Lecture C2. Discrete Time Markov Chain2 Solution

Reinforcement Learning Study

2021-01-10

차 례

C2의 파이썬 구현은 모두가 비슷하고, 올바른 답을 제출하였습니다. 다만 솔루션 선정 기준은 lecture note와 가장 비슷하게 구현한 학생의 풀이를 가져왔습니다.

Method 1 -eigen-decomposition p.12

in r

```
P <- array(c(0.7, 0.5, 0.3, 0.5),dim =c(2,2))
eigen(t(P)) #eigen-decomposition for P^t
```

```
## eigen() decomposition
## $values
## [1] 1.0 0.2
##
## $vectors
##           [,1]       [,2]
## [1,] 0.8574929 -0.7071068
## [2,] 0.5144958  0.7071068
```

```
x_1 <-eigen(t(P))$vectors[,1]
x_1
```

```
## [1] 0.8574929 0.5144958
```

```
v<-x_1/sum(x_1)
v
```

```
## [1] 0.625 0.375
```

in Python (김봉석)

```
import numpy as np
P = np.array([[0.7, 0.3],[0.5, 0.5]])
egien_value, egien_vector = np.linalg.eig(P.T) #eigen-decomposition for P^t
print("egien_value :\n",egien_value)
```

```
## egien_value :
##  [1.  0.2]
```

```
print("egien_vector :\n",egien_vector)
```

```
## egien_vector :
##  [[ 0.85749293 -0.70710678]
##  [ 0.51449576  0.70710678]]
```

```
x_1=egien_vector[:,0]
print(x_1)
```

```
## [0.85749293 0.51449576]
```

```
v=x_1/np.sum(x_1)
print(v)
```

```
## [0.625 0.375]
```

종합의견

P를 저장할때 사용한 메트릭스 자료형의 차이를 보였습니다. 이후 과정은 np.linalg.eig 를 사용하여 동일합니다.

저장 방법은 다음과 같이 3가지 방식으로 나뉩니다.

```
import numpy as np
P=np.matrix([[0.7,0.3],[0.5,0.5]]) # 방법1 np.matrix
P=np.array([[0.7, 0.3],[0.5, 0.5]]) # 방법2 np.array 2차원 배열
P=np.array([0.7,0.3,0.5,0.5]).reshape(2,2) # 방법3 np.array  1차원 배열 reshape
```

Method 2- system of linear equation p.15

in R

```
P<- array(c(0.7, 0.5, 0.3, 0.5), dim =c(2,2))
n<- nrow(P)
I<-diag(n)
A<-cbind(P-I,rep(1,n))
b<-array(c(rep(0,n),1),dim =c(1, n+1))
A
```

```
##      [,1] [,2] [,3]
## [1,] -0.3  0.3    1
## [2,]  0.5 -0.5    1
```

```
b
```

```
##      [,1] [,2] [,3]
## [1,]    0    0    1
```

```
v <- solve(A %*%t(A),A%*%t(b))
v
```

```
##       [,1]
## [1,] 0.625
## [2,] 0.375
```

in Python (손민상)

```
import numpy as np
P=np.array([[0.7, 0.3],[0.5, 0.5]])
n=len(P) # n=|S|
I=np.identity(n) # identity matrix
A=np.c_[P-I,np.repeat(1,n)] ## np._c_ equalt to cbind in r
b=np.append(np.repeat(0,n), np.array(1))
print(A)
```

```
## [[-0.3  0.3  1. ]
##  [ 0.5 -0.5  1. ]]
```

```
print(b)
```

```
## [0 0 1]
```

```
v=np.linalg.solve(np.dot(A,A.T),np.dot(A,b.T))
print(v)
```

```
## [0.625 0.375]
```

in Python (박재민)

```
P = np.array([[0.7,0.3],[0.5,0.5]])
n = P.shape[0] #nrow
I = np.identity(n)
rep=np.array([[1],[1]])
A = np.hstack([P-I, rep])
b=np.array([0,0,1])
sol = np.linalg.solve(np.dot(A,A.T),np.dot(A,b.T))
print(A)
```

```
## [[-0.3  0.3  1. ]
##  [ 0.5 -0.5  1. ]]
```

```
print(b)
```

```
## [0 0 1]
```

```
print(sol)
```

```
## [0.625 0.375]
```

종합의견

마찬가지로 대다수가 올바른 답이며, lecture note와 가장 비슷하게 구현한 학생의 풀이를 가져왔습니다.

