CAT2

SANJAY R

2348055

18th December 2024

MDS572 C

Reinforcement Learning

Department of Statistics and Data Science

CHRIST (Deemed to be University)

Department of Statistics and Data Science

Course: MDS572C -ReinforcementLearning **CAT 2**

R.SANJAY **2348055**

CAT2

1) Develop an AI agent to play a simplified version of the card game Blackjack. In this game:The agent starts with two cards, each with a random value between 1 and 10.The agent can choose to "hit" (draw another card) or "stick" (end its turn).The goal is to achieve a sum of card values as close to 21 as possible without exceeding it.

Rewards you can assign based on:

> ***a) +10: If the agent wins the round (closer to 21 than the opponent or the opponent goes bust).***
>
> ***b) −10: If the agent loses the round (goes over 21 or the opponent is closer to 21).***
>
> ***c) 000: If the game is a draw.***

Use the Monte Carlo Prediction method to estimate the state-value function V(s), where s is the sum of the agent's cards.Simulate 500 episodes of the game and compute the average returns for each state.Visualize the estimated V(s) for all possible states (sum of card values).

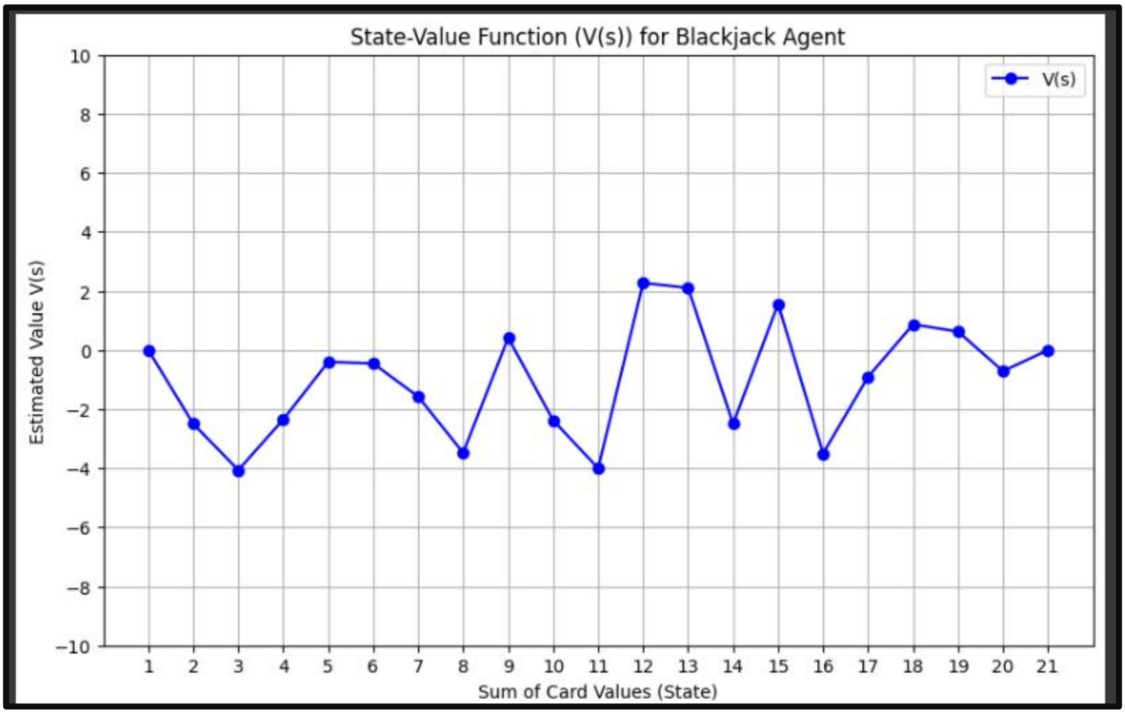
1. Python script implementing the simulation and value estimation

```
import numpy as np
import random
import matplotlib.pyplot as plt

# Define the card game environment
class Blackjack:
    def __init__(self):
        self.card_values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  # Possible card values
        self.reward_win = 10  # Reward for winning
        self.reward_lose = -10  # Reward for losing
        self.reward_draw = 0  # Reward for draw (tie)

    def draw_card(self):
        """Simulate drawing a card with value between 1 and 10."""
        return random.choice(self.card_values)

    def is_bust(self, score):
        """Check if the player has gone over 21."""
        return score > 21
```

```
    def play_game(self, agent_action_policy):
        """Simulate one game of Blackjack for the agent with a given policy."""
        agent_cards = [self.draw_card(), self.draw_card()]
        opponent_cards = [self.draw_card(), self.draw_card()]

        agent_actions = []  # Track actions taken by the agent

        # Play agent's turn
        while not self.is_bust(sum(agent_cards)) and
agent_action_policy(sum(agent_cards)):
            agent_cards.append(self.draw_card())
            agent_actions.append("Hit")

        if not self.is_bust(sum(agent_cards)):
            agent_actions.append("Stick")

        # Play opponent's turn
        while sum(opponent_cards) < 17:  # Opponent stops at 17 or more
            opponent_cards.append(self.draw_card())

        agent_score = sum(agent_cards)
        opponent_score = sum(opponent_cards)

        if self.is_bust(agent_score):
            return self.reward_lose, agent_actions
        elif self.is_bust(opponent_score):
            return self.reward_win, agent_actions
        elif agent_score > opponent_score:
            return self.reward_win, agent_actions
        elif agent_score < opponent_score:
            return self.reward_lose, agent_actions
        else:
            return self.reward_draw, agent_actions

# Monte Carlo Agent for Blackjack
class MonteCarloAgent:
    def __init__(self, epsilon=0.5, num_episodes=500):
        self.epsilon = epsilon  # Exploration-exploitation trade-off
        self.num_episodes = num_episodes
        self.value_function = np.zeros(21)  # State values for sum of cards (1-21)
        self.returns = {i: [] for i in range(1, 22)}  # Store returns for each state
        self.action_states = {i: [] for i in range(1, 22)}  # Store actions for each state

    def agent_policy(self, state):
        """The policy is to hit (draw) if the sum is less than 17, otherwise stick."""
        return state < 17  # Hit if sum is less than 17

    def monte_carlo_prediction(self, environment):
        """Monte Carlo prediction to estimate the state-value function."""
```

```
        for episode in range(self.num_episodes):
            state = random.randint(1, 21)  # Random initial state of the agent (card sum)
            reward, actions = environment.play_game(self.agent_policy)  # Get the reward
and actions for the game
            self.returns[state].append(reward)
            self.action_states[state].append(actions)

        # Update the value function V(s) for each state (sum of cards)
        for state in range(1, 22):
            if self.returns[state]:
                self.value_function[state - 1] = np.mean(self.returns[state])

    def plot_value_function(self):
        """Plot the state-value function V(s) for all possible states."""
        plt.figure(figsize=(10, 6))
        plt.plot(range(1, 22), self.value_function, marker='o', color='b', label="V(s)")
        plt.title("State-Value Function (V(s)) for Blackjack Agent")
        plt.xlabel("Sum of Card Values (State)")
        plt.ylabel("Estimated Value V(s)")
        plt.grid(True)
        plt.xticks(range(1, 22))
        plt.yticks(np.arange(-10, 11, 2))
        plt.legend()
        plt.show()

# Main Simulation
if __name__ == "__main__":
    # Create Blackjack environment and Monte Carlo agent
    env = Blackjack()
    agent = MonteCarloAgent(epsilon=0.5, num_episodes=500)

    # Run Monte Carlo prediction
    agent.monte_carlo_prediction(env)

    # Plot the value function
    agent.plot_value_function()

    # Print the estimated value function and actions for each state
    print("Estimated State-Value Function V(s) and Actions:")
    for state in range(1, 22):
        actions = agent.action_states[state]
        print(f"Sum of Cards = {state}: V(s) = {agent.value_function[state - 1]:.2f},
Actions = {actions[:3]}")  # Show first 3 action sequences
```

