TP CHAINE DE MARKOV (DISTRIBUTION STATIONNAIRE, FONCTION VALEUR ET POLITIQUE OPTIMALE)

July 20, 2023

Université d'Abomey-Calavi
The Abdus Salam International Center for Theoretical Physics
INSTITUT DE MATHEMATIQUES ET DE SCIENCES PHYSIQUES

Filière: Recherche Opérationnelle

Niveau: MASTER II

PROJET

Optimisation Stochastique

##

$Présenté par : $

####

METSANOU TSAFACK Dexter Stephane

####

SOSSOU EDOU Rose

0.1 TP1

L'objectif de ce TP est de pratiquer la modélisation et la résolution du problème.

Pour une population de clients de grande importance, nous avons 4 concurrents pour un abonnement de média télé (pay-TV, for example) avec une distribution relativement stable mais changeante de [0.55, 0.2, 0.1, 0.15], le dernier groupe de 15% n'ayant aucune souscription particulière, et préfèrent le téléchargement gratuit à la demande.

Le second plus grand concurrent (b) vient de lancer un nouveau produit premium et le dominant suspecte qu'il restructure les parts de marché. Ils souhaitent savoir quel sera l'impact sur leurs parts de marché s'ils ne font rien. Il souhaite aussi comprendre la dynamique leurs pertes de clients et comment cela se rapport à leur part de marché

Une étude du marché produit les probabilités estimées suivantes qu'un client change de fournisseur de service: P = [0.9262, 0.0385, 0.01, 0.0253; 0.01, 0.94, 0.01, 0.04; 0.01, 0.03, 0.92, 0.04; 0.035, 0.035, 0.035, 0.895];

0.1.1 1 - Calculer de la distribution stationnaire des parts de marché

1-a Par résolution de l'équation de Rouché Capelli

1-b Par simulation d'une loi multinomiale

[110]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import *
from scipy import linalg as li
```

0.1.2 Calcul de la distribution stationnaire des parts de marché par itération successive

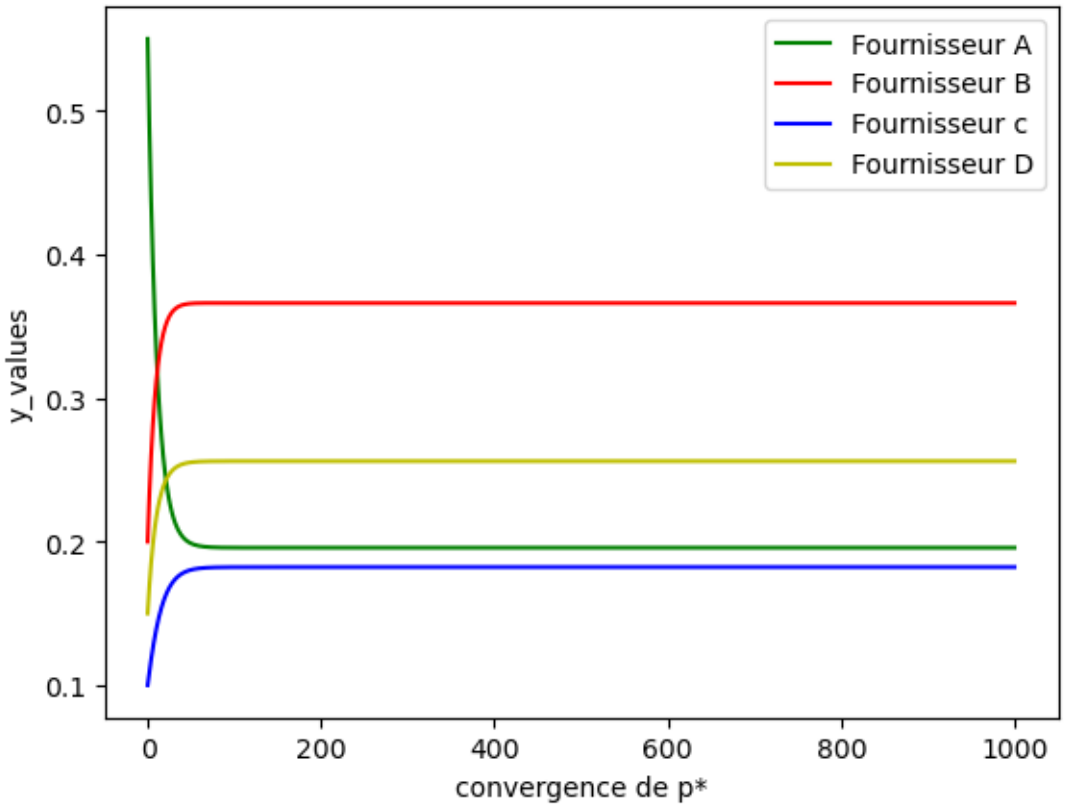
[111]:
```
#matricede trqnsition
P = [[0.9262, 0.0385, 0.01, 0.0253],
     [0.01, 0.94, 0.01, 0.04],
     [0.01, 0.03, 0.92, 0.04],
     [0.035, 0.035, 0.035, 0.895]]
transition = np.array(P)
transition
```

[111]:
```
array([[0.9262, 0.0385, 0.01  , 0.0253],
       [0.01  , 0.94  , 0.01  , 0.04  ],
       [0.01  , 0.03  , 0.92  , 0.04  ],
       [0.035 , 0.035 , 0.035 , 0.895 ]])
```

[113]:
```
#calcul de la distribution staitonnaire par succession d'iteration

