TP InfoResp
Page Rank

30 septembre 2024

Dans ce TP, nous allons étudier comment implémenter une version simple de l'algorithme Page Rank, proposé par les concepteurs du moteur de recherche Google pour évaluer l'importance des pages web.

Nous vous rappelons que cet algorithme repose sur le calcul de la distribution stationnaire d'une chaîne de Markov irréductible et apériodique.

Pour obtenir un algorithme efficace, nous verrons comment utiliser le fait que les graphes manipulés sont peu denses (*sparse* en anglais).

Ce TP se divise en trois parties :

— étudier le calcul de la distribution stationnaire dans une chaîne de Markov ;
— étudier une version de l'algorithme Page Rank pour des graphes dirigés et l'évaluer sur des graphes aléatoires ;
— étudier une solution visant à modifier le score obtenu par Page Rank à l'aide d'une ferme de liens.

1 Distribution stationnaire d'une chaîne de Markov

Pour cette partie, on suppose qu'on manipule une matrice carrée, représentée en Python comme une liste de listes, de dimension n. On suppose de plus que la matrice est stochastique, c'est-à-dire que pour chaque ligne, la somme des valeurs est égale à 1.

De plus, on manipulera des vecteurs de même dimension que la matrice. Etant donné un vecteur X de dimension n, on note X_i sa i-ème composante, pour $i \in \{0, \ldots, n-1\}$.

Écrire une fonction qui réalise le produit d'un vecteur (ligne) X avec une matrice M, c'est-à-dire le produit $X \cdot M$. On rappelle que le résultat est un vecteur (ligne) Y de même dimension que X, et dont la j-ème case contient la valeur $\sum_{i=0}^{n-1} X_i \times M_{i,j}$.

```python
def produit_vec_mat(X,M):
    # x : vecteur ligne de dimension n
    # M : matrice carrée stochastique de dimension n
    # retourne un vecteur ligne de dimension n, correspondant à X.M
    Y = []
    for j in range(len(M)):
        scal_prod = 0
        for i in range(len(X)):
            scal_prod += X[i]*M[i][j]
        Y.append(scal_prod)
    return Y
```

Vous pouvez tester votre fonction avec l'exemple suivant, vous devez retrouver les valeurs données dans les transparents du cours dernier (slide 42).

```python
M = [
    [.6,.1,.3],
```

```
    [.6,.4,0],
    [.7,0,.3]
]

X = [.7,.3,0]

print(produit_vec_mat(X,M))
```

Attendu : [.6, .19, .21]

Etant donnés deux vecteurs X et Y de même dimension n, on définit la distance entre ces vecteurs comme $d(X,Y) = \sum_{i=0}^{n-1} |X_i - Y_i|$. Compléter le code de la fonction suivante qui réalise le calcul de cette distance. Vous pouvez utiliser la fonction `abs` de Python qui calcule la valeur absolue d'un nombre réel.

```
def distance(X,Y):
    # X, Y : vecteurs de même dimension
    if len(X)!=len(Y):
        return null
    dist = 0
    # TODO ...
    return dist
```

```
def distance(X,Y):
    # X, Y : vecteurs de même dimension
    if len(X)!=len(Y):
        return null
    dist = 0
    for i in range(len(X)):
        dist+=abs(X[i]-Y[i])
    return dist
```

On suppose dans la suite que la chaîne de Markov représentée par M satisfait les conditions d'application du théorème assurant l'existence d'une unique distribution stationnaire, c'est-à-dire que la chaîne est bien irréductible et apériodique (voir les slides pour des détails). Un algorithme possible pour calculer cette distribution, appelée méthode des puissances, consiste à calculer les termes de la suite $(u_i)_{i\in\mathbb{R}}$ suivante :

$$\begin{cases} u_0 = [\frac{1}{n}, \ldots, \frac{1}{n}] \\ u_{i+1} = u_i \cdot M \end{cases}$$

Dans l'écriture précédente, les termes u_i représentent tous des vecteurs de dimension n.

