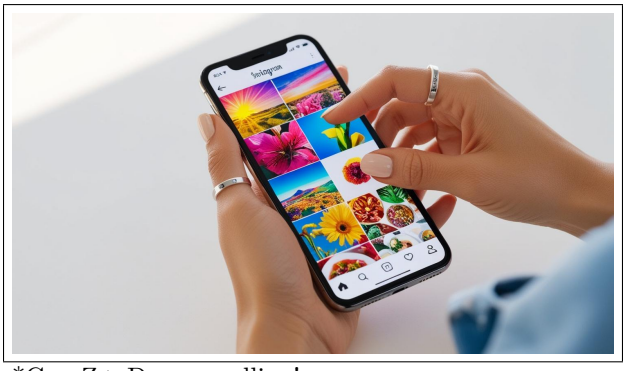
Mastering Markov Chains: Bridging the Gap Between Theory and Real-World Applications in Human Behavior

Minor In AI, IIT Ropar
27th March, 2025

Welcome, aspiring AI explorers! This module is designed to guide you on a fascinating journey into the world of Markov Chains, a powerful tool for understanding and modeling sequential decision-making, especially in scenarios where human behavior plays a central role. We'll start with the core ideas, build your intuition, and then translate that understanding into practical Python code. Don't worry if you're new to Python or AI – we'll take it one step at a time.

The Instagram Scroll: A Day in the Life of a Markov Chain

Imagine you're scrolling through Instagram. What happens? You see a post, maybe you like it, comment on it, or keep scrolling. Sometimes you get bored and close the app. This seemingly random behavior can actually be modeled using a Markov Chain.
Think about it: what you do *right now* is influenced by what you are doing *right now*. For instance, if you are currently scrolling, there's a pretty good chance you'll keep scrolling. But, eventually, something might catch your eye, leading you to like and comment. Or, you might realize you have an exam to study for, and you close the app.


*Gen Z+ Doomscrolling!

Traditional machine learning often focuses on historical data. "You've liked 100 cat videos in the past, so we'll show you more cat videos!" But what if today you're feeling more like dogs? Or what if you're simply tired of cute animals and want to look at landscapes? Markov Chains allow us to consider that *current state* of mind. This is especially relevant to model the uncertainty associated with real-time behavior.
Let's say, for simplicity, there are three possible actions:

1. **Scrolling**
2. **Liking/Commenting**
3. **Closing the App**

A Markov Chain would model the *probabilities* of transitioning between these states. For example:

- If you are currently **Scrolling**, there's a 70% chance you'll continue **Scrolling**, a 20% chance you'll **Like/Comment**, and a 10% chance you'll **Close the App**.

- If you are currently **Liking/Commenting**, there's a 30% chance you'll stay in **Liking/Commenting**, a 50% chance you'll go back to **Scrolling**, and a 20% chance you'll **Close the App**.
- If you have **Closed the App**, there's an 80% chance you'll start **Scrolling** again, a 15% chance you'll go straight to **Liking/Commenting**, and only a 5% chance you'll keep the app closed.

These probabilities tell us how likely you are to move from one state (activity) to another. The key idea is that your next action *only* depends on your current action, not on the entire history of your Instagram usage. That's the defining characteristic of a Markov Chain!
This simple example illustrates the core concept. Now, let's delve into the theory and see how we can implement this using Python.

Building Your First Markov Chain in Python

Now, let's see how to represent the above Instagram transitions using a Markov Chain in Python using the `NumPy` library.

```
import numpy as np

# Create a transition matrix
transition_matrix = np.array([
    [0.7, 0.2, 0.1],  # Probabilities from Scrolling to Scrolling, Liking/Commenting, Closing
    [0.5, 0.3, 0.2],  # Probabilities from Liking/Commenting to Scrolling, Liking/Commenting, Closin
    [0.8, 0.15, 0.05] # Probabilities from Closing to Scrolling, Liking/Commenting, Closing
])

# Possible states
states = ["Scrolling", "Liking/Commenting", "Closing"]

# Initial state
current_state = "Scrolling"
print(f"You are currently {current_state}")

# Simulate the next state based on the transition matrix
current_state_index = states.index(current_state) #Find index for the current state
next_state_index = np.random.choice(len(states), p=transition_matrix[current_state_index]) #Return a
next_state = states[next_state_index] # Map index back to a state
print(f"You will transition to {next_state}")
```

Code Breakdown:

1. `import numpy as np`: Imports the NumPy library, essential for numerical operations, especially for working with arrays.
2. `transition_matrix = np.array(...)`: This is the heart of our Markov Chain. It's a 2D NumPy array representing the probabilities of transitioning between states. Each row represents the current state, and each column represents the possible next states. The values in the matrix are the probabilities.
   - The **first row** (`[0.7, 0.2, 0.1]`) represents the probabilities *when you are currently scrolling.* 70% chance of continuing to scroll, 20% chance of liking/commenting, and 10% chance of closing the app.
   - The **second row** (`[0.5, 0.3, 0.2]`) represents the probabilities *when you are currently liking/commenting.* 50% chance of going back to scrolling, 30% chance of continuing to like/comment, and 20% chance of closing the app.
   - The **third row** (`[0.8, 0.15, 0.05]`) represents the probabilities *when you have closed the app.* 80% chance of starting scrolling again, 15% chance of going straight to liking/commenting, and only a 5% chance of keeping the app closed.

