IMPLEMENTATION OF BAYESIAN BELIEF NETWORKS

Exp.No : 1
Date :

Aim: Write a program to construct a Bayesian network considering medical data. Use this mode demonstrate the diagnosis of heart patients using a standard Heart Disease Data Set.

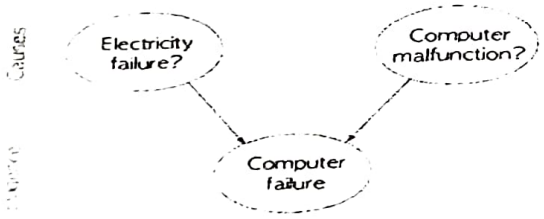
Theory: A Bayesian network is a directed acyclic graph in which each edge corresponds to a conditional dependency, and each node corresponds to a unique random variable.

Bayesian network consists of two major parts: a directed acyclic graph and a set of conditional probability distributions

- The directed acyclic graph is a set of random variables represented by nodes.
- The conditional probability distribution of a node (random variable) is defined for every possible outcome of the preceding causal node(s).

For illustration, consider the following example. Suppose we attempt to turn on our computer, but the computer does not start (observation/evidence). We would like to know which of the possible causes of computer failure is more likely. In this simplified illustration, we assume only two possib causes of this misfortune: electricity failure and computer malfunction.

The corresponding directed acyclic graph is depicted in below figure.



The goal is to calculate the posterior conditional probability distribution of each of the possible unobserved causes given the observed evidence, i.e. P [Cause | Evidence].

Data Set:

Title: Heart Disease Databases
The Cleveland database contains 76 attributes, but all published experiments refer to using a subset of 14 of them. In particular, the Cleveland database is the only one that has been used by ML researchers to this date. The "Heartdisease" field refers to the presence of heart disease in the patient. It is integer valued from 0 (no presence) to 4.

Algorithm:

Step 1: Importing the required libraries
Step 2: Defining and visualizing the Model
Step 3: Print probability distributions
Step 4: Calculate the probability of a burglary if John and Mary calls (0: True, 1: False)

Step 5: Calculate the probability of alarm starting if there is a burglary and an earthquake (0: True, 1: False)
Step 6: Print posterior probability

Program:

```
import pgmpy.models
import pgmpy.inference
import networkx as nx
import pylab as plt
model = pgmpy.models.BayesianModel([('Burglary', 'Alarm'),
                                    ('Earthquake', 'Alarm'),
                                    ('Alarm', 'JohnCalls'),
                                    ('Alarm', 'MaryCalls')])
cpd_burglary = pgmpy.factors.discrete.TabularCPD('Burglary', 2, [[0.001], [0.999]])
cpd_earthquake = pgmpy.factors.discrete.TabularCPD('Earthquake', 2, [[0.002], [0.998]])
cpd_alarm = pgmpy.factors.discrete.TabularCPD('Alarm', 2, [[0.95, 0.94, 0.29, 0.001],
                                                           [0.05, 0.06, 0.71, 0.999]],
                                    evidence=['Burglary', 'Earthquake'],
                                    evidence_card=[2, 2])
cpd_john = pgmpy.factors.discrete.TabularCPD('JohnCalls', 2, [[0.90, 0.05],
                                                              [0.10, 0.95]],
                                    evidence=['Alarm'],
                                    evidence_card=[2])
cpd_mary = pgmpy.factors.discrete.TabularCPD('MaryCalls', 2, [[0.70, 0.01],
                                                              [0.30, 0.99]],
                                    evidence=['Alarm'],
                                    evidence_card=[2])
model.add_cpds(cpd_burglary, cpd_earthquake, cpd_alarm, cpd_john, cpd_mary)
model.check_model()
# Print probability distributions
print('Probability distribution, P(Burglary)')
print(cpd_burglary)
print()
print('Probability distribution, P(Earthquake)')
print(cpd_earthquake)
print()
print('Joint probability distribution, P(Alarm | Burglary, Earthquake)')
print(cpd_alarm)
print()
print('Joint probability distribution, P(JohnCalls | Alarm)')
print(cpd_john)
print()
print('Joint probability distribution, P(MaryCalls | Alarm)')
print(cpd_mary)
print()
```

```
infer = pgmpy.inference.VariableElimination(model)

# Calculate the probability of a burglary if John and Mary calls (0: True, 1: False)
posterior_probability = infer.query(['Burglary'], evidence={'JohnCalls': 0, 'MaryCalls': 0})
# Print posterior probability
print('Posterior probability of Burglary if JohnCalls(True) and MaryCalls(True)')
print(posterior_probability)
print()


# Calculate the probability of alarm starting if there is a burglary and an earthquake (0: True, 1: False)
posterior_probability = infer.query(['Alarm'], evidence={'Burglary': 0, 'Earthquake': 0})
# Print posterior probability
print('Posterior probability of Alarm sounding if Burglary(True) and Earthquake(True)')
print(posterior_probability)
```

Output:

```
Probability distribution, P(Burglary)
+-------------+-------+
| Burglary(0) | 0.001 |
+-------------+-------+
| Burglary(1) | 0.999 |
+-------------+-------+

Probability distribution, P(Earthquake)
+---------------+-------+
| Earthquake(0) | 0.002 |
+---------------+-------+
| Earthquake(1) | 0.998 |
+---------------+-------+

Joint probability distribution, P(Alarm | Burglary, Earthquake)
+-----------+--------------+--------------+--------------+--------------
+
| Burglary  | Burglary(0)  | Burglary(0)  | Burglary(1)  | Burglary(1)
|
+-----------+--------------+--------------+--------------+--------------
+
| Earthquake | Earthquake(0) | Earthquake(1) | Earthquake(0) | Earthquake(1)
|
+-----------+--------------+--------------+--------------+--------------
+
| Alarm(0)  | 0.95         | 0.94         | 0.29         | 0.001
|
+-----------+--------------+--------------+--------------+--------------
+
| Alarm(1)  | 0.05         | 0.06         | 0.71         | 0.999
|
```

```
+------------+--------------+--------------+--------------+--------------
+
```

Joint probability distribution, P(JohnCalls | Alarm)

Alarm	Alarm(0)	Alarm(1)
JohnCalls(0)	0.9	0.05
JohnCalls(1)	0.1	0.95

Joint probability distribution, P(MaryCalls | Alarm)

Alarm	Alarm(0)	Alarm(1)
MaryCalls(0)	0.7	0.01
MaryCalls(1)	0.3	0.99

Posterior probability of Burglary if JohnCalls(True) and MaryCalls(True)

Burglary	phi(Burglary)
Burglary(0)	0.2842
Burglary(1)	0.7158

Posterior probability of Alarm sounding if Burglary(True) and Earthquake(True
)

Alarm	phi(Alarm)
Alarm(0)	0.9500
Alarm(1)	0.0500

Result: Thus written a program to construct a Bayesian network considering medical data.

APPROXIMATE INFERENCES IN BAYESIAN NETWORK

Exp.No : 2
Date :

Aim: Write a program to construct a Bayesian network and get it approximate inferences.