def markov(matrice_trans, distribution_init, num_iterations):
    num_states = len(distribution_init)
    distribution_courante = np.array(distribution_init)
    liste = [distribution_init]
    for i in range(num_iterations):
        distribution_courante = np.dot(distribution_courante, matrice_trans)
        liste.append(distribution_courante)

    return distribution_courante,liste

# Exemple de notre chaîne de Markov
matrice_trans = transition

# Nombre d'itération de simulation
num_iterations = 1000
```

```python
# Distribution initiale des états
distribution_init = np.array([0.55, 0.2, 0.1, 0.15])

# Calcul de la distribution stationnaire
distribution_stationnaire, Liste = markov(matrice_trans, distribution_init,␣
 ↪num_iterations)

print("Distribution stationnaire (p*) des parts de marché :\n\n",␣
 ↪distribution_stationnaire)
#print(Liste)

# Convergence de p*
x0,x1,x2,x3= [],[],[],[]
t = np.linspace(0,len(Liste))

for i in range (len(Liste)):
    x0.append(Liste[i][0])
    x1.append(Liste[i][1])
    x2.append(Liste[i][2])
    x3.append(Liste[i][3])

plt.plot(x0,'g',label="Fournisseur A")
plt.plot(x1,'r',label="Fournisseur B")
plt.plot(x2,'b',label="Fournisseur c")
plt.plot(x3,'y',label="Fournisseur D")
plt.xlabel("convergence de p*")
plt.ylabel("y_values")
plt.legend()
plt.show()
```

```
Distribution stationnaire (p*) des parts de marché :

 [0.19571035 0.36604048 0.18222808 0.25602109]
```



0.1.3 1-a Calcul de la distribution stationnaire des parts de marché par résolution de l'équation de Rouché Capelli

[114]:
```
# Matrice de transition P
P = transition

# Matrice identité d'ordre 4
I4 = np.eye(4)

# Matrice A
A = np.vstack([P.T - I4, np.ones(4)])

# Vecteur b
b = np.array([0, 0, 0, 0, 1])

# Matrice (AT * A)
A_trans_A = np.dot(A.T, A)

# Vecteur (AT * b)
A_trans_b = np.dot(A.T, b)
```

```
# Résolution du système d'équations linéaires pour trouver le vecteur ( *)
distribution_stationnaire = np.linalg.solve(A_trans_A, A_trans_b)

print("Distribution stationnaire (p*) des parts de marché :\n\n",␣
 ↪distribution_stationnaire)
```

```
Distribution stationnaire (p*) des parts de marché :

 [0.19571035 0.36604048 0.18222808 0.25602109]
```

0.1.4 1-b Calcul de la distribution stationnaire des parts de marché par simulation d'une loi multinomiale

[62]:
```
#implementer la chaine de Markov pour avoir la distribution stationnaire avec␣
 ↪la loi multinomiale

def markov_multi(matrice_trans, etat_init, num_iterations):
    # nombre d'etat
    nombre_etat = len(matrice_trans)

