R P Sarathy
Institute of Technology

An Autonomous Institution

Approved by AICTE | Accredited By NAAC A++ | Affiliated to Anna University

Salem Bengaluru Highway NH - 7, Poosaripatty, Kadayampatty Taluk, Salem - 636305.

Admin Office: 93449-72274, Admission Cell: 93449-72275,73977-56003,

Register No: ______

Name of the Laboratory: ______

Certified that this is bonafide record of work done by

______ *of* ______ *year/semester*

______ *Branch for the year 20 - 20*

Staff Incharge **Head of the Department**

Submitted for the University Practical Examination

held on ______

Internal Examiner **External Examiner**

INDEX

Serial No.	Date		Page No.	Marks awarded	Remarks

INDEX

Serial No.	Date		Page No.	Marks awarded	Remarks

EXPERIMENT: 1	Implementation of Uninformed search algorithms (BFS, DFS)
Date:	

Aim:

To implement and compare Uninformed Search Algorithms: Breadth-First Search (BFS) and Depth-First Search (DFS) for traversing or searching graph data structures.

1. Breadth-First Search (BFS)

Algorithm:

1. Start by adding the root node to a queue.
2. While the queue is not empty:
   - Dequeue a node from the front.
   - If the node is the goal, return the result.
   - Else, enqueue all its unvisited neighbors.
3. Mark each visited node to avoid processing it multiple times.

Program (Python):

```python
from collections import deque

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

    while queue:
        node = queue.popleft()

        if node == goal:
            return f"Goal {goal} found"

        if node not in visited:
            visited.add(node)
            queue.extend(neighbor for neighbor in graph[node] if neighbor not in visited)

    return f"Goal {goal} not found"

# Graph as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

start_node = 'A'
goal_node = 'F'
print(bfs(graph, start_node, goal_node))
```

Output:

```
[Running] python -u "c:\Users\Admin\OneDrive\SEM 4\AIML\Code\EXE_1.py"
Goal F found

[Done] exited with code=0 in 0.083 seconds
```

2. Depth-First Search (DFS)

Algorithm:

1. Mark each visited node to avoid loops.
    - Else, push all its unvisited neighbors onto the stack.
    - If the node is the goal, return the result.
    - Pop a node from the stack.
2. While the stack is not empty:
3. Start by pushing the root node to a stack.

Program (Python):

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

    while stack:
        node = stack.pop()

        if node == goal:
            return f"Goal {goal} found"

        if node not in visited:
            visited.add(node)
            stack.extend(neighbor for neighbor in graph[node] if neighbor not in visited)

    return f"Goal {goal} not found"

# Graph as an adjacency list (same as above)
print(dfs(graph, start_node, goal_node))
```

Output:

```
[Running] python -u "c:\Users\Admin\OneDrive\SEM 4\AIML\Code\EXE_1.py"
Goal F found

[Done] exited with code=0 in 0.083 seconds
```

Result:

EXPERIMENT: 2	Implementation of Uninformed search algorithms (BFS, DFS)
Date:	

Aim:

To implement and compare Informed Search Algorithms: A* and Memory-Bounded A* for finding the optimal path from a start node to a goal node in a graph.

1. *A Search Algorithm**

Algorithm:

1. Initialize the open list (priority queue) with the start node.
2. Initialize the closed list as empty.
3. While the open list is not empty:
    - Select the node from the open list with the lowest f(n) where f(n) = g(n) + h(n), g(n) is the cost from the start to n, and h(n) is the heuristic estimate from n to the goal.
    - If the node is the goal, return the path.
    - Otherwise, expand the node by evaluating its neighbors, and add unvisited neighbors to the open list.
    - Move the current node to the closed list.

Program (Python):

```python
import heapq

def a_star(graph, start, goal, h):
    open_list = []
    heapq.heappush(open_list, (0, start))
    came_from = {}
    g_score = {start: 0}
    f_score = {start: h[start]}

    while open_list:
        _, current = heapq.heappop(open_list)

        if current == goal:
            return reconstruct_path(came_from, current)

        for neighbor, cost in graph[current].items():
            tentative_g = g_score[current] + cost
            if neighbor not in g_score or tentative_g < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score[neighbor] = tentative_g + h[neighbor]
                heapq.heappush(open_list, (f_score[neighbor], neighbor))

    return "Path not found"

def reconstruct_path(came_from, current):
    total_path = [current]
    while current in came_from:
        current = came_from[current]
        total_path.append(current)
    return total_path[::-1]

