Réduction de dimension

Gabriel Romon

17 juin 2017

1 Introduction

1.1 Présentation du projet

Suite aux progrès récents réalisés dans la capture, le stockage, l'analyse et la visualisation des données, il est devenu courant, dans une multitude de domaines scientifiques, d'en manipuler de larges volumes possédant des dimensions élevées. Afin d'interpréter de telles quantités d'information, il est nécessaire d'employer des techniques qui permettent de réduire les dimensions. Notre projet répond à la question suivante : *étant donné un nuage de m points $x_1, \dots, x_m$ dans $\mathbb{R}^n$, avec m grand, comment résumer l'information contenue dans cet ensemble de points ?* Nous avons tenté de répondre à cette question en suivant deux approches fondées sur la factorisation de matrices.

1.2 Lien avec la factorisation de matrices

Commençons par illustrer le lien entre la réduction de dimension et la factorisation de matrices. Considérons un nuage de m points $x_1, \dots, x_m \in \mathbb{R}^n$ avec m grand devant n. Chaque x_i correspond à un individu, et chaque dimension $j \in [\![1, n]\!]$ correspond à une variable, de sorte que le nuage considéré décrit les valeurs de n variables prises par m individus, où le nombre d'individus est grand. Le nuage est représenté par la matrice $X = \begin{pmatrix} — & x_1 & — \\ & \vdots & \\ — & x_m & — \end{pmatrix}$ de taille $m \times n$. On cherche à factoriser X^T de façon approchée en un produit de deux matrices de rangs inférieurs : étant donné $k \ll n \ll m$, on cherche W et H respectivement de taille $n \times k$ et $k \times m$ telles que

$$\underset{(n\times m)}{X^T} \approx \underset{(n\times k)}{W} \cdot \underset{(k\times m)}{H}$$

On a alors réduit la taille de l'information contenue dans X, c'est-à-dire le nombre de coefficients, car $n \cdot k + k \cdot m = (n+m)k \ll nm$ pourvu que k soit faible devant n et m.
Par ailleurs, la factorisation $X^T = WH$ est intéressante en elle-même : si on note $w_1, \dots, w_k$ les colonnes de W, alors chaque colonne de X^T (ie chaque individu) s'écrit comme combinaison linéaire de $w_1, \dots, w_k$, de sorte qu'on peut considérer les w_j comme une base de l'espace des individus, voire comme des individus-type, à partir desquels les autres individus s'obtiennent par combinaison linéaire.

1.3 Déroulement du projet

Nous avons étudié deux techniques permettant d'obtenir une factorisation approchée du nuage. Dans un premier temps nous sommes revenus sur l'Analyse en Composantes Principales (ACP) qui avait été introduite en cours de statistique et en analyse de données, avec pour objectif de l'implémenter en Python dans son intégralité sans utiliser de librairie extérieure comme Scipy ou Numpy, et nous l'avons appliquée à la compression d'images. Ensuite, nous avons étudié la Non-negative Matrix Factorization (NMF), une méthode assez récente où X,W et H sont astreintes à avoir des coefficients positifs ou nuls.

2 L'Analyse en Composantes Principales

2.1 Principe

L'ACP est une méthode très répandue en analyse de données introduite par Pearson au début du 20ème siècle. Fondamentalement, l'ACP sur le nuage $x_1, \dots, x_m$ revient, étant donné $k \ll m$, à chercher une famille de vecteurs orthonormés $v_1, \dots, v_k$ dans $\mathbb{R}^n$ tels que chaque x_i s'écrit de façon approchée comme combinaison linéaire de $v_1, \dots, v_k$. L'ACP permet, pour k donné, de trouver les $v_1, \dots, v_k$ qui fournissent la meilleure approximation en un sens que nous allons préciser. Quantitativement, le problème d'approximation, une fois le nuage centré, est défini par

