Tutorial V.II - Recap with Randomness

Programming: Everyday Decision-Making Algorithms

Introduction

From now on, it is rather difficult to give you tasks that you cannot solve within minutes by yourself with the help of AI. Therefore, today's tutorial will be more open-ended than usual. All tasks are designed to be solved together with AI, but you are nonetheless the human in the loop. Thus, your responsiblities are to control the AI and to learn how to use it by doing. Instructions and explanations will only be given sparsely, as you are encouraged to use AI to ask for help and explanations to explore the tasks on your own.

Let's explore how randomness affects our daily decision-making. As always, we will use Python and simulate various scenarios where randomness plays a crucial role.

Section 1 - Optimal Stopping with Random Candidates

Optimal stopping is a classic problem in decision theory where you want to select the best candidate from a list of candidates. As seen in the lecture, the 37% rule is a simple strategy that can be used to solve this problem.

Exercise 1.1 - The 37% Rule

Let's implement the 37% rule from Optimal Stopping by using random candidates. Your task is to simulate the process of selecting the best candidate out of a list of random candidates. Remember, the 37% rule says that you should reject the first 37% of candidates and then select the next candidate who is better than all previous candidates.

First, let's import the necessary library:

```
import random
```

In total, you have 32 candidates. Screen the candidates one by one and compute the `best_candidate_index` to find the best candidate.

How to start with AI pair programming

1. In Cursor, you can select the code or text you want to work with and then press `Ctrl+L` to open the chat.
2. In the chat, you can ask the AI to help you complete the task or ask it to explain the code, the task, or related concepts
3. While writing code, you can press `Ctrl+K` to let the AI help you write the code directly in the editor.
4. Furthermore, the AI will try to understand your code and might make suggestions on how to improve it or on how to continue. Just press `TAB` to accept a suggestion.

```
random.seed(42) # Sets the random seed for reproducibility
candidates = [random.random() for _ in range(32)]

# YOUR CODE BELOW
```

```
# Test your solution with the following code
assert best_candidate_index == 24, "The selected candidate is not the best
one"
print(r"All tests passed, you have successfully implemented the 37% rule!")
```

Section 2 - Explore vs Exploit

Remember the explore vs. exploit dilemma? Should you try a new restaurant (explore) or return to your favorite (exploit)? Let's simulate this decision-making process!

Exercise 2.1 - Visiting Restaurants

Imagine you're new to a city with 10 restaurants. Each restaurant has a "true" quality rating (unknown to you) between 0 and 5 stars. Each time you visit a restaurant, you get to experience its rating. In total, you will visit restaurants 15 times.

Your goal is to create a simulation that computes the average satisfaction over 15 days based on an explore vs. exploit strategy. First, you should try each restaurant once (pure exploration). Then, you should just exploit your favorite restaurant for the remaining visits. Save the average satisfaction in `average_satisfaction` and the index of the best restaurant in `best_restaurant_index`.

```
import random

# Set up the restaurants (true qualities are unknown to the visitor!)
random.seed(59)
qualities = [random.uniform(1, 5) for _ in range(10)]

# YOUR CODE BELOW
```

```
# Test your solution with the following code
assert average_satisfaction > 3.7, "The average satisfaction should be
greater than 3.7"
assert best_restaurant_index == 2, "The best restaurant should be number 3"
print(r"All tests passed, you have successfully implemented the explore vs.
exploit strategy!")
```

Section 3 - Caching

Caching is a technique that can be used to speed up computations by storing the results of expensive computations and reusing them when the same inputs occur again. Let's implement a simple caching mechanism in Python.

Exercise 3.1 - Random Menu Generator

Create a function `generate_daily_menu()` that generates a daily menu by randomly selecting items from different categories. As our "cache" - the daily menu - is rather small, we only want to receive one menu for each day as a list from the function.

```
appetizers = ["Sorted Salad", "Binary Bruschetta", "Array of Antipasti"]
mains = ["Loop Lasagna", "Python Pasta", "Recursive Risotto"]
desserts = ["Binary Brownie", "Cache Cookie", "Stack Sundae"]

# YOUR CODE HERE
```

```
# Test your solution with the following code
menu = generate_daily_menu()
assert len(menu) == 3, "Menu should contain exactly 3 items"
assert menu[0] in appetizers, "Appetizer should be from the appetizers
list"
assert menu[1] in mains, "Main course should be from the mains list"
assert menu[2] in desserts, "Dessert should be from the desserts list"

# Example usage
print(r"All tests passed, you have successfully implemented the menu
generator!")
print(f"Today's menu:\nAppetizer: {menu[0]}\nMain: {menu[1]}\nDessert:
{menu[2]}")
```

Section 4 - Scheduling

Scheduling is a classic problem in operations research and in daily life. Let's implement a simple scheduling algorithm that tries to find the optimal schedule for a given set of tasks with the earliest due date strategy.