Theory: A method of estimating probabilities in Bayesian networks also called 'Monte Carlo' algorithms. We will discuss two types of algorithms: direct sampling and Markov chain sampling. Exact inference becomes intractable for large multiply-connected networks Variable elimination can have exponential time and space complexity exact inference is strictly HARDER than NP-complete problems (#P-hard). Computing posterior and marginal probabilities constitutes the backbone of almost all inferences in Bayesian networks. These computations are known to be intractable in general, both to compute exactly and to approximate (e.g., by sampling algorithms).

Algorithm:

Defining Bayesian Structure

Defining the CPDs:

Associating the CPDs with the network structure.

Inferring the posterior probability

Program:

```
from pgmpy.models import BayesianNetwork
from pgmpy.factors.discrete import TabularCPD
import networkx as nx
import pylab as plt
# Defining Bayesian Structure
model = BayesianNetwork([('Guest', 'Host'), ('Price', 'Host')])
# Defining the CPDs:
cpd_guest = TabularCPD('Guest', 3, [[0.33], [0.33], [0.33]])
cpd_price = TabularCPD('Price', 3, [[0.33], [0.33], [0.33]])
cpd_host = TabularCPD('Host', 3, [[0, 0, 0, 0, 0.5, 1, 0, 1, 0.5],
                                  [0.5, 0, 1, 0, 0, 0, 1, 0, 0.5],
                                  [0.5, 1, 0, 1, 0.5, 0, 0, 0, 0]],
                      evidence=['Guest', 'Price'], evidence_card=[3, 3])
```

```
# Associating the CPDs with the network structure.
model.add_cpds(cpd_guest, cpd_price, cpd_host)
model.check_model()
# Infering the posterior probability
from pgmpy.inference import VariableElimination

infer = VariableElimination(model)
posterior_p = infer.query(['Host'], evidence={'Guest': 2, 'Price': 2})
print(posterior_p)
nx.draw(model, with_labels=True)
plt.savefig('model.png')
plt.close()
```

Output:

```
Finding Elimination Order :      0/0 [00:57<?, ?it/s]
    0/0 [00:00<?, ?it/s]
+---------+-------------+
| Host    |   phi(Host) |
+=========+=============+
| Host(0) |      0.5000 |
+---------+-------------+
| Host(1) |      0.5000 |
+---------+-------------+
| Host(2) |      0.0000 |
+---------+-------------+
```

Result : Thus the program to implement Bayesian using python was executed and verified it approximate inferences successfully.

IMPLEMENTS DECISION PROBLEMS FOR VARIOUS REAL-WORLD APPLICATIONS

Exp.No : 3
Date :

Aim: Write a program to implement decision problems for various real-world applications.

Theory: A decision tree is a support tool with a tree-like structure that models probable outcom cost of resources, utilities, and possible consequences. Decision trees provide a way to pres algorithms with conditional control statements. They include branches that represent decisio making steps that can lead to a favorable result. Decision trees are one of the best forms of learni algorithms based on various learning methods. They boost predictive models with accuracy, ease interpretation, and stability. The tools are also effective in fitting non-linear relationships since the can solve data-fitting challenges, such as regression and classifications.

Types of Decisions

There are two main types of decision trees that are based on the target variable, i.e., categorical variable decision trees and continuous variable decision trees.

1. Categorical variable decision tree

A categorical variable decision tree includes categorical target variables that are divided into categories. For example, the categories can be yes or no. The categories mean that every stage of the decision process falls into one category, and there are no in-betweens.

2. Continuous variable decision tree

A continuous variable decision tree is a decision tree with a continuous target variable. For example, the income of an individual whose income is unknown can be predicted based on available information such as their occupation, age, and other continuous variables

Applications of Decision Trees

1. Assessing prospective growth opportunities
2. Using demographic data to find prospective clients
3. Serving as a support tool in several fields

Algorithm:

Importing the required packages
Function importing Dataset
Function to split the dataset
Function to perform training with giniIndex.
Function to make predictions
Function to calculate accuracy
Driver code

Program:

```
# Importing the required packages
import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report

# Function importing Dataset
def importdata():
        balance_data = pd.read_csv(
'https://archive.ics.uci.edu/ml/machine-learning-'+
'databases/balance-scale/balance-scale.data',
        sep= ',', header = None)

        # Printing the dataswet shape
        print ("Dataset Length: ", len(balance_data))
        print ("Dataset Shape: ", balance_data.shape)

        # Printing the dataset obseravtions
        print ("Dataset: ",balance_data.head())
        return balance_data

# Function to split the dataset
def splitdataset(balance_data):

        # Separating the target variable
        X = balance_data.values[:, 1:5]
        Y = balance_data.values[:, 0]

        # Splitting the dataset into train and test
        X_train, X_test, y_train, y_test = train_test_split(
        X, Y, test_size = 0.3, random_state = 100)

        return X, Y, X_train, X_test, y_train, y_test

# Function to perform training with giniIndex.
def train_using_gini(X_train, X_test, y_train):

        # Creating the classifier object
        clf_gini = DecisionTreeClassifier(criterion = "gini",
                        random_state = 100,max_depth=3, min_samples_leaf=5)

        # Performing training
        clf_gini.fit(X_train, y_train)
        return clf_gini

# Function to perform training with entropy.
def tarin_using_entropy(X_train, X_test, y_train):
```

```
        # Decision tree with entropy
        clf_entropy = DecisionTreeClassifier(
                        criterion = "entropy", random_state = 100,
                        max_depth = 3, min_samples_leaf = 5)

        # Performing training
        clf_entropy.fit(X_train, y_train)
        return clf_entropy


# Function to make predictions
def prediction(X_test, clf_object):

        # Predicton on test with giniIndex
        y_pred = clf_object.predict(X_test)
        print("Predicted values:")
        print(y_pred)
        return y_pred

# Function to calculate accuracy
def cal_accuracy(y_test, y_pred):

        print("Confusion Matrix: ",
                confusion_matrix(y_test, y_pred))

        print ("Accuracy : ",
        accuracy_score(y_test,y_pred)*100)

        print("Report : ",
        classification_report(y_test, y_pred))

# Driver code
def main():
        # Building Phase
        data = importdata()
        X, Y, X_train, X_test, y_train, y_test = splitdataset(data)
        clf_gini = train_using_gini(X_train, X_test, y_train)
        clf_entropy = tarin_using_entropy(X_train, X_test, y_train)

        # Operational Phase
        print("Results Using Gini Index:")

        # Prediction using gini
        y_pred_gini = prediction(X_test, clf_gini)
        cal_accuracy(y_test, y_pred_gini)

        print("Results Using Entropy:")
        # Prediction using entropy
        y_pred_entropy = prediction(X_test, clf_entropy)
```

```
    cal_accuracy(y_test, y_pred_entropy)

# Calling main function
if __name__=="__main__":
    main()
```

