

LIST OF EXPERIMENTS

- 1. Implementation of Uninformed search algorithms (BFS, DFS)**
- 2. Implementation of Informed search algorithms (A*, memory-bounded A*)**
- 3. Implement naive Bayes models**
- 4. Implement Bayesian Networks**
- 5. Build Regression models**
- 6. Build decision trees and random forests**
- 7. Build SVM models**
- 8. Implement ensembling techniques**
- 9. Implement clustering algorithms**
- 10. Implement EM for Bayesian networks**
- 11. Build simple NN models**
- 12. Build deep learning NN models**

Date:**Aim:**

To write a Python program to implement Breadth First Search (BFS).

Algorithm:

- Step 1. Start
- Step 2. Put any one of the graph's vertices at the back of the queue.
- Step 3. Take the front item of the queue and add it to the visited list.
- Step 4. Create a list of that vertex's adjacent nodes. Add those which are not within the visited list to the rear of the queue.
- Step 5. Continue steps 3 and 4 till the queue is empty.
- Step 6. Stop

Program:

```
graph = {
    '5' : ['3','7'],
    '3' : ['2', '4'],
    '7' : ['8'],
    '2' : [],
    '4' : ['8'],
    '8' : []
}

visited = [] # List for visited nodes.
queue = []   #Initialize a queue

def bfs(visited, graph, node): #function for BFS
    visited.append(node)
    queue.append(node)

    while queue:           # Creating loop to visit each node
        m = queue.pop(0)
        print (m, end = " ")

        for neighbour in graph[m]:
            if neighbour not in visited:
                visited.append(neighbour)
                queue.append(neighbour)

# Driver Code
print("Following is the Breadth-First Search")
bfs(visited, graph, '5') # function calling
```

Viva questions:

1. What is BFS and how does it differ from other search algorithms such as DFS or A* search?
2. Can you describe the steps of a BFS algorithm and explain how it works?
3. Can you explain the time and space complexity of a BFS algorithm?
4. Can you give an example of a real-world problem that can be solved using BFS?
5. What does the "visited array" in BFS refer to?

Result:

Thus the Python program to implement Breadth First Search (BFS) was developed successfully.

Ex. No.1.B

UNINFORMED SEARCH ALGORITHM - DFS

Date:

Aim:

To write a Python program to implement Depth First Search (DFS).

Algorithm:

Step 1.Start

Step 2.Put any one of the graph's vertex on top of the stack.

Step 3.After that take the top item of the stack and add it to the visited list of the vertex.

Step 4.Next, create a list of that adjacent node of the vertex. Add the ones which aren't in the visited list of vertexes to the top of the stack.

Step 5.Repeat steps 3 and 4 until the stack is empty.

Step 6.Stop

Program:

```
graph = {
    '5': ['3','7'],
    '3': ['2', '4'],
    '7': ['8'],
    '2': [],
    '4': ['8'],
    '8': []
}

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node): #function for dfs
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

# Driver Code
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

Viva Questions:

1. What is DFS and how does it differ from other search algorithms such as BFS or A* search?
2. Can you describe the steps of a DFS algorithm and explain how it works?
3. How does DFS handle loops or repeated states in graph?
4. Can you explain the time and space complexity of a DFS algorithm?
5. Can you give an example of a real-world problem that can be solved using DFS?

Result:

Thus the Python program to implement Depth First Search (DFS) was developed successfully.

Ex. No.2. A

INFORMED SEARCH ALGORITHM

Date:

A* SEARCH

Aim:

To write a Python program to implement A* search algorithm.

Algorithm:

- Step 1: Create a priority queue and push the starting node onto the queue. Initialize minimum value (min_index) to location 0.
- Step 2: Create a set to store the visited nodes.
- Step 3: Repeat the following steps until the queue is empty:
 - 3.1: Pop the node with the lowest cost + heuristic from the queue.
 - 3.2: If the current node is the goal, return the path to the goal.
 - 3.3: If the current node has already been visited, skip it.
 - 3.4: Mark the current node as visited.
 - : Expand the current node and add its neighbors to the queue.
- Step 4: If the queue is empty and the goal has not been found, return None (no path found).
- Step 5: Stop

Program:

