National University of Computer & Emerging Sciences, Karachi

Computer Science Department
Spring 2023, Lab Manual – 09

Course Code: AI-2002	Course: Artificial Intelligence Lab
Instructor(s):	**Kariz Kamal, Saeeda Kanwal, Sania Urooj, Shakir Hussain, Omer Qureshi, Muhammad Ali Fatmi.**

Contents:

Objective

1. Introduction to BN, DBN & HMM
2. Explore how to use python to solve BN, DBN & HMM problems and then apply these skills to a couple of challenge problems.
3. Introduction to different Libraries of python.

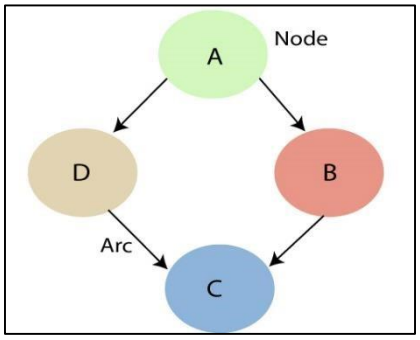
Bayesian Network

A Bayesian network is a probabilistic graphical model which represents a set of variables and their conditional dependencies using a directed acyclic graph. It is also called a Bayes network, belief network, decision network, or Bayesian model. Bayesian networks are probabilistic, because these networks are built from a probability distribution, and also use probability theory for prediction and anomaly detection. Real world applications are probabilistic in nature, and to represent the relationship between multiple events, we need a Bayesian network. It can also be used in various tasks including prediction, anomaly detection, diagnostics, automated insight, reasoning, time series prediction, and decision making under uncertainty.

Bayesian Network can be used for building models from data and expert opinions, and it consists of two parts:

- Directed Acyclic Graph
- Table of conditional probabilities.

The generalized form of Bayesian network that represents and solve decision problems under uncertain knowledge is known as an Influence diagram. A Bayesian network graph is made up of nodes and Arcs (directed links), where:



- Each node corresponds to the random variables, and a variable can be continuous or discrete.
- Arc or directed arrows represent the causal relationship or conditional probabilities between random variables. These directed links or arrows connect the pair of nodes in the graph.
  These links represent that one node directly influence the other node, and if there is no directed link that means that nodes are independent with each other
- In the above diagram, A, B, C, and D are random variables represented by the nodes of the network graph.
- If we are considering node B, which is connected with node A by a directed arrow, then node A is called the parent of Node B.
- Node C is independent of node A.

The Bayesian network has mainly two components:

- **Causal Component**
- **Actual numbers**

Each node in the Bayesian network has condition probability distribution **$P(X_i \mid Parent(X_i))$**, which determines the effect of the parent on that node.
Bayesian network is based on Joint probability distribution and conditional probability. So let's first understand the joint probability distribution:

Bayesian Networks Python

In this demo, we'll be using Bayesian Networks to solve the famous Monty Hall Problem. For those of you who don't know what the Monty Hall problem is, let me explain:
The Monty Hall problem named after the host of the TV series, 'Let's Make A Deal', is a paradoxical probability puzzle that has been confusing people for over a decade.
So this is how it works. The game involves three doors, given that behind one of these doors is a car and the remaining two have goats behind them. So you start by picking a random door, say #2. On the other hand, the host knows where the car is hidden and he opens another door, say #1 (behind which there is a goat). Here's the catch, you're now given a choice, the host will ask you if you want to pick door #3 instead of your first choice i.e. #2.

Is it better if you switch your choice or should you stick to your first choice?