Output :

```
Dataset Length:  625
Dataset Shape:  (625, 5)
Dataset:      0  1  2  3  4
0  B  1  1  1  1
1  R  1  1  1  2
2  R  1  1  1  3
3  R  1  1  1  4
4  R  1  1  1  5
Results Using Gini Index:
Predicted values:
['R' 'L' 'R' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'R' 'L' 'L' 'L' 'R' 'L' 'R' 'L'
 'L' 'R' 'L' 'R' 'L' 'L' 'R' 'L' 'L' 'L' 'R' 'L' 'L' 'L' 'R' 'L' 'L' 'L'
 'L' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'R' 'L' 'R'
 'R' 'L' 'R' 'R' 'L' 'L' 'R' 'R' 'L' 'L' 'L' 'L' 'L' 'R' 'R' 'L' 'L' 'R'
 'R' 'L' 'R' 'L' 'R' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'L' 'R' 'R' 'L' 'R' 'L'
 'R' 'R' 'L' 'L' 'L' 'R' 'R' 'L' 'L' 'L' 'R' 'L' 'R' 'R' 'R' 'R' 'R' 'R'
 'R' 'L' 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'R' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'L'
 'L' 'L' 'L' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R'
 'L' 'L' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'R' 'R'
 'L' 'L' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'L' 'R' 'R'
 'L' 'R' 'R' 'L' 'L' 'R' 'R' 'R']
Confusion Matrix:  [[ 0  6  7]
 [ 0 67 18]
 [ 0 19 71]]
Accuracy :  73.40425531914893
Report :                precision    recall  f1-score   support

           B       0.00      0.00      0.00        13
           L       0.73      0.79      0.76        85
           R       0.74      0.79      0.76        90

    accuracy                           0.73       188
   macro avg       0.49      0.53      0.51       188
weighted avg       0.68      0.73      0.71       188

Results Using Entropy:
Predicted values:
['R' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'R' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'L'
 'L' 'R' 'L' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'L' 'L'
 'L' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'R' 'L' 'L' 'R' 'L' 'L' 'R' 'L' 'L'
 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L' 'R' 'L' 'L' 'R' 'L' 'L' 'L' 'R'
 'R' 'L' 'R' 'L' 'R' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'L' 'R' 'R' 'L' 'R' 'L'
 'R' 'R' 'L' 'L' 'L' 'R' 'R' 'L' 'L' 'L' 'R' 'L' 'L' 'R' 'R' 'R' 'R' 'R'
 'R' 'L' 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L'
 'L' 'L' 'L' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R'
 'L' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'R' 'R'
 'R' 'L' 'R' 'L' 'R' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'L' 'L' 'L' 'L' 'R'
 'R' 'R' 'L' 'L' 'L' 'R' 'R' 'R']
Confusion Matrix:  [[ 0  6  7]
```

```
[ 0 63 22]
[ 0 20 70]]
Accuracy :  70.74468085106383
Report :
              precision    recall  f1-score   support

           B       0.00      0.00      0.00        13
           L       0.71      0.74      0.72        85
           R       0.71      0.78      0.74        90

    accuracy                           0.71       188
   macro avg       0.47      0.51      0.49       188
weighted avg       0.66      0.71      0.68       188
```

RESULT:

Thus the program to implement decision problems for various real-world applications using python was executed and verified successfully.

LEARN VARIOUS BAYESIAN PARAMETERS

Exp.No : 4
Date :

Aim: Write a program to learn various Bayesian parameters.

Theory: In Bayesian machine learning we use the Bayes rule to infer model parameters (theta) from data (D):

$$P(\text{theta} \mid D) = P(D \mid \text{theta}) * P(\text{theta}) / P(\text{data})$$

All components of this are probability distributions.

P(data) is something we generally cannot compute, but since it's just a normalizing constant, it doesn't matter that much. When comparing models, we're mainly interested in expressions containing theta, because P(data) stays the same for each model.

P(theta) is a prior, or our belief of what the model parameters might be. Most often our opinion in this matter is rather vague and if we have enough data, we simply don't care. Inference should converge to probable theta as long as it's not zero in the prior. One specifies a prior in terms of a parametrized distribution - see Where priors come from.

P(D | theta) is called likelihood of data given model parameters. The formula for likelihood is model-specific. People often use likelihood for evaluation of models: a model that gives higher likelihood to real data is better.

Finally, P(theta | D), a posterior, is what we're after. It's a probability distribution over model parameters obtained from prior beliefs and data. When one uses likelihood to get point estimates of model parameters, it's called maximum-likelihood estimation, or MLE. If one also takes the prior into account, then it's maximum a posteriori estimation (MAP). MLE and MAP are the same if the prior is uniform.

Algorithm:

Objective function
Grid-based sample of the domain [0,1]
Sample the domain without noise
Sample the domain with noise
Find best result
Plot the points with noise
Plot the points without noise

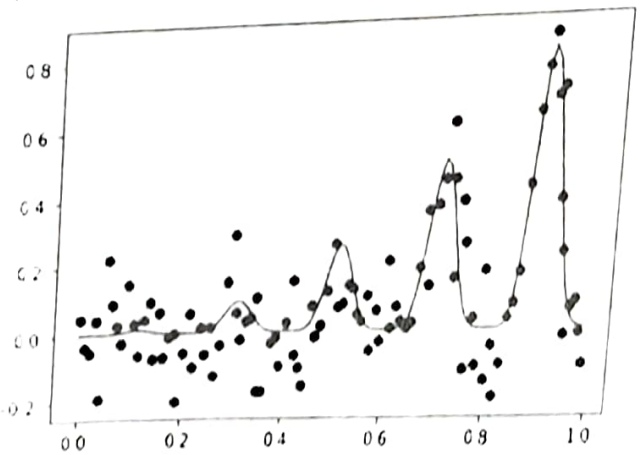
Program :

```
# example of the test problem
from math import sin
from math import pi
from numpy import arange
```

```
from numpy import argmax
from numpy.random import normal
from matplotlib import pyplot

# objective function
def objective(x, noise=0.1):
noise = normal(loc=0, scale=noise)
return (x**2 * sin(5 * pi * x)**6.0) + noise

# grid-based sample of the domain [0,1]
X = arange(0, 1, 0.01)
# sample the domain without noise
y = [objective(x, 0) for x in X]
# sample the domain with noise
ynoise = [objective(x) for x in X]
# find best result
ix = argmax(y)
print('Optima: x=%.3f, y=%.3f' % (X[ix], y[ix]))
# plot the points with noise
pyplot.scatter(X, ynoise)
# plot the points without noise
pyplot.plot(X, y)
# show the plot
pyplot.show()
```

OUTPUT :

```
Optima: x=0.900, y=0.810
```



RESULT:

Thus the program to learn various Bayesian parameters using python was executed and verified successfully.

IMPLEMENTATION OF HIDDEN MARKOV MODELS

Exp.No : 5
Date :

Aim: Write a program to implementation of hidden markov models.

Theory: In a regular Markov Chain we are able to see the states and their associated transition probabilities. However, in a Hidden Markov Model (HMM), the Markov Chain is hidden but we can infer its properties through its given observed states.

- If the weather is Sunny, I have a 90% chance of being happy and 10% chance of being sad.
- If the weather is Rainy, I have a 30% chance of being happy and 70% chance of being sad.

These associated probabilities of the observed states (Happy, Sad) are known as the emission probabilities.