1. 마찬가지로 위와 동일하게 P를 저장하는 방법의 차이를 보입니다.

   ex) np.array(2차원), np.array.reshape(1차원 reshape), np.matrix

2. Identity matrix를 만들때 사용하는 함수의 차이를 보였습니다

   ex) np.identity , np.eys

3. A를 만들때 사용하는 함수의 차이를 보였습니다

   ex) np._c(대다수) np.hstack(1명) np.column_stack(1명)

위의 방법의 조합에 따라 코드는 상이하며 결과는 모두 동일합니다.

```python
import numpy as np

a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# 세로로 붙여서 2차원 배열을 만듬, 사용방법은 다르지만 동일 결과 반환
print(np.c_[a,b])
```

```
## [[1 4]
##  [2 5]
##  [3 6]]
```

```python
print(np.column_stack([a,b]))
```

```
## [[1 4]
##  [2 5]
##  [3 6]]
```

Limiting Probability -Motivation p.17

in R

```
library(expm)
```

```
## Loading required package: Matrix

##

## Attaching package: 'expm'

## The following object is masked from 'package:Matrix':
##
##     expm
```

```
P <- array(c(0.7,0.5,0.3,0.5), dim = c(2,2))
P %*% P
```

```
##      [,1] [,2]
## [1,] 0.64 0.36
## [2,] 0.60 0.40
```

```
P %^% 3
```

```
##       [,1]  [,2]
## [1,] 0.628 0.372
## [2,] 0.620 0.380
```

```
P %^% 4
```

```
##        [,1]   [,2]
## [1,] 0.6256 0.3744
## [2,] 0.6240 0.3760
```

```
P %^% 20
```

```
##       [,1]  [,2]
## [1,] 0.625 0.375
## [2,] 0.625 0.375
```

in Python (백종민)

```
from numpy.linalg import matrix_power
P = np.array([0.7,0.5,0.3,0.5]).reshape(2,2).T
print(P)
```

```
## [[0.7 0.3]
##  [0.5 0.5]]
```

```
print(matrix_power(P,2))
```

```
## [[0.64 0.36]
##  [0.6  0.4 ]]
```

```
print(matrix_power(P,3))
```

```
## [[0.628 0.372]
##  [0.62  0.38 ]]
```

```
print(matrix_power(P,4))
```

```
## [[0.6256 0.3744]
##  [0.624  0.376 ]]
```

```
print(matrix_power(P,20))
```

```
## [[0.625 0.375]
##  [0.625 0.375]]
```

in Python(이성호)

```
from sympy import *
p = Matrix([[0.7,0.3],[0.5,0.5]])
```

```
np.dot(P,P) #matrix multiplication
```

```
## array([[0.64, 0.36],
##        [0.6 , 0.4 ]])
```

```
p**3
```

```
## Matrix([
## [0.628, 0.372],
## [ 0.62,  0.38]])
```

```
p**4
```

```
## Matrix([
## [0.6256, 0.3744],
## [ 0.624,  0.376]])
```

```
p**20
```

```
## Matrix([
## [0.625000000000003, 0.374999999999996],
## [0.624999999999993, 0.375000000000006]])
```

종합의견

마찬가지로 P를 저장하는 방법에 따라

1. np.array를 사용한경우 matrixpower함수를 사용
2. np. matrix 사용한경우 **연산자를 사용했습니다.

Limiting Probability- p.19

The limiting distribution may or may not exit. For example

in r

```r
library(expm)
P<-array(c(0,1,0,1,0), dim=c(2,2))
P
```

```
##      [,1] [,2]
## [1,]    0    0
## [2,]    1    1
```

```r
P %^% 2
```

```
##      [,1] [,2]
## [1,]    0    0
## [2,]    1    1
```

```r
P %^% 3
```

```
##      [,1] [,2]
## [1,]    0    0
## [2,]    1    1
```

in Python(김봉석)

```
from numpy.linalg import matrix_power

P=np.array([[0,1],[1,0]])
print(P)
```

```
## [[0 1]
##  [1 0]]
```

```
print(matrix_power(P,2))
```

```
## [[1 0]
##  [0 1]]
```

```
print(matrix_power(P,3))
```

```
## [[0 1]
##  [1 0]]
```

in Python(권태현)

```
import numpy as np
matrix = np.matrix
p = matrix([[0,1],[1,0]])
print(p**2)
```

```
## [[1 0]
##  [0 1]]
```

```
print(p**3)
```

```
## [[0 1]
##  [1 0]]
```

종합의견

마찬가지로 P를 저장하는 방법에 따라

1. np.array를 사용한경우 matrixpower함수를 사용
2. np. matrix 사용한경우 ** 연산자를 사용했습니다.

```r
"Done, Lecture C2. Discrete Time Markov Chain2 "
```

```
## [1] "Done, Lecture C2. Discrete Time Markov Chain2 "
```