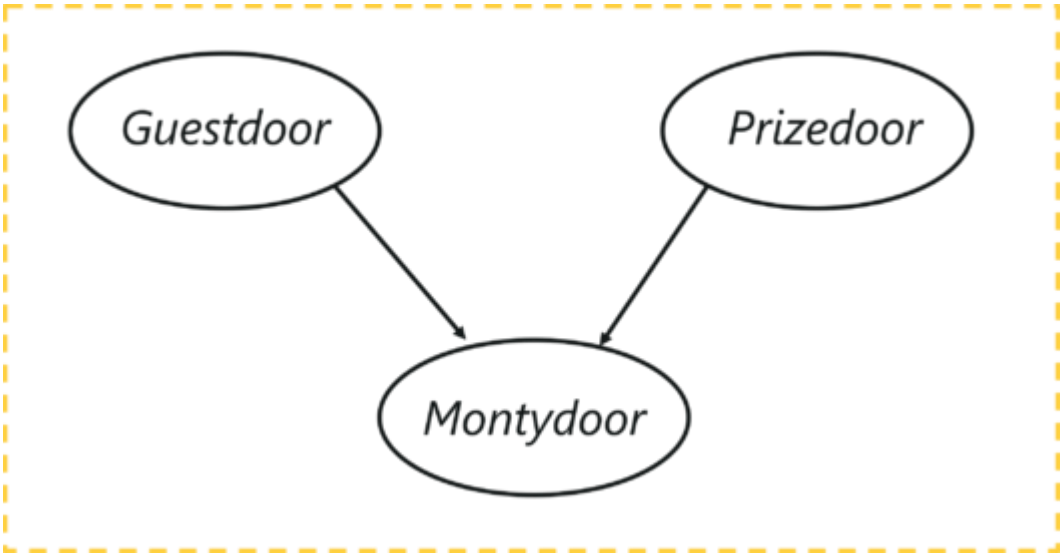
This is exactly what we're going to model. We'll be creating a Bayesian Network to understand the probability of winning if the participant decides to switch his choice.

Now let's get started.

The first step is to build a Directed Acyclic Graph.

The graph has three nodes, each representing the door chosen by:

1. The door selected by the Guest
2. The door containing the prize (car)
3. The door Monty chooses to open



Let's understand the dependencies here, the door selected by the guest and the door containing the car are completely random processes. However, the door Monty chooses to open is dependent on both the doors; the door selected by the guest, and the door the prize is behind. Monty has to choose in such a way that the door does not contain the prize and it cannot be the one chosen by the guest.

```
#Import required packages
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 d
oor
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 Ne
tworks" )
network.add_states(d1, d2, d3)
network.add_edge(d1, d3)
network.add_edge(d2, d3)
network.bake()
```

```
beliefs = network.predict_proba({ 'guest' : 'A' })
beliefs = map(str, beliefs)
print("n".join( "{}t{}".format( state.name, belief ) for state, belief in z
ip( 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 )))
```

Dynamic Bayesian Networks

A Bayesian network is a snapshot of the system at a given time and is used to model systems that are in some kind of equilibrium state. Unfortunately, most systems in the world change over time and sometimes we are interested in how these systems evolve over time more than we are interested in their equilibrium states. Whenever the focus of our reasoning is change of a system over time, we need a tool that is capable of modeling dynamic systems.

