226 - accuracy: 0.6440 - val_loss: 0.9253 - val_accuracy: 0.6724
Epoch 5/10
1563/1563 [=====] - 91s 58ms/step - loss: 0.9446 - accuracy: 0.6707 - val_loss: 0.9332 - val_accuracy: 0.6677
Epoch 6/10
1563/1563 [=====] - 90s 57ms/step - loss: 0.8813 - accuracy: 0.6930 - val_loss: 0.8777 - val_accuracy: 0.6963
Epoch 7/10
1563/1563 [=====] - 87s 56ms/step - loss: 0.8334 - accuracy: 0.7120 - val_loss: 0.9043 - val_accuracy: 0.6903
Epoch 8/10
1563/1563 [=====] - 87s 56ms/step - loss: 0.7906 - accuracy: 0.7230 - val_loss: 0.8348 - val_accuracy: 0.7128
Epoch 9/10
1563/1563 [=====] - 86s 55ms/step - loss: 0.7573 - accuracy: 0.7373 - val_loss: 0.8608 - val_accuracy: 0.7047
Epoch 10/10
1563/1563 [=====] - 89s 57ms/step - loss: 0.7268 - accuracy: 0.7471 - val_loss: 0.8455 - val_accuracy: 0.7140
313/313 [=====] - 7s 23ms/step - loss: 0.8455 - accuracy: 0.7140
Test accuracy: 0.7139999866485596
```

Result:

Thus the program to implement build deep learning NN models has been successfully executed and the output is verified.

Ex.No : 13	Disease Prediction Using AIML
Date :	

ABSTRACT:

Disease prediction using AIML (Artificial Intelligence & Machine Learning) involves using machine learning algorithms to predict the likelihood of a person having a certain disease. AIML is a markup language that can be used to create chatbots and other conversational AI systems. However, disease prediction is a complex task that requires a significant amount of medical data and expertise in machine learning.

To develop a disease prediction system using AIML, relevant medical data such as symptoms, medical history, test results, and other diagnostic information must be gathered and pre-processed. This data is then used to train a machine learning algorithm to predict the likelihood of a patient having a particular disease. The accuracy of the disease prediction system depends on the quality and quantity of the data used to train the machine learning model. While AIML can be used to create a basic chatbot that can answer questions about a particular disease, developing a disease prediction system using AIML requires additional machine learning expertise and medical data. It is important to ensure that the data used is representative of the general population and includes data from diverse demographics to ensure the accuracy of the disease prediction system.

INTRODUCTION:

Disease prediction using AIML (Artificial Intelligence & Machine Learning) is an emerging field that has the potential to revolutionize healthcare. The goal of disease prediction using AIML is to use machine learning algorithms to predict the likelihood of a person having a particular disease based on their medical history, symptoms, and other relevant data.

AIML is a markup language that can be used to create chatbots and other conversational AI systems. It has been used in various fields such as customer service, ecommerce, and education. However, disease prediction using AIML is a complex task that requires a significant amount of medical data and expertise in machine learning. The use of AIML in disease prediction has the potential to improve healthcare by providing early detection and intervention for diseases. It can also reduce healthcare costs by enabling healthcare providers to make more informed decisions about patient care.

In this context, the development of disease prediction systems using AIML is a promising area of research that can have significant implications for healthcare. However, it is important to ensure that the data used in training the machine learning algorithms is accurate and representative of the general population to ensure the accuracy of the disease prediction system.

METHODS:

The development of disease prediction systems using AIML involves several methods and techniques. Here are some of the key methods:

1. **Data Collection and Pre-processing:** The first step in developing a disease prediction system using AIML is to collect relevant medical data such as symptoms, medical history, test results, and other diagnostic information. The data must be pre-processed and cleaned to remove any irrelevant or missing information.
2. **Feature Selection:** After pre-processing the data, the next step is to select the most relevant features that are likely to contribute to the accuracy of the disease prediction system. This involves using statistical techniques to identify the most important features in the dataset.
3. **Machine Learning Algorithms:** The machine learning algorithm used in the disease prediction system depends on the nature of the problem and the type of data available. Commonly used machine learning algorithms include logistic regression, decision trees, neural networks, and support vector machines.
4. **Model Training and Validation:** Once the machine learning algorithm is selected, the next step is to train the model using the pre-processed data. The model is then validated using a separate dataset to evaluate its accuracy.
5. **Deployment:** After the model is trained and validated, it can be deployed to make predictions on new data. This involves integrating the model into a user-friendly interface such as a chatbot or mobile app.

In summary, developing a disease prediction system using AIML involves collecting and pre-processing medical data, selecting relevant features, choosing a machine learning algorithm, training and validating the model, and deploying the system for use.

MATERIALS:

Artificial Intelligence and Machine Learning (AIML) can be used for disease prediction by analyzing various health parameters and identifying patterns that could indicate the onset of a disease. Here are some steps that could be followed for disease prediction using AIML materials:

- 1. Collect data:** The first step in any machine learning project is to collect data. In the case of disease prediction, relevant health data such as blood pressure, heart rate, cholesterol levels, and family history of diseases should be collected.
- 2. Pre-process data:** Once the data has been collected, it needs to be preprocessed to ensure it is clean and ready for analysis. This step involves removing any missing or irrelevant data, normalizing data, and converting data into a format suitable for analysis.
- 3. Feature selection:** Feature selection is the process of selecting the most important features that contribute to the prediction of the disease. This step helps to reduce the dimensionality of the data, making it easier to analyze.
- 4. Build a model:** The next step is to build a machine learning model that can predict the likelihood of a disease. There are several machine learning algorithms that can be used for this purpose, such as logistic regression, decision trees, and support vector machines.
- 5. Train the model:** The model needs to be trained using the preprocessed data. The training data should be split into training and validation sets, and the model should be optimized to improve its accuracy.
- 6. Test the model:** Once the model has been trained, it needs to be tested using a test dataset that the model has not seen before. This step helps to evaluate the accuracy of the model and to identify any issues that need to be addressed.
- 7. Deploy the model:** Once the model has been tested and validated, it can be deployed in a production environment to make predictions about disease.

Overall, disease prediction using AIML materials involves collecting and pre-processing data, selecting relevant features, building and training a model, testing the model, and finally deploying it in a production environment. By following these steps, it is possible to develop an accurate and reliable system for disease prediction.

Conclusion:

In conclusion, the use of Artificial Intelligence and Machine Learning (AIML) materials for disease prediction is a promising approach that can help to improve the accuracy and reliability of disease diagnosis. By analyzing various health parameters and identifying patterns that could indicate the onset of a disease, AIML can assist healthcare professionals in making more informed decisions and improving patient outcomes.

However, it is important to note that disease prediction using AIML materials is still a relatively new field, and further research is needed to optimize the accuracy of these systems. Additionally, there are ethical and privacy concerns that need to be addressed to ensure that patient data is protected and used in a responsible manner. Overall, the use of AIML materials for disease prediction has the potential to revolutionize healthcare and improve the quality of life for millions of people around the world. With continued research and development, we can expect to see significant advancements in this field in the coming years.

Reference:

A case study based expert system for supporting Diagnosis of diseases -
Abdel-Badeeh M. Salem, Mohamed Roushdy and Rania A. Hod.