    # etat courant
    etat_courant = etat_init

    # chaine de parcours des etats
    state_chain = [etat_courant]

    for i in range(num_iterations):
        # Obtenir les probabilités de transition à partir de l'état actuel
        transition_probs = matrice_trans[etat_courant]

        # Simuler le prochain état à l'aide de la distribution multinomiale
        next_state = np.random.choice(nombre_etat, p=transition_probs)

        # Mettre à jour l'état actuel
        etat_courant = next_state

        # Ajouter l'état simulé à la chaîne simulée
        state_chain.append(etat_courant)

    return state_chain
```

[118]:
```
#Appel de fonction à notre chaîne de Markov et calcul de la distribution␣
 ↪stationnaire (loi multinomiale)
matrice_trans = transition

# État initial
etat_init = 0
```

```
# Nombre d'étapes de simulation
num_iteratios = 70000

# Simuler une chaîne de Markov de 10000 étapes à partir de l'état initial
simulated_chain = markov_multi(matrice_trans, etat_init, num_iteratios)

#print("Chaîne simulée :", simulated_chain)

# Calcul de la distribution stationnaire en comptant les occurrences de chaque␣
 ↪état dans la chaîne simulée
distribution_stationnaire = np.bincount(simulated_chain, weights=None,␣
 ↪minlength=len(matrice_trans)) / (num_iteratios+1)

print("Distribution stationnaire (p*) des parts de marché :\n\n",␣
 ↪distribution_stationnaire)
```

```
Distribution stationnaire (p*) des parts de marché :

 [0.19461151 0.3661662  0.18282596 0.25639634]
```

0.2 TP2

0.2.1 2- Determinons la fonction valeur optimal(V*(s)) et la politique optimale (* (s)) pour le processus décisinnels de Markov (MDP)

2-a Par l'algorithme d'itération par valeurs (value-iteration)

2-b Par l'algorithme d'itération par politiques (policy-iteration)

Données

[119]:
```
#Ensemble d'etat
S=[0,1,2,3,4,5]

#Ensemble d'action
A=['up','down','left','right']
```

Differentes fonctions que nous utiliserons dans la suite pour implementer nos deux algorithmes value-iteration et policy-ietreation

[120]:
```
# Fonction de qui retourne la probabilite de transition d'un etat s vers un␣
 ↪etat s' suivant l'action a effectuée

def P(s,a,fs):
    r=0
    if(a=='up'):
            if(s==0):
                if(fs==s):
                    r=0.9
```

```
            elif(fs==1):
                r=0.1
            else:
                r=0
        elif(s==1):
            if(fs==s):
                r=0.8
            elif(fs==2 or fs==0):
                r=0.1
            else:
                r=0
        elif(s==2):
            if(fs==s):
                r=0.9
            elif(fs==1):
                r=0.1
            else:
                r=0
        elif(s==3):
            if(fs==s):
                r=0.1
            elif(fs==0):
                r=0.8
            elif(fs==4):
                r=0.1
            else:
                r=0
        elif(s==4):
            if(fs==1):
                r=0.8
            elif(fs==3 or fs==5):
                r=0.1
            else:
                r=0
        elif(s==5):
            if(fs==s):
                r=0.1
            elif(fs==2):
                r=0.8
            elif(fs==4):
                r=0.1
            else:
                r=0
elif(a=='down'):
        if(s==0):
            if(fs==3):
                r=0.8
```

```
                elif(fs==1 or fs==0):
                    r=0.1
                else:
                    r=0
            elif(s==1):
                if(fs==4):
                    r=0.8
                elif(fs==2 or fs==0):
                    r=0.1
                else:
                    r=0
            elif(s==2):
                if(fs==5):
                    r=0.8
                elif(fs==1 or fs==2):
                    r=0.1
                else:
                    r=0
            elif(s==3):
                if(fs==s):
                    r=0.9
                elif(fs==5):
                    r=0.1
                else:
                    r=0
            elif(s==4):
                if(fs==s):
                    r=0.8
                elif(fs==3 or fs==5):
                    r=0.1
                else:
                    r=0
            elif(s==5):
                if(fs==s):
                    r=0.9
                elif(fs==4):
                    r=0.1
                else:
                    r=0
    elif(a=='right'):
            if(s==0):
                if(fs==s):
                    r=0.1
                elif(fs==1):
                    r=0.8
                elif(fs==3):
                    r=0.1
```

