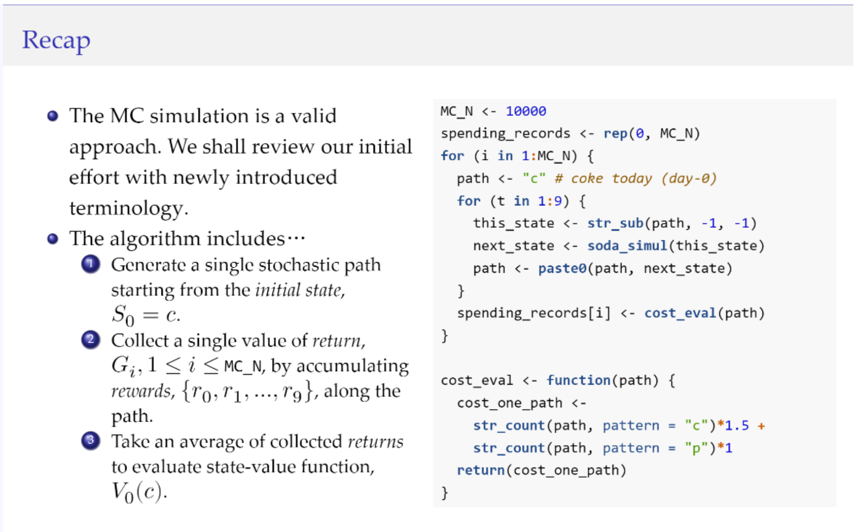
D1_Solution

Reinforcement Learning Study

2021-01-26

차 례

Recap (P. 10) 손민상




```python
import numpy as np

def soda_simul(this_state):
    n=np.random.random()

    if this_state=='c':
        if n<=0.7:
            next_state='c'
        else:
            next_state='p'
    else:
        if n<=0.5:
            next_state='c'
        else:
            next_state='p'

    return next_state

def cost_eval(path):
    cost_one_path=path.count('c')*1.5+path.count('p')*1
    return cost_one_path

MC_N=10000
spending_records=np.zeros((MC_N,))

for i in range(MC_N):
    path='c' # coke today (day-0)

    for t in range(9):
        this_state=path[-1]
        next_state=soda_simul(this_state)
        path+=next_state

    spending_records[i]=cost_eval(path)
```

```python
print(spending_records)
```

```
## [13.  14.  13.  ... 13.5 14.5 13.5]
```

Recap (P. 10) R code 박재민

```r
MC_N <- 10000
spending_records <- rep(0, MC_N)
for (i in 1:MC_N) {
  path <- "c" # coke today (day-0)
  for (t in 1:9) {
    this_state <- str_sub(path, -1, -1)
    next_state <- soda_simul(this_state)
    path <- paste0(path, next_state)
  }
  spending_records[i] <- cost_eval(path)
}
cost_eval <- function(path) {
  cost_one_path <-
    str_count(path, pattern = "c")*1.5 +
    str_count(path, pattern = "p")*1
  return(cost_one_path)
}
```

MC simulation for estimating state-value function (P. 11) 백종민

```python
def state_value_function(num_episode):
  episode_i = 0
  cum_sum_G_i  = 0
  # number of episode(iteration)
  while episode_i < num_episode:
    path = 'c' # initial state
    # generate stochastic path(episode)
    for t in range(1,10):
      this_state = path[-1]
      next_state = soda_simul(this_state)
      path += next_state
    # print(path)
    # calculate sum of rewards
    G_i = cost_eval(path)
    cum_sum_G_i += G_i
    episode_i += 1
  V_t = cum_sum_G_i/num_episode
  return V_t
```

MC simulation for estimating *state-value function*

- Formally, for a *finite-horizon MRP*, the following is MC simulation for estimating *state-value function*.

```
# MC evaluation for state-value function
# with state s, time 0, reward r, time-horizon H
episode_i <- 0
cum_sum_G_i <- 0
while episode_i < num_episode
  Generate an stochastic path starting from state s and time 0.
  Calculate return G_i <- sum of rewards from time 0 to time H-1.
  cum_sum_G_i <- cum_sum_G_i + G_i
  episode_i <- episode_i + 1
State-value-fn V_t(s) <- cum_sum_G_i/num_episode
return V_t(s)
```

```
state_value_function(10000)
```

```
## 13.33665
```

