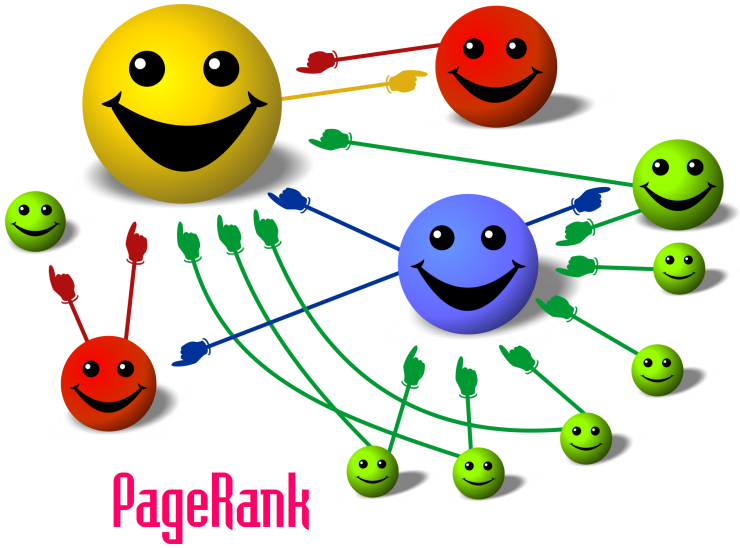
Rapport de projet

Algèbre linéaire appliquée

Matthéo RAPENNE & Ghilen TAGNIT HAMMOU

Contents



L'une des applications célèbres du modèle des Chaînes de Markov est l'algorithme PageRank de Google. PageRank est l'algorithme d'analyse des liens concourant au système de classement des pages Web utilisé par le moteur de recherche Google, il affiche les réponses dans un ordre reposant sur un indice de popularité des pages web. Le PageRank n'est qu'un indicateur parmi d'autres dans l'algorithme qui permet de classer les pages du Web dans les résultats de recherche de Google. Son principe de base est d'attribuer à chaque page une valeur (ou score) proportionnelle au nombre de fois que passerait par cette page un utilisateur parcourant le graphe du Web en cliquant aléatoirement, sur un des liens apparaissant sur chaque page. Ainsi, une page a un PageRank d'autant plus important qu'est grande la somme des PageRanks des pages qui pointent vers elle (elle comprise, s'il y a des liens internes) . Le but de ce projet est d'implémenter le calcul de l'indice: il s'agit d'un calcul de vecteur propre par la méthode de la puissance itérée.

1 Prise en main

1.1 Calcul des matrices H, S et G

1) Calcul de la matrice des hyperliens notée H :

```
H1 = c(0, 0, 1, 0) # ligne 1 de H
H2 = c(0, 0, 0.5, 0.5) # ligne 2 de H
H3 = c(0, 0.5, 0, 0.5) # ligne 3 de H
H4 = c(0, 0, 0, 0) # ligne 4 de H
H = matrix(c(H1, H2, H3, H4), nrow = 4, byrow = TRUE)
H
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0  0.0  1.0  0.0
## [2,]    0  0.0  0.5  0.5
## [3,]    0  0.5  0.0  0.5
## [4,]    0  0.0  0.0  0.0
```

2) Calcul de la matrice stochastique déduite notée S :

```
e = c(1,1,1,1)
a = c(0,0,0,1) # P4 est la seule page qui ne pointe vers aucune autre page
S = H + a%*%(1/4*t(e))
S
```

```
##      [,1] [,2] [,3] [,4]
## [1,] 0.00 0.00 1.00 0.00
## [2,] 0.00 0.00 0.50 0.50
## [3,] 0.00 0.50 0.00 0.50
## [4,] 0.25 0.25 0.25 0.25
```

3) Calcul de la matrice de Google notée G :