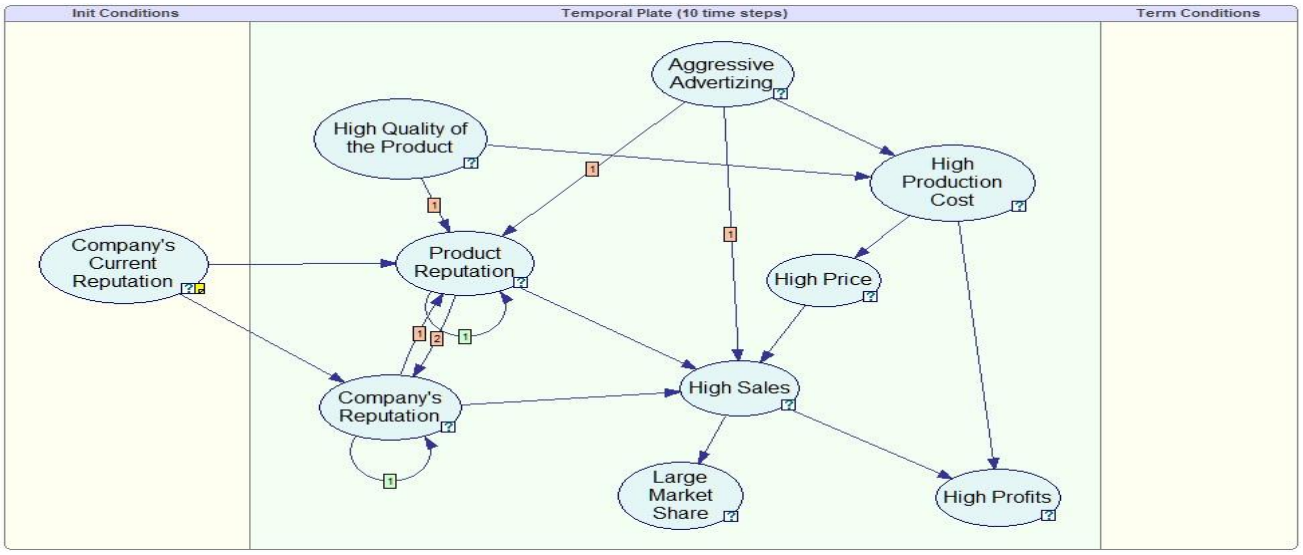
A dynamic Bayesian network (DBN) is a Bayesian network extended with additional mechanisms that are capable of modeling influences over time. The temporal extension of Bayesian networks does not mean that the network structure or parameters changes dynamically, but that a dynamic system is modeled. In other words, the underlying process, modeled by a DBN, is stationary. A DBN is a model of a stochastic process.

The structure of a dynamic Bayesian network and its interpretation

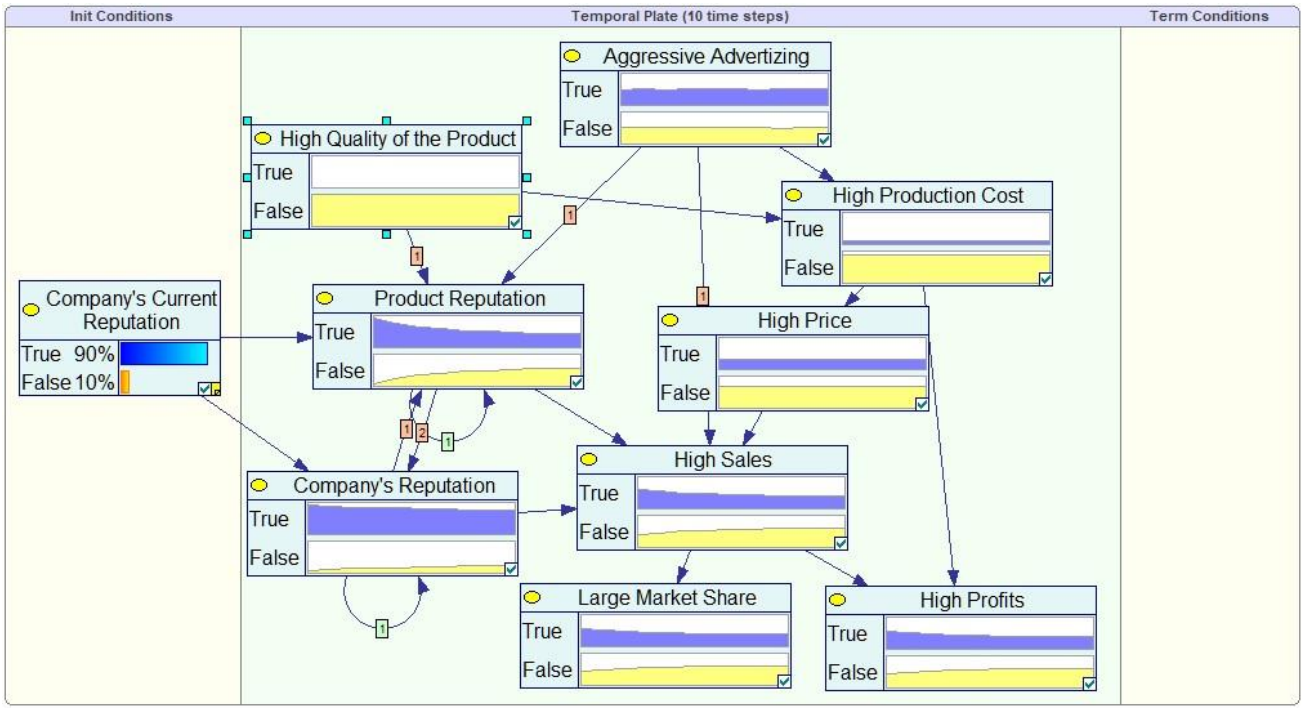
Consider a decision problem faced by a manufacturer, who is conscious of the fact that the quality of a new product will come at a price (high production cost), which will drive the product price up and, effectively, lead to decrease of sales, profits, and market share. On