# Example graph and heuristic
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'D': 1, 'E': 3},
    'C': {'E': 1},
    'D': {'F': 2},
    'E': {'F': 1},
    'F': {}
}

heuristic = {
    'A': 5, 'B': 4, 'C': 2,
    'D': 2, 'E': 1, 'F': 0
}

start_node = 'A'
goal_node = 'F'
print(a_star(graph, start_node, goal_node, heuristic))
```

Output:

```
[Running] python -u "c:\Users\Admin\OneDrive\SEM 4\AIML\Code\EXE_2.py"
['A', 'B', 'D', 'F']

[Done] exited with code=0 in 0.067 seconds
```

2. *Memory-Bounded A (MA)* Algorithm**

Algorithm:

1. Similar to A*, but limits the memory used by storing only a subset of nodes.
2. If memory exceeds a defined limit, remove the least promising nodes (with the highest f(n) value) from the open list to free up space.
3. In case of pruning, store information about the best path to revisit later if necessary.

Program (Python):

```python
import heapq

def memory_bounded_a_star(graph, start, goal, h, memory_limit):
    open_list = []
    heapq.heappush(open_list, (0, start))
    came_from = {}
    g_score = {start: 0}
    f_score = {start: h[start]}
    closed_list = set()

    while open_list:
        if len(open_list) > memory_limit:
            # Prune the least promising node
            open_list = sorted(open_list, key=lambda x: x[0])[:-1]

        _, current = heapq.heappop(open_list)

        if current == goal:
            return reconstruct_path(came_from, current)

        closed_list.add(current)

        for neighbor, cost in graph[current].items():
            tentative_g = g_score[current] + cost
            if neighbor not in g_score or tentative_g < g_score[neighbor]:
                if neighbor not in closed_list:
                    came_from[neighbor] = current
                    g_score[neighbor] = tentative_g
                    f_score[neighbor] = tentative_g + h[neighbor]
                    heapq.heappush(open_list, (f_score[neighbor], neighbor))

    return "Path not found"

# Example usage with memory limit
memory_limit = 4  # Limiting memory to store only 4 nodes at a time
print(memory_bounded_a_star(graph, start_node, goal_node, heuristic, memory_limit))
```

Output:

```
[Running] python -u "c:\Users\Admin\OneDrive\SEM 4\AIML\Code\EXE_2.py"
['A', 'B', 'D', 'F']

[Done] exited with code=0 in 0.101 seconds
```

Result:

EXPERIMENT: 3	Implement naïve Bayes models
Date:	

Aim:

To implement a Naïve Bayes classifier that uses probability theory to predict the class of given data points based on input features, assuming independence among features.

Algorithm:

1. **Calculate Prior Probabilities:**
   Determine the probability of each class in the training data.
2. **Compute Likelihood:**
   For each feature and class, compute the likelihood using a probability distribution (e.g., Gaussian for continuous data).
3. **Apply Bayes' Theorem:**
   Combine the prior and likelihood to compute the posterior probability for each class.
4. **Predict Class:**
   Assign the data point to the class with the highest posterior probability.

Program (Python):

```python
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, classification_report

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

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Initialize and train the Naïve Bayes model (GaussianNB for continuous data)
model = GaussianNB()
model.fit(X_train, y_train)

# Predict using the test set
y_pred = model.predict(X_test)

# Calculate accuracy and display classification report
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred)

print("Accuracy:", accuracy)
print("Classification Report:\n", report)
```

Output:

```
[Running] python -u "c:\Users\Admin\OneDrive\SEM 4\AIML\Code\EXE_3.py"
Accuracy: 0.9777777777777777
Classification Report:
               precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      0.92      0.96        13
           2       0.93      1.00      0.96        13

    accuracy                           0.98        45
   macro avg       0.98      0.97      0.97        45
weighted avg       0.98      0.98      0.98        45


[Done] exited with code=0 in 6.026 seconds
```

Result:

EXPERIMENT: 4	Implement Bayesian Networks
Date:	

Aim:

To implement a Bayesian Network that models probabilistic relationships among variables and performs inference to compute the likelihood of events given observed evidence.

Algorithm:

1. **Define the Network Structure:**
   Identify nodes (random variables) and their dependencies. For example, in the classic burglary scenario, nodes such as Burglary, Earthquake, Alarm, JohnCalls, and MaryCalls are connected by cause-effect relationships.
2. **Specify Conditional Probability Distributions (CPDs):**
   For each node, define the CPD. This includes prior probabilities for root nodes and conditional probabilities for nodes with parents.
3. **Construct the Bayesian Network:**
   Use a library (e.g., pgmpy) to create the network and add the nodes and CPDs.
4. **Validate the Model:**
   Check that the CPDs are consistent and that the network is correctly specified.
5. **Perform Inference:**
   Use an inference algorithm (e.g., Variable Elimination) to compute the posterior probability of a node given evidence from other nodes.

Program (Python):