Program :

```
def viterbi(obs, states, start_p, trans_p, emit_p):
  V = [{}]

  for st in states:
    V[0][st] = {"prob": start_p[st] * emit_p[st][obs[0]], "prev": None}

  # Run Viterbi when t > 0

  for t in range(1, len(obs)):
    V.append({})

    for st in states:
      max_tr_prob = max(V[t-1][prev_st]["prob"]*trans_p[prev_st][st] for prev_st in states)

      for prev_st in states:
        if V[t-1][prev_st]["prob"] * trans_p[prev_st][st] == max_tr_prob:
          max_prob = max_tr_prob * emit_p[st][obs[t]]
          V[t][st] = {"prob": max_prob, "prev": prev_st}
          break

for line in dptable(V):
```

```
        print(line)

    opt = []

    # The highest probability
    max_prob = max(value["prob"] for value in V[-1].values())
    previous = None

    # Get most probable state and its backtrack
    for st, data in V[-1].items():
        if data["prob"] == max_prob:
            opt.append(st)
            previous = st
            break

    # Follow the backtrack till the first observation
    for t in range(len(V) - 2, -1, -1):
        opt.insert(0, V[t + 1][previous]["prev"])
        previous = V[t + 1][previous]["prev"]

    print('The steps of states are ' + ' '.join(opt) + ' with highest probability of %s' % max_prob)


def dptable(V):
    # Print a table of steps from dictionary
    yield " ".join(("%12d" % i) for i in range(len(V)))

    for state in V[0]:
        yield "%.7s: " % state + " ".join("%.7s" % ("%f" % v[state]["prob"]) for v in V)

#The function viterbi takes the following arguments: obs is the sequence of observations, e.g.
['normal', 'cold', 'dizzy']; states is the set of hidden states; start_p is the start probability; trans_p are
the transition probabilities; and emit_p are the emission probabilities. For simplicity of code, we
assume that the observation sequence obs is non-empty and that trans_p[i][j] and emit_p[i][j] is
defined for all states i,j.

#In the running example, the forward/Viterbi algorithm is used as follows:


obs = ('normal', 'cold', 'dizzy')
states = ('Healthy', 'Fever')
start_p = {'Healthy': 0.6, 'Fever': 0.4}
trans_p = {
  'Healthy' : {'Healthy': 0.7, 'Fever': 0.3},
  'Fever' : {'Healthy': 0.4, 'Fever': 0.6}
  }
emit_p = {
  'Healthy' : {'normal': 0.5, 'cold': 0.4, 'dizzy': 0.1},
  'Fever' : {'normal': 0.1, 'cold': 0.3, 'dizzy': 0.6}
  }
```

viterbi(obs,states,start_p,trans_p,emit_p)

OUTPUT

```
                0             1             2
Healthy:  0.30000       0.08400       0.00588
Fever:    0.04000       0.02700       0.01512
The steps of states are Healthy Healthy Fever with highest probability of 0.0
1512
```

Result:

Thus the program to implementation of hidden markov models using python was executed and verified successfully.

IMPLEMENTATION OF EM ALGORITHM FOR HMM

Exp.No : 6
Date :

Aim: Write a program to implement EM algorithm for HMM

Theory:

GMMs are probabilistic models that assume all the data points are generated from a mixture of several Gaussian distributions with unknown parameters. They differ from k-means clustering in that GMMs incorporate information about the center(mean) and variability(variance) of each clusters and provide posterior probabilities.

In the example mentioned earlier, we have 2 clusters: people who like the product and people who don't. If we know which cluster each customer belongs to (the labels), we can easily estimate the parameters(mean and variance) of the clusters, or if we know the parameters for both clusters, we can predict the labels. Unfortunately, we don't know either one. To solve this chicken and egg problem, the Expectation-Maximization Algorithm (EM) comes in handy.

EM is an iterative algorithm to find the maximum likelihood when there are latent variables. The algorithm iterates between performing an expectation (E) step, which creates a heuristic of the posterior distribution and the log-likelihood using the current estimate for the parameters, and a maximization (M) step, which computes parameters by maximizing the expected log-likelihood from the E step. The parameter-estimates from M step are then used in the next E step. In the following sections, we will delve into the math behind EM, and implement it in Python from scratch.

Program:

```
import numpy as np
import pandas as pd
from scipy import stats
from scipy.special import logsumexp
from sklearn.mixture import GaussianMixture
from matplotlib import pyplot as plt


def GMM_sklearn(x, weights=None, means=None, covariances=None):
  model = GaussianMixture(n_components=2,
                covariance_type='full',
                tol=0.01,
                max_iter=1000,
                weights_init=weights,
                means_init=means,
                precisions_init=covariances)
  model.fit(x)
  print("\nscikit learn:\n\tphi: %s\n\tmu_0: %s\n\tmu_1: %s\n\tsigma_0: %s\n\tsigma_1: %s"
        % (model.weights_[1], model.means_[0, :], model.means_[1, :], model.covariances_[0, :],
model.covariances_[1, :]))
  return model.predict(x), model.predict_proba(x)[:,1]
```

```
def get_random_psd(n):
    x = np.random.normal(0, 1, size=(n, n))
    return np.dot(x, x.transpose())


def initialize_random_params():
    params = {'phi': np.random.uniform(0, 1),
              'mu0': np.random.normal(0, 1, size=(2,)),
              'mu1': np.random.normal(0, 1, size=(2,)),
              'sigma0': get_random_psd(2),
              'sigma1': get_random_psd(2)}
    return params


def learn_params(x_labeled, y_labeled):
    n = x_labeled.shape[0]
    phi = x_labeled[y_labeled == 1].shape[0] / n
    mu0 = np.sum(x_labeled[y_labeled == 0], axis=0) / x_labeled[y_labeled == 0].shape[0]
    mu1 = np.sum(x_labeled[y_labeled == 1], axis=0) / x_labeled[y_labeled == 1].shape[0]
    sigma0 = np.cov(x_labeled[y_labeled == 0].T, bias= True)
    sigma1 = np.cov(x_labeled[y_labeled == 1].T, bias=True)
    return {'phi': phi, 'mu0': mu0, 'mu1': mu1, 'sigma0': sigma0, 'sigma1': sigma1}


def e_step(x, params):
    np.log([stats.multivariate_normal(params["mu0"], params["sigma0"]).pdf(x),
            stats.multivariate_normal(params["mu1"], params["sigma1"]).pdf(x)])
    log_p_y_x = np.log([1-params["phi"], params["phi"]])[np.newaxis, ...] + \
                np.log([stats.multivariate_normal(params["mu0"], params["sigma0"]).pdf(x),
            stats.multivariate_normal(params["mu1"], params["sigma1"]).pdf(x)]).T
    log_p_y_x_norm = logsumexp(log_p_y_x, axis=1)
    return log_p_y_x_norm, np.exp(log_p_y_x - log_p_y_x_norm[..., np.newaxis])


def m_step(x, params):
    total_count = x.shape[0]
    _, heuristics = e_step(x, params)
    heuristic0 = heuristics[:, 0]
    heuristic1 = heuristics[:, 1]
    sum_heuristic1 = np.sum(heuristic1)
    sum_heuristic0 = np.sum(heuristic0)
    phi = (sum_heuristic1/total_count)
    mu0 = (heuristic0[..., np.newaxis].T.dot(x)/sum_heuristic0).flatten()
    mu1 = (heuristic1[..., np.newaxis].T.dot(x)/sum_heuristic1).flatten()
    diff0 = x - mu0
    sigma0 = diff0.T.dot(diff0 * heuristic0[..., np.newaxis]) / sum_heuristic0
    diff1 = x - mu1
    sigma1 = diff1.T.dot(diff1 * heuristic1[..., np.newaxis]) / sum_heuristic1
```

