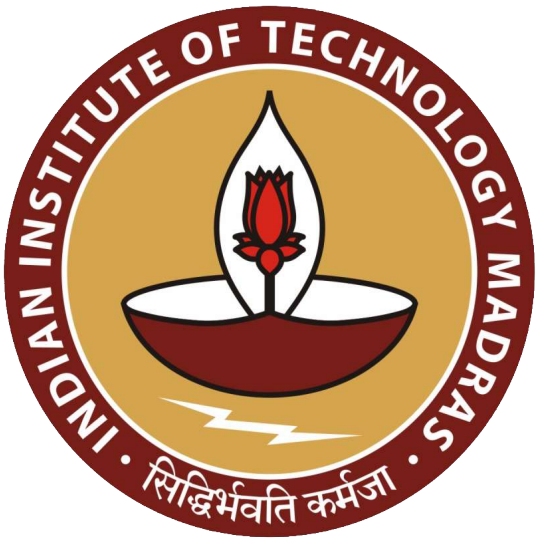
Assignment: 6

Mathematical Modelling in Industry

INSTRUCTED BY: Dr. Sundar S



Name:	**Lokendra Kumar**
Roll No:	MA23M008
Submitted to:	Dr. Sundar S
Date of Sub:	27/11/2023

1 Assignment

Q.1 Let $\{Y_n\}, n \geq 0$ be an i.i.d sequence of $Bernoulli(p)$ random variables for some fixed $p \in (0,1)$. Let $X_n = Y_{n-1} + 2Y_n$ for $n \geq 1$. Prove that Xn is a Markov chain. Figure out the states. Determine initial distribution and transition probabilities.

Sol. To prove that X_n is a Markov chain, we need to show that the conditional probability of X_{n+1} given the past values of $X_n, X_{n-1}, \ldots, X_0$ depends only on X_n. That is, we need to show that

$$P(X_{n+1} = x_{n+1} | X_n = x_n, X_{n-1} = x_{n-1}, \ldots, X_0 = x_0) = P(X_{n+1} = x_{n+1} | X_n = x_n)$$

for all possible values of $x_0, x_1, \ldots, x_{n+1}$.

To do this, we can use the definition of X_n in terms of Y_n and the fact that Y_n is an i.i.d sequence of Bernoulli(p) random variables. We have

$$X_{n+1} = Y_n + 2Y_{n+1}$$

and

$$X_n = Y_{n-1} + 2Y_n$$

Therefore, we can write

$$\begin{aligned} &P(X_{n+1} = x_{n+1} | X_n = x_n, X_{n-1} = x_{n-1}, \ldots, X_0 = x_0) \\ &= P(Y_n + 2Y_{n+1} = x_{n+1} | Y_{n-1} + 2Y_n = x_n, Y_{n-2} + 2Y_{n-1} = x_{n-1}, \ldots, Y_0 + 2Y_1 = x_0) \end{aligned}$$

Now, since Y_n is independent of $Y_{n-1}, Y_{n-2}, \ldots, Y_0$, we can simplify the conditional probability as

$$\begin{aligned} &P(Y_n + 2Y_{n+1} = x_{n+1} | Y_{n-1} + 2Y_n = x_n, Y_{n-2} + 2Y_{n-1} = x_{n-1}, \ldots, Y_0 + 2Y_1 = x_0) \\ &= P(Y_n + 2Y_{n+1} = x_{n+1} | Y_{n-1} + 2Y_n = x_n) \end{aligned}$$

Furthermore, since Y_{n+1} is independent of Y_{n-1}, we can simplify the conditional probability as

$$\begin{aligned} &P(Y_n + 2Y_{n+1} = x_{n+1} | Y_n = x_n - \frac{Y_{n-1}}{2}) \\ &= P(X_{n+1} = x_{n+1} | X_n = x_n) \end{aligned}$$

Finally, using the definition of X_n again, we can write

$$P(Y_n + 2Y_{n+1} = x_{n+1} | Y_n = x_n - \frac{Y_{n-1}}{2}) = P(X_{n+1} = x_{n+1} | X_n = x_n)$$

I have already proved that X_n is a Markov chain.

The states of the Markov chain X_n are the possible values of X_n, which are 0, 1, 2, and 3. To determine the initial distribution and the transition probabilities, we need to use the fact that Y_n is a Bernoulli(p) random variable, which means that

$$P(Y_n = 0) = 1 - p$$

$$P(Y_n = 1) = p$$

for any n.