the other hand, high product quality will positively impact the product reputation over time and the product reputation will, again over time, impact the reputation of the company. These impact directly sales and profits. The problem is complex, as reputation (both products and companies) is to some degree a self-propelling process, i.e., reputation at a given point in time, in addition to other factors, impacts reputation in the immediate future. Company's reputation will also impact the reputation of its products, including the current product. The decision is not a one-shot decision, whose effects we will know immediately but rather a decision whose effects will unfold over time. In order to assess the consequences of this decision, the manufacturer needs to model time explicitly and use dynamic Bayesian networks.

The following GeNIe screen shows the structure of a model representing this problem. The way that GeNIe represents dynamic models is different from other representations in that is does not focus on time slices but rather on variables. This representation allows for a compact modeling of higher order temporal influences. One thing that we can notice right away in the screen shot below is that there are special arcs in the model that are labeled by numbers inside little squares. These arcs represent temporal influences and the numbers denote their order. An influence of the first order, represents an influence spanning over one time step. Influences of higher order, represent slower influences that span over multiple time steps. The number in the square expresses the number of time steps over which the influence spans. We can see that both Product Reputation and Company's Reputation impact themselves in the near future. There is an influence of Company's Reputation on Product Reputation and a slower, two-step influence of Product Reputation on Company's Reputation. Product Reputation will thus slowly impact Company's Reputation. Aggressive Advertising will impact both Product Reputation and High Sales over time.



Dynamic Bayesian network model allows us to calculate how probabilities of interest change over time. This is of vital interest to decision who deal with consequences of their decisions over time. The following plot shows the same model with nodes viewed as bar charts and High Quality of the Product set to False. We can see the marginal probabilities changing over the time horizon of 10 steps (this is a model parameter that can be easily changed).



Dynamic Bayesian Networks Python

- PGMPY implementation
- PyAgrum implementation

Run the code Files PGMPY and PyAgrum

PGMPY:

Suppose we have a sensor network that consists of 3 sensors measuring temperature, humidity, and pressure. The state of the environment (e.g., whether it's sunny or rainy) affects the temperature and humidity, and the pressure is affected by both the temperature and humidity. We want to use a DBN to model the relationships between the environment, the sensors, and the measurements over time.

