Monte Carlo Methods

February 15, 2023

1 5.1 Monte Carlo Prediction

Suppose we wish to estimate $v_\pi(s)$, the value of a state s under policy π, given a set of episodes obtained by following π and passing through s. Each occurrence of state s in an episode is called a *visit* to s. Of course, s may be visited multiple times in the same episode; let us call the first time it is visited in an episode the *first-visit* to s. The *first-vist* MC method estimates $v_\pi(s)$ as the average of the returns following first visits to s, whereas the *every-visit* MC method averages the returns following all visits to s

First-visit MC prediction, for estimating $V \approx v_\pi$ Input: a policy π to be evaluated Initialize: $V(s) \in R$, arbitrarily, for all $s \in S$ $Returns(s) \leftarrow$ an empty list, for all $s \in S$ Loop forever (for each episode): Generate an episode following π: $(S_0, A_0, R_1, S_1), (S_1, A_1, R_2, S_2), ..., (S_{T-1}, A_{T-1}, S_T, R_T)$ $G \leftarrow 0$ Loop for each step of episode, $t = T-1, T-2, ..., 0$: $G \leftarrow \gamma G + R_{t+1}$ If S_t does not appear in $S_0, S_1, ..., S_{t-1}$: Append G to $Returns(S_t)$ $V(S_t) \leftarrow average(Returns(S_t))$

Ex 5.1 Blackjack

[1]:
```python
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import clear_output

def hit():
    return np.random.choice([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10])

def get_sums(cards, index, cur, sums):
    if index == len(cards):
        sums.append(cur)
        return
    if cards[index] == 1:
        get_sums(cards, index + 1, cur + 1, sums)
        get_sums(cards, index + 1, cur + 11, sums)
    else:
        get_sums(cards, index + 1, cur + cards[index], sums)

def get_valid_sum(sums):
    sums.sort()
    if sums[0] > 21:
```

```
        return sums[0]
    for i in range(len(sums) - 1, -1, -1):
        if sums[i] <= 21:
            return sums[i]

def get_sum(cards):
    sums = list()
    get_sums(cards, 0, 0, sums)
    return get_valid_sum(sums)

def act(s, a):
    player_cards = [1, s[0] - 11] if s[2] else [s[0]]
    dealer_cards = [s[1]]
    if a == 'hit':
        player_cards.append(hit())
        player_sum = get_sum(player_cards)
        if player_sum > 21:
            return (None, -1)
        elif player_sum == 21:
            dealer_sum = get_sum(dealer_cards)
            while dealer_sum < 17:
                dealer_cards.append(hit())
                dealer_sum = get_sum(dealer_cards)
            if dealer_sum > 21:
                return (None, 1)
            elif dealer_sum == player_sum:
                return (None, 0)
            else:
                return (None, 1)
        else:
            return ((player_sum, s[1], sum(player_cards) != player_sum), 0)
    elif a == 'stick':
        player_sum = get_sum(player_cards)
        dealer_sum = get_sum(dealer_cards)
        while dealer_sum < 17:
            dealer_cards.append(hit())
            dealer_sum = get_sum(dealer_cards)
        if dealer_sum > 21:
            return (None, 1)
        elif dealer_sum > player_sum:
            return (None, -1)
        elif dealer_sum == player_sum:
            return (None, 0)
        else:
            return (None, 1)

def get_episode(s, policy):
```

```
    a = policy[s]
    s_, r = act(s, a)
    episode = [(s_, r, s, a)]
    while s_ is not None:
        s = s_
        a = policy[s]
        s_, r = act(s, a)
        episode.append((s_, r, s, a))
    return episode
```

[2]:
```
states = [(i, j, k) for i in range(12, 22, 1) for j in range(1, 11, 1) for k in␣
 ↪[True, False]]
actions = ['hit', 'stick']
policy = {s: 'hit' if s[0] < 20 else 'stick' for s in states}

V = {s: [0, 0] for s in states}

for i in range(500000):
    if (i + 1) % 1000 == 0:
        clear_output(wait=True)
        print(f'{(i + 1) / 500000}')
    s_0 = states[np.random.choice(len(states))]
    episode = get_episode(s_0, policy)
    V[s_0][0] += episode[-1][1]
    V[s_0][1] += 1
```