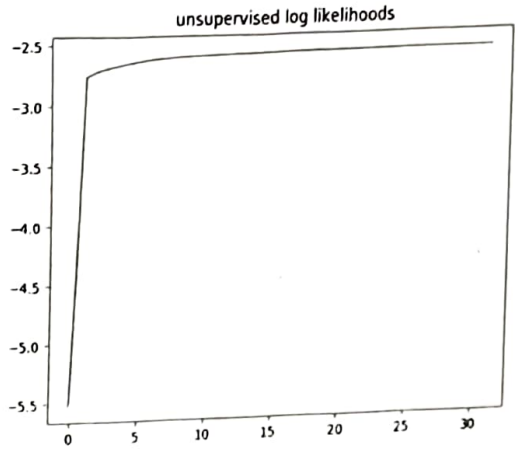
```
    params = {'phi': phi, 'mu0': mu0, 'mu1': mu1, 'sigma0': sigma0, 'sigma1': sigma1}
    return params


def get_avg_log_likelihood(x, params):
    loglikelihood, _ = e_step(x, params)
    return np.mean(loglikelihood)


def run_em(x, params):
    avg_loglikelihoods = []
    while True:
        avg_loglikelihood = get_avg_log_likelihood(x, params)
        avg_loglikelihoods.append(avg_loglikelihood)
        if len(avg_loglikelihoods) > 2 and abs(avg_loglikelihoods[-1] - avg_loglikelihoods[-2]) <
0.0001:
            break
        params = m_step(x_unlabeled, params)
    print("\tphi: %s\n\tmu_0: %s\n\tmu_1: %s\n\tsigma_0: %s\n\tsigma_1: %s"
          % (params['phi'], params['mu0'], params['mu1'], params['sigma0'], params['sigma1']))
    _, posterior = e_step(x_unlabeled, params)
    forecasts = np.argmax(posterior, axis=1)
    return forecasts, posterior, avg_loglikelihoods


def unsupervised_gmm(x_unlabeled):
    params = initialize_random_params()
    weights = [1 - params["phi"], params["phi"]]
    means = [params["mu0"], params["mu1"]]
    covariances = [params["sigma0"], params["sigma1"]]
    sklearn_forecasts, posterior_sklearn = GMM_sklearn(x_unlabeled, weights, means, covariances)
    forecasts, posterior, loglikelihoods = run_em(x_unlabeled, params)
    print("total steps: ", len(loglikelihoods))
    plt.plot(loglikelihoods)
    plt.title("unsupervised log likelihoods")
    plt.savefig("unsupervised.png")
    plt.close()
    return pd.DataFrame({'forecasts': forecasts, 'posterior': posterior[:,1],
                         'sklearn_forecasts': sklearn_forecasts,
                         'posterior_sklearn': posterior_sklearn})


# def semi_supervised_gmm(x_unlabeled):
#     data_labeled = pd.read_csv("data/labeled.csv")
#     x_labeled = data_labeled[["x1", "x2"]].values
#     y_labeled = data_labeled["y"].values
#     params = learn_params(x_labeled, y_labeled)
#     weights = [1 - params["phi"], params["phi"]]
#     means = [params["mu0"], params["mu1"]]
#     covariances = [params["sigma0"], params["sigma1"]]
```

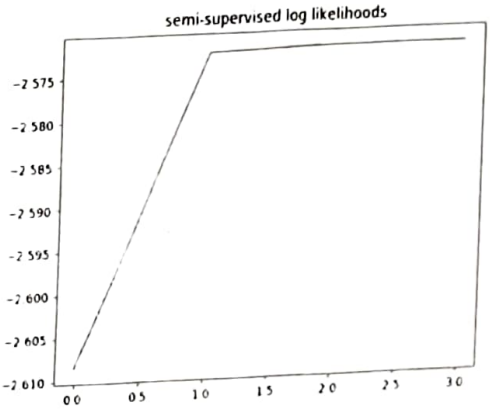
```
#     sklearn_forecasts, posterior_sklearn = GMM_sklearn(x_unlabeled, weights, means, covariances)
#     forecasts, posterior, loglikelihoods = run_em(x_unlabeled, params)
#     print("total steps: ", len(loglikelihoods))
#     plt.plot(loglikelihoods)
#     plt.title("semi-supervised log likelihoods")
#     plt.savefig("semi-supervised.png")
#     return pd.DataFrame({'forecasts': forecasts, 'posterior': posterior[:, 1],
#                          'sklearn_forecasts': sklearn_forecasts,
#                          'posterior_sklearn': posterior_sklearn})


if __name__ == '__main__':
    data_unlabeled = pd.read_csv("data/unlabeled.csv")
    x_unlabeled = data_unlabeled[["x1", "x2"]].values

    # Unsupervised learning
    print("unsupervised: ")
    random_params = initialize_random_params()
    unsupervised_forecastsforecasts, unsupervised_posterior, unsupervised_loglikelihoods =
run_em(x_unlabeled, random_params)
    print("total steps: ", len(unsupervised_loglikelihoods))
    plt.plot(unsupervised_loglikelihoods)
    plt.title("unsupervised log likelihoods")
    plt.savefig("unsupervised.png")
    plt.close()

    # Semi-supervised learning
    print("\nsemi-supervised: ")
    data_labeled = pd.read_csv("data/labeled.csv")
    x_labeled = data_labeled[["x1", "x2"]].values
    y_labeled = data_labeled["y"].values
    learned_params = learn_params(x_labeled, y_labeled)
    semisupervised_forecasts, semisupervised_posterior, semisupervised_loglikelihoods =
run_em(x_unlabeled, learned_params)
    print("total steps: ", len(semisupervised_loglikelihoods))
    plt.plot(semisupervised_loglikelihoods)
    plt.title("semi-supervised log likelihoods")
    plt.savefig("semi-supervised.png")

    # Compare the forecats with Scikit-learn API
    learned_params = learn_params(x_labeled, y_labeled)
    weights = [1 - learned_params["phi"], learned_params["phi"]]
    means = [learned_params["mu0"], learned_params["mu1"]]
    covariances = [learned_params["sigma0"], learned_params["sigma1"]]
    sklearn_forecasts, posterior_sklearn = GMM_sklearn(x_unlabeled, weights, means, covariances)

    output_df = pd.DataFrame({'semisupervised_forecasts': semisupervised_forecasts,
                              'semisupervised_posterior': semisupervised_posterior[:, 1],
                              'sklearn_forecasts': sklearn_forecasts,
```