$$\underset{\substack{H \text{ s.e.v de } \mathbb{R}^n \\ \dim H = k}}{\operatorname{argmax}} \sum_{i=1}^{m} \|p_H(x_i)\|^2$$

qu'on peut encore écrire

$$\underset{v_1,\dots,v_k \text{ orthonormés}}{\operatorname{argmax}} \sum_{i=1}^{m} \sum_{j=1}^{k} \langle v_j, x_i \rangle^2$$

dont la solution est donnée par les k vecteurs propres associés aux k plus grandes valeurs propres de la matrice symétrique semi-définie positive $X^T X$ où comme précédemment,

$$X = \begin{pmatrix} — & x_1 & — \\ & \vdots & \\ — & x_m & — \end{pmatrix}$$

En principe, et dans la pratique, l'ACP revient donc à déterminer un nombre restreint de vecteurs propres d'une matrice symétrique semi-définie positive.

2.2 Un algorithme naïf : la méthode des puissances

Nous avons d'abord implémenté un algorithme itératif simple qui, pour une matrice symétrique semi-définie positive, converge vers un vecteur propre associé à la plus grande valeur propre. Nous le présentons ci-après, mais il convient d'abord d'ajouter quelques précisions sur notre implémentation des matrices en Python.

2.2.1 Gestion des matrices

Etant donné que nous avons décidé, pour l'ACP, d'éviter les librairies de calcul scientifique, nous avons dû coder nous-mêmes toutes les fonctions afférentes au calcul matriciel : multiplication de deux matrices, multiplication d'une matrice par un scalaire, transposée, etc. Une matrice est représentée par la liste de ses lignes, donc par une liste de listes, et la plupart du temps, un vecteur par une matrice-colonne.
Par ailleurs, par souci de concision et de performance [1], nous avons utilisé autant que possible la compréhension de liste au lieu de boucles `for`.

Toutes les fonctions de calcul matriciel que nous avons implémentées se situent dans le fichier `calcmat.py`

2.2.2 Algorithme des puissances

La méthode des puissances repose sur un algorithme itératif simple décrit ci-dessous. Il n'est a priori pas nécessaire de diviser par la norme à chaque itération mais il est conseillé de le faire dans toutes les sources que nous avons consultées, afin d'éviter tout overflow ou underflow.

Notons A la matrice dont on veut calculer le vecteur propre et $\lambda_1 \geq \lambda_2$ les deux plus grandes valeurs propres de A avec multiplicité. Dans [2], il est démontré que l'algorithme converge à condition que $\lambda_1 > \lambda_2$ et que x_0 ne soit pas dans l'orthogonal du sous-espace propre associé à λ_1. Alors la vitesse de convergence de

Algorithme 1 Algorithme des puissances : première implémentation

Entrées : $A \in S_n^+(\mathbb{R})$, x_0 vecteur unitaire aléatoire, ϵ niveau de précision

1: $x_1 := \frac{Ax_0}{\|Ax_0\|}$
2: **while** $\|x_k - x_{k-1}\| > \epsilon$ **do**
3: $\quad x_{k+1} := \frac{Ax_k}{\|Ax_k\|}$
4: **end while**
5: **return** x_k

l'algorithme est en $O\left(\left|\frac{\lambda_2}{\lambda_1}\right|\right)$. Si les deux plus grandes valeurs propres sont proches, il est possible que l'algorithme converge lentement.

Nous proposons une autre implémentation de cet algorithme qui calcule A^{2^k} par multiplication successive et applique la matrice obtenue à x_0. Le nombre de puissances à calculer est ici fixé à l'avance.