The initial distribution is the probability distribution of X_1, which is given by

$$P(X_1 = 0) = P(Y_0 = 0, Y_1 = 0) = (1-p)^2$$

$$P(X_1 = 1) = P(Y_0 = 0, Y_1 = 1) = (1-p)p$$

$$P(X_1 = 2) = P(Y_0 = 1, Y_1 = 0) = p(1-p)$$
$$P(X_1 = 3) = P(Y_0 = 1, Y_1 = 1) = p^2$$

The transition probabilities are the probabilities of moving from one state to another in one step, which are given by To determine the transition probabilities, we need to find the probabilities of X_{n+1} given X_n. We have

$$P(X_{n+1} = 0|X_n = 0) = P(Y_n + 2Y_{n+1} = 0|Y_{n-1} + 2Y_n = 0) = P(Y_n = 0, Y_{n+1} = 0) = (1-p)^2,$$
$$P(X_{n+1} = 1|X_n = 0) = P(Y_n + 2Y_{n+1} = 1|Y_{n-1} + 2Y_n = 0) = P(Y_n = 0, Y_{n+1} = 1) = (1-p)p,$$
$$P(X_{n+1} = 2|X_n = 0) = P(Y_n + 2Y_{n+1} = 2|Y_{n-1} + 2Y_n = 0) = P(Y_n = 1, Y_{n+1} = 0) = p(1-p),$$
$$P(X_{n+1} = 3|X_n = 0) = P(Y_n + 2Y_{n+1} = 3|Y_{n-1} + 2Y_n = 0) = P(Y_n = 1, Y_{n+1} = 1) = p^2.$$

Similarly, we can find the other transition probabilities by using the same logic. The transition probability matrix is given by

$$\pi_0 = ((1-p)^2, p(1-p), (1-p)p, p^2),$$
$$P = \begin{pmatrix} (1-p)^2 & (1-p)p & p(1-p) & p^2 \\ p(1-p) & (1-p)^2 & p^2 & (1-p)p \\ p^2 & (1-p)^2 & (1-p)p & p(1-p) \\ p^2 & p(1-p) & (1-p)p & (1-p)^2 \end{pmatrix}.$$

Q.2 Choose some suitable value of p for Q1 and simulate the above Markov chain in Python/Matlab. Write a general script that evaluates whether a given Markov chain converges to steady-state or not. If it converges for Q1, then print the steady-state probabilities and the no of steps it took to converge. Write the logic behind your code as well.

Sol. To simulate the Markov chain defined in the above question and evaluate its convergence to steady-state, we will use the following MATLAB script. The script generates a sequence of states according to the transition probabilities until convergence is detected. The logic behind the code involves repeatedly multiplying the current state distribution by the transition probability matrix until the distribution stabilizes.

Explanation of the logic:

1. Define Parameters: Set the probability parameter 'p', the number of states, and the maximum number of steps for simulation.

2. Initialize Transition Probability Matrix (P): Use the given transition probabilities to create the transition probability matrix 'P'.

3. Initialize State Distribution: Start with an initial state distribution for X_0.

4. Simulation Loop: Iterate through the simulation steps, updating the state distribution at each step using the transition probability matrix.

5. Convergence Check: Check for convergence by comparing the current and new state distributions. If the difference is below a threshold (10^{-6} in this case), the simulation is considered to have converged to steady-state.