'posterior_sklearn': posterior_sklearn})

print("\n%s%% of forecasts matched." % (output_df[output_df["semisupervised_forecasts"] output_df["sklearn_forecasts"]].shape[0] /output_df.shape[0] * 100))

OUTPUT:






```
semi-supervised:
    phi: 0.586349881794546
    mu_0: [-1.04546727 -1.02704636]
    mu_1: [0.98763329 0.99661118]
    sigma_0: [[0.36018609 0.30853357]
 [0.30853357 0.75384027]]
    sigma_1: [[0.7196797  0.1437903 ]
 [0.1437903  0.30853791]]
total steps:  4

scikit learn:
    phi: 0.59647894226803
    mu_0: [-1.06169376 -1.0563389 ]
    mu_1: [0.96408565 0.98206315]
    sigma_0: [[0.35027155 0.29629092]
 [0.29629092 0.73083581]]
    sigma_1: [[0.74510804 0.16156928]
 [0.16156928 0.32021029]]

99.4% of forecasts matched.
```

Result:

Thus the program to implement EM algorithm for HMM using python was executed and verified successfully.

IMPLEMENTATION OF THE REINFORCEMENT LEARNING FOR VARIOUS REWARD BASED APPLICATIONS

Exp.No : 7
Date :

Aim: Write a program to implement the Reinforcement learning for various reward based applications.

Theory :

Reinforcement Learning is a type of Machine Learning paradigms in which a learning algorithm is trained not on preset data but rather based on a feedback system. These algorithms a touted as the future of Machine Learning as these eliminate the cost of collecting and cleaning the data. we are going to demonstrate how to implement a basic Reinforcement Learning algorithm which is called the Q-Learning technique. In this demonstration, we attempt to teach a bot to reach its destination using the Q-Learning technique.

Algorithm:

Step 1: Importing the required libraries
Step 2: Defining and visualising the graph
Step 3: Defining the reward the system for the bot
Step 4: Defining some utility functions to be used in the training
Step 5: Training and evaluating the bot using the Q-Matrix
Step 6: Defining and visualizing the new graph with the environmental clues
Step 7: Defining some utility functions for the training process
Step 8: Visualising the Environmental matrices
Step 9: Training and evaluating the model

Program:

```
import numpy as np
import pylab as pl
import networkx as nx
edges = [(0, 1), (1, 5), (5, 6), (5, 4), (1, 2),
                 (1, 3), (9, 10), (2, 4), (0, 6), (6, 7),
                 (8, 9), (7, 8), (1, 7), (3, 9)]
goal = 10
G = nx.Graph()
G.add_edges_from(edges)
pos = nx.spring_layout(G)
nx.draw_networkx_nodes(G, pos)
nx.draw_networkx_edges(G, pos)
nx.draw_networkx_labels(G, pos)
pl.show()
```

```
MATRIX_SIZE = 11
M = np.matrix(np.ones(shape =(MATRIX_SIZE, MATRIX_SIZE)))
M *= -1
for point in edges:
        print(point)
        if point[1] == goal:
                M[point] = 100

        else:
                M[point] = 0

        if point[0] == goal:
                M[point[::-1]] = 100

        else:
                M[point[::-1]]= 0
                # reverse of point


M[goal, goal]= 100
print(M)
# add goal point round trip
Q = np.matrix(np.zeros([MATRIX_SIZE, MATRIX_SIZE]))
gamma = 0.75
# learning parameter
initial_state = 1

# Determines the available actions for a given state
def available_actions(state):
  current_state_row = M[state, ]
  available_action = np.where(current_state_row >= 0)[1]
  return available_action

available_action = available_actions(initial_state)

# Chooses one of the available actions at random
def sample_next_action(available_actions_range):
  next_action = int(np.random.choice(available_action, 1))
  return next_action


action = sample_next_action(available_action)

def update(current_state, action, gamma):

max_index = np.where(Q[action, ] == np.max(Q[action, ]))[1]
if max_index.shape[0] > 1:
  max_index = int(np.random.choice(max_index, size = 1))
else:
```

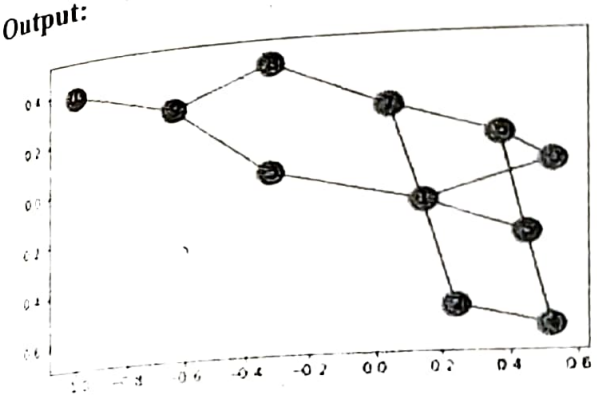
```
    max_index = int(max_index)
  max_value = Q[action, max_index]
  Q[current_state, action] = M[current_state, action] + gamma * max_value
  if (np.max(Q) > 0):
   return(np.sum(Q / np.max(Q)*100))
  else:
   return (0)
# Updates the Q-Matrix according to the path chosen

update(initial_state, action, gamma)

scores = []
for i in range(1000):
    current_state = np.random.randint(0, int(Q.shape[0]))
    available_action = available_actions(current_state)
    action = sample_next_action(available_action)
    score = update(current_state, action, gamma)
    scores.append(score)

 # print("Trained Q matrix:")
 # print(Q / np.max(Q)*100)
 # You can uncomment the above two lines to view the trained Q matrix

 # Testing
 current_state = 0
 steps = [current_state]

 while current_state != 10:

    next_step_index = np.where(Q[current_state, ] == np.max(Q[current_state, ]))[1]
    if next_step_index.shape[0] > 1:
       next_step_index = int(np.random.choice(next_step_index, size = 1))
    else:
       next_step_index = int(next_step_index)
   steps.append(next_step_index)
   current_state = next_step_index

print("Most efficient path:")
print(steps)

pl.plot(scores)
pl.xlabel('No of iterations')
pl.ylabel('Reward gained')
pl.show()

# Defining the locations of the police and the drug traces
```

```
police = [2, 4, 5]
drug_traces = [3, 8, 9]

G = nx.Graph()
G.add_edges_from(edges)
mapping = {0:'0 - Detective', 1:'1', 2:'2 - Police', 3:'3 - Drug traces',
       4:'4 - Police', 5:'5 - Police', 6:'6', 7:'7', 8:'Drug traces',
       9:'9 - Drug traces', 10:'10 - Drug racket location'}

H = nx.relabel_nodes(G, mapping)
pos = nx.spring_layout(H)
nx.draw_networkx_nodes(H, pos, node_size =[200, 200, 200, 200, 200, 200, 200, 200])
nx.draw_networkx_edges(H, pos)
nx.draw_networkx_labels(H, pos)
pl.show()

Q = np.matrix(np.zeros([MATRIX_SIZE, MATRIX_SIZE]))
env_police = np.matrix(np.zeros([MATRIX_SIZE, MATRIX_SIZE]))
env_drugs = np.matrix(np.zeros([MATRIX_SIZE, MATRIX_SIZE]))
initial_state = 1

# Same as above
def available_actions(state):
    current_state_row = M[state, ]
    av_action = np.where(current_state_row >= 0)[1]
    return av_action

# Same as above
def sample_next_action(available_actions_range):
    next_action = int(np.random.choice(available_action, 1))
    return next_action

# Exploring the environment
def collect_environmental_data(action):
    found = []
    if action in police:
        found.append('p')
    if action in drug_traces:
        found.append('d')
    return (found)


available_action = available_actions(initial_state)
action = sample_next_action(available_action)

def update(current_state, action, gamma):
```

```
    max_index = np.where(Q[action, ] == np.max(Q[action, ]))[1]
    if max_index.shape[0] > 1:
        max_index = int(np.random.choice(max_index, size = 1))
    else:
        max_index = int(max_index)
    max_value = Q[action, max_index]
    Q[current_state, action] = M[current_state, action] + gamma * max_value
    environment = collect_environmental_data(action)
    if 'p' in environment:
      env_police[current_state, action] += 1
    if 'd' in environment:
      env_drugs[current_state, action] += 1
    if (np.max(Q) > 0):
      return(np.sum(Q / np.max(Q)*100))
    else:
      return (0)
# Same as above
update(initial_state, action, gamma)

def available_actions_with_env_help(state):
    current_state_row = M[state, ]
    av_action = np.where(current_state_row >= 0)[1]

    # if there are multiple routes, dis-favor anything negative
    env_pos_row = env_matrix_snap[state, av_action]

    if (np.sum(env_pos_row < 0)):
        # can we remove the negative directions from av_act?
        temp_av_action = av_action[np.array(env_pos_row)[0]>= 0]
        if len(temp_av_action) > 0:
            av_action = temp_av_action
    return av_action
# Determines the available actions according to the environment
scores = []
for i in range(1000):
    current_state = np.random.randint(0, int(Q.shape[0]))
    available_action = available_actions(current_state)
    action = sample_next_action(available_action)
    score = update(current_state, action, gamma)

# print environmental matrices
print('Police Found')
print(env_police)
print('')
print('Drug traces Found')
print(env_drugs)
```