Écrire une fonction qui prend en argument la matrice M, un nombre de pas K, et un seuil de précision ϵ, et qui effectue le calcul de la suite précédente jusqu'à ce que deux termes successifs soient à distance au plus ϵ, ou, si cela n'arrive pas, jusqu'au K-ème terme de la suite.

```
def calcul_distrib_stat(M,K,eps):
    # M : matrice carrée stochastique de dimension n
    # On suppose en plus que la chaine associée est irréductible et apé
    riodique
    # K : nombre de pas
```

```
    # eps : précision (flottant)
    n = len(M)
    u = n * [1/n]
    for i in range(K):
        u_next = produit_vec_mat(u,M)
        if distance(u_next,u)<eps:
            return u_ext
        u = u_next
    return u_next
```

À nouveau, comparez le résultat obtenu par votre fonction avec ceux des transparents du cours (slide 44).

```
stat = calcul_distrib_stat(M,10,10**(-5))
print(stat)
# Attendu : [.62,.1,.27] (aux approximations près)
```

Retour sur le produit matriciel. La complexité du produit matriciel implémenté est en $O(n^2)$, où n indique la dimension de la matrice. Par conséquent, dans le cas où la matrice M est peu dense, c'est-à-dire que beaucoup de cases sont égales à 0, il est plus intéressant d'obtenir un algorithme linéaire dans le nombre de cases non vide, c'est-à-dire dans le nombre d'arêtes du graphe. Pour cela, on suppose à présent que la matrice M est représentée comme un dictionnaire de dictionnaires. Plus précisément, on associe à chaque sommet un dictionnaire, qui donne, pour chaque sommet cible, le poids de la transition correspondante. Seules les transitions de poids non nul sont indiquées dans ce dictionnaire. On suppose ici que les sommets sont numérotés de 0 à $n-1$. Un exemple de cette représentation est donné ci-dessous, correspondant à la matrice précédente :

```
sparseM = {
    0 : {0:.6, 1:.1, 2:.3},
    1 : {0:.6, 1:.4},
    2 : {0:.7, 2:.3}
}
for source in sparseM:
    for target in sparseM[source]:
        print(source,target,sparseM[source][target])
```

Écrire un algorithme pour le calcul du produit matriciel $X \cdot M$ utilisant cette représentation. Vous vérifierez votre résultat avec l'exemple précédent.

```
def produit_vec_mat_sparse(X,M):
    # x : vecteur ligne de dimension n
    # M : dictionnaire de dictionnaire représentant la matrice stochastique
     de dimension n
    # retourne un vecteur ligne de dimension n, correspondant à X.M
    n = len(M)
    Y = n * [0]
    for i in range(n):
        for j in M[i]:
            Y[j] += X[i] * M[i][j]
```

```
    return Y

print(produit_vec_mat(X,M))
print(produit_vec_mat_sparse(X,sparseM))
# Attendu : [.6, .19, .21]
```

2 Implémentation de Page Rank

Dans cette partie, on va travailler avec des graphes produits à l'aide de la librairie `networkx`.
On rappelle que l'algorithme de Page Rank peut être résumé de la façon suivante :

1. On construit à partir du graphe la matrice stochastique M dans laquelle les poids sont obtenus en mettant une probabilité uniforme sur les transitions sortantes.
2. On modifie cette matrice en tenant compte des puits (saut uniforme sur tous les sommets du graphe), on obtient ainsi la matrice M_2. On ajoute ensuite une petite probabilité de saut arbitraire, afin d'avoir une chaîne de Markov irréductible et apériodique. On obtient ainsi la matrice P, appelée matrice Page Rank.
3. On calcule la distribution stationnaire associée à P.