- Fonction pour calculer la matrice de Google
- Entrées: matrice carrée des hyperliens H, vecteur colonne e, vecteur colonne v, paramètre α
- Sortie: matrice carrée de Google G

```
matrice_Google <- function(H, e, v, alpha){
  n = ncol(H) # taille de H
  h = apply(H,1,sum)
  a = matrix(1*(h==0)) # Calcul du vecteur colonne a
  S = H + a%*%(1/n*e) # Calcul de la matrice S
  G = alpha*S + (1-alpha)*(e%*%t(v)) # Calcul de la matrice de Google
  return(G)
}
```

- La matrice de Google G est une matrice stochastique à coefficients strictement positifs.
- La somme des coefficients sur chaque ligne de G vaut 1.
- G vérifie les hypothèses du théorème de Perron-Frobenius.

```
e = c(1,1,1,1)
v = 1/4*e
alpha = 0.85

G = matrice_Google(H, e, v, alpha)
G
```

```
##        [,1]   [,2]   [,3]   [,4]
## [1,] 0.0375 0.0375 0.8875 0.0375
## [2,] 0.0375 0.0375 0.4625 0.4625
## [3,] 0.0375 0.4625 0.0375 0.4625
## [4,] 0.2500 0.2500 0.2500 0.2500
```

1.2 Implémentation de l'algorithme *puissanceiteree*

- Fonction *puissanceiteree* pour calculer le vecteur PageRank
- Entrées: matrice carrée stochastique $A > 0$ de taille n, vecteur ligne stochastique $x_0 \geq 0$ d'initialisation, tolérance $\epsilon > 0$ sur le résidu, borne iter_max sur le nombre d'itérations
- Sorties: vecteur ligne stochastique $x > 0$ tel que $xA = x$, et nombre k d'itérations

```
puissanceiteree <- function(A, x_0, epsilon, iter_max){
  z = x_0%*%A
```

```
  eta = sum(abs(z-x_0)) # norme 1
  x = z
  k = 1
  while(eta > epsilon & k < iter_max){
    z = x%*%A
    eta = sum(abs(z-x))
    x = z
    k = k + 1
  }
  return(c(x,k))
}
```

```
pi_0 = v # v = 1/4*e
epsilon = 10**-2
iter_max = 1000000


c = puissanceiteree(G, pi_0, epsilon, iter_max)
vectx = c[1:4] # vecteur ligne stochastique
vectx
```

```
## [1] 0.1104066 0.2413493 0.3054072 0.3428369
```

```
iterk = c[5] # nombre d'itérations
iterk
```

```
## [1] 6
```

- La méthode converge en 6 itérations.
- x = (0.1104066 0.2413493 0.3054072 0.3428369) -> c'est le vecteur PageRank
- La page 1 a la plus petite valeur associée à x car aucune page ne pointe vers elle.
- La page 2 est la seule à être visitée par une seule page (la page 3).
- Mais la page 2 pointe vers deux pages (les pages 3 et 4) ce qui explique que sa valeur associée à x est supérieure à celle de la page 1.
- Les pages 3 et 4 sont visitées par deux pages.
- Les valeurs des pages 3 et 4 associées à x sont donc supérieures à celle de la page 2.
- La page 3 pointe vers les pages 2 et 4 alors que la page 4 pointe vers aucune autre page.
- Donc la valeur de la page 4 associée à x est supérieure a celle de la page 3.
- Le classement des pages de la moins à la plus "populaire" est donc $1 < 2 < 3 < 4$.
- La page 1 est la moins "populaire".
- La page 4 est la plus "populaire".