```python
# Import necessary libraries
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([('Burglary', 'Alarm'),
                       ('Earthquake', 'Alarm'),
                       ('Alarm', 'JohnCalls'),
                       ('Alarm', 'MaryCalls')])

# Define CPDs for each node

# Prior for Burglary (0: No burglary, 1: Burglary)
cpd_burglary = TabularCPD(variable='Burglary', variable_card=2,
                          values=[[0.999], [0.001]])

# Prior for Earthquake (0: No earthquake, 1: Earthquake)
cpd_earthquake = TabularCPD(variable='Earthquake', variable_card=2,
                            values=[[0.998], [0.002]])

# CPD for Alarm given Burglary and Earthquake
# The values are arranged for the combinations: [Burglary=0, Earthquake=0], [0,1], [1,0], [1,1]
cpd_alarm = TabularCPD(variable='Alarm', variable_card=2,
                       values=[[0.999, 0.71, 0.06, 0.05],   # Alarm = 0 (False)
                               [0.001, 0.29, 0.94, 0.95]],  # Alarm = 1 (True)
                       evidence=['Burglary', 'Earthquake'],
                       evidence_card=[2, 2])

# CPD for JohnCalls given Alarm
cpd_john = TabularCPD(variable='JohnCalls', variable_card=2,
                      values=[[0.95, 0.10],  # John does not call
                              [0.05, 0.90]], # John calls
                      evidence=['Alarm'], evidence_card=[2])

# CPD for MaryCalls given Alarm
cpd_mary = TabularCPD(variable='MaryCalls', variable_card=2,
                      values=[[0.99, 0.30],  # Mary does not call
                              [0.01, 0.70]], # Mary calls
                      evidence=['Alarm'], evidence_card=[2])

# Add CPDs to the model
model.add_cpds(cpd_burglary, cpd_earthquake, cpd_alarm, cpd_john, cpd_mary)

# Validate the model structure and CPDs
assert model.check_model(), "Model invalid: Check CPDs and structure."

# Perform inference using Variable Elimination
infer = VariableElimination(model)

# Example Query:
# Calculate the probability of a burglary given that both John and Mary called (1 indicates 'True')
query_result = infer.query(variables=['Burglary'], evidence={'JohnCalls': 1, 'MaryCalls': 1})
print(query_result)
```

Output:

```
+-------------+-----------------+
| Burglary    |   phi(Burglary) |
+=============+=================+
| Burglary(0) |          0.7158 |
+-------------+-----------------+
| Burglary(1) |          0.2842 |
+-------------+-----------------+

[Done] exited with code=0 in 13.645 seconds
```

Result:

EXPERIMENT: 5	Build Regression models
Date:	

Aim:

To build and evaluate regression models that predict continuous target values from input features. The example demonstrates a linear regression model to illustrate data preparation, model training, and evaluation.

Algorithm:

1. **Data Preparation:**
    - Load and inspect the dataset.
    - Preprocess the data (handle missing values, scale features if necessary).
2. **Train-Test Split:**
    - Divide the dataset into training and testing subsets.
3. **Model Training:**
    - Initialize a regression model (e.g., Linear Regression).
    - Fit the model using the training data.
4. **Model Evaluation:**
    - Predict target values on the test set.
    - Compute evaluation metrics such as Mean Squared Error (MSE) and R-squared (coefficient of determination).
5. **Result Analysis:**
    - Analyze the output metrics to determine the model's performance.

Program (Python):

```python
import numpy as np
import pandas as pd
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Load the California Housing dataset
data = fetch_california_housing()
X = pd.DataFrame(data.data, columns=data.feature_names)
y = data.target

# Split the data into training and testing sets (70% train, 30% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Initialize the Linear Regression model
model = LinearRegression()

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

# Predict the target for test set
y_pred = model.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

# Output evaluation results
print("Mean Squared Error:", mse)
print("R-squared:", r2)

```

Output:

```
[Running] python -u "c:\Users\Admin\OneDrive\SEM 4\AIML\Code\tempCodeRunnerFile.py"
Mean Squared Error: 0.5305677824766755
R-squared: 0.5957702326061662

[Done] exited with code=0 in 16.576 seconds
```

Result:

EXPERIMENT: 6	Build decision trees and random forests
Date:	

Aim:

To build and compare Decision Tree and Random Forest models for classification tasks using a sample dataset, highlighting their performance and generalization capabilities.

Algorithm:

1. **Data Preparation:**
   - Load a classification dataset (e.g., Iris).
   - Split the dataset into training and testing sets.
2. **Model Training:**
   - Train a Decision Tree classifier on the training data.
   - Train a Random Forest classifier on the training data.
3. **Model Evaluation:**
   - Predict class labels on the test set.
   - Evaluate the models using metrics like accuracy and a classification report.

Program (Python):

