main

August 7, 2025

0.1 Laboratorio 04 - Métodos de Monte Carlo

Integrantes: - Ricardo Méndez - Melissa Pérez

Repositorio: https://github.com/MelissaPerez09/Lab04-CC3104

0.1.1 Task 01

1. ¿Cómo afecta la elección de la estrategia de exploración (exploring starts vs soft policy) a la precisión de la evaluación de políticas en los métodos de Monte Carlo?

    a. Considere la posibilidad de comparar el desempeño de las políticas evaluadas con y sin explorar los inicios o con diferentes niveles de exploración en políticas blandas.

    - De manera general, Exploring Starts ofrece evaluaciones más precisas pero necesita un control del entorno, mientras que las soft policies ofrecen una solución práctica y más aplicable, aunque con menor cobertura teórica.
    - Al usar Exploring Starts, se fuerza al agente a comenzar desde cualquier estado y acción al azar. Esto ayuda mucho porque garantiza que explore todo el espacio posible, lo que hace que la evaluación de la política sea más precisa.
    - Una soft policy, como una política -greedy, permite al agente elegir la mejor acción la mayoría del tiempo, pero de vez en cuando prueba otras acciones. Mantiene la exploración constante mientras aprende.

2. En el contexto del aprendizaje de Monte Carlo fuera de la póliza, ¿cómo afecta la razón de muestreo de importancia a la convergencia de la evaluación de políticas? Explore cómo la razón de muestreo de importancia afecta la estabilidad y la convergencia.

    - En un aprendizaje off-policy, se aprender sobre una política (objetivo) usando datos generados por otra (la de comportamiento). Para ajustar esto, se utiliza muestreo de importancia, que es una razón que indica qué tanto se debe corregir lo aprendido para que refleje la política objetivo.

3. ¿Cómo puede el uso de una soft policy influir en la eficacia del aprendizaje de políticas óptimas en comparación con las políticas deterministas en los métodos de Monte Carlo? Compare el desempeño y los resultados de aprendizaje de las políticas derivadas de estrategias épsilon-greedy con las derivadas de políticas deterministas.

    - Una soft policy como -greedy sigue explorando aunque ya tenga buenas acciones, porque siempre deja un espacio pequeño para probar nuevas cosas. Esto es bueno porque ayuda a descubrir mejores soluciones que quizá no se habían considerado.

- En cambio, una política determinista elige siempre la mejor acción conocida. Es más rápida en tomar decisiones, pero si esa "mejor acción" no es realmente la óptima global, se puede quedar atascada.

4. ¿Cuáles son los posibles beneficios y desventajas de utilizar métodos de Monte Carlo off-policy en comparación con los on-policy en términos de eficiencia de la muestra, costo computacional y velocidad de aprendizaje?

   - Los principales beneficios a considerar es que los algoritmos off-policy ofrecen una eficiencia de muestra porque puede usar cualquier experiencia recolectada, el costo computacional es más alto por el uso de muestreo de importancia, la velocidad de aprendizaje depende de si las políticas no son muy distintas. Por último, mencionar que su principal desventaja es que es más sensible y difícil de ajustar.
   - Por otro lado, con los algoritmos on-policy se tiene el caso contrario a lo mencionado con anterioridad, la eficiencia de muestra usa únicamente la que genera bajo su propia política, el costo computacional es más bajo porque los cálculos son más directos y la velocidad de aprendizaje es más estable pero depende de la exploración. Su principal desventaja es que es menos eficiente con los datos porque deben ser controlados.

0.1.2 Task 02

[306]:
```
import numpy as np
import random
```

[307]:
```
# definicion del entorno
class InventoryEnvironment:
    def __init__(self):
        self.products = ['product_A', 'product_B']
        self.max_stock = 10 # Pueden cambiar este número si gustan
        self.demand = {'product_A': [0, 1, 2], 'product_B': [0, 1, 2]}
        self.restock_cost = {'product_A': 5, 'product_B': 7}
        self.sell_price = {'product_A': 10, 'product_B': 15}
        self.state = None

    def reset(self):
        self.state = {product: random.randint(0, self.max_stock) for product in␣
 ↪self.products}
        return self.state

    def step(self, action):
        reward = 0
        for product in self.products:
            stock = self.state[product]
            restock = action[product]
            self.state[product] = min(self.max_stock, stock + restock)
            demand = random.choice(self.demand[product])
            sales = min(demand, self.state[product])
            self.state[product] -= sales
```

```
            reward += sales * self.sell_price[product] - restock * self.
 ↪restock_cost[product]
        return self.state, reward

# Init el ambiente
env = InventoryEnvironment()
```

[308]:
```
# generar episodios
def generate_episode(policy, days=30, start_state=None, start_action=None):
    episode = []

    state = env.reset()

    if start_state:
        env.state = start_state.copy()
        state = env.state

    for day in range(days):
        if day == 0 and start_action:
            action = start_action
        else:
            action = policy(state)

        if isinstance(action, tuple):
            action = policy(state)
            pass

        next_state, reward = env.step(action)
        episode.append((state.copy(), action.copy(), reward))
        state = next_state

    return episode
```