```
import heapq

class Node:
    def __init__(self, state, parent, cost, heuristic):
        self.state = state
        self.parent = parent
        self.cost = cost
        self.heuristic = heuristic

    def __lt__(self, other):
        return (self.cost + self.heuristic) < (other.cost + other.heuristic)

def astar(start, goal, graph):
    heap = []
    heapq.heappush(heap, (0, Node(start, None, 0, 0)))
    visited = set()

    while heap:
        (cost, current) = heapq.heappop(heap)

        if current.state == goal:
            path = []
            while current is not None:
                path.append(current.state)
```

```

        current = current.parent
        # Return reversed path
        return path[::-1]

    if current.state in visited:
        continue

    visited.add(current.state)

    for state, cost in graph[current.state].items():
        if state not in visited:
            heuristic = 0 # replace with your heuristic function
            heappq.heappush(heap, (cost, Node(state, current, current.cost + cost, heuristic)))

    return None # No path found

graph = {
    'A': {'B': 1, 'D': 3},
    'B': {'A': 1, 'C': 2, 'D': 4},
    'C': {'B': 2, 'D': 5, 'E': 2},
    'D': {'A': 3, 'B': 4, 'C': 5, 'E': 3},
    'E': {'C': 2, 'D': 3}
}
start = 'A'
goal = 'E'

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

```

Viva Questions:

1. What is A* search and what makes it different from other search algorithms?
2. How does the A* algorithm choose which node to expand next?
3. Can you explain how the heuristic function is used in the A* algorithm and what role it plays in the search process?
4. How does the cost function used in A* search differ from the heuristic function?
5. What are the advantages and disadvantages of using A* search compared to other search algorithms like breadth-first search or depth-first search?

Result:

Thus the python program for A* Search was developed and the output was verified successfully.

Aim:

To write a Python program to implement memory- bounded A* search algorithm.

Algorithm:

Step 1: Create a priority queue and push the starting node onto the queue.

Step 2: Create a set to store the visited nodes.

Step 3: Set a counter to keep track of the number of nodes expanded.

Step 4: Repeat the following steps until the queue is empty or the node counter exceeds the max_nodes:

4.1: Pop the node with the lowest cost + heuristic from the queue.

4.2: If the current node is the goal, return the path to the goal.

4.3: If the current node has already been visited, skip it.

4.4: Mark the current node as visited.

: Increment the node counter.

: Expand the current node and add its neighbors to the queue.

Step 5: If the queue is empty and the goal has not been found, return None (no path found).

Step 6: Stop

Program:

```
import heapq
```

```
class Node:
```

```
    def __init__(self, state, parent, cost, heuristic):
```

```
        self.state = state
```

```
        self.parent = parent
```

```
        self.cost = cost
```

```
        self.heuristic = heuristic
```

```
    def __lt__(self, other):
```

```
        return (self.cost + self.heuristic) < (other.cost + other.heuristic)
```

```
def astar(start, goal, graph, max_nodes):
```

```
    heap = []
```

```
    heapq.heappush(heap, (0, Node(start, None, 0, 0)))
```

```
    visited = set()
```

```
    node_counter = 0
```

```
    while heap and node_counter < max_nodes:
```

```
        (cost, current) = heapq.heappop(heap)
```

```
        if current.state == goal:
```



```

    path = []
    while current is not None:
        path.append(current.state)
        current = current.parent
    return path[::-1]

    if current.state in visited:
        continue

    visited.add(current.state)

    node_counter += 1

    for state, cost in graph[current.state].items():
        if state not in visited:
            heuristic = 0
            heapq.heappush(heap, (cost, Node(state, current, current.cost + cost, heuristic)))

    return None

# Example usage

graph = {'A': {'B': 1, 'C': 4},
        'B': {'A': 1, 'C': 2, 'D': 5},
        'C': {'A': 4, 'B': 2, 'D': 1},
        'D': {'B': 5, 'C': 1}}