MC simulation for estimating state-value function (P. 11) R code 박재민

```
episode_i <- 0
cum_sum_G_i <- 0
while episode_i < num_episode
  #Generate an stochastic path starting from state s and time 0.
  #Calculate return G_i <- sum of rewards from time 0 to time H-1.
  cum_sum_G_i <- cum_sum_G_i + G_i
  episode_i <- episode_i + 1
State-value-fn V_t(s) -- cum_sum_G_i/num_episode
return V_t(s)
```

For general t, Exercise (P. 17) 손민상

For general t,

$$
\begin{aligned}
V_t(s) &= \mathbb{E}[G_t|S_t = t] \\
&= \mathbb{E}[r_t + r_{t+1} + r_{t+2} + \cdots + r_\infty|S_t = s] \\
&= \mathbb{E}[r_t|S_t] + \mathbb{E}[r_{t+1} + r_{t+2} + \cdots + r_\infty|S_t = s] \\
&= R(s) + \mathbb{E}[r_{t+1} + r_{t+2} + \cdots + r_\infty|S_t = s] \\
&= R(s) + \mathbb{E}[G_{t+1}|S_t = s, S_{t+1} = s'] \\
&= R(s) + \mathbb{E}[G_{t+1}|S_{t+1} = s'](\because Markov\ property) \\
&= R(s) + \sum_{s \in s'} P_{ss'} V_{t+1}(s')
\end{aligned}
$$

P. 20 권도윤

```
import numpy as np
P = np.array([0.7,0.3,0.5,0.5]).reshape(2,2)
R = np.array([1.5,1.0]).reshape(2,1)
H = 10
v_t1 = np.array([0,0]).reshape(2,1)
t = H-1 # time-horizon

while (t>=0):
    v_t = R+ np.dot(P,v_t1)
    t = t-1
    v_t1 = v_t
print(v_t)
```

```
P <- array(c(0.7,0.5,0.3,0.5), dim=c(2,2))
R <- array(c(1.5,1.0), dim=c(2,1))
H <- 10 # time-horizon
v_t1 <- array(c(0,0), dim=c(2,1)) # v_{t+1}

t <- H-1
while (t >= 0) {
  v_t  <- R + P %*% v_t1
  t    <- t-1
  v_t1 <- v_t
}
v_t
```

```
##         [,1]
## [1,] 13.35937
## [2,] 12.73438
```

- Thus, we have the following *state-value function*.
    - $V_0(c) = 13.359375$
    - $V_0(p) = 12.734375$

```
## [[13.35937498]
##  [12.73437504]]
```

Page 21 백종민

```
#Backward induction for state-value function
#with transition prob mat P , reward vector R, time-horizon H, state-value vector v

import numpy as np
P = np.array([0.7,0.3,0.5,0.5]).reshape(2,2)
R = np.array([1.5,1.0]).reshape(2,1)
def state_value_function(P,R,H):
    t = H-1
    globals()['V_{}'.format(H)] = np.array([0,0]).reshape(2,1)
    while t >= 0:
        globals()['V_{}'.format(t)] = R+np.dot(P,globals()['V_{}'.format(t+1)])
        t = t-1
    return globals()['V_{}'.format(t+1)]
state_value_function(P,R,10)
```

```
## array([[13.35937498],
##        [12.73437504]])
```

- Formally, for a *finite-horizon MRP*, the following is *backward induction* for estimating *state-value function*.

```
# Backward induction for state-value function
# with transition prob mat P, reward vector R, time-horizon H, state-value vector v_{}
1: v_H <- zero-column vector
2: t <- H-1
3: while t >= 0
4:   v_t <- R + P*v_{t+1}
5:   t <- t-1
9: return v_t # this is v_0(s) for all s, because t=0 at this point
```

```
"D1_Solution"
```