1.3 Impact du vecteur de personnalisation v

Données : H, e, π_0, α, ϵ, iter_max sont comme précédemment

1.3.1 Calcul du vecteur PageRank pour $v1$

```
v1 = c(0.1,0.4,0.1,0.4)
G1 = matrice_Google(H, e, v1, alpha)
```

```
c1 = puissanceiteree(G1, pi_0, epsilon, iter_max)
vectx1 = c1[1:4]
vectx1
```

```
## [1] 0.09315082 0.25860515 0.28079692 0.36744711
```

```
iterk1 = c1[5]
iterk1
```

```
## [1] 6
```

- La méthode converge en 6 itérations.
- $x = (0.09315082\ 0.25860515\ 0.28079692\ 0.36744711)$ -> c'est le vecteur PageRank
- Les valeurs des pages 1 et 3 associées à x diminuent. -> on leur y accorde moins d'importance
- Les valeurs des pages 2 et 4 associées à x augmentent. -> on leur y accorde plus d'importance
- Le classement des pages de la moins à la plus "populaire" est $1 < 2 < 3 < 4$.
- La page 1 est la moins "populaire"
- La page 4 est la plus "populaire"

1.3.2 Calcul du vecteur PageRank pour $v2$

```
v2 = c(0.02,0.48,0.02,0.48)
G2 = matrice_Google(H, e, v2, alpha)
c2 = puissanceiteree(G2, pi_0, epsilon, iter_max)
vectx2 = c2[1:4]
vectx2
```

```
## [1] 0.08394772 0.26780825 0.26767145 0.38057258
```

```
iterk2 = c2[5]
iterk2
```

```
## [1] 6
```

- La méthode converge en 6 itérations.
- $x = (0.08394772\ 0.26780825\ 0.26767145\ 0.38057258)$ -> c'est le vecteur PageRank
- Les valeurs des pages 1 et 3 associées à x diminuent. -> on leur y accorde encore moins d'importance
- Les valeurs des pages 2 et 4 associées à x augmentent. -> on leur y accorde encore plus d'importance
- Le classement des pages de la moins à la plus "populaire" est $1 < 3 < 2 < 4$.
- La page 1 est la moins "populaire".
- La page 4 est la plus "populaire".
- Cette fois-ci, on remarque que la page 2 est plus "populaire" que la page 3.

2 Approfondissement

2.1 Implémentation de l'algorithme *PageRank*

- Fonction *PageRank* pour calculer le vecteur PageRank
- Entrées: matrice carrée $H \geq 0$ de taille n, vecteur stochastique de personnalisation v, paramètre α dans]0,1[, vecteur ligne stochastique $\pi_0 \geq 0$ d'initialisation, tolérance $\epsilon > 0$ sur le résidu, borne iter_max sur le nombre d'itérations

- Sorties: vecteur ligne PageRank π, et nombre k d'itérations

```
PageRank <- function(H, v, alpha, pi_0, epsilon, iter_max){
  n = ncol(H) # taille de H
  h = apply(H,1,sum)
  a = matrix(1*(h==0)) # Calcul du vecteur colonne a
  z = alpha*(pi_0%*%H + (pi_0%*%a)%*%e/n) + (1-alpha)*v
  eta = sum(abs(z-pi_0))
  pi = z
  k = 1
  while(eta > epsilon & k < iter_max){
    z = alpha*(pi%*%H + (pi%*%a)%*%e/n) + (1-alpha)*v
    eta = sum(abs(z-pi))
    pi = z
    k = k + 1
  }
  return(c(pi,k))
}
```

Données : e, π_0, v, v_1, v_2, α, ϵ, iter_max sont comme précédemment

```
library(Matrix) # pour utiliser le mode SPARSE

# Matrice H stockée en mode sparse
H1 = c(0.0, 0.0, 1.0, 0.0) # ligne 1 de H
H2 = c(0.0, 0.0, 0.5, 0.5) # ligne 2 de H
H3 = c(0.0, 0.5, 0.0, 0.5) # ligne 3 de H
H4 = c(0.0, 0.0, 0.0, 0.0) # ligne 4 de H
H = Matrix(c(H1, H2, H3, H4), nrow = 4, byrow = TRUE, sparse = TRUE)
H
```

```
## 4 x 4 sparse Matrix of class "dgCMatrix"
##
## [1,] . .   1.0 .
## [2,] . .   0.5 0.5
## [3,] . 0.5 .   0.5
## [4,] . .   .   .
```

2.1.1 Calcul du vecteur PageRank pour v

```
# v = 1/n*e avec n = 4
c = PageRank(H, v, alpha, pi_0, epsilon, iter_max)
vectx = c[1]
vectx
```

```
## [[1]]
## 1 x 4 Matrix of class "dgeMatrix"
##           [,1]      [,2]      [,3]      [,4]
## [1,] 0.1104066 0.2413493 0.3054072 0.3428369
```

```
iterk = c[2]
iterk
```

```
## [[1]]
## [1] 6
```

2.1.2 Calcul du vecteur PageRank pour $v1$

```
# v1 = (0.1,0.4,0.1,0.4)
c1 = PageRank(H, v1, alpha, pi_0, epsilon, iter_max)
vectx1 = c1[1]
vectx1
```

```
## [[1]]
## 1 x 4 Matrix of class "dgeMatrix"
##            [,1]      [,2]      [,3]      [,4]
## [1,] 0.09315082 0.2586051 0.2807969 0.3674471
```

```
iterk1 = c1[2]
iterk1
```

```
## [[1]]
## [1] 6
```

2.1.3 Calcul du vecteur PageRank pour $v2$

```
# v2 = (0.02,0.48,0.02,0.48)
c2 = PageRank(H, v2, alpha, pi_0, epsilon, iter_max)
vectx2 = c2[1]
vectx2
```

```
## [[1]]
## 1 x 4 Matrix of class "dgeMatrix"
##            [,1]      [,2]      [,3]      [,4]
## [1,] 0.08394772 0.2678082 0.2676714 0.3805726
```

```
iterk2 = c2[2]
iterk2
```

```
## [[1]]
## [1] 6
```

Les fonctions *puissanceiteree* et *PageRank* retournent les mêmes résultats.

2.2 Implémentation de l'algorithme *GenererH*

- Fonction *genererH* qui construit une matrice creuse
- Entrées: n = nombre de pages, m = nombre d'hyperliens au total par ligne de H
- Sortie: matrice d'hyperliens aléatoire H

```
genererH <- function(n, m){
  lignes = c()
```

```
  # ensemble contenant les numéros de lignes de H comptés avec multiplicité occ_i
  colonnes = c()
  # ensemble des colonnes qui contiennent des coefficients non nuls associées à
  # chaque ligne de H
  non_nuls = c()
  # ensemble des coefficients non nuls de H
  for(page_i in 1:n){ # page_i est un indice parcourant les lignes de H
    nb_liens = sample(0:m,1) # page_i pointe vers nb_liens pages
    if(nb_liens != 0){
      pages_liees = c() # page_j est dans page_liees si page_i pointe vers la
      # page_j
      while(length(pages_liees) != nb_liens){
        page_j = sample(1:n,1)
        # page_j est différente de page_i considérée car une page ne peut pas
        # pointer vers elle-même
        if(page_j %in% pages_liees == FALSE & page_j != page_i){
          pages_liees = append(pages_liees, page_j) # page_i pointe vers page_j
        }
      }
      occ_i = rep(page_i, nb_liens)
      # ensemble contenant nb_liens occurences de page_i
      terme_i = rep(1/nb_liens,nb_liens)
      # ensemble contenant les coefficients non nuls de la ligne i de H comptés
      # avec multiplicité nb_liens
      lignes = c(lignes, occ_i)
      colonnes = c(colonnes, pages_liees)
      non_nuls = c(non_nuls, terme_i)
    }
  }
  return(sparseMatrix(i = lignes, j = colonnes, x = non_nuls, dims = c(n,n)))
}
```

- Les pages sont assimilées aux lignes de H, désignées par la variable page_i.
- Les pages j vers lesquelles une page i pointe sont assimilées aux colonnes j de H.
- Pour chaque ligne i, on tire aléatoirement un nombre de colonnes nb_liens, compris entre 1 et m, dont les coefficients sont non nuls.
- On choisit aléatoirement ces nb_liens colonnes entre 1 et n qui forment l'ensemble pages_liees.
- Par définition, la page i pointe vers nb_liens autres pages différentes de page i.
- Ainsi, le coefficient de la ligne i de H affecté aux colonnes correspondantes aux pages ci-dessus est 1/nb_liens.
- On obtient une matrice des hyperliens H stockée en mode sparse.