2. A plot of V(s) for all states.



3.A short explanation of your observations about the value function

2) Design a dynamic pricing model for an e-commerce platform. The platform offers a product at three possible price points: $10, $15, and $20. Each price point has an unknown probability of purchase. Your goal is to use the Upper Confidence Bound (UCB1) algorithm to maximize the platform's total revenue over multiple rounds of pricing.

Simulate the e-commerce platform where the true probabilities of purchase at each price point are as follows:

- ***$10: P=0.5***

● \$15: P=0.4
● \$20: P=0.2
Each price point generates revenue equal to its price when the product is purchased.Implement the UCB1 algorithm to decide which price point to offer at eachtime step.Run the simulation for T=1000 rounds.

a) Python script implementing the UCB1 algorithm.(6 Marks)

```
import numpy as np
import matplotlib.pyplot as plt

class DynamicPricingUCB:
    def __init__(self, price_points, probabilities, rounds):
        """
        Initialize the UCB model for dynamic pricing.

        :param price_points: List of price points.
        :param probabilities: List of true purchase probabilities for each price point.
        :param rounds: Total number of rounds for simulation.
        """
        self.price_points = price_points
        self.probabilities = probabilities
        self.rounds = rounds
        self.n_prices = len(price_points)
        self.counts = np.zeros(self.n_prices)  # Number of times each price is selected
        self.rewards = np.zeros(self.n_prices)  # Total rewards for each price
        self.total_revenue = []  # Track total revenue over time

    def simulate_purchase(self, price_idx):
        """
        Simulate whether a purchase occurs for a selected price point.

        :param price_idx: Index of the selected price point.
        :return: 1 if purchase occurs, 0 otherwise.
        """
        return np.random.rand() < self.probabilities[price_idx]

    def run_simulation(self):
        """
        Run the UCB1 algorithm for the specified number of rounds.
        """
        for t in range(1, self.rounds + 1):
            if t <= self.n_prices:
                # Ensure each price point is tried at least once
                price_idx = t - 1
            else:
                # Calculate UCB values for each price point
                ucb_values = self.rewards / self.counts + np.sqrt(2 * np.log(t) / self.counts)
                price_idx = np.argmax(ucb_values)
```