First, we'll define the structure of the DBN using the **DynamicBayesianNetwork** class from **pgmpy.models**.

```
from pgmpy.models import DynamicBayesianNetwork as DBN
from pgmpy.factors.discrete import TabularCPD
# define the DBN structure
dbn = DBN()
```

```
# define the nodes (variables) and their temporal dependencies
dbn.add_nodes_from(['Environment', 'Temperature', 'Humidity', 'Pressure', 'Sensor1', 'Sensor2', 'Sensor3'])
```

```
dbn.add_edges_from([('Environment', 'Temperature'), ('Environment', 'Humidity'),
                    ('Temperature', 'Pressure'), ('Humidity', 'Pressure'),
                    ('Temperature', 'Sensor1'), ('Humidity', 'Sensor2'), ('Pressure', 'Sensor3')])

# define the temporal structure
dbn.add_temporal_edges([('Environment', 'Environment'), ('Temperature', 'Temperature'),
                        ('Humidity', 'Humidity'), ('Pressure', 'Pressure'),
                        ('Sensor1', 'Sensor1'), ('Sensor2', 'Sensor2'), ('Sensor3', 'Sensor3')])

# define the initial distributions for the nodes
environment_init = TabularCPD(variable='Environment', variable_card=2, values=[[0.5], [0.5]])
temperature_init = TabularCPD(variable='Temperature', variable_card=2, values=[[0.5], [0.5]])
humidity_init = TabularCPD(variable='Humidity', variable_card=2, values=[[0.5], [0.5]])
pressure_init = TabularCPD(variable='Pressure', variable_card=2, values=[[0.5], [0.5]])
sensor1_init = TabularCPD(variable='Sensor1', variable_card=2, values=[[0.5], [0.5]])
sensor2_init = TabularCPD(variable='Sensor2', variable_card=2, values=[[0.5], [0.5]])
sensor3_init = TabularCPD(variable='Sensor3', variable_card=2, values=[[0.5], [0.5]])

dbn.add_cpds(environment_init, temperature_init, humidity_init, pressure_init, sensor1_init, sensor2_init, sensor3_init)
```

This code defines the structure of the DBN using add_nodes_from and add_edges_from, and adds the temporal dependencies using add_temporal_edges. We also define the initial distributions for each node using TabularCPD, and add them to the model using add_cpds.

Next, we can use the DynamicBayesianNetworkSampler class from pgmpy.sampling to generate a sample from the DBN.

```
from pgmpy.sampling import DynamicBayesianNetworkSampler as DBNSampler
# generate a sample from the DBN
dbn_sampler = DBNSampler(dbn)
sample = dbn_sampler.forward_sample(n_samples=10)
print(sample)
```

This code uses **DBNSampler** to generate a sample from the DBN. We specify that we want to generate 10 samples using **forward_sample**. The resulting sample is a **pandas** dataframe containing

PYAGRUM

```
import pyAgrum as gum
import pyAgrum.lib.dynamicBN as gdyn

# create a new dynamic Bayesian network
dbn = gdyn.DynBayesNet()
# define the nodes (variables) and their temporal dependencies
dbn.addNode("Environment", 2)
dbn.addNode("Temperature", 2)
dbn.addNode("Humidity", 2)
dbn.addNode("Pressure", 2)
dbn.addNode("Sensor1", 2)
dbn.addNode("Sensor2", 2)
dbn.addNode("Sensor3", 2)

dbn.addArc("Environment", "Temperature")
dbn.addArc("Environment", "Humidity")
dbn.addArc("Temperature", "Pressure")
dbn.addArc("Humidity", "Pressure")
dbn.addArc("Temperature", "Sensor1")
dbn.addArc("Humidity", "Sensor2")
dbn.addArc("Pressure", "Sensor3")

# define the initial distributions for the nodes
environment_init = gum.Prior(0.5)
temperature_init = gum.Prior(0.5)
humidity_init = gum.Prior(0.5)
pressure_init = gum.Prior(0.5)
sensor1_init = gum.Prior(0.5)
sensor2_init = gum.Prior(0.5)
sensor3_init = gum.Prior(0.5)

dbn.cpt("Environment")[0] = environment_init.toarray()
dbn.cpt("Environment")[1] = 1 - environment_init.toarray()
dbn.cpt("Temperature")[0] = temperature_init.toarray()
dbn.cpt("Temperature")[1] = 1 - temperature_init.toarray()
dbn.cpt("Humidity")[0] = humidity_init.toarray()
dbn.cpt("Humidity")[1] = 1 - humidity_init.toarray()
dbn.cpt("Pressure")[0] = pressure_init.toarray()
dbn.cpt("Pressure")[1] = 1 - pressure_init.toarray()
dbn.cpt("Sensor1")[0] = sensor1_init.toarray()
dbn.cpt("Sensor1")[1] = 1 - sensor1_init.toarray()
dbn.cpt("Sensor2")[0] = sensor2_init.toarray()
dbn.cpt("Sensor2")[1] = 1 - sensor2_init.toarray()
dbn.cpt("Sensor3")[0] = sensor3_init.toarray()
dbn.cpt("Sensor3")[1] = 1 - sensor3_init.toarray()
```

```
# generate a sample from the DBN
dbn.generate(10)
print(dbn)
```