2.2.1 Création d'une matrice creuse de taille 5

Les 5 pages contiennent au plus chacune 3 hyperliens.

```
set.seed(8) # pour toujours avoir le même tirage donc la même matrice H
A <- genererH(5, 3)
A
```

```
## 5 x 5 sparse Matrix of class "dgCMatrix"
##
```

```
## [1,] .         0.3333333 .         0.3333333 0.3333333
## [2,] .         .         1.0000000 .         .
## [3,] 0.5000000 .         .         0.5000000 .
## [4,] .         .         .         .         .
## [5,] 0.3333333 0.3333333 0.3333333 .         .
```

2.3 Temps de calcul des fonctions *puissanceiteree* et *PageRank*

```
n = 10000
m = 50
H = genererH(n, m)
alpha = 0.5
epsilon = 10**-3
e = c(rep(1,n)) # vecteur colonne constitué uniquement de 1
v = 1/n*e
pi_0 = v # initialisation uniforme
iter_max = 1000000
```

2.3.1 Calcul du temps d'exécution de *puissanceiteree*

```
start_time1 = Sys.time()
G = matrice_Google(H, e, v, alpha)
C1 = puissanceiteree(G, pi_0, epsilon, iter_max)
end_time1 = Sys.time()
Time1 = end_time1 - start_time1
Time1
```

```
## Time difference of 13.65058 secs
```

2.3.2 Calcul du temps d'exécution de *PageRank*

```
start_time2 = Sys.time()
C2 = PageRank(H, v, alpha, pi_0, epsilon, iter_max)
end_time2 = Sys.time()
Time2 = end_time2 - start_time2
Time2
```

```
## Time difference of 2.984844 secs
```

- Il y a une nette différence de temps de calcul du PageRank entre les deux fonctions.
- La fonction *PageRank* est toujours plus rapide que la fonction *puissanceiteree*.
- *PageRank* manipule une matrice très creuse H tandis que *puissanceiteree* prend comme argument en entrée une matrice G grande et pleine.
- Pour économiser de la place en mémoire et du temps, il est donc préférable d'utiliser la fonction *PageRank*.

2.3.3 Cas d'une matrice de taille plus grande

Données : α, ϵ, iter_max sont comme précédemment

```
n = 20000
m = 50
set.seed(8) # pour avoir la même matrice H que la prochaine question
H = genererH(n, m)
e = c(rep(1, n))
v = 1/n*e
pi_0 = v # initialisation uniforme
```

Calcul du temps d'exécution de *puissanceiteree* :

```
start_time1 = Sys.time()
G = matrice_Google(H, e, v, alpha)
C1 = puissanceiteree(G, pi_0, epsilon, iter_max)
end_time1 = Sys.time()
Time1 = end_time1 - start_time1
Time1
```

```
## Time difference of 1.589292 mins
```

- Entre 4 et 5 minutes pour $n = 30000$
- Impossible à déterminer pour $n = 50000$, beaucoup trop long + bug de RStudio

Calcul du temps d'exécution de *PageRank* :

```
start_time2 = Sys.time()
C2 = PageRank(H, v, alpha, pi_0, epsilon, iter_max)
end_time2 = Sys.time()
Time2 = end_time2 - start_time2
Time2
```

```
## Time difference of 15.93702 secs
```

- Environ 40 secondes pour $n = 30000$
- Environ 3 et 4 minutes pour $n = 50000$

- L'exécution de *PageRang* est plus rapide que *puissanceiteree* pour $n = 30000$.
- Le calcul de la matrice G est extrêmement coûteux pour $n = 50000$.
- Les calculs nécessitent donc beaucoup de mémoire ce qui, dans notre cas, fait bugger RStudio.
- Alors que *PageRank* fonctionne pour $n = 50000$.
- On constate que *PageRank* est beaucoup plus performant que *puissanceiteree*.
- Les opérations effectuées par *PageRank* sont moins coûteuses que celles de *puissanceiteree*.
- Il est conseillé d'utiliser *PageRank* pour limiter l'utilisation de la mémoire et gagner du temps.