Algorithme 2 Algorithme des puissances : deuxième implémentation

Entrées : $A \in S_n^+(\mathbb{R})$, x_0 vecteur unitaire aléatoire, $k \geq 1$

1: **for** $j = 1, \ldots, k$ **do**
2: $\quad A_j = A_{j-1} \cdot A_{j-1}$
3: **end for**
4: **return** $\frac{A_k x_0}{\|A_k x_0\|}$

2.2.3 Déflation

Nous avons pour l'instant répondu partiellement au problème : nous sommes capables d'approcher un seul vecteur propre v_1, mais qu'en est-il de ceux associés à des valeurs propres inférieures ?
La connaissance de v_1 nous donne une valeur approchée de λ_1 et il suffit en fait de faire l'opération suivante, dite de déflation [3] : $A \leftarrow A - \frac{\lambda_1}{\|v_1\|^2} v_1 v_1^T$ et d'appliquer une nouvelle fois la méthode des puissances. On obtient donc k vecteurs propres de A en appliquant successivement la méthode des puissances et en modifiant A à chaque étape.

2.2.4 Pratique

Nos implémentations de la méthode des puissances sont situées dans le fichier `power.py`

L'algorithme des puissances est présenté dans les ouvrages d'analyse numérique que nous avons consultés comme un algorithme de base qui permet de calculer

un nombre restreint de vecteurs propres d'une matrice. Nous avons pu confirmer la convergence de l'algorithme par des tests sur des matrices aléatoires. Avec la première implémentation, même pour des matrices de petite taille (10×10) nous avons observé que le nombre d'étapes nécessaires pour converger avec une précision de 10^{-4} était plutôt élevé (de l'ordre de 20 itérations pour le premier vecteur propre, 30 pour le deuxième). En faisant des essais sur des matrices plus grandes (de l'ordre 100×100) et toujours avec la première implémentation, nous avons rencontré un problème d'overflow dans le calcul des normes (le vecteur de départ étant aléatoire ce problème survient fréquemment, mais pas tout le temps).
En ce qui concerne la deuxième implémentation, nous n'avons plus observé le même problème d'overflow (cette implémentation ne faisant appel qu'à un seul calcul de norme). Cependant le nombre d'itérations est fixé à l'avance, ce qui ne permet pas vraiment de contrôler la précision de la convergence. De plus, un trop grand nombre d'itérations mène à un overflow dans le calcul des coefficients de la matrice (car ils peuvent diverger). En pratique, pour des matrices de l'ordre 100×100, il n'a pas été possible de dépasser le calcul de A^{2^8}.

La méthode des puissances est certes facile à implémenter et fournit des résultats acceptables pour de petites matrices, en revanche, il est inenvisageable de l'utiliser pour traiter des données réelles (même une simple image 512×512).

2.3 Autre méthode de calcul des vecteurs propres

La méthode des puisssances ne convenant pas, nous avons dû nous tourner vers d'autres moyens. Nous avons recherché comment la fonction `scipy.linalg.eigh` détermine les éléments propres d'une matrice symétrique réelle. Nous avons examiné son code source dans un premier temps, situé à l'adresse [4] et nous en avons déduit qu'elle appelait une routine Fortran de la librairie Lapack nommée `dsyev` dont la source est disponible en [5]. Son fonctionnement est mieux expliqué en [6] : la matrice symétrique est d'abord réduite sous forme tridiagonale T (tous ses coefficients sont 0 sauf peut-être sur la diagonale et les deux bandes au-dessus et en-dessous d'elle), puis l'algorithme détermine les éléments propres de T. Les méthodes de Lapack sont le fruit de plusieurs années de recherche et il était hors de notre portée de tenter de les reproduire.

Des techniques plus abordables sont cependant décrites dans la référence [7]. Ce sont ces algorithmes que nous avons choisi d'implémenter en Python, toujours sans recours à Numpy ou Scipy.

2.3.1 Principe

Nous présentons un algorithme qui, contrairement à la méthode des puissances, calcule toutes les valeurs propres et tous les vecteurs propres d'une matrice symé-

trique semi-définie positive. Il procède en deux temps : étant donné une matrice $A \in S_n^+(\mathbb{R})$, la première étape consiste à trouver P orthogonale et T tