Compte Rendu de TP :
Algèbre Linéaire Appliquée

Cyprien MANGEOT - Arslane MEDJAHDI - Aloyse PHULPIN

25 janvier 2022

Dans ce compte rendu nous nous intéressons à la mise en place du programme "Page Rank" de Google et qui a fait sa popularité.

1 Prise en main :

On supposera dans cette partie que le web est constitué des quatres pages données ci-dessous ; les caractères en gras indiquent les hyperliens.

— P_1 : Matrice des hyperliens
La matrice des hyperliens code les liens entre les pages du web. **La matrice de Google**, employée par le célèbre moteur de recherche repose sur cette matrice.

— P_2 : Matrice stochastique
Une matrice stochastique est une matrice carrée dont les coefficients sont positifs et dont la somme des coefficients vaut 1 sur chaque ligne. **La matrice de Google** est un exemple célèbre de matrice stochastique, dont **le vecteur propre** de la valeur propre dominante fournit le PageRank.

— P_3 : Matrice de Google
La matrice de Google est **une matrice stochastique** déduite de la matrice des hyperliens de l'ensemble des pages web. Le vecteur PageRank de classement des pages par popularité est un **vecteur propre** associé à la plus grande valeur propre de cette matrice.

— P_4 : Vecteur propre
Si A est une matrice réelle $n*n$, on dit que le vecteur $v \in \mathbb{R}^{\kappa}$ est un vecteur propre de A si :

$$v \neq 0 \text{ et } \exists \lambda \in \mathbb{C} \; tel \; que \; : Av = \lambda v.$$

1. Posons $\alpha = 0,85$ et le vecteur v uniforme : $v = \frac{1}{n}e$.

   (a) La matrice H correspond à la matrice des hyperliens, pour la trouver, il suffit d'appliquer la formule suivante :

   $$H_{ij} = \begin{cases} \frac{1}{|P_{ij}|} & \text{si } P_i \text{ possède un lien qui pointe vers } P_j. \\ 0 & \text{sinon.} \end{cases}$$

   Où $|P_i|$ est le nombre d'hyperliens sortants de la page P_i vers la page. En suivant cette méthode, on obtient la matrice des hyperliens :

   $$\boxed{H = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1/2 & 1/2 \\ 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 0 & 0 \end{pmatrix}}$$

   (b) Pour trouver la matrice S, il suffit d'appliquer la formule :

   $S = H + a\frac{1}{n}e^T$ où a est un vecteur colonne défini par :

   $$a = (a_i)_{1 \leq i \leq 4} = \begin{cases} 1 & \text{si } P_i \text{ ne pointe vers aucune page.} \\ 0 & \text{sinon.} \end{cases}$$

   et où $e^T = (1, 1, 1, 1)$.

   En suivant cette méthode, on obtient la matrice stochastique :

   $$\boxed{S = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1/2 & 1/2 \\ 0 & 1/2 & 0 & 1/2 \\ 1/4 & 1/4 & 1/4 & 1/4 \end{pmatrix}}$$

(c) On se propose maintenant de trouver G la matrice de Google, donnée par la formule :

$$G = \alpha S + (1-\alpha)ev^T \text{ avec } v = (v_i) \in \mathbb{R}^4 \text{ ; } v > 0 \text{ et } \alpha \in]0;1[.$$

On obtient :

$$\boxed{G = \frac{1}{80}\begin{pmatrix} 3 & 3 & 71 & 3 \\ 3 & 3 & 37 & 37 \\ 3 & 37 & 3 & 37 \\ 20 & 20 & 20 & 20 \end{pmatrix}}$$

2. Voici l'algorithme de puissance itérée avec l'utilisation de la normalisation, celui-ci dépend de a et de v pour la suite du TP.

```python
def puissance_itere(S, x0, v, eps, n_max, a):
    ''' Calcul Page Rank (x ; x@A=x)
    Entrée :
        S : Matrice stochastisque
        x0 : vecteur stochastique d'initialisation
        v : vecteur d'initialisation
        eps : tolérance
        n_max : itération max
        a : 0<a<1
    Sortie:
        x : Page Rank
        k : nombre d'itérations
    '''
    n = len(x0)
    z = a*x0@S+(1-a)*v
    res = np.linalg.norm(z-x0, 1)
    x = z
    k = 1
    while res > eps and k < n_max:
        z = a*x@S+(1-a)*v
        res = np.linalg.norm(z-x, 1)
        x = z
        k += 1
    return x, k
```

3. A l'aide de la fonction Puissance itérée, nous allons calculer le PageRank π des pages de l'exemple avec les paramètres suivants :

$$\alpha = 0.85,\ \epsilon = 10^{-2},\ x_0 = v = \tfrac{1}{4}e,$$

Ces paramètres seront valables pour les questions 1) à 5). La matrice G a été calculé dans la première question.