Explanation:

This code uses the DynBayesNet class from pyAgrum.lib.dynamicBN to create a new dynamic Bayesian network. We add the nodes using addNode, and define their dependencies using addArc. We define the initial distributions for each node using Prior, and set the CPTs for each node using cpt.

Finally, we generate a sample from the DBN using generate, and print the resulting DBN using print.

Markov Model

Markov models are named after Andrey Markov, who first developed them in the early 1900s. Markov models are a type of probabilistic model that is used to predict the future state of a system, based on its current state. In other words, Markov models are used to predict the future state based on the current hidden or observed states. Markov model is a finite-state machine where each state has an associated probability of being in any other state after one step. They can be used to model real-world problems where hidden and observable states are involved. Markov models can be classified into hidden and observable based on the type of information available to use for making predictions or decisions. Hidden Markov models deal with hidden variables that cannot be directly observed but only inferred from other observations, whereas in an observable model also termed as Markov chain, hidden variables are not involved.

To better understand Markov models, let's look at an example. Say you have a bag of marbles that contains four marbles: two red marbles and two blue marbles. You randomly select a marble from the bag, note its color, and then put it back in the bag. After repeating this process several times, you begin to notice a pattern: The probability of selecting a red marble is always two out of four, or 50%. This is because the probability of selecting a particular color of marble is determined by the number of that color of marble in the bag. In other words, the past history (i.e., the contents of the bag) determines the future state (i.e., the probability of selecting a particular color of marble).

Hidden Markov Model

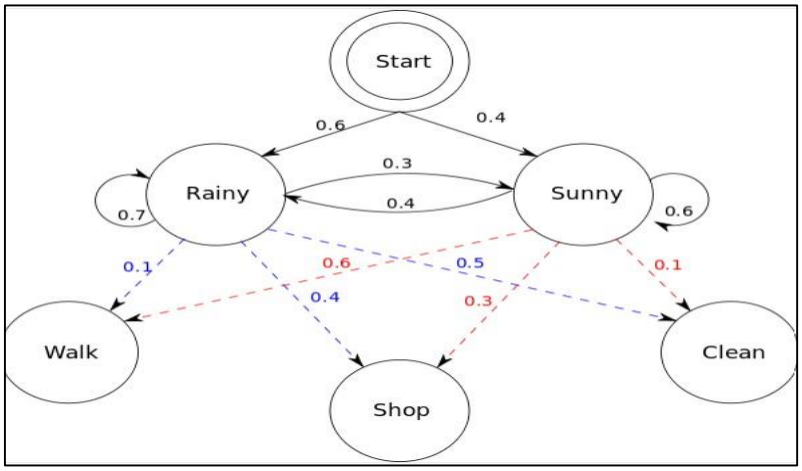
Hidden Markov models (HMMs) are a type of statistical modeling that has been used for several years. They have been applied in different fields such as medicine, computer science, and data science. The Hidden Markov model (HMM) is the foundation of many modern-day data science algorithms. It has been used in data science to make efficient use of observations for successful predictions or decisionmaking processes.

The hidden Markov model (HMM) is another type of Markov model where there are few states which are hidden. This is where HMM differs from a Markov chain. HMM is a statistical model in which the system being modeled are Markov processes with unobserved or hidden states. It is a hidden variable model which can give an observation of another hidden state with the help of the Markov assumption. The hidden state is the term given to the next possible variable which cannot be directly observed but can be inferred by observing one or more states according to Markov's assumption. **Markov assumption is the assumption that a hidden variable is dependent only on the previous hidden state.** Mathematically, the probability of being in a state at a time t depends only on the state at the time (t-1). It is termed a **limited horizon assumption**. Another Markov assumption states that the **conditional distribution over the next state, given the current state, doesn't change over time**. This is also termed a **stationary process assumption**.