start = 'A'
goal = 'D'
max_nodes = 10

result = astar(start, goal, graph, max_nodes)
print(result)

```

Viva Questions:

1. What is memory bounded A* search and how does it differ from traditional A* search?
2. How does memory bounded A* search help in handling large state spaces?
3. What is the basic idea behind memory bounded A* search and how does it work?
4. Can you explain the trade-off between optimality and memory usage in memory bounded A* search?
5. How does memory bounded A* search handle the problem of node replanning and how does it impact the performance of the search?

Result:

Thus the python program for memory-bounded A* search was developed and the output was verified successfully.

Date:**Aim:**

To write a python program to implement Naïve Bayes model.

Algorithm:

- Step 1. Load the libraries: import the required libraries such as pandas, numpy, and sklearn.
- Step 2. Load the data into a pandas dataframe.
- Step 3. Clean and preprocess the data as necessary. For example, you can handle missing values, convert categorical variables into numerical variables, and normalize the data.
- Step 4. Split the data into training and test sets using the **train_test_split** function from scikit-learn.
- Step 5. Train the Gaussian Naive Bayes model using the training data.
- Step 6. Evaluate the performance of the model using the test data and the **accuracy_score** function from scikit-learn.
- Step 7. Finally, you can use the trained model to make predictions on new data.

Program:

```
import pandas as pd
import numpy as np
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

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

# Split the data into training and test sets
X = df.drop('buy_computer', axis=1)
y = df['buy_computer']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)

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

# Test the model
y_pred = model.predict(X_test.values)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

```
# Make a prediction on new data
new_data = np.array([[35, 60000, 1, 100]])
prediction = model.predict(new_data)
print("Prediction:", prediction)
```

Sample data.csv file

```
age,income,student,credit_rating,buy_computer
30,45000,0,10,0
32,54000,0,100,0
35,61000,1,10,1
40,65000,0,50,1
45,75000,0,100,0
```

Viva Questions:

1. What is Naive Bayes and how does it work?
2. Can you discuss the different types of Naive Bayes models?
3. Why is Naive Bayes considered "naive"?
4. What are the advantages and disadvantages of using Naive Bayes?
5. Can you give some real-world examples where Naive Bayes has been applied successfully?

Result:

Thus the Python program for implementing Naïve Bayes model was developed and the output was verified successfully.

Date:**Aim:**

To write a python program to implement a Bayesian network for the Monty Hall problem.

Algorithm:

- Step 1. Start by importing the required libraries such as math and pomegranate.
- Step 2. Define the discrete probability distribution for the guest's initial choice of door
- Step 3. Define the discrete probability distribution for the prize door
- Step 4. Define the conditional probability table for the door that Monty picks based on the guest's choice and the prize door
- Step 5. Create State objects for the guest, prize, and Monty's choice
- Step 6. Create a Bayesian Network object and add the states and edges between them
- Step 7. Bake the network to prepare for inference
- Step 8. Use the predict_proba method to calculate the beliefs for a given set of evidence
- Step 9. Display the beliefs for each state as a string.
- Step 10. Stop

Program:

```
import math
from pomegranate import *

# Initially the door selected by the guest is completely random
guest = DiscreteDistribution({'A': 1./3, 'B': 1./3, 'C': 1./3})

# The door containing the prize is also a random process
prize = DiscreteDistribution({'A': 1./3, 'B': 1./3, 'C': 1./3})

# The door Monty picks, depends on the choice of the guest and the prize door
monty = ConditionalProbabilityTable(
    [['A', 'A', 'A', 0.0],
     ['A', 'A', 'B', 0.5],
     ['A', 'A', 'C', 0.5],
     ['A', 'B', 'A', 0.0],
```

```

['A', 'B', 'B', 0.0],
['A', 'B', 'C', 1.0],
['A', 'C', 'A', 0.0],
['A', 'C', 'B', 1.0],
['A', 'C', 'C', 0.0],
['B', 'A', 'A', 0.0],
['B', 'A', 'B', 0.0],
['B', 'A', 'C', 1.0],
['B', 'B', 'A', 0.5],
['B', 'B', 'B', 0.0],
['B', 'B', 'C', 0.5],
['B', 'C', 'A', 1.0],
['B', 'C', 'B', 0.0],
['B', 'C', 'C', 0.0],
['C', 'A', 'A', 0.0],
['C', 'A', 'B', 1.0],
['C', 'A', 'C', 0.0],
['C', 'B', 'A', 1.0],
['C', 'B', 'B', 0.0],
['C', 'B', 'C', 0.0],
['C', 'C', 'A', 0.5],
['C', 'C', 'B', 0.5],
['C', 'C', 'C', 0.0]], [guest, prize])