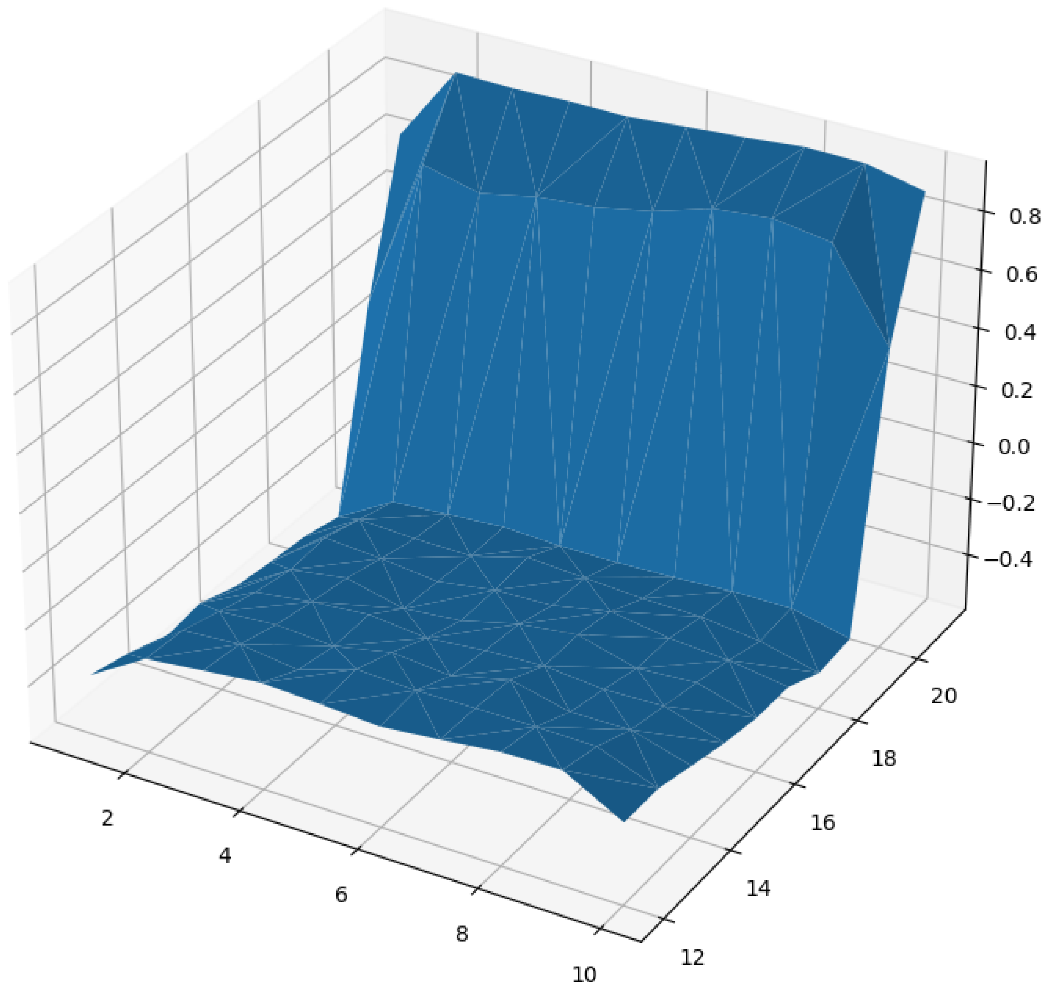
```
1.0
```

[3]:
```
fig = plt.figure(figsize=(16, 9))
ax = plt.axes(projection="3d")

X = [s[1] for s in states if s[2]]
Y = [s[0] for s in states if s[2]]
Z = [V[s][0] / V[s][1] for s in states if s[2]]

ax.plot_trisurf(X, Y, Z)

plt.show()
```



Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$ Initialize: $\pi(s) \in A(s)$ (arbitrarily), for all $s \in S$ $Q(s,a) \in R$ (arbitrarily), for all $s \in S, a \in A(s)$ $Returns(s,a) \leftarrow$ empty list, for all $s \in S, a \in A(s)$ Loop forever (for each episode): Choose $S_0 \in S, A_0 \in A(S_0)$ randomly such that all pairs have probability > 0 Generate an episode from S_0, A_0, following π: $(S_0, A_0, R_1, S_1), (S_1, A_1, R_2, S_2), ..., (S_{T-1}, A_{T-1}, R_T, S_T)$ $G \leftarrow 0$ Loop for each step of episode, $t = T-1, T-2, ..., 0$: $G \leftarrow \gamma G + R_{t+1}$ Unless the pair S_t, A_t appears in $(S_0, A_0, R_1, S_1), (S_1, A_1, R_2, S_2), ..., (S_{T-1}, A_{T-1}, R_T, S_T)$: Append G to $Returns(S_t, A_t)$ $Q(S_t, A_t) \leftarrow average(Returns(S_t, A_t))$ $\pi(S_t) \leftarrow argmax_a Q(S_t, a)$