```python
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, classification_report

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

# Split the dataset (70% train, 30% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Decision Tree Model
dt_model = DecisionTreeClassifier(random_state=42)
dt_model.fit(X_train, y_train)
dt_pred = dt_model.predict(X_test)

# Random Forest Model
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)
rf_pred = rf_model.predict(X_test)

# Evaluation
print("Decision Tree Accuracy:", accuracy_score(y_test, dt_pred))
print("Decision Tree Classification Report:\n", classification_report(y_test, dt_pred))

print("Random Forest Accuracy:", accuracy_score(y_test, rf_pred))
print("Random Forest Classification Report:\n", classification_report(y_test, rf_pred))
```

Output:

```
[Running] python -u "c:\Users\Admin\OneDrive\SEM 4\AIML\Code\tempCodeRunnerFile.py"
Decision Tree Accuracy: 1.0
Decision Tree Classification Report:
               precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45

Random Forest Accuracy: 1.0
Random Forest Classification Report:
               precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45


[Done] exited with code=0 in 2.464 seconds
```

Result:

EXPERIMENT: 7	Build SVM models
Date:	

Aim:

To build and evaluate Support Vector Machine (SVM) models for classification tasks, demonstrating their effectiveness on datasets like the Iris dataset.

Algorithm:

1. **Data Preparation:**
   - Load the dataset (e.g., Iris dataset).
   - Split the data into training and testing sets.
2. **Model Training:**
   - Initialize an SVM classifier with chosen hyperparameters.
   - Train the SVM model on the training set.
3. **Model Evaluation:**
   - Predict the labels for the test set.
   - Evaluate the model using accuracy and classification metrics.

Program (Python):

```python
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report

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

# Split dataset into training and testing sets (70% train, 30% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Initialize the SVM classifier with a linear kernel
svm_model = SVC(kernel='linear', random_state=42)

# Train the model
svm_model.fit(X_train, y_train)

# Predict using the test set
y_pred = svm_model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred)

print("SVM Accuracy:", accuracy)
print("Classification Report:\n", report)

```

Output:

```
[Running] python -u "c:\Users\Admin\OneDrive\SEM 4\AIML\Code\EXE_7.py"
SVM Accuracy: 1.0
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45


[Done] exited with code=0 in 1.607 seconds
```

Result:

EXPERIMENT: 8	Implement ensembling techniques
Date:	

Aim:

To implement ensemble techniques by combining multiple machine learning models (base learners) to improve prediction accuracy and robustness.

Algorithm:

1. **Data Preparation:**
   - Load and preprocess the dataset.
   - Split the data into training and testing sets.
2. **Train Base Models:**
   - Initialize several base classifiers (e.g., Decision Tree, SVM, Logistic Regression).
3. **Ensemble Model Construction:**
   - Combine base models using an ensemble strategy such as a Voting Classifier.
4. **Model Evaluation:**
   - Train the ensemble model.
   - Evaluate its performance on the test set using accuracy and other metrics.

Program (Python):

```python
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import VotingClassifier
from sklearn.metrics import accuracy_score, classification_report

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

# Split dataset into training and testing sets (70% train, 30% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Initialize base models
dt_model = DecisionTreeClassifier(random_state=42)
svm_model = SVC(kernel='linear', probability=True, random_state=42)
lr_model = LogisticRegression(max_iter=200, random_state=42)

# Create the ensemble model using a Voting Classifier
ensemble_model = VotingClassifier(
    estimators=[('dt', dt_model), ('svm', svm_model), ('lr', lr_model)],
    voting='soft'  # Use soft voting to average predicted probabilities
)

# Train the ensemble model
ensemble_model.fit(X_train, y_train)

# Predict using the test set
y_pred = ensemble_model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred)

print("Ensemble Model Accuracy:", accuracy)
print("Classification Report:\n", report)
```

Output:

```
[Running] python -u "c:\Users\Admin\OneDrive\SEM 4\AIML\Code\EXE_8.py"
Ensemble Model Accuracy: 1.0
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45