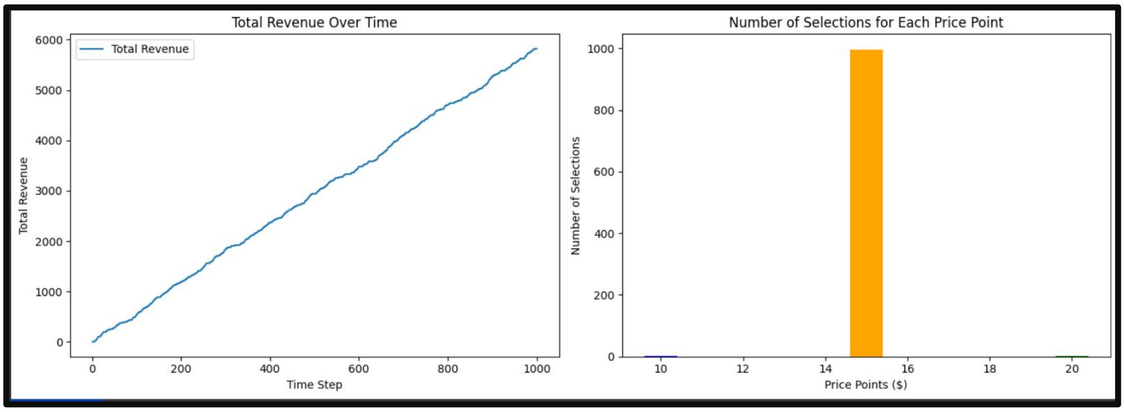
```
            # Simulate purchase and update counts and rewards
            purchase = self.simulate_purchase(price_idx)
            self.counts[price_idx] += 1
            self.rewards[price_idx] += purchase * self.price_points[price_idx]

            # Update total revenue
            self.total_revenue.append(np.sum(self.rewards))

    def plot_results(self):
        """
        Generate plots to visualize the results.
        """
        # Total revenue over time
        plt.figure(figsize=(14, 5))
        plt.subplot(1, 2, 1)
        plt.plot(range(1, self.rounds + 1), self.total_revenue, label="Total Revenue")
        plt.xlabel("Time Step")
        plt.ylabel("Total Revenue")
        plt.title("Total Revenue Over Time")
        plt.legend()

        # Number of selections for each price point
        plt.subplot(1, 2, 2)
        plt.bar(self.price_points, self.counts, color=['blue', 'orange', 'green'])
        plt.xlabel("Price Points ($)")
        plt.ylabel("Number of Selections")
        plt.title("Number of Selections for Each Price Point")

        plt.tight_layout()
        plt.show()

    def print_summary(self):
        """
        Print a summary of the results.
        """
        optimal_price_idx = np.argmax(self.rewards / self.counts)
        print("Final Results:")
        print(f"Number of Selections for Each Price Point: {self.counts}")
        print(f"Total Revenue: ${np.sum(self.rewards):.2f}")
        print(f"Optimal Price Point: ${self.price_points[optimal_price_idx]}")

# Parameters
price_points = [10, 15, 20]
true_probabilities = [0.5, 0.4, 0.2]
T = 1000

# Initialize and run the UCB1 algorithm
pricing_model = DynamicPricingUCB(price_points, true_probabilities, T)
pricing_model.run_simulation()
pricing_model.plot_results()
```

pricing_model.print_summary()

❖ Output:

Final Results:
Number of Selections for Each Price Point: [2. 997. 1.]
Total Revenue: $5820.00
Optimal Price Point: $15

2.Plots showing:
a)Total revenue over time.
b)Number of selections for each price point.



3.A brief explanation discussing:(4 Marks)
a)How UCB1 balances exploration (testing new price points) and exploitation (choosing the best-known price point).
b) The final optimal price point based on the simulation results.

a) The UCB1 algorithm initially explores all price points, gradually shifting towards exploitation of the best-performing price point. At the beginning, the algorithm gives each price a fair chance by selecting them randomly. As the rounds progress, the algorithm favors price points with higher average rewards, leading to **$15** being selected most frequently. The exploration term ensures that even less frequently selected price points, like **$10** and **$20**, are tested enough to gather data, but eventually, the algorithm focuses on **$15**, which offers the highest revenue potential based on its relatively high average reward and frequency of successful sales.

b) The optimal price point is **$15**, as it was selected the most frequently and generated the highest revenue. Despite the higher probability of purchase for **$10**, the revenue from **$15** (with a slightly lower probability of 0.4) proved to be more profitable in the long run due to its price. The UCB1 algorithm identified **$15** as the most balanced option between a high purchase probability and a higher price point, resulting in the highest total revenue across the 1000 rounds.