[ ]:
```
def get_episode(s, a, policy):
    s_, r = act(s, a)
    episode = [(s_, r, s, a)]
    while s_ is not None:
```

```
        s = s_
        a = policy[s]
        s_, r = act(s, a)
        episode.append((s_, r, s, a))
    return episode
```

[ ]:
```
states = [(i, j, k) for i in range(12, 22, 1) for j in range(1, 11, 1) for k in␣
 ↪[True, False]]
actions = ['hit', 'stick']
policy = {s: 'hit' if s[0] < 20 else 'stick' for s in states}
Q = {(s, a): [0, 0] for s in states for a in actions}

count = 0
while True:
    if count != 0 and count % 1000000 == 0:
        fig = plt.figure(figsize=(16, 9))
        ax = plt.axes(projection="3d")
        X = [s[1] for s, a in Q if s[2]]
        Y = [s[0] for s, a in Q if s[2]]
        Z = [Q[(s, a)][0] / Q[(s, a)][1] for s, a in Q if s[2]]
        ax.plot_trisurf(X, Y, Z)
        plt.savefig(f'./blackjack/{count}.png')
        plt.close()
    s_0 = states[np.random.choice(len(states))]
    a_0 = actions[np.random.choice(len(actions))]
    episode = get_episode(s_0, a_0, policy)
    Q[(s_0, a_0)][0] += episode[-1][1]
    Q[(s_0, a_0)][1] += 1
    try:
        policy[s_0] = 'hit' if np.argmax([Q[(s_0, 'hit')][0] / Q[(s_0,␣
 ↪'hit')][1], Q[(s_0, 'stick')][0] / Q[(s_0, 'stick')][1]]) == 0 else 'stick'
    except:
        pass
    count += 1
```

[ ]:
```
# fig = plt.figure(figsize=(16, 9))
# ax = plt.axes(projection="3d")

# X = [s[1] for s, a in Q if s[2]]
# Y = [s[0] for s, a in Q if s[2]]
# Z = [Q[(s, a)][0] / Q[(s, a)][1] for s, a in Q if s[2]]

# ax.plot_trisurf(X, Y, Z)

# plt.show()
```

2 Monte Carlo Control without Exploring Starts

On-policy methods attempt to evaluate or improve the policy that is used to make decisions, whereas off-policy methods evaluate or improve a policy different from that used to generate the data. **Monte Carlo ES** is an example of an on-policy method.

On-policy first-visit MC control (for ϵ-soft policies), estimates $\pi \approx \pi_*$ Algorithm parameter: small $\epsilon > 0$ Initialize: $\pi \leftarrow$an arbitrary ϵ-soft policy $Q(s,a) \in R$(arbitaraily), for all $s \in S, a \in A(s)$ $Returns(s,a) \leftarrow$empty list, for for all $s \in S, a \in A(s)$ Repeat forever (for each episode): Generate an episode follwing π: $(S_1, R_1, S_0, A_0), (S_2, R_2, S_1, A_1), ..., (S_T, R_T, S_{T-1}, A_{T-1})$ $G \leftarrow 0$ Loop for each step of episode, $t = T-1, T-2, ..., 0$: $G \leftarrow \gamma G + R_{t+1}$ Unless the pair S_t, A_t appears in $(S_1, R_1, S_0, A_0), (S_2, R_2, S_1, A_1), ..., (S_T, R_T, S_{T-1}, A_{T-1})$: Append G to $Returns(S_t, A_t)$ $A^* \leftarrow argmax_a Q(S_t, a)$ (with ties broken arbitrarily) For all $a \in A(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \epsilon + \epsilon/|A(S_t)| & if a = A^* \\ \epsilon/|A(S_t)| & if a \neq A^* \end{cases}$$

Now we only achieve the best policy among the ϵ-soft policies, but on the other hand, we have eliminated the assumption of exploring starts.

```
[ ]:
```