[Done] exited with code=0 in 1.665 seconds
```

Result:

EXPERIMENT: 9	Implement clustering algorithms
Date:	

Aim:

To implement clustering algorithms to group similar data points into clusters without prior knowledge of labels, demonstrating unsupervised learning using the K-Means clustering algorithm.

Algorithm:

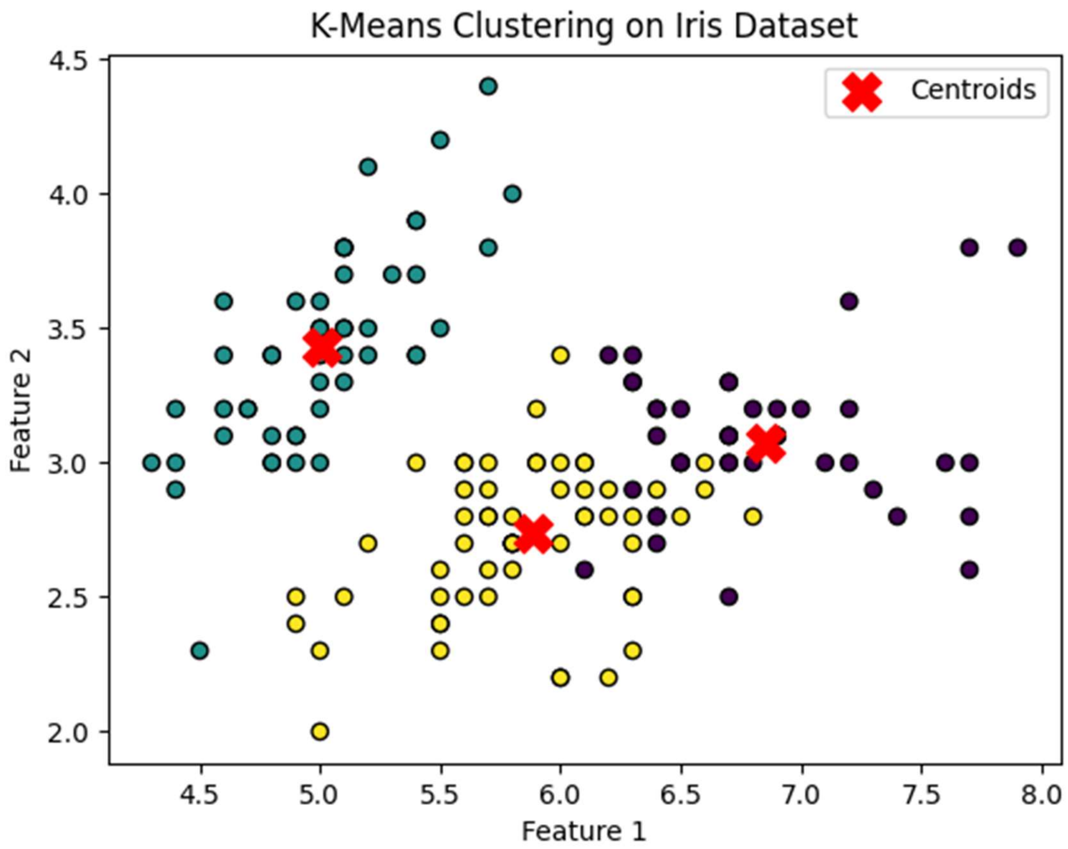
1. **Data Preparation:**
    - Load and preprocess the dataset (e.g., Iris dataset).
2. **Clustering Process:**
    - **Initialization:** Randomly select k initial centroids.
    - **Assignment:** Assign each data point to the nearest centroid.
    - **Update:** Recalculate centroids as the mean of points in each cluster.
    - **Iteration:** Repeat the assignment and update steps until convergence (i.e., centroids no longer change or a maximum iteration is reached).
3. **Evaluation:**
    - Analyze the clustering output by reviewing cluster centers, inertia (sum of squared distances), and if applicable, visualizing clusters.

Program (Python):

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

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

# Set the number of clusters (for Iris dataset, usually k=3 is appropriate)
k = 3

# Initialize and fit the KMeans clustering model
kmeans = KMeans(n_clusters=k, random_state=42)
kmeans.fit(X)

# Predict the cluster labels for the dataset
labels = kmeans.labels_
centroids = kmeans.cluster_centers_
inertia = kmeans.inertia_

# Print clustering results
print("Cluster Labels:", labels)
print("Cluster Centers:\n", centroids)
print("Inertia:", inertia)

# (Optional) Visualize clusters using the first two features
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', marker='o', edgecolor='k')
plt.scatter(centroids[:, 0], centroids[:, 1], c='red', marker='X', s=200, label='Centroids')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('K-Means Clustering on Iris Dataset')
plt.legend()
plt.show()

```

Output:

```
Cluster Labels: [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 0 2 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 2 0 0 0 0 2 0 0 0 0
 0 0 2 2 0 0 0 0 2 0 2 0 2 0 0 2 2 0 0 0 0 0 2 0 0 0 0 2 0 0 0 2 0 0 0 2 0
 0 2]
Cluster Centers:
 [[6.85384615 3.07692308 5.71538462 2.05384615]
 [5.006      3.428      1.462      0.246     ]
 [5.88360656 2.74098361 4.38852459 1.43442623]]
Inertia: 78.8556658259773

```


K-Means Clustering on Iris Dataset
Centroids
Feature 1
Feature 2
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0
2.0
2.5
3.0
3.5
4.0
4.5

Result:

EXPERIMENT: 10	Implement EM for Bayesian networks
Date:	

Aim:

To implement the Expectation-Maximization (EM) algorithm for parameter estimation in a Bayesian network with latent variables, using synthetic data to learn the conditional probability distributions.