```

```

d1 = State(guest, name="guest")
d2 = State(prize, name="prize")
d3 = State(monty, name="monty")

```

```

# Building the Bayesian Network

```

```

network = BayesianNetwork("Solving the Monty Hall Problem With Bayesian Networks")
network.add_states(d1, d2, d3)
network.add_edge(d1, d3)
network.add_edge(d2, d3)
network.bake()

```

```

# Compute the probabilities for each scenario

```

```

beliefs = network.predict_proba({'guest': 'A'})
print("\n".join("{}\t{}".format(state.name, str(belief)) for state, belief in zip(network.states,
beliefs)))

```

```

beliefs = network.predict_proba({'guest': 'A', 'monty': 'B'})
print("\n".join("{}\t{}".format(state.name, str(belief)) for state, belief in zip(network.states,
beliefs)))

```

```

beliefs = network.predict_proba({'guest': 'A', 'prize': 'B'})
print("\n".join("{}\t{}".format(state.name, str(belief)) for state, belief in zip(network.states,
beliefs)))

```

Viva Questions:

1. What is a Bayesian network and how does it work?
2. What are the key differences between Bayesian networks and other probabilistic models such as Naive Bayes or Markov Networks?
3. What is the purpose of the directed edges in a Bayesian network and how are they used to perform probabilistic inference?
4. Can you discuss some of the challenges in constructing Bayesian networks and how they can be addressed?
5. What are some real-world applications of Bayesian networks and how have they been used in these applications?

Result:

Thus, the Python program for implementing Bayesian Networks was successfully developed and the output was verified.

Date:

Aim:

To write a Python program to build Regression models

Algorithm:

- Step 1. Import necessary libraries: numpy, pandas, matplotlib.pyplot, LinearRegression, mean_squared_error, and r2_score.
- Step 2. Create a numpy array for waist and weight values and store them in separate variables.
- Step 3. Create a pandas DataFrame with waist and weight columns using the numpy arrays.
- Step 4. Extract input (X) and output (y) variables from the DataFrame.
- Step 5. Create an instance of LinearRegression model.
- Step 6. Fit the LinearRegression model to the input and output variables.
- Step 7. Create a new DataFrame with a single value of waist.
- Step 8. Use the predict() method of the LinearRegression model to predict the weight for the new waist value.
- Step 9. Calculate the mean squared error and R-squared values using mean_squared_error() and r2_score() functions respectively.
- Step 10. Plot the actual and predicted values using matplotlib.pyplot.scatter() and matplotlib.pyplot.plot() functions.

Program:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# import sample data using pandas
waist = np.array([70, 71, 72, 73, 74, 75, 76, 77, 78, 79])
weight = np.array([55, 57, 59, 61, 63, 65, 67, 69, 71, 73])
data = pd.DataFrame({'waist': waist, 'weight': weight})

# extract input and output variables
X = data[['waist']]
y = data['weight']

# fit a linear regression model
model = LinearRegression()
model.fit(X, y)
```

```

# make predictions on new data
new_data = pd.DataFrame({'waist': [80]})
predicted_weight = model.predict(new_data[['waist']])
print("Predicted weight for new waist value:", int(predicted_weight))

#calculate MSE and R-squared
y_pred = model.predict(X)
mse = mean_squared_error(y, y_pred)
print('Mean Squared Error:', mse)
r2 = r2_score(y, y_pred)
print('R-squared:', r2)