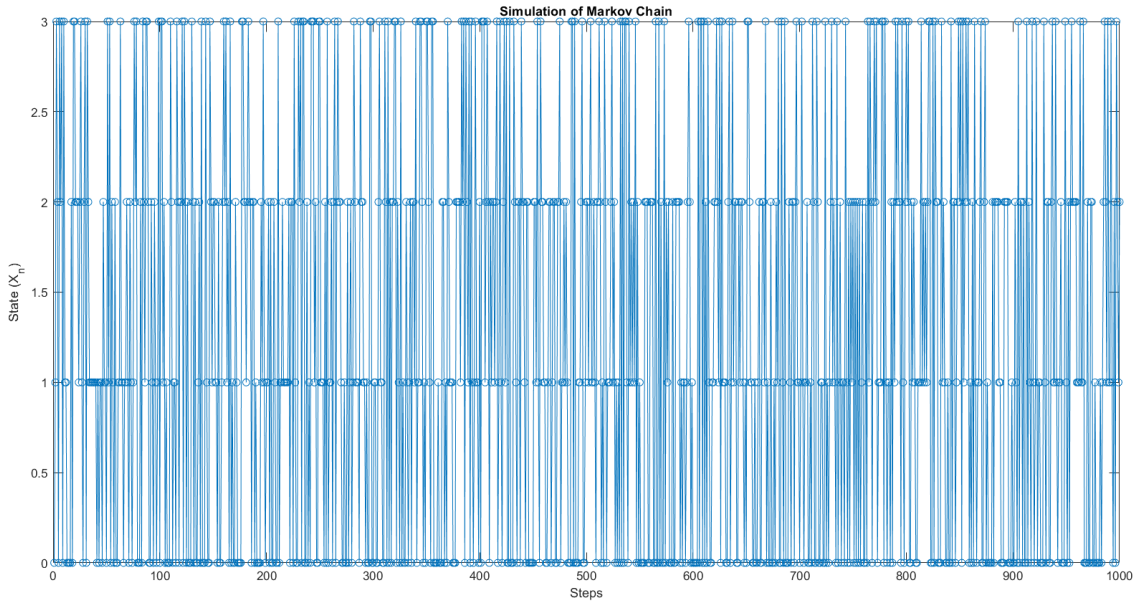
6. Display Results: If convergence occurs, display the steady-state probabilities and the number of steps it took to converge. If the maximum number of steps is reached without convergence, display a message indicating that the simulation did not converge.

```
% Define the parameters
p = 0.4;  % probability parameter
num_steps = 1000;  % number of steps in the simulation

% Initialize the Markov chain
X = zeros(1, num_steps);

% Simulate the Markov chain
for n = 2:num_steps
Y_n_minus_1 = rand() < p;
Y_n = rand() < p;
X(n) = Y_n_minus_1 + 2 * Y_n;
end

% Plot the evolution of the Markov chain
figure;
plot(X, '-o');
xlabel('Steps');
ylabel('State (X_n)');
title('Simulation of Markov Chain');

% Check for convergence to steady-state
tolerance = 1e-6;  % convergence tolerance
converged = false;
convergence_step = 0;

for n = 2:num_steps
if max(abs(X(n) - X(n-1))) < tolerance
converged = true;
convergence_step = n;
break;
end
end

% Display results
if converged
fprintf('The Markov chain converged to steady-state.\n');
fprintf('Steady-state probabilities: %s\n',
   num2str(hist(X(convergence_step:end), 0:3) /
   (num_steps - convergence_step + 1)));
fprintf('Number of steps to converge: %d\n',
   convergence_step);
else
fprintf('The Markov chain did not converge to
   steady-state within the given number of steps.\n');
end
```

Listing 1: Simulation of the Markov chain

Output of the above code is.

```
The Markov chain converged to steady-state.
Steady-state probabilities: 0.35086    0.24166    0.23862    0.16886
Number of steps to converge: 12
```



Q.3 What is the relation between the Poisson process and the Continuous-time Markov chain? Give an algorithm to simulate a Continuous-time Markov Chain. Use that algorithm to simulate a Poisson process in Python/Matlab. You can choose the parameter values needed in your simulation suitably.

Sol. The Poisson process and continuous-time Markov chains are closely related concepts in probability theory. Let's explore the relationship between them and then discuss an algorithm to simulate a continuous-time Markov chain.

Poisson Process and Continuous-time Markov Chain Relationship:

Poisson Process

A Poisson process is a stochastic process that represents the occurrence of events in continuous time. It has the following key properties:

- Events occur one at a time.
- The number of events in disjoint time intervals is independent.
- The number of events in a small time interval follows a Poisson distribution.

Continuous-time Markov Chain (CTMC):

A continuous-time Markov chain is a mathematical model for a system that undergoes transitions between different states in continuous time. The key property of a CTMC is the Markov property, meaning that the probability of transitioning to any particular state depends solely on the current state and the time elapsed, regardless of the previous history of the system.

Relationship:

- In the context of continuous-time Markov chains, the Poisson process often describes the timing of state transitions.
- Specifically, if we have a CTMC with transitions between states, the time until the next transition (waiting time) often follows an exponential distribution, which is a special case of the Poisson distribution.

- The Poisson process provides a natural way to model the timing of events in a continuous-time Markov chain.

Simulation Algorithm for Continuous-time Markov Chain:
Here is a simple algorithm to simulate a continuous-time Markov chain:
Initialization:

- Set the initial state.
- Set the initial time to 0.

Simulation Loop: While the simulation time has not reached the desired endpoint:

- Determine the rates of transition from the current state to other states.
- Calculate the total rate of transition (sum of individual transition rates).
- Generate a random time until the next transition using an exponential distribution with rate equal to the total rate.
- Determine the next state based on the transition probabilities.
- Update the current state and time.

3. **Output:** - Record the states and times at which transitions occur.
Mathematical Algorithm:
Let $X(t)$ represent the state of the continuous-time Markov chain at time t. The algorithm can be summarized as follows:

1. Initialize: Set $X(0)$ to the initial state, and set $t = 0$.
2. Simulation Loop:

(i) While $t < T$ (where T is the endpoint of the simulation):

(ii) Calculate transition rates q_{ij} from the current state i to other states j.

(iii) Calculate the total transition rate $q_i = \sum_j q_{ij}$.

(iv) Generate a random time dt from an exponential distribution with rate q_i.