   Important: Notice that the probabilities in each row *must add up to 1.* This ensures that all possible transitions from a given state are accounted for.

3. `states = ["Scrolling", "Liking/Commenting", "Closing"]`: A simple list of the possible states we can be in.
4. `current_state = "Scrolling"`: Here, we manually specify the starting state.
5. `current_state_index = states.index(current_state)`: Gets the index of the current state from the list of possible states.
6. `next_state_index = np.random.choice(len(states), p=transition_matrix[current_state_index])`: This is where the "magic" happens. The next step is randomly selected using probabilities from the transition matrix based on the current state.
   - `np.random.choice(len(states), ...)`: Chooses a random index from the range of states in the chain.
   - *`p=transition_matrix[current_state_index]`*: Takes the appropriate row from the transition matrix (based on the current state) for the list of probabilities for the next state.
7. `next_state = states[next_state_index]`: Maps the index of the next state back to the name of the next state.
8. `print(...)`: Prints the next state

When you run this code, it will randomly choose the next state based on the probabilities defined in the `transition_matrix`. Each time you run it, you might get a different result!

Understanding the Transition Matrix

Let's break down the transition matrix in more detail:

```
[0.7, 0.2, 0.1],  # Probabilities from Scrolling
[0.5, 0.3, 0.2],  # Probabilities from Liking/Commenting
[0.8, 0.15, 0.05] # Probabilities from Closing
```

- **Rows**: Each row represents the *current state*. The first row is "Scrolling", the second is "Liking/Commenting", and the third is "Closing".
- **Columns**: Each column represents the *possible next states*. The first column is "Scrolling", the second is "Liking/Commenting", and the third is "Closing".
- **Values**: The values at each intersection represent the *probability* of moving from the row's current state to the column's next state.
  - `transition_matrix[0][0]` (0.7): The probability of going from "Scrolling" to "Scrolling" (staying in the same state).
  - `transition_matrix[0][1]` (0.2): The probability of going from "Scrolling" to "Liking/Commenting".
  - `transition_matrix[2][0]` (0.8): The probability of going from "Closing" back to "Scrolling".

Understanding how to read and interpret the transition matrix is crucial to grasping the behavior of the Markov Chain.

From States to Simulation

Now that we understand the basics, let's create a simulation to observe the Markov Chain in action over a longer period:

```
import numpy as np

# Transition Matrix (same as before)
transition_matrix = np.array([
    [0.7, 0.2, 0.1],
    [0.5, 0.3, 0.2],
    [0.8, 0.15, 0.05]
```

```
])

# States (same as before)
states = ["Scrolling", "Liking/Commenting", "Closing"]

# Initial State
current_state = "Scrolling"

# Number of steps to simulate
num_steps = 15

# Simulation Loop
print(f"Starting simulation from {current_state}:")
for i in range(num_steps):
    current_state_index = states.index(current_state) #Find index for the current state
    next_state_index = np.random.choice(len(states), p=transition_matrix[current_state_index]) #Retu
    current_state = states[next_state_index] # Map index back to a state
    print(f"Step {i+1}: You are now {current_state}")
```

Key Improvements:

- **Simulation Loop:** The `for` loop simulates the Markov Chain for a specified number of steps (`num_steps = 15`).
- **State Update:** In each step, the `current_state` is updated to the `next_state`, reflecting the transition based on the probabilities.
- **Output:** The code now prints the state at each step of the simulation, giving us a sequence of states representing the user's "journey" through Instagram.

Run this code, and you'll see a different sequence of Instagram activities each time, based on the defined probabilities! This showcases the power of Markov Chains in simulating real-world, probabilistic behavior.

Beyond Instagram: Applications of Markov Chains

While our Instagram example is illustrative, Markov Chains have a wide range of applications:

- **Speech Recognition**: Predicting the next word in a sentence.
- **Financial Modeling**: Predicting stock prices or market movements.
- **Genetics**: Modeling the evolution of genes.
- **Recommendation Systems**: Suggesting products or content based on user behavior.
- **Weather Forecasting**: Predicting future weather conditions.

The core principle remains the same: understanding the probabilities of transitioning between different states allows us to model and predict sequential events.

Balancing Past Data and Current State

As we've seen, Markov Chains emphasize the importance of the *current state* in determining the next action. However, it's crucial to remember that traditional AI models rely heavily on *historical data*. The best approach often involves finding a balance between these two perspectives.
Consider our Instagram example again. While the current state (scrolling, liking, closing) strongly influences the next action, historical data could provide valuable insights into user preferences. For instance, if a user consistently likes posts from a particular account, the probability of liking a future post from that account might be higher.
By combining historical data with the principles of Markov Chains, we can create more accurate and personalized models of human behavior.

Congratulations! You've taken your first steps into the fascinating world of Markov Chains. You've learned the core concepts, seen how to implement them in Python, and explored a range of real-world applications.
Remember, this is just the beginning. As you continue your journey, explore more advanced topics such as Hidden Markov Models, reinforcement learning, and the application of Markov Chains to more complex problems. The possibilities are endless!