# plot the actual and predicted values
plt.scatter(X, y, marker='*', edgecolors='g')
plt.scatter(new_data, predicted_weight, marker='*', edgecolors='r')
plt.plot(X, y_pred, color='y')
plt.xlabel('Waist (cm)')
plt.ylabel('Weight (kg)')
plt.title('Linear Regression Model')
plt.show()

```

Viva questions:

1. What is a regression model?
2. What are the different types of regression models?
3. How do you determine which predictor variables to include in a regression model?
4. What is the difference between simple linear regression and multiple linear regression?
5. What are some common challenges in regression analysis and how can they be overcome?

Result:

Thus the Python program to build a simple linear Regression model was developed successfully.

Date:**Aim:**

To write a Python program to build decision tree and random forest.

Algorithm:

- Step 1. Import necessary libraries: numpy, matplotlib, seaborn, pandas, train_test_split, LabelEncoder, DecisionTreeClassifier, plot_tree, and RandomForestClassifier.
- Step 2. Read the data from 'flowers.csv' into a pandas DataFrame.
- Step 3. Extract the features into an array X, and the target variable into an array y.
- Step 4. Encode the target variable using the LabelEncoder.
- Step 5. Split the data into training and testing sets using train_test_split function.
- Step 6. Create a DecisionTreeClassifier object, fit the model to the training data, and visualize the decision tree using plot_tree.
- Step 7. Create a RandomForestClassifier object with 100 estimators, fit the model to the training data, and visualize the random forest by displaying 6 trees.
- Step 8. Print the accuracy of the decision tree and random forest models using the score method on the test data.

Program:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.ensemble import RandomForestClassifier

# read the data
data = pd.read_csv('flowers.csv')
X = data.iloc[:, :-1].values
y = data.iloc[:, -1].values

# encode the labels
le = LabelEncoder()
y = le.fit_transform(y)

# split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

# create and fit a decision tree model
tree = DecisionTreeClassifier().fit(X_train, y_train)
```

```

# visualize the decision tree
plt.figure(figsize=(10,6))
plot_tree(tree, filled=True)
plt.title("Decision Tree")
plt.show()

# create and fit a random forest model
rf = RandomForestClassifier(n_estimators=100, random_state=0).fit(X_train, y_train)

# visualize the random forest
plt.figure(figsize=(20,12))
for i, tree_in_forest in enumerate(rf.estimators_[ :6]):
    plt.subplot(2, 3, i+1)
    plt.axis('off')
    plot_tree(tree_in_forest, filled=True, rounded=True)
    plt.title("Tree " + str(i+1))
plt.suptitle("Random Forest")
plt.show()

# calculate and print the accuracy of decision tree and random forest
print("Accuracy of decision tree: {:.2f}".format(tree.score(X_test, y_test)))
print("Accuracy of random forest: {:.2f}".format(rf.score(X_test, y_test)))

```

Sample flowers.csv

```

Sepal_length,Sepal_width,Petal_length,Petal_width,Flower
4.6,3.2,1.4,0.2,Rose
5.3,3.7,1.5,0.2,Rose
5.3,3.1,1.4,0.2,Rose
7.3,2.4,4.7,1.4,Jasmin
6.4,3.2,4.5,1.5,Jasmin
7.1,3.5,5.9,2.1,Lotus
6.3,2.9,5.6,1.8,Lotus

```

Viva Questions:

1. What is the difference between a decision tree and a random forest?
2. How do you determine the best split at each node of a decision tree?
3. How do you prevent overfitting when building a decision tree?
4. How does the number of trees in a random forest affect the accuracy and performance of the model?
5. Can you explain how feature importance is calculated in a random forest model?

Result:

Thus the Python program to build decision tree and random forest was developed successfully.

Date:**Aim:**

To write a Python program to build SVM model.

Algorithm:

- Step 1.Import the necessary libraries (matplotlib.pyplot, numpy, and svm from sklearn).
- Step 2.Define the features (X) and labels (y) for the fruit dataset.
- Step 3.Create an SVM classifier with a linear kernel using svm.SVC(kernel='linear').
- Step 4.Train the classifier on the fruit data using clf.fit(X, y).
- Step 5.Plot the fruits and decision boundary using plt.scatter(X[:, 0], X[:, 1], c=colors),
where colors is a list of colors assigned to each fruit based on its label.
- Step 6.Create a meshgrid to evaluate the decision function using
np.meshgrid(np.linspace(xlim[0], xlim[1], 100), np.linspace(ylim[0], ylim[1], 100)).
- Step 7.Use the decision function to create a contour plot of the decision boundary and
margins using ax.contour(xx, yy, Z, colors='k', levels=[-1, 0, 1], alpha=0.5,
linestyles=['--', '-', '--']).
- Step 8.Show the plot using plt.show().