Exercise 4.1 - Scheduling Tasks

Let's create a task scheduler that handles tasks with different due dates. The scheduler should sort tasks by their due date (Earliest Due Date First - EDD strategy) to minimize potential delays.

Your task is to create a function `schedule_tasks(tasks)` that:

- Sorts tasks by due date
- Creates a schedule with start and end times for each task
- Returns the complete schedule as an pandas DataFrame
- Keep the same column names as in the dictionary

```
# Pre-generated tasks (don't modify this!)
# Times are represented in hours from start (hour 0)
# For example, due_time: 24 means it's due 24 hours from start
import pandas as pd

tasks = [
    {
        "name": "Project Review",
        "duration": 2,  # hours
        "due_time": 12  # due in 12 hours
    },
    {
        "name": "Team Meeting",
        "duration": 4,
        "due_time": 6
    },
    {
        "name": "Client Presentation",
        "duration": 3,
        "due_time": 7
    },
    {
        "name": "Email Updates",
        "duration": 4,
        "due_time": 11
    },
    {
        "name": "Planning Session",
        "duration": 2,
        "due_time": 9
    }
]

# YOUR CODE HERE
```

```
# Test your solution with the following code
schedule_df = schedule_tasks(tasks)
assert schedule_df.shape == (5, 4), "The schedule should have 5 rows and 4
columns"
assert schedule_df.iloc[0]['name'] == "Team Meeting", "First task should be
'Team Meeting'"
assert schedule_df.iloc[4]['name'] == "Project Review", "Fifth task should
be 'Project Review'"
print(r"All tests passed, you have successfully implemented the task
scheduler!")
```

Section 5 - Randomness

In our daily lives, we often encounter situations where randomness plays a crucial role in decision-making. From choosing a restaurant for dinner to selecting which tasks to tackle first, incorporating some randomness can actually lead to better outcomes than strictly deterministic approaches.

Solving your travel problem by brute force

Imagine you want to travel during the semester break and you want to visit 6 cities and then return to Hamburg. Your aim is to find the route that is the cheapest to travel. Implement a heuristic that randomly tries 10 different routes and picks the cheapest one. Save the costs in `min_cost`.

```
# City names and cost matrix
import pandas as pd
import random

cities = ["New York", "London", "Tokyo", "Sydney", "Barcelona", "Hamburg"]

cost_matrix = [
    [0, 700, 1500, 2000, 800, 900],  # New York
    [700, 0, 1200, 1800, 300, 700],  # London
    [1500, 1200, 0, 1000, 1100, 1400], # Tokyo
    [2000, 1800, 1000, 0, 1700, 1500], # Sydney
    [800, 300, 1100, 1700, 0, 600],  # Barcelona
    [900, 700, 1400, 1500, 600, 0]   # Hamburg
]

# Create a DataFrame
cost_df = pd.DataFrame(cost_matrix, index=cities, columns=cities)

# Display the DataFrame
print(cost_df)

# YOUR CODE HERE
```

```
           New York  London  Tokyo  Sydney  Barcelona  Hamburg
New York          0     700   1500    2000        800      900
London          700       0   1200    1800        300      700
Tokyo          1500    1200      0    1000       1100     1400
Sydney         2000    1800   1000       0       1700     1500
Barcelona       800     300   1100    1700          0      600
Hamburg         900     700   1400    1500        600        0
```

```
# Test your solution with the following code
assert min_cost < 6000, "The cheapest route should be cheaper than 6000.
Try again, if you think your algorithm works correctly."
print(r"All tests passed, you have successfully implemented the route
planner!")
```

Conclusion

We've explored how randomness affects decision-making through:

- Explore vs. Exploit trade-offs
- Optimal stopping with random elements
- Caching a random subset of data
- Scheduling tasks with due dates
- Finding the cheapest route for a travel problem

Remember that randomness isn't always bad - sometimes it's the best strategy we have!

In the next tutorial, we'll dive deeper into some free-form tasks!