```
                else:
                    r=0
            elif(s==1):
                if(fs==s or fs==4):
                    r=0.1
                elif(fs==2):
                    r=0.8
                else:
                    r=0
            elif(s==2):
                if(fs==s):
                    r=0.9
                elif(fs==5):
                    r=0.1
                else:
                    r=0
            elif(s==3):
                if(fs==s or fs==0):
                    r=0.1
                elif(fs==4):
                    r=0.8
                else:
                    r=0
            elif(s==4):
                if(fs==1 or fs==s):
                    r=0.1
                elif(fs==5):
                    r=0.8
                else:
                    r=0
            elif(s==5):
                if(fs==s):
                    r=0.9
                elif(fs==2):
                    r=0.1
                else:
                    r=0
    elif(a=='left'):
            if(s==0):
                if(fs==s):
                    r=0.9
                elif(fs==3):
                    r=0.1
                else:
                    r=0
            elif(s==1):
                if(fs==0):
```

```
                r=0.8
            elif(fs==s or fs==4):
                r=0.1
            else:
                r=0
        elif(s==2):
            if(fs==s or fs==5):
                r=0.1
            elif(fs==1):
                r=0.8
            else:
                r=0
        elif(s==3):
            if(fs==s):
                r=0.9
            elif(fs==0):
                r=0.1
            else:
                r=0
        elif(s==4):
            if(fs==1 or fs==s):
                r=0.1
            elif(fs==3):
                r=0.8
            else:
                r=0
        elif(s==5):
            if(fs==s or fs==2):
                r=0.1
            elif(fs==4):
                r=0.8
            else:
                r=0


    return r
```

[121]:
```
# Fonction qui retourne la recompense recue lorsqu'on effectue une action a␣
 ↪etant dans un etat s

def R(s,a):
    r=0
    Pa1=0.1
    Pa2=0.1
    Pa=0.8
    if(a=='up'):
```

```
        if(s==0):
            r=0*Pa-100*Pa1+0*Pa2
        elif(0<s<4 or s==5):
            r=0
        else:
            r=10
    elif(a=='down'):
        if(s==0):
            r=Pa*(-100)+Pa1*(-100)
        elif(0<s<4 or s==5):
            r=0
        else:
            r=Pa1*(2*10)
    elif(a=='right'):
        if(s==0):
            r=(Pa1)*(-100)
        elif(0<s<4 or s==5):
            r=0
        else:
            r=10*(Pa+Pa1)
    else:#(a=='left'):
        if(s==0):
            r=((0.1+0.8))*(-100)#miro_medium
        elif(0<s<4 or s==5):
            r=0
        else:
            r=10*(Pa+Pa1)

    return r
```

[122]:
```
# Fonction qui retoure le recompense du systeme

def gain(etats,action):
    liste = []
    Liste = []
    for i in range(len(etats)):
        for j in range (len(action)):
            liste.append(R(etats[i],action[j]))
        Liste.append(liste)
        liste = []
    return Liste
```

[123]:
```
# Fonction qui retoure la matrice de transition du systeme suivant l'etat et␣
 ↪l'action effectuée

def Trans_prob(etats, actions):
    liste = []
```

```
    Liste = []
    LISTE =[]
    for i in range(len(etats)):
        for j in range (len(actions)):
            for k in range (len(etats)):
                liste.append(P(etats[i],actions[j],etats[k]))
            Liste.append(liste)
            liste = []
        LISTE.append(Liste)
        Liste = []
    return LISTE
```

0.2.2 2-a Algorithme d'itération par valeurs (value-iteration)

[124]:
```
#value iteration

def value_iteration(transition_probs, Gain, gamma, epsilon):
    nombre_etat = len(transition_probs)
    nombre_action = len(transition_probs[0])

    # Initialisation des valeurs V(s) pour chaque état
    V = [0] * nombre_etat

    while True:
        # Copie de la valeur précédente pour vérifier la convergence
        V_prev = V.copy()

        # Mise à jour des valeurs pour chaque état s
        for s in range(nombre_etat):
            Q_s = []

            # Calcul des Q-values pour chaque action a dans l'état s
            for a in range(nombre_action):
                q_sa = Gain[s][a]
                for s_prim in range(nombre_etat):
                    q_sa += gamma * transition_probs[s][a][s_prim] *␣
↪V_prev[s_prim]
                Q_s.append(q_sa)