Program:

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn import svm

# Define the fruit features (size and color)
X = np.array([[5, 2], [4, 3], [1, 7], [2, 6], [5, 5], [7, 1], [6, 2], [5, 3], [3, 6], [2, 7], [6, 3], [3, 3],
[1, 5], [7, 3], [6, 5], [2, 5], [3, 2], [7, 5], [1, 3], [4, 2]])

# Define the fruit labels (0=apples, 1=oranges)
y = np.array([0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0])

# Create an SVM classifier with a linear kernel
clf = svm.SVC(kernel='linear')

# Train the classifier on the fruit data
clf.fit(X, y)

# Plot the fruits and decision boundary
colors = ['red' if label == 0 else 'yellow' for label in y]
plt.scatter(X[:, 0], X[:, 1], c=colors)
ax = plt.gca()
ax.set_xlabel('Size')
```

```
ax.set_ylabel('Color')
xlim = ax.get_xlim()
ylim = ax.get_ylim()

# Create a meshgrid to evaluate the decision function
xx, yy = np.meshgrid(np.linspace(xlim[0], xlim[1], 100), np.linspace(ylim[0], ylim[1], 100))
Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Plot the decision boundary and margins
ax.contour(xx, yy, Z, colors='k', levels=[-1, 0, 1], alpha=0.5, linestyle=['--', '-', '--'])
plt.show()
```

Viva Questions:

1. What is an SVM model?
2. What is the kernel function in SVM?
3. How do you choose the optimal value of C in SVM?
4. What is the decision boundary in SVM?
5. What is the purpose of the mesh grid in the code?

Result:

Thus, the Python program to build an SVM model was developed, and the output was successfully verified.

EX NO.: 8

Implement EM for Bayesian Networks

DATE :

Aim:

To write a python program to implement EM for Bayesian Networks.

Algorithm:

1. Start with a Bayesian network that has some variables that are not directly observed. For example, suppose we have a Bayesian network with variables A, B, C, and D, where A and D are observed and B and C are latent.
2. Initialize the parameters of the network. This includes the conditional probabilities for each variable given its parents, as well as the prior probabilities for the root variables.
3. E-step: Compute the expected sufficient statistics for the latent variables. This involves computing the posterior probability distribution over the latent variables given the observed data and the current parameter estimates. This can be done using the forward-backward algorithm or the belief propagation algorithm.
4. M-step: Update the parameter estimates using the expected sufficient statistics computed in step 3. This involves maximizing the likelihood of the data with respect to the parameters of the network, given the expected sufficient statistics.
5. Repeat steps 3-4 until convergence. Convergence can be measured by monitoring the change in the log-likelihood of the data, or by monitoring the change in the parameter estimates.

Program :

```
import numpy as np
import math

# Define the Bayesian network structure
nodes = ['A', 'B', 'C', 'D']
parents = {
    'B': ['A'],
    'C': ['A'],
    'D': ['B', 'C']
}
```

```

probabilities = {
    'A': np.array([0.6, 0.4]),
    'B': np.array([[0.3, 0.7], [0.9, 0.1]]),
    'C': np.array([[0.2, 0.8], [0.7, 0.3]]),
    'D': np.array([[[0.1, 0.9], [0.6, 0.4]], [[0.7, 0.3], [0.8, 0.2]]])
}

# Define the observed data
data = {'A': 1, 'D': 0}

# Define the EM algorithm
def em_algorithm(nodes, parents, probabilities, data, max_iterations=100):
    # Initialize the parameters
    for node in nodes:
        if node not in data:
            probabilities[node] = np.ones(probabilities[node].shape) /
probabilities[node].shape[0]

    # Run the EM algorithm
    for iteration in range(max_iterations):
        # E-step: Compute the expected sufficient statistics
        expected_counts = {}
        for node in nodes:
            if node not in data:
                expected_counts[node] = np.zeros(probabilities[node].shape)

        # Compute the posterior probability distribution over the latent variables given
the observed data
        joint_prob = np.ones(probabilities['A'].shape)
        for node in nodes:
            if node in data:
                joint_prob *= probabilities[node][data[node]]
            else:
                parent_probs = [probabilities[parent][data[parent]] for parent in
parents[node]]
                joint_prob *= probabilities[node][tuple(parent_probs)]
        posterior = joint_prob / np.sum(joint_prob)