Algorithm:

1. **Initialization**: Start with random guesses for parameters (e.g., prior probabilities and conditional probabilities).
2. **Expectation Step (E-Step)**:
   - Compute the posterior distribution of latent variables given observed data and current parameters.
   - Calculate expected counts for latent variables using Bayes' theorem.
3. **Maximization Step (M-Step)**:
   - Update parameters by maximizing the expected log-likelihood from the E-step.
   - Normalize expected counts to compute new probabilities.
4. **Convergence Check**: Repeat E and M steps until parameters stabilize (change < tolerance).

Program (Python):

```python
import numpy as np

# Generate synthetic data
np.random.seed(42)
p_z_true = 0.6
p_x_given_z_true = {0: 0.3, 1: 0.8}
n_samples = 1000
Z = np.random.binomial(1, p_z_true, n_samples)
X = np.array([np.random.binomial(1, p_x_given_z_true[z]) for z in Z])

# Initialize parameters
p_z_current = 0.5
p_x_z0_current = 0.5
p_x_z1_current = 0.5
max_iter = 100
tolerance = 1e-4
log_likelihoods = []

for iteration in range(max_iter):
    # E-Step: Compute posteriors P(Z=1 | X)
    e_z1 = []
    for x in X:
        if x == 1:
            prob_z1 = p_z_current * p_x_z1_current
            prob_z0 = (1 - p_z_current) * p_x_z0_current
        else:
            prob_z1 = p_z_current * (1 - p_x_z1_current)
            prob_z0 = (1 - p_z_current) * (1 - p_x_z0_current)
        total = prob_z0 + prob_z1
        e_z1.append(prob_z1 / total if total != 0 else 0)
    e_z1 = np.array(e_z1)

    # M-Step: Update parameters
    new_p_z = np.mean(e_z1)
    numerator_x1_z0 = np.sum((X == 1) * (1 - e_z1))
    denominator_z0 = np.sum(1 - e_z1)
    new_p_x_z0 = numerator_x1_z0 / denominator_z0 if denominator_z0 != 0 else 0
    numerator_x1_z1 = np.sum((X == 1) * e_z1)
    denominator_z1 = np.sum(e_z1)
    new_p_x_z1 = numerator_x1_z1 / denominator_z1 if denominator_z1 != 0 else 0

    # Check convergence
    deltas = [
        abs(new_p_z - p_z_current),
        abs(new_p_x_z0 - p_x_z0_current),
        abs(new_p_x_z1 - p_x_z1_current)
    ]
    p_z_current, p_x_z0_current, p_x_z1_current = new_p_z, new_p_x_z0, new_p_x_z1

    # Log likelihood
    log_likelihood = 0
    for x in X:
        if x == 1:
            term = (1 - p_z_current)*p_x_z0_current + p_z_current*p_x_z1_current
        else:
            term = (1 - p_z_current)*(1 - p_x_z0_current) + p_z_current*(1 - p_x_z1_current)
        log_likelihood += np.log(term) if term != 0 else 0
    log_likelihoods.append(log_likelihood)

    if all(delta < tolerance for delta in deltas):
        print(f"Converged at iteration {iteration + 1}")
        break

print(f"Estimated P(Z=1): {p_z_current:.4f} (True: 0.6)")
print(f"Estimated P(X=1|Z=0): {p_x_z0_current:.4f} (True: 0.3)")
print(f"Estimated P(X=1|Z=1): {p_x_z1_current:.4f} (True: 0.8)")
```

Output:

An example output might be:

```
[Running] python -u "c:\Users\Admin\OneDrive\SEM 4\AIML\Code\EXE_10.py"
Converged at iteration 2
Estimated P(Z=1): 0.5000 (True: 0.6)
Estimated P(X=1|Z=0): 0.6080 (True: 0.3)
Estimated P(X=1|Z=1): 0.6080 (True: 0.8)

[Done] exited with code=0 in 1.025 seconds
```

Result:

EXPERIMENT: 11	Build simple NN models
Date:	

Aim

To build a simple feedforward neural network (NN) model for classification using TensorFlow/Keras, trained on the MNIST dataset to recognize handwritten digits (0-9).