On fait le test de la façon suivante :

```
print("==========Question 3============")
S = np.array([[0, 0, 1, 0], [0, 0, 1/2, 1/2],
              [0, 1/2, 0, 1/2], [1/4, 1/4, 1/4, 1/4]])
v = np.array([1/4, 1/4, 1/4, 1/4])

print(puissance_itere(S, x0, v, eps, n_max, a))
```

On obtient que $\pi = [0.11040664, 0.24134933, 0.30540718, 0.34283685]$ et $k = 6$.
La méthode converge en 6 itérations. On remarque que les probabilités stationnaires sont croissanteS. Ceci est parfaitement logique, si l'on note P_i les pages internet, tous les états i communiquent et plus i est grand, plus P_i renvoie vers un grand nombre de pages et reçoit d'hyperliens, donc plus $\pi(i)$ est élevé.

4. Dans cette question, nous allons observer l'impact du vecteur de personnalisation v. Recalculons le PageRank π en faisant varier le vecteur v. On fait le test de la façon suivante :

```
print("==========Question 4============")
v = np.array([0.1, 0.4, 0.1, 0.4])
print(puissance_itere(S, x0, v, eps, n_max, a))

v = np.array([0.02, 0.48, 0.02, 0.48])
print(puissance_itere(S, x0, v, eps, n_max, a))
```

   (a) Premier cas : $v = [0.1, 0.4, 0.1, 0.4]$

   On obtient que $\pi = [0.09315082, 0.25860515, 0.28079692, 0.36744711]$, en 6 itérations.

   (b) Deuxième cas : $v = [0.02, 0.48, 0.02, 0.48]$

   On obtient que $\pi = [0.08394772, 0.26780825, 0.26767145, 0.38057258]$, en 6 itérations.

(c) Conclusion :
On remarque le le PageRank π change en fonction du vecteur de personnalisation v. Plus la i-ème coordonée de v est grande, plus le temps asymptotique passé sur la page P_i augmentera. On constate donc dans le deuxième cas que les temps asymptotiques passés sur les 2ème et 4ème pages augmentent. Et ceux passés sur la 1ère et 3ème pages diminuent.

5. Voici le code de l'algorithme Page Rank :

```
def PageRank(H, v, a, x0, eps, n_max):
    ''' Calcul Page Rank (x ; x@G=G ; avec simplification)
    Entrée :
        H : Matrice des Hyperliens
        v : vecteur de personnalisation
        a : 0<a<1
        x0 : vecteur stochastique d'initialisation
        eps : tolérance
        n_max : itération max
    Sortie:
        x : Page Rank
        k : nombre d'itérations
    '''
    n=len(x0)
    # On calcule les donnnés d'initalisation
    # On crée le vecteur des liens 'artificielles'.
    l = e = np.ones(n)
    e = e*(1/n)
    row = H.nonzero()[0]
    #On supprimer les mutiplicité des indices
    #pour parcourir au max n éléments
    row = set(row)
    row = list(row)
    for i in row :
        l[i]=0
    z = a*(x0@H+(x0@l)*e)+(1-a)*v
    res = np.linalg.norm(z-x0, 1)
    x = z
    k = 1
    # On calcul le vecteur stochastique,
    # on s'arrête si la cv convient
    #ou si on dépasse le nb d'itérations max.
    while res > eps and k < n_max:
        z = a*(x@H+(x@l)*e)+(1-a)*v
        res = np.linalg.norm(z-x, 1)
        x = z
        k += 1
    return x, k
```

On exécute la fonction avec les paramètres des exemples précédent, comme suit :

```
print("==========Question 5============")
# Paramètres C.S.R
row = np.array([0, 1, 1, 2, 2])
col = np.array([2, 2, 3, 1, 3])
data = np.array([1, 1/2, 1/2, 1/2, 1/2])
H = csr_matrix((data, (row, col)), shape=(4, 4))

v = np.array([1/4, 1/4, 1/4, 1/4])
print(PageRank(H, v, a, x0, eps, 10))

v = np.array([0.1, 0.4, 0.1, 0.4])
print(PageRank(H, v, a, x0, eps, 10))

v = np.array([0.02, 0.48, 0.02, 0.48])
print(PageRank(H, v, a, x0, eps, 10))
```

L'affichage de ce test nous donne :

```
==========Question 5============
(array([0.11040664, 0.24134933, 0.30540718, 0.34283685]), 6)
(array([0.09315082, 0.25860515, 0.28079692, 0.36744711]), 6)
(array([0.08394772, 0.26780825, 0.26767145, 0.38057258]), 6)
```

Par conséquent, on retrouve bien les valeurs numériques des exemples précédents.

6. Voici le code de la fonction *genererH* qui prend en argument deux entiers n et m, et qui génère une matrice des hyperliens aléatoire H représentant un web constitué de n pages contenant au plus m hyperliens.

```
def genererH(n, m):
    '''Génère une matrice google, stochastique, de n lignes et m hyperliens
    Entrée :
        n : nb lignes
        m : nb colonnes
    Sortie :
        H : Matrice des Hyperliens stockée en mode sparse (C.S.R)'''
    # Row  : indice des lignes / Col : indice des colonnes
    # Data : coefficients / Indice : indices des colonnes de H
    row = [] ; col = [] ; data = [] ; Indice = []
    for i in range(n):
        Indice += [i]
    # On parcours chaque ligne
    for i in range(n):
        # On enlève la colonne corespondant à l'élement diagonal
        # (aucune page ne renvoie sur elle même).
        Indice.remove(i)
        # k le nombre d'hyperliens, max m.
        k = random.randint(0, m)
        if k != 0:
            # On rempli data et row des k indice corespondants
            # 1/k et la ième ligne.
            data += [1/k]*k
            row += [i]*k
            # On sélectionne un échantillon aléatoire sans remise
            # dans les colonnes disponibles (tout sauf j=i).
            col += random.sample(Indice, k)
        # On rajoute la colonne corespondant à l'élement diagonal.
        Indice.append(i)
    row = np.array(row)
    col =np.array(col)
    data = np.array(data)
    return csr_matrix((data, (row, col)), shape=(n,n))
```

Voici un exemple de matrice H (de dimension $n = 5$ avec au plus $m = 3$ hyperliens) générée par cette fonction :

```
==========Question 6============
[[0.         0.         0.33333333 0.33333333 0.33333333]
 [0.         0.         0.         0.         0.        ]
 [0.5        0.         0.         0.5        0.        ]
 [0.         1.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.        ]]
```

L'affichage de cette matrice en mode sparse nous donne :

```
(0, 2)    0.3333333333333333
(0, 3)    0.3333333333333333
(0, 4)    0.3333333333333333
(2, 0)    0.5
(2, 3)    0.5
(3, 1)    1.0
```

7. A l'aide de la fonction *genererH*, nous allons générer une matrice H (de dimension $n = 10000$ avec au plus $m = 50$ hyperliens). Puis nous allons observer à l'aide de *tic*, *toc* les temps de calcul du PageRank π selon que l'on utilise la méthode générale puissance itérée ou la méthode dédiée PageRank.

```
print("==========Question 7============")
n = 10000 ; m = 50 ; a = 0.5
eps = 10**(-3)
v = x0 = np.array([1/n]*n)
n_max=1000
H = genererH(n, m)
start = time.time()
PageRank(H, v, a, x0, eps,n_max)
end = time.time()
print("Temps PageRank :", end-start)

start1 = time.time()
S = matrice_S(H)
puissance_itere(S, x0, v, eps, n_max, a)
end1 = time.time()
print("Temps Puissance itéré :", end1-start1)
```

L'affichage de cette exécution nous donne :

```
==========Question 7============
Temps PageRank : 0.03979086875915527
Temps Puissance itéré : 0.6861710548400879
```

On remarque un écart de temps de calcul conséquent : en moyenne $0,6$ secondes. Autrement dit, Page Rank prend 4% du temps de calcul de l'algorithme de puissance itéré pour s'exécuter. Le gain de temps de calcul est donc considérable car il ne fera qu'augmenter avec la taille des matrices.

8. Tout d'abord on génère la matrice H avec la fonction $genererH$ (de dimension $n = 100000$ avec au plus $m = 50$ hyperliens). Dans cette question on mettera en évidence l'importance du stockage en mode Sparse et des simplifications de calculs. Une fois générée, on teste les fonctions Puissance Itérée et PagRank avec cette matrice H.