2.4 Scénario

- On suppose que la matrice des hyperliens H de la question précédente représente les pages du web.

- Nous accceptons de bien classer 5 pages d'un commerçant, notées P_6,...,P_{10} par notre moteur de recherche.
- Les questions précédentent suggèrent fortement d'utiliser la fonction *PageRank* pour calculer le vecteur PageRank π.
- Les valeurs du PageRank π correspondent à la "popularité" des pages associées.
- La fonction *PageRank* prend 6 arguments en entrée : H, v, α, π_0, ϵ, iter_max.
- Nous décidons de fixer le paramètre α à 0.5 comme le cas précédent pour ne pas copier Google (qui préconise α à 0.85).
- Nous fixons aussi la tolérance ϵ à 10^{-3} et le nombre d'itérations iter_max à 1000000.
- Comme dans les cas précédents, le vecteur stochastique π_0 d'initialisation sera $\frac{1}{n}e$ où $n = 100\ 000$ et e est un vecteur colonne de taille $n \times 1$ composé uniquement de 1.
- On suppose que la matrice H est connue et stockée en mode sparse pour économiser du temps et de la mémoire.
- Nous allons modifier le vecteur de personnalisation v afin d'augmenter la "popularité" des 5 pages du commerçant.
- Pour se faire, on considère v tel que $v_i = \frac{1}{5}$ si $6 \leq$ i ≤ 10 et 0 sinon.
- On compare le PageRank π obtenu pour le vecteur v ci-dessus et le vecteur uniforme $\frac{1}{n}e$.
- On remarque alors que les valeurs des pages 6 à 10 associées à π sont plus élevées pour v.
- Ceci est intéressant car, dans ce cas, les pages du commerçant sont mieux classées par notre moteur.

Comme RStudio n'est pas adapté pour faire des calculs sur des matrices de grande taille, on se limitera à n = 20000. Le procédé est identique pour une matrice de taille 100000.

```
n = 20000
m = 50
set.seed(8) # pour avoir la même matrice H que la question précédente
H = genererH(n, m)
e = c(rep(1, n))
pi_0 = 1/n*e
alpha = 0.5
epsilon = 10**-3
iter_max = 1000000
```

2.4.1 Calcul du PageRank avec $\frac{1}{n}e$, vecteur uniforme

```
v = 1/n*e
C1 = PageRank(H, v, alpha, pi_0, epsilon, iter_max)
C1Pages6_10 = C1[[1]][,6:10] # avant modification du vecteur de personnalisation
C1Pages6_10 # valeurs des pages 6 à 10 du commmerçant associées au PageRank
```

```
## [1] 5.842515e-05 4.241267e-05 4.792216e-05 5.325783e-05 5.018740e-05
```

2.4.2 Calcul du PageRank avec v, vecteur modifié

```
v[1:5] = 0
v[6:10] = 1/5
v[11:n] = 0
```

```r
C2 = PageRank(H, v, alpha, pi_0, epsilon, iter_max)
C2Pages6_10 = C2[[1]][,6:10] # après modification du vecteur de personnalisation
C2Pages6_10 # valeurs des pages 6 à 10 du commmerçant associées au PageRank
```

```
## [1] 0.1000214 0.1000025 0.1000059 0.1000046 0.1000075
```

- On constate une nette différence selon le vecteur de personnalisation v choisi.
- Ainsi, les 5 pages du commerçant sont beaucoup plus "populaires" lorsqu'on modifie v comme pour le dernier test.
- Les pages $P_6, \ldots, P_{10}$ du commerçant seront donc mieux répertoriées dans notre moteur de recherche.