A Markov model is made up of two components: the state transition and hidden random variables that are conditioned on each other. A hidden Markov model consists of five important components:

- Initial probability distribution: An initial probability distribution over states, πi is the probability that the Markov chain will start in state i. Some states j may have πj = 0, meaning that they cannot be initial states. The initialization distribution defines each hidden variable in its initial condition at time t=0 (the initial hidden state).
- One or more hidden states
- Transition probability distribution: A transition probability matrix where each aijaij represents the probability of moving from state i to state j. The transition matrix is used to show the hidden state to hidden state transition probabilities. A sequence of observations
- Emission probabilities: A sequence of observation likelihoods, also called emission probabilities, each expressing the probability of an observation oioi being generated from a state I. The emission probability is used to define the hidden variable in terms of its next hidden state. It represents the conditional distribution over an observable output for each hidden state at time t=0.

Let's understand the above using the hidden Markov model representation shown below:



The hidden Markov model in the above diagram represents the process of predicting whether someone will be found to be walking, shopping, or cleaning on a particular day depending upon whether the day is rainy or sunny. The following represents five components of the hidden Markov model in the above diagram:

HMM Python

```
from hmmlearn import hmm
import numpy as np

# define the model
model = hmm.MultinomialHMM(n_components=5)

# specify the parameters of the model
model.startprob_ = np.array([1.0, 0.0, 0.0, 0.0, 0.0])
model.transmat_ = np.array([[0.0, 1.0, 0.0, 0.0, 0.0],
                            [0.5, 0.0, 0.5, 0.0, 0.0],
                            [0.0, 0.5, 0.0, 0.5, 0.0],
                            [0.0, 0.0, 0.5, 0.0, 0.5],
                            [0.0, 0.0, 0.0, 0.0, 1.0]])
model.emissionprob_ = np.array([[1.0, 0.0, 0.0, 0.0, 0.0],
                               [0.0, 0.5, 0.5, 0.0, 0.0],
                               [0.0, 0.0, 1.0, 0.0, 0.0],
                               [0.0, 0.0, 0.5, 0.5, 0.0],
                               [0.0, 0.0, 0.0, 0.0, 1.0]])

# generate a sequence of observations from the model
X, state_sequence = model.sample(n_samples=10)


# fit the model to the data
```

```
model.fit(X)

# predict the most likely sequence of hidden states for a new sequence of observations
logprob, state_sequence = model.decode(X)
```

Explanation:

In this example, we use the **MultinomialHMM** class, which is a type of HMM with discrete emission probabilities. We have 5 hidden states, corresponding to the letters "h", "e", "l", "l", and "o". The **startprob_** attribute specifies that the first state is always the "h" state. The **transmat_** attribute specifies the transition probabilities between states. For example, the probability of transitioning from the "h" state to the "e" state is 1.0. The **emissionprob_** attribute specifies the probabilities of emitting each possible observation (i.e., each letter) from each state.

We then generate a sequence of 10 observations (i.e., letters) from the model using the **sample** method, and fit the model to the data using the **fit** method.

Finally, we use the **decode** method to predict the most likely sequence of hidden states for a new sequence of observations. The **logprob** variable contains the log-likelihood of the observation sequence given the model.

TASKS

Write Python Program for each task, create sample space and calculate the probability and show the output.

1. A manufacturing firm employs three analytical plans for the design and development of a particular product. For cost reasons, all three are used at varying times. In fact, plans 1, 2, and 3 are used for 30%, 20%, and 50% of the products, respectively. The defect rate is different for the three procedures as follows: P(D | P1)=0.01, P(D | P2)=0.03, P(D | P3)=0.02, where P(D | Pj) is the probability of a defective product, given plan j. If a random product was observed and found to be defective, which plan was most likely used and thus responsible?