Algorithm

1. **Data Preparation**:
   - Load MNIST dataset (28x28 grayscale images of digits).
   - Normalize pixel values to [0, 1].
   - Flatten images to 784-dimensional vectors.
   - Split into training and test sets.
2. **Model Architecture**:
   - Input layer: 784 neurons (one per pixel).
   - Hidden layer: 128 neurons with ReLU activation.
   - Output layer: 10 neurons (one per digit) with softmax activation.
3. **Training**:
   - Loss: Sparse Categorical Crossentropy (for integer labels).
   - Optimizer: Stochastic Gradient Descent (SGD).
   - Metric: Accuracy.
   - Train for 10 epochs.
4. **Evaluation**:
   - Test accuracy calculation.

Program (Python):

```python
import tensorflow as tf
from keras.models import Sequential
from keras.layers import Dense, Flatten
from keras.datasets import mnist
from keras.utils import to_categorical

# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalize the pixel values (scale between 0 and 1)
x_train, x_test = x_train / 255.0, x_test / 255.0

# One-hot encode the labels
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)

# Build a simple Neural Network model
model = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])

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

# Train the model with a validation split of 10%
model.fit(x_train, y_train, epochs=5, batch_size=32, validation_split=0.1)

# Evaluate the model on the test set
loss, accuracy = model.evaluate(x_test, y_test)
print("Test Loss:", loss)
print("Test Accuracy:", accuracy)
```

Output:

[Running] python -u "c:\Users\Admin\OneDrive\SEM 4\AIML\Code\EXE_11.py"

Epoch 1/5

1/1688 [..............................] - ETA: 9:53 - loss: 2.3978 - accuracy: 0.1250

34/1688 [..............................] - ETA: 2s - loss: 1.4847 - accuracy: 0.6250

....

1688/1688 [==============================] - 4s 2ms/step - loss: 0.2789 – accuracy: 0.9211 - val_loss: 0.1401 - val_accuracy: 0.9613

Epoch 2/5

```
   1/1688 [..............................] - ETA: 0s - loss: 0.1407 - accuracy: 0.9062
  27/1688 [..............................] - ETA: 3s - loss: 0.1097 - accuracy: 0.9641
........................
1688/1688 [==============================] - 3s 2ms/step - loss: 0.1241 –
accuracy: 0.9634 - val_loss: 0.1061 - val_accuracy: 0.9677
Epoch 3/5

   1/1688 [..............................] - ETA: 2s - loss: 0.0572 - accuracy: 1.0000
  32/1688 [..............................] - ETA: 2s - loss: 0.0865 - accuracy: 0.9746
........................
1688/1688 [==============================] - 3s 2ms/step - loss: 0.0849 –
accuracy: 0.9745 - val_loss: 0.0826 - val_accuracy: 0.9767
Epoch 4/5

   1/1688 [..............................] - ETA: 3s - loss: 0.1030 - accuracy: 0.9688
  33/1688 [..............................] - ETA: 2s - loss: 0.0796 - accuracy: 0.9773
........................
1688/1688 [==============================] - 3s 2ms/step - loss: 0.0629 –
accuracy: 0.9807 - val_loss: 0.0853 - val_accuracy: 0.9747
Epoch 5/5

   1/1688 [..............................] - ETA: 2s - loss: 0.0758 - accuracy: 0.9688
  35/1688 [..............................] - ETA: 2s - loss: 0.0505 - accuracy: 0.9893
........................
313/313 [==============================] - 0s 866us/step - loss: 0.0805 –
accuracy: 0.9748
Test Loss: 0.08050397038459778
Test Accuracy: 0.9747999906539917

[Done] exited with code=0 in 18.83 seconds
```

Result

EXPERIMENT: 12	Build deep learning NN models
Date:	

Aim

To build a deep learning neural network (DLNN) model using convolutional neural networks (CNNs) for image classification on the MNIST dataset, achieving higher accuracy than a simple feedforward NN.

Algorithm

1. **Data Preparation**:
   - Load MNIST dataset (28x28 grayscale images).
   - Reshape data to include channel dimension (required for CNNs).
   - Normalize pixel values to [0, 1].
   - One-hot encode labels.
2. **Model Architecture**:
   - **Convolutional Layers**:
     - Conv2D (32 filters, 3x3 kernel, ReLU activation).
     - MaxPooling2D (2x2 pool size).
     - Conv2D (64 filters, 3x3 kernel, ReLU activation).
     - MaxPooling2D (2x2 pool size).
   - **Dense Layers**:
     - Flatten layer to convert 2D features to 1D.
     - Dense layer (128 neurons, ReLU activation).
     - Dropout layer (0.5 rate for regularization).
     - Output layer (10 neurons, softmax activation).
3. **Training**:
   - Loss: Categorical Crossentropy.
   - Optimizer: Adam.
   - Metrics: Accuracy.
   - Train for 10 epochs with batch size 64.
4. **Evaluation**:
   - Test accuracy and loss calculation.