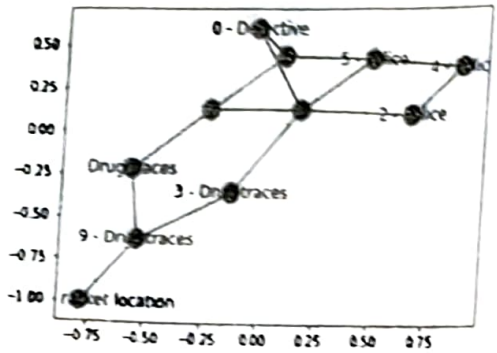
```
scores = []
for i in range(1000):
    current_state = np.random.randint(0, int(Q.shape[0]))
    available_action = available_actions_with_env_help(current_state)
    action = sample_next_action(available_action)
    score = update(current_state, action, gamma)
    scores.append(score)

pl.plot(scores)
pl.xlabel('Number of iterations')
pl.ylabel('Reward gained')
pl.show()
```

Output:

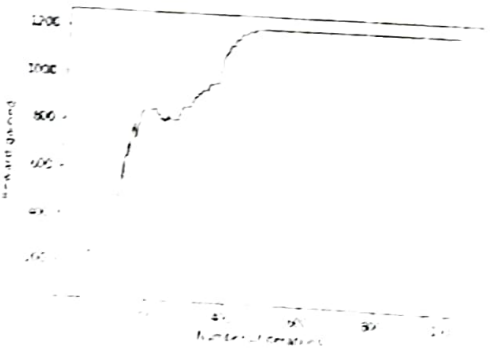


```
[[  0.   0.   1.  -1.   1.  -1.   0.  -1.  -1.  -1.  -1.]
 [  0.   0.   0.   1.   0.   1.   0.   1.  -1.  -1.]
 [  0.   0.   1.   0.   1.   1.   1.   1.   1.  -1.]
 [  1.   0.   1.   1.   1.   1.  -1.  -1.   0.  -1.]
 [  0.   0.   1.   1.   1.   0.   1.   1.   1.   1.]
 [  1.   0.   1.   1.   0.   1.   0.  -1.   1.  -1.]
 [  0.   1.   1.   1.   1.   0.   1.   0.  -1.  -1.]
 [  1.   0.   1.   1.   1.   1.   0.  -1.   0.  -1.]
 [  1.   1.   1.   0.   1.  -1.   1.  -1.   0.  -1.]
 [  1.   1.   1.   1.   1.  -1.   1.   1.  -1.   0.]]
```

```
Most efficient path:
[0, 1, 3, 9, 10]
```




```
Police Found
[[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. 23.  0.  0. 14.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. 51.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. 51.  0.  0. 37.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. 29.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0. 32.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]

Drug traces Found
[[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0. 12.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. 40.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. 29.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0. 52.  0.]
 [ 0.  0.  0. 36.  0.  0.  0.  0. 37.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0. 47.  0.]]
```



Result:

Thus the program to implement the Reinforcement learning for various reward based applications using python was executed and verified successfully.

MINI-PROJECT – IMPLEMENTATION OF THE APPROXIMATE INFERENCES IN BAYESIAN NETWORK.

Exp.No : 8
Date :

Aim: Create a project and to implement the Approximate inferences in Bayesian network

Algorithm:

Step 1: Importing the required libraries
Step 2: Defining and visualizing the graph
Step 3: Defining the reward the system for the bot
Step 4: Defining some utility functions to be used in the training
Step 5: Training and evaluating the bot
Step 6: Defining and visualizing the new graph with the environmental clues
Step 7: Defining some utility functions for the training process
Step 8: Visualising the Environmental matrices
Step 9: Training and evaluating the model