[309]:
```
# exploring starts
def random_action():
    return {
        'product_A': random.randint(0, env.max_stock),
        'product_B': random.randint(0, env.max_stock)
    }

def random_policy(state):
    return {
        'product_A': random.randint(0, 2),
        'product_B': random.randint(0, 2)
    }

def exploring_starts(num_episodes=500):
```

```
    episodes = []
    for _ in range(num_episodes):
        start_state = {
            'product_A': random.randint(0, env.max_stock),
            'product_B': random.randint(0, env.max_stock)
        }
        start_action = random_action()
        episode = generate_episode(policy=random_policy,␣
 ↪start_state=start_state, start_action=start_action)
        episodes.append(episode)
    return episodes
```

[ ]:
```
# soft policy
epsilon = 0.1

def greedy_policy(Q, state):
    state_key = (state['product_A'], state['product_B'])
    if state_key in Q:
        max_action = max(Q[state_key], key=Q[state_key].get)
        return {'product_A': max_action[0], 'product_B': max_action[1]}
    else:
        return random_policy(state)

def epsilon_greedy_policy(Q, epsilon):
    epsilon = epsilon
    def policy(state):
        if random.random() < epsilon:
            return random_policy(state)
        else:
            return greedy_policy(Q, state)
    return policy
```

[311]:
```
def off_policy_mc_control(num_episodes=500, days=30, epsilon=0.1):
    Q = {}
    returns_count = {}
    rewards_per_day = [0] *  days

    for _ in range(num_episodes):
        policy = epsilon_greedy_policy(Q, epsilon)  # generar la política con Q␣
 ↪actualizado
        episode = generate_episode(policy=policy, days=days)

        G = 0
        for t in reversed(range(len(episode))):
            state, action, reward = episode[t]
            G += reward
            rewards_per_day[t] += reward
```

```
            state_key = (state['product_A'], state['product_B'])
            action_key = (action['product_A'], action['product_B'])

            if state_key not in Q:
                Q[state_key] = {}
            if action_key not in Q[state_key]:
                Q[state_key][action_key] = 0
                returns_count[(state_key, action_key)] = 0

            returns_count[(state_key, action_key)] += 1
            alpha = 1 / returns_count[(state_key, action_key)]
            Q[state_key][action_key] += alpha * (G - Q[state_key][action_key])

    return Q, rewards_per_day
```

[312]:
```
# Exploring starts
episodes = exploring_starts(num_episodes=500)

# Rewards per day
rewards_per_day = [sum(reward for _, _, reward in episode) for episode in␣
 ↪episodes]
avg_reward = sum(rewards_per_day) / len(rewards_per_day)
print(f"Average reward per day (exploring starts): {avg_reward}")
```

```
Average reward per day (exploring starts): 317.41
```

[313]:
```
# Soft Policy
Q = {}
epsilon_greedy = epsilon_greedy_policy(Q, epsilon)
episodes = []
for _ in range(500):
    episodes.append(generate_episode(policy=epsilon_greedy, days=30))
# Rewards per day
rewards_per_day = [sum(reward for _, _, reward in episode) for episode in␣
 ↪episodes]
avg_reward = sum(rewards_per_day) / len(rewards_per_day)
print(f"Average reward per day (exploring starts): {avg_reward}")
```

```
Average reward per day (exploring starts): 339.808
```

[314]:
```
# Off-policy
Q, rewards = off_policy_mc_control(num_episodes=500, days=30)

# Rewards per day
avg_reward = sum(rewards) / len(rewards)
print(f"Average reward per day (off-policy): {avg_reward}")
```

```
Average reward per day (off-policy): 5987.066666666667
```

0.1.3 Preguntas

1. ¿Cuál es el valor estimado de mantener diferentes niveles de existencias para cada producto? El valor estimado de mantener niveles distintos de stock es el encontrar un balance entre evitar pérdidas de ventas (no tener producto disponible) y minimizar costos de restock.
2. ¿Cómo afecta el valor epsilon en la política blanda al rendimiento? Afecta en el momento de la exploración. Depende del ambiente que se trate si esto puede ser positivo, negativo o no tener algún efecto, ya que está la posibilidad de que al explorar mucho se encuentren mejores políticas, se encuentren peores, o no sume ni reste.
3. ¿Cuál es el impacto de utilizar el aprendizaje fuera de la política en comparación con el aprendizaje dentro de la política? Usar un aprendizaje off-policy permite al agente aprender una mejor política mientras explora, lo que aprovecha mejor los datos. Es menos prudente que un agente on-policy ya que solo busca maximizar su reward sin limites establecidos. En el ambiente en uso (inventario de ventas), off-policy puede ser una mejor decisión ya que encontrará más rápido cómo maximizar las ganancias netas.