   Pour réaliser cette question, on a besoin d'une fonction qui prend la matrice H et qui nous renvoie la matrice S stockée en mode Sparse. Voici donc cette fonction appelée $matrice\_S\_Sparse$:

```
def matrice_S_Sparse(H):
    ''' Calcul de S en mode Sparse en fonction deH
    Entrée :
        H : Matrice des Hyperliens
    Sortie :
        S : Matrice Stochastique stockée en mode Sparse
    '''
    m, n = H.shape
    # On calcule la matrice des liens 'artificielles' en mode sparse.
    l =np.ones(n)
    row = H.nonzero()[0]
    row = set(row)
    row = list(row)
    for i in row:
        l[i] = 0
    L=[]
    for j in range(n):
        if l[j]!=0:
            L+=[j]*n
    c =[]
    for k in range(n):
        c += [k]
    c = c*(n-len(row))
    d = [1/n]*len(c)
    # On construit la matrice des lignes 'artificielles'
    F=csr_matrix((d, (L, c)), shape=(n,n))
    return H+F
```

   Répondons maintenant à la question. Plusieurs cas peuvent être distingués :
   Premier cas :
   Si on stocke H en matrice numpy, la fonction ne s'exécute pas et nous renvoie un message d'erreur pour nous signifier le manque de mémoire vive.
   Deuxième cas :
   Si on stocke H en mode sparse, la fonction Puissance Itérée peut cette fois-ci effectuer le calcul mais en une minute (en moyenne), ce qui n'est pas du tout optimal. On pourrait même dire que ce n'est pas un programme viable au vu du nombre de pages existantes sur internet (estimé à plusieurs milliards).

Troisième cas :
En revanche, si on exécute la fonction PageRank, le temps de calcul est réduit à moins d'une demi-seconde, ce qui est plus réaliste en considérant la quantité de pages internet connues.

Voici la manière dont nous avons comparé le temps de calcul du PageRank :

```
print("==========Question 8============")
n = 100000 ; m = 50
v = x0 = np.array([1/n]*n)
H = genererH(n, m)

start = time.time()
x,k=PageRank(H, v, a, x0, eps, n_max)
end = time.time()
print("Temps PageRank :", end-start)

start1 = time.time()
S = matrice_S_Sparse(H)
puissance_itere(S, x0, v, eps, n_max, a)
end1 = time.time()
print("Temps Puissance itéré, H en mode Sparse:", end1-start1)
```

Voici les résultats obtenus :

```
==========Question 8============
Temps PageRank : 0.2803988456726074
Temps Puissance itéré, H en mode Sparse: 49.67033886909485
```

Conclusion :
On remarque l'importance de chacune de nos optimisations : sans le mode Sparse on demande bien trop d'espace mémoire. Sans les simplifications de calculs, on peut obtenir un résultat mais dans un temps bien trop long pour que cela soit intéressant. C'est finalement en stockant la matrice en mode sparse qu'on obtient un résultat avec un temps de calcul intéressant.

9. On se place dans le contexte de la question 8. On considère que les 5 pages $P6, ..., P10$ sont celles d'un commerçant. On veut faire en sorte de mettre en avant les pages de ce commerçant.
On applique les observations faites lors des questions 1 à 5 : on augmente les coefficients de v[6] à v[10] du vecteur de personnalisation, tout en conservant la stochasticité du vecteur. Comme suit :

```
print("==========Question 9============")
#Sur l'exemple précédent, on a un vecteur v de taille n=100000
#Chaque coefficient de v vaut 1/n
#On print les probabilités stationnaires des pages 6 à 10 de notre commerçant
print(x[5:11])

#On remplace les personnalisations des pages 6 a 10 (ici on double)
o=0
for e in range(6,11):
    v[e]= v[e]*2
    o+=v[e]

#on recalcule les autres coefficients tel que sum(v)=1
for e in range(6):
    v[e]=(1-o)*(1/(n-5))
x,k=PageRank(H, v, a, x0, eps, n_max)

#On print à nouveau les probabilités stationnaires des pages 6 à 10
print(x[5:11])
```

On affiche ensuite les probabilités stationnaires des pages du commerçants avant et après modification des vecteur v. Et on constate ci-dessous que ses pages ont bien été mises en avant.

```
==========Question 9============
Valeurs des coefficients du PageRank pour les pages 6 à 10 avec un
vecteur v uniforme :
[1.01478361e-05 9.49183503e-06 8.16210850e-06 8.65559482e-06
 7.62298499e-06 9.55990737e-06]

Valeurs des coefficient du PageRank pour les pages 6 à 10 avec un vecteur
v dont les coordonnées 6 à 10 ont été doublées :
[1.01477019e-05 1.44918753e-05 1.31628309e-05 1.36556076e-05
 1.26229984e-05 1.45599750e-05]
```

Par conséquent, en augmentant certaines coordonnées du vecteur de personnalisation v, cela revient à mettre en avant les pages concernées.