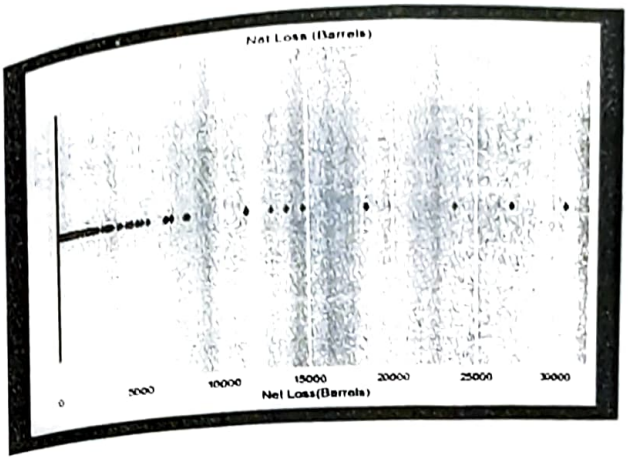
Program:

```
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
data=pd.read_csv('E:\BALA\AI\Lab programs\pgms\database.csv')
cols_of_interest=['Accident Date/Time','Accident State','Pipeline Location','Liquid Type','Net
Loss(Barrels)','All Costs']
data=data[cols_of_interest]
data_summary=print(data[['All Costs','Net Loss(Barrels)']])
data['All Costs']=data['All Costs']/1000000
plt.style.use('seaborn')
sns.boxplot(data['All Costs'],data = data)
plt.title('Costs of Accident')
plt.show()
plt.close()
sns.boxplot(data['Net Loss(Barrels)'],data=data)
plt.title('Net Loss (Barrels)')
plt.show()
data['Accident Date/Time']=pd.to_datetime(data['Accident Date/Time'])
totaltimespan=np.max(data['Accident Date/Time'])-np.min(data['Accident Date/Time'])
totaltime_hour=(totaltimespan.days*24+totaltimespan.seconds/(3600))
totaltime_month=(totaltimespan.days+totaltimespan.seconds/(3600*24))*12/365
lmda_h=len(data)/totaltime_hour
lmda_m=len(data)/totaltime_month
print('Estimated no.of Accident per hour:{}'.format(lmda_h))
print('Estimated no.of Accident per month:{}'.format(lmda_m))
import math
```

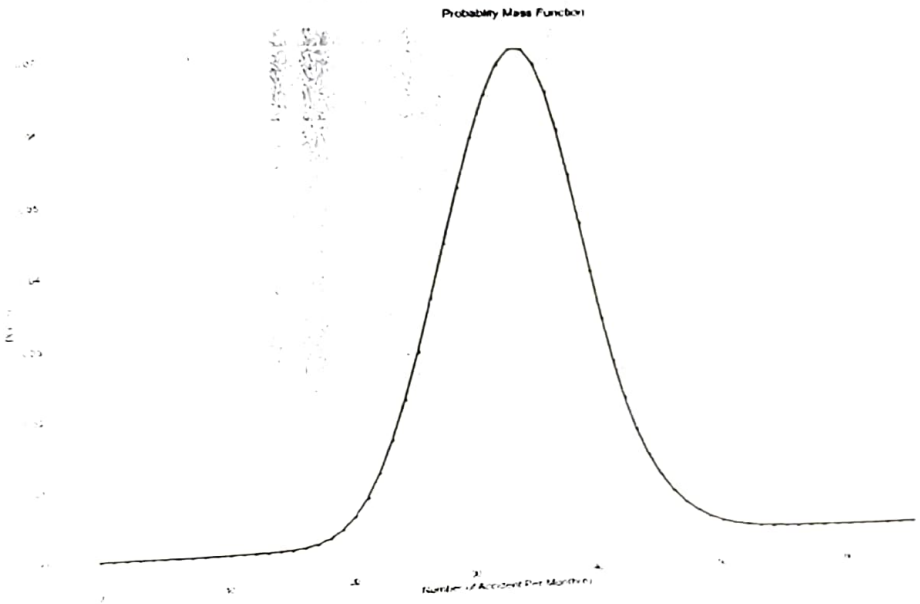
```
import matplotlib.pyplot as plt
X={}
for i in range(66):
  X[i]=math.pow(2.71828,-1*33)*math.pow(33,i)/math.factorial(i)
p_Poission=pd.DataFrame(X.items(),columns=['X','PX'])
plt.style.use('seaborn')
fig=plt.subplots(figsize=(15,10))
plt.plot(p_Poission['X'],p_Poission['PX'],marker='.',color = 'purple',linestyle='solid')
plt.xlabel('Number of Accident Per Month(n)')
plt.ylabel('P(X<=n)')
plt.title('Probablity Mass Function')
plt.show()
plt.close()
def cdf(data):
    n=len(data)
    x=np.sort(data)
    y=np.arange(1,n+1)/n
    return x,y
np.random.seed(42)
sample_Poission=np.random.poisson(33,10000)
x,y=cdf(sample_Poission)
fig=plt.subplots(figsize=(15,10))
plt.plot(x,y,marker='',alpha=0.5,color ='purple',linestyle='solid')
plt.xlabel('Number of Accident Per Month(n)')
plt.ylabel('P(X<=n)')
plt.title('Cummulative Distribution Function')
plt.show()
```

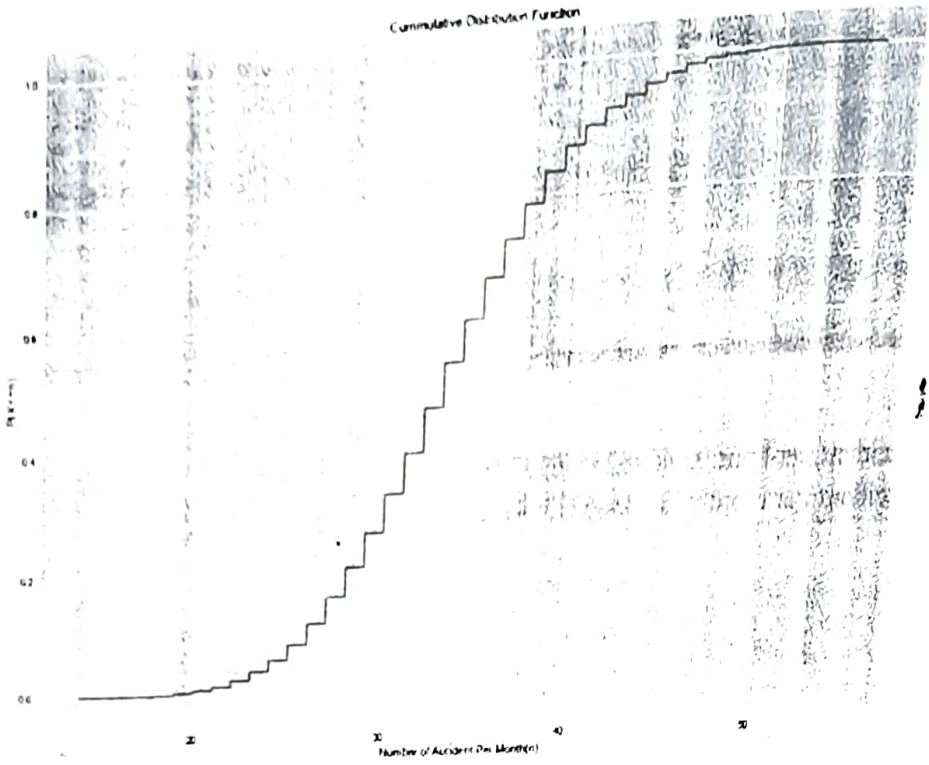
OUTPUT:

```
      All Costs  Net Loss(8arrels)
0          1627               21.0
1          4008                0.0
2           200                2.0
3         11540                0.0
4         29650                2.0
...         ...                ...
2790      61015                0.0
2791     105400              580.0
2792      15050                0.0
2793      41428                0.0
2794      45800                0.0

[2795 rows x 2 columns]
```



Estimated no.of Accident per hour:0.04540255169379675
Estimated no.of Accident per month:33.14386273647162





Result:

Thus the created the project and implemented the approximate inferences in Bayesian network usin
python was executed and verified successfully.