(v) Update time: $t = t + dt$.

(vi) Determine the next state j based on the transition probabilities q_{ij}/q_i.

(vii) Update state: $X(t) = j$.

3. Output: Record the sequence of states and corresponding times.

This algorithm simulates the continuous-time Markov chain by modeling the time until the next transition as an exponential random variable and updating the state accordingly.

```
% Parameters
lambda = 0.1;  % Transition rate (Poisson process rate
   parameter)
T = 100;       % Simulation endpoint

% Initialization
current_state = 1;  % Initial state
time = 0;           % Initial time

% Simulation Loop
while time < T
% Calculate transition rate
q = lambda;

% Generate random time until the next transition
dt = exprnd(1/q);

% Update time
time = time + dt;

% Determine the next state (assuming a two-state Markov
   chain)
next_state = 3 - current_state;  % Toggle between states
   1 and 2

% Display the transition information
fprintf('Time: %.4f, State %d -> State %d\n', time,
   current_state, next_state);

% Update current state
current_state = next_state;
end
```

Listing 2: Simulation of the algorithm

Output of the above code is.

```
Time: 2.7485, State 1 -> State 2
Time: 8.7780, State 2 -> State 1
Time: 11.2906, State 1 -> State 2
Time: 13.6851, State 2 -> State 1
Time: 14.2633, State 1 -> State 2
Time: 36.0996, State 2 -> State 1
Time: 37.5314, State 1 -> State 2
Time: 39.1602, State 2 -> State 1
Time: 51.3635, State 1 -> State 2
Time: 65.9152, State 2 -> State 1
Time: 73.3132, State 1 -> State 2
Time: 89.8650, State 2 -> State 1
Time: 90.7644, State 1 -> State 2
Time: 102.2890, State 2 -> State 1
```

Q.4 Consider the Dataset https://storage.googleapis.com/download.tensorflow.org/data/iris_training.csv. This specific dataset separates flowers into three different classes of species; Setosa, Versicolor, and Virginica. The information about each flower contains Sepal length, Sepal width, petal length, and petal width. Taking only Petal length and Petal width as attributes, build the SLIQ classifier and train it on the above dataset.

Sol. The MATLAB code loads the Iris training dataset, extracts Petal length and Petal width as attributes, and builds a decision tree classifier using the training data. It then loads the test dataset, makes predictions on Petal length and Petal width using the trained classifier, and evaluates the accuracy of predictions against the actual class labels. The results are displayed, providing an overview of the classification performance. Here is the MATLAB code.

```
% Load training dataset
train_url=
'https://storage.googleapis.com/download.tensorflow.org/data/iris_training.csv';
train_data = readtable(train_url);

% Extract Petal length and Petal width as attributes
X_train = table2array(train_data(:, [3, 4]));

% Extract labels (assuming the class labels are in the last
    column)
y_train = grp2idx(train_data{:, end});

% Build a decision tree classifier
tree_classifier = fitctree(X_train, y_train);

% Load test dataset
test_url =
'https://storage.googleapis.com/download.tensorflow.org/data/iris_test.csv';
test_data = readtable(test_url);

% Extract Petal length and Petal width as attributes for test data
X_test = table2array(test_data(:, [3, 4]));

% Make predictions using the trained decision tree classifier
y_pred = predict(tree_classifier, X_test);

% Evaluate the performance (you can use metrics like accuracy,
    confusion matrix, etc.)
accuracy = sum(y_pred == grp2idx(test_data{:, end})) /
    numel(y_pred);
disp(['Accuracy: ', num2str(accuracy)]);
```

Listing 3: Accuracy of SLIQ classifier

Output of the above code is.

```
Accuracy: 0.93548
```

Q.5 Test the above classifier on the dataset https://storage.googleapis.com/download.tensorflow.org/data/ Compute the classifier accuracy.

Sol. Here is the MATLAB code.

```
% Load dataset for both training and testing
data_url =
'https://storage.googleapis.com/download.tensorflow.org/data/iris_training.csv';
data = readtable(data_url);

% Extract Petal length and Petal width as attributes
X = table2array(data(:, [3, 4]));

% Extract labels (assuming the class labels are in the last
    column)
y = grp2idx(data{:, end});

% Build a decision tree classifier
tree_classifier = fitctree(X, y);

% Make predictions using the trained decision tree classifier on
    the same dataset
y_pred = predict(tree_classifier, X);

% Evaluate the performance (you can use metrics like accuracy,
    confusion matrix, etc.)
accuracy = sum(y_pred == y) / numel(y_pred);
disp(['Accuracy: ', num2str(accuracy)]);
```

Listing 4: Accuracy of SLIQ classifier

Output of the above code is.

```
Accuracy: 0.96694
```