On va se baser sur un générateur aléatoire de graphe pour étudier ce processus.

```
import networkx as nx
import matplotlib.pyplot as plt

n = 10
G = nx.random_k_out_graph(n, 3, 0.5)
nx.draw(G, with_labels = True)
plt.show()
```

Sur les environnements du département, les bibliothèques ne sont pas installées. On peut passer par docker :

```
$ mkdir $HOME/TP
$ docker run -it --volume="$HOME/TP:/TP" python bash
$ pip install networkx matplotlib
$ cd TP
```

Pour générer un PDF, il faut désactiver le mode interactif :

```
import matplotlib
matplotlib.use("Agg")
```

Puis générer le PDF avec :

```
plt.savefig("exemple.pdf")
```

Vérifier visuellement si le graphe obtenu est irréductible et apériodique (vérifier que le graphe est connexe sera suffisant en pratique) et régénérer le graphe sinon. À partir de la matrice stochastique que l'on peut dériver de ce graphe, on peut simplifier l'algorithme Page Rank en appliquant directement la méthode des puissances sur cette matrice. C'est-à-dire, que l'on ne se basera que sur la première étape et que l'on ignorera les 2 dernières étapes.

```
# Matrice stochastique associée
M = nx.stochastic_graph(G)
for x in M:
    print(x,":",M[x])
sparseM = {x: {y: sum(M[x][y][k]["weight"] for k in M[x][y]) for y in M[x]}
     for x in M}

def calcul_distrib_stat(M,K,eps):
    # M : matrice carrée stochastique de dimension n
    # On suppose en plus que la chaine associée est irréductible et apé
    riodique
    # K : nombre de pas
    # eps : précision (flottant)
    n = len(M)
    u = n * [1/n]
    for i in range(K):
        u_next = produit_vec_mat_sparse(u,M)
        if distance(u_next,u)<eps:
            return u_ext
        u = u_next
    return u_next

stat = calcul_distrib_stat(sparseM,10,10**(-5))
```

Comparer le résultat avec ce que renvoie la méthode suivante :

```
pr_nx = nx.pagerank(G)
plt.hist([pr_nx[i] for i in pr_nx])
```

Les sommets qui ont les plus forts scores restent les mêmes avec les 2 méthodes. Par contre, la méthode simplifiée fournit des scores qui convergent vers zéro pour les puits.

3 Fermes de liens

On rappelle ici le principe de la ferme de liens, présenté en cours (voir slides 54 et suivants). L'objectif est d'améliorer le score Page Rank obtenu par une page web. Pour cela, des pages inutiles, que l'on contrôle, sont ajoutées, et pointées par et vers la page dont on veut améliorer le score.

Nous allons travailler avec un graphe obtenue par l'algorithme utilisé précédemment, possédant 200 noeuds. Écrire un programme permettant de créer un tel graphe. Afficher l'histogramme des scores Page Rank associé.

Écrire une fonction qui prend en argument un graphe, et retourne un tuple de deux éléments donnant le sommet dont le score Page Rank est minimal, avec son score Page Rank.

```
def compute_min_PR(G):
    # G : DiGraph
    pr_nx = nx.pagerank(G)
    i_min = min(pr_nx, key = pr_nx.get)
    return (i_min, pr_nx[i_min])
```

Écrire une fonction qui prend en entrée un graphe G, l'indice i du sommet dont on veut modifier le score, et un entier K indiquant le nombre de pages web dans la ferme de liens, et qui modifie le graphe G de sorte à créer la ferme de liens comme indiqué plus haut.

```
def construire_ferme(G,i,K):
    # G : graphe orienté
    # i : un sommet de G
    # K : un entier, nombre de pages de la ferme à ajouter à G, reliées à i
    n = len(G)
    for k in range(K):
        G.add_edge(i,n+k)
        G.add_edge(n+k,i)
```

Nous allons étudier l'effet de l'ajout de la ferme de liens de façon empirique. Pour cela, nous allons suivre les étapes suivantes :

1. construire un graphe
2. identifier le sommet i de score minimal
3. ajouter la ferme de liens de taille K
4. mesurer le nouveau score Page Rank du sommet i

Ecrire un programme qui réalise les étapes précédentes, pour un même graphe, en faisant varier K de sorte à pouvoir tracer une courbe montrant l'évolution du nouveau score de i en fonction de K.

```
i_min, _ = compute_min_PR(G)
pr = []
for _ in range(100):
    construire_ferme(G, i_min, 1)
    pr.append(nx.pagerank(G)[i_min])
plt.plot(pr)
plt.show()
```