STA 130 HW2

September 23, 2024

Q2:

[3]:
```python
import numpy as np

all_doors = [1, 2, 3]
initial_choice = 1  # Player's initial choice
wins = 0
reps = 100000

for _ in range(reps):
    winning_door = np.random.choice(all_doors)  # Randomly select the winning␣
 ↪door

    # Monty reveals a losing door that is neither the player's initial choice␣
 ↪nor the winning door
    remaining_doors = [door for door in all_doors if door != initial_choice and␣
 ↪door != winning_door]
    monty_reveals = np.random.choice(remaining_doors)

    # Player swaps to the other remaining door
    swap_choice = [door for door in all_doors if door != initial_choice and␣
 ↪door != monty_reveals][0]

    # Check if the swapped choice is the winning door
    if swap_choice == winning_door:
        wins += 1

win_rate = wins / reps
print(f"Win rate by switching: {win_rate:.2f}")
```

```
Win rate by switching: 0.67
```

Key Improvements: List Comprehensions: The code uses list comprehensions to remove the need for explicit removal and try-except blocks, making the logic cleaner.

Monty's reveal: We create a list of remaining doors that are neither the player's choice nor the winning door. Swap choice: After Monty reveals a door, the player switches to the remaining door. No unnecessary list conversions: The original code converted tuples to lists to remove elements, which adds complexity. This version directly works with lists.

Improved variable names: remaining_doors, monty_reveals, and swap_choice provide clearer meanings.

Explanation: Monty reveals a door: The host reveals one of the losing doors, chosen from doors that are not the player's initial choice or the winning door. Player swaps: The player switches their choice to the remaining door. Check win: The code checks if the new choice (after swapping) is the winning door. Win rate: The final win rate is computed and displayed. This version should be easier to follow and explain while preserving the logic of the original Monty Hall simulation.

Q3:

[ ]:
```
secret_winning_door = np.random.choice(all_door_options)
```

The winning door is randomly chosen from the three doors.

[ ]:
```
all_door_options_list.remove(secret_winning_door)
```

The list all_door_options_list is a mutable version of all_door_options. The code removes the winning door from this list, leaving the two non-winning doors.

[ ]:
```
try:
    all_door_options_list.remove(my_door_choice)
except:
    pass
```

If the player's original choice matches the winning door, the code tries to remove it:

[ ]:
```
goat_door_reveal = np.random.choice(all_door_options_list)
all_door_options_list.remove(goat_door_reveal)
```

Revealing a "goat" door: Monty Hall (the host) reveals a "goat" door (a losing door) from the remaining doors

[ ]:
```
my_door_choice = all_door_options_list[0]
```

Swap strategy: Now that one of the losing doors is revealed, the player uses the "swap strategy," which means they switch their choice to the remaining door

[ ]:
```
if my_door_choice == secret_winning_door:
    i_won += 1
```

Checking if the player won: After switching, the code checks whether the new choice matches the secret winning door. If so, it increments the win counter i_won

[ ]:
```
i_won / reps
```

At the end of the loop, the proportion of times the player wins (by using the switch strategy) is calculated

Q6

ChatBot Interaction for Understanding the Monte Hall Problem:The ChatBot was able to explain the Monte Hall problem simulation code quickly and effectively. By walking through the code and clarifying the steps, the ChatBot provided a clear and logical explanation. The code itself was straightforward, and the ChatBot's explanation made it easier to grasp the probability mechanics behind it.

ChatBot Interaction for Understanding the "Markovian ChatBot" Code:When it came to understanding the "Markovian ChatBot" code, the ChatBot provided a solid breakdown of the core components, like building the Markov chain and generating text. For an initial walkthrough of the code, it was very effective. The ChatBot even pointed out how characters are used as the basis for transitions and how the state transitions occur using probabilities.The more complicated extensions, like character-specific Markov chains and bigram dependency, required a bit more effort to explain.

Overall Assessment of ChatBots for Troubleshooting and Understanding Code:

Quick Responses: The ChatBot was able to quickly break down the logic of various codes, whether it was for a simulation or Markov chain text generation. Code Debugging: In past experiences, using ChatBots to troubleshoot coding errors has been useful. The ChatBot can usually point out syntax issues or help with logic flow, which makes debugging faster. Conceptual Understanding: ChatBots are great for grasping the underlying theory of algorithms and simulations. Concepts like probability or Markov chains can be difficult, but the ChatBot makes them more digestible by going step-by-step through the code.

For more advanced extensions, like the character-specific Markov chains or bigrams, the ChatBot might need some hints or clarifications to fully understand the nuances. It sometimes needs extra guidance to pinpoint what is different in more complex modifications.

Q7

My perspective on AI-driven tools has shifted from viewing them as mere quick fixes to recognizing them as valuable aids for deeper learning. They now play a crucial role in my learning process, helping not only with troubleshooting but also with mastering and reinforcing concepts in coding, statistics, and data science. However, since AI isn't always perfect, it's important to maintain our own judgment and continue learning independently.

Q8:https://chatgpt.com/share/66eb61c8-8f08-800f-b382-26810dbd6ec5

link of Chat GPT:https://chatgpt.com/share/66eb61f5-c7e8-800f-be3d-3a78e78c8552