            # Mise à jour de la valeur de l'état s en prenant le maximum parmi␣
↪les Q-values
            V[s] = max(Q_s)

        # Vérification de la convergence
        max_diff = max(abs(V[s] - V_prev[s]) for s in range(nombre_etat))
        if max_diff < epsilon:
            break
```

```
    # Calcul de la politique optimale *(s) pour chaque état s
    politique = []
    for s in range(nombre_etat):
        Q_s = []
        for a in range(nombre_action):
            q_sa = Gain[s][a]
            for s_prim in range(nombre_etat):
                q_sa += gamma * transition_probs[s][a][s_prim] * V[s_prim]
            Q_s.append(q_sa)

        # Sélection de l'action a* qui maximise Q(s, a)
        meilleur_action = Q_s.index(max(Q_s))
        politique.append(meilleur_action)

    return V, politique
```

[125]:
```
# Application

#parametres
transition_probs = Trans_prob(S,A)
Gain = gain (S,A)
gamma = 0.9  # Facteur d'escompte
epsilon=1e-6 #tolerence

# Appel de l'algorithme pour obtenir les valeurs V* et la politique optimale *
valeur_optimale, strategie_optimale = value_iteration(transition_probs, Gain,␣
 ↪gamma,epsilon)
Action = [A[i] for i in strategie_optimale]
dictionnaire = dict(zip(S, Action))

print("\n Valeurs optimales V*(s) :\n\n", valeur_optimale,"\n\n")
print("Politique optimale *(s) :\n\n", dictionnaire)
```

```
 Valeurs optimales V*(s) :

 [20.70808431880798, 35.58014061375226, 32.83399357725108, 35.85173757340623,
42.72410346736096, 37.051003324241265]


Politique optimale *(s) :

 {0: 'right', 1: 'down', 2: 'down', 3: 'right', 4: 'right', 5: 'left'}
```

0.2.3 2-b Algorithme d'itération par politiques (policy-iteration)

[126]:
```
#policy-iteration

def policy_iteration(transition_probs, Gain, gamma, epsilon):
    nombre_etats, nombre_action = len(transition_probs),len(transition_probs[0])

    # Initialisation de la politique aléatoire
    politique = np.ones(nombre_etats, dtype=int)

    while True:
        # Étape d'évaluation de la politique
        V = np.zeros(nombre_etats)
        while True:
            V_new = np.zeros(nombre_etats)
            for s in range(nombre_etats):
                a = politique[s]
                V_new[s] = np.sum(transition_probs[s][a] * (Gain[s][a] + gamma␣
 ↪* V))
            if np.max(np.abs(V_new - V)) < epsilon:
                break
            V = V_new

        # Étape d'amélioration de la politique
        politique_stable = True
        for s in range(nombre_etats):
            action_courante = politique[s]
            Q = np.zeros(nombre_action)
            for a in range(nombre_action):
                Q[a] = np.sum(transition_probs[s][a] * (Gain[s][a] + gamma * V))
            best_action = np.argmax(Q)
            if action_courante != best_action:
                politique_stable = False
            politique[s] = best_action

        if politique_stable:
            break

    return  V,politique
```

[127]:
```
# Application

#parametres
transition_probs = Trans_prob(S,A)
Gain = gain (S,A)
gamma = 0.9  # Facteur d'escompte
epsilon=1e-6 #tolerence
```

```
# Appel de l'algorithme pour obtenir les valeurs V* et la politique optimale  *
valeur_optimale, strategie_optimale = policy_iteration(transition_probs, Gain,␣
 ↪gamma,epsilon)
Action = [A[i] for i in strategie_optimale]
dictionnaire = dict(zip(S, Action))

print("\n Valeurs optimales V*(s) :\n\n", valeur_optimale,"\n\n")
print("Politique optimale  *(s) :\n\n", dictionnaire)
```

```
 Valeurs optimales V*(s) :

 [20.70808418 35.58014047 32.83399344 35.85173743 42.72410333 37.05100318]


Politique optimale  *(s) :

 {0: 'right', 1: 'down', 2: 'down', 3: 'right', 4: 'right', 5: 'left'}
```