        # Compute the expected sufficient statistics
        for node in nodes:
            if node not in data:
                if node in parents:
                    parent_probs = [probabilities[parent][data[parent]] for parent in
parents[node]]

```

```

        expected_counts[node] = np.sum(posterior *
probabilities[node][tuple(parent_probs)], axis=0)
    else:
        expected_counts[node] = np.sum(posterior * probabilities[node], axis=0)

    # M-step: Update the parameter estimates
    for node in nodes:
        if node not in data:
            probabilities[node] = expected_counts[node] / np.sum(expected_counts[node])

    # Check for convergence
    if iteration > 0 and np.allclose(prev_probabilities, probabilities):
        break
    prev_probabilities = np.copy(probabilities)

    return probabilities

# Run the EM algorithm
probabilities = em_algorithm(nodes, parents, probabilities, data)

# Print the final parameter estimates
for node in nodes:
    print(node, probabilities[node])

```

Result :

Thus the Python program to Implement EM for Bayesian Networks was developed successfully.

EX NO.: 9

Build Simple NN Models

DATE :

Aim:

To write a python program to build simple NN models.

Algorithm:

1. Define the input and output data.
2. Choose the number of layers and neurons in each layer. This depends on the problem you are trying to solve.
3. Define the activation function for each layer. Common choices are ReLU, sigmoid, and tanh.
4. Initialize the weights and biases for each neuron in the network. This can be done randomly or using a pre-trained model.
5. Define the loss function and optimizer to be used during training. The loss function measures how well the model is doing, while the optimizer updates the weights and biases to minimize the loss.
6. Train the model on the input data using the defined loss function and optimizer. This involves forward propagation to compute the output of the model, and backpropagation to compute the gradients of the loss with respect to the weights and biases. The optimizer then updates the weights and biases based on the gradients.
7. Evaluate the performance of the model on new data using metrics such as accuracy, precision, recall, and F1 score.

Program:

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense

# Define the input and output data
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])

# Define the model
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'])

# Fit the model to the data
model.fit(X, y, epochs=1000, batch_size=4)

# Evaluate the model on new data
test_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
predictions = model.predict(test_data)
print(predictions)
```

Result:

Thus the Python program to build simple NN Models was developed successfully.

EX NO.: 10

Deep Learning NN Models

DATE :

Aim:

To write a python program to implement deep learning of NN models.

Algorithm:

1. Import the necessary libraries, such as numpy and keras.
2. Load or generate your dataset. This can be done using numpy or any other data manipulation library.
3. Preprocess your data by performing any necessary normalization, scaling, or other transformations.
4. Define your neural network architecture using the Keras Sequential API. Add layers to the model using the add() method, specifying the number of units, activation function, and input dimensions for each layer.
5. Compile your model using the compile() method. Specify the loss function, optimizer, and any evaluation metrics you want to use.
6. Train your model using the fit() method. Specify the training data, validation data, batch size, and number of epochs.
7. Evaluate your model using the evaluate() method. This will give you the loss and accuracy metrics on the test set.
8. Use your trained model to make predictions on new data using the predict() method.

Program:

```
# Import necessary libraries
import numpy as np
from keras.models import Sequential
from keras.layers import Dense

# Define the neural network model
model = Sequential()
model.add(Dense(units=64, activation='relu', input_dim=100))
model.add(Dense(units=10, activation='softmax'))

# Compile the model
```

```
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])

# Generate some random data for training and testing
data = np.random.random((1000, 100))
labels = np.random.randint(10, size=(1000, 1))
one_hot_labels = keras.utils.to_categorical(labels, num_classes=10)

# Train the model on the data
model.fit(data, one_hot_labels, epochs=10, batch_size=32)

# Evaluate the model on a test set
test_data = np.random.random((100, 100))
test_labels = np.random.randint(10, size=(100, 1))
test_one_hot_labels = keras.utils.to_categorical(test_labels, num_classes=10)
loss_and_metrics = model.evaluate(test_data, test_one_hot_labels, batch_size=32)
print("Test loss:", loss_and_metrics[0])
print("Test accuracy:", loss_and_metrics[1])
```

Result:

Thus the Python program to implement deep learning of NN Models was developed successfully.