Program (Python):

```python
import tensorflow as tf
from keras import layers, models

# Load and preprocess data
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.reshape(-1, 28, 28, 1).astype('float32') / 255.0
x_test = x_test.reshape(-1, 28, 28, 1).astype('float32') / 255.0
y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)

# Build CNN model
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(10, activation='softmax')
])

# Compile and train
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
history = model.fit(x_train, y_train, epochs=10, batch_size=64,
                    validation_data=(x_test, y_test))

# Evaluate
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
print(f"Test accuracy: {test_acc * 100:.2f}%")
```

Output:

[Running] python -u "c:\Users\Admin\OneDrive\SEM 4\AIML\Code\EXE_12.py"

Epoch 1/10

1/938 [..............................] - ETA: 7:48 - loss: 2.3010 - accuracy: 0.0625

....

938/938 [==============================] - 17s 18ms/step - loss: 0.2410 - accuracy: 0.9268 - val_loss: 0.0495 - val_accuracy: 0.9838

Epoch 2/10

1/938 [..............................] - ETA: 19s - loss: 0.0466 - accuracy: 0.9844

4/938 [..............................] - ETA: 15s - loss: 0.0563 - accuracy: 0.9805

....

938/938 [==============================] - 18s 19ms/step - loss: 0.0862 - accuracy: 0.9749 - val_loss: 0.0364 - val_accuracy: 0.9883

Epoch 3/10

1/938 [..............................] - ETA: 29s - loss: 0.0491 - accuracy: 0.9688

4/938 [..............................] - ETA: 16s - loss: 0.0711 - accuracy: 0.9766

....

938/938 [==============================] - 19s 20ms/step - loss: 0.0631 - accuracy: 0.9813 - val_loss: 0.0306 - val_accuracy: 0.9889

Epoch 4/10

1/938 [..............................] - ETA: 16s - loss: 0.0188 - accuracy: 1.0000

4/938 [..............................] - ETA: 18s - loss: 0.0633 - accuracy: 0.9805

....

938/938 [==============================] - 19s 21ms/step - loss: 0.0498 - accuracy: 0.9845 - val_loss: 0.0285 - val_accuracy: 0.9900

Epoch 5/10

1/938 [..............................] - ETA: 20s - loss: 0.0220 - accuracy: 1.0000

4/938 [..............................] - ETA: 16s - loss: 0.0394 - accuracy: 0.9883

....

938/938 [==============================] - 19s 20ms/step - loss: 0.0418 - accuracy: 0.9875 - val_loss: 0.0247 - val_accuracy: 0.9915

Epoch 6/10

1/938 [..............................] - ETA: 24s - loss: 0.0753 - accuracy: 0.9688

3/938 [..............................] - ETA: 25s - loss: 0.0393 - accuracy: 0.9844

……………………

937/938 [============================>.] - ETA: 0s - loss: 0.0356 - accuracy: 0.9893

938/938 [==============================] - 20s 21ms/step - loss: 0.0356 - accuracy: 0.9893 - val_loss: 0.0215 - val_accuracy: 0.9917

Epoch 7/10

1/938 [..............................] - ETA: 18s - loss: 0.0489 - accuracy: 0.9688

4/938 [..............................] - ETA: 18s - loss: 0.0277 - accuracy: 0.9883

……………………

938/938 [==============================] - 20s 21ms/step - loss: 0.0315 - accuracy: 0.9903 - val_loss: 0.0225 - val_accuracy: 0.9923

Epoch 8/10

1/938 [..............................] - ETA: 22s - loss: 0.0513 - accuracy: 0.9688

4/938 [..............................] - ETA: 17s - loss: 0.0186 - accuracy: 0.9922

……………………

938/938 [==============================] - 21s 22ms/step - loss: 0.0277 - accuracy: 0.9913 - val_loss: 0.0233 - val_accuracy: 0.9923

Epoch 9/10

1/938 [..............................] - ETA: 26s - loss: 0.0122 - accuracy: 1.0000

3/938 [..............................] - ETA: 23s - loss: 0.0093 - accuracy: 1.0000

……………………

938/938 [==============================] - 19s 21ms/step - loss: 0.0221 - accuracy: 0.9929 - val_loss: 0.0255 - val_accuracy: 0.9922

Test accuracy: 99.22%

[Done] exited with code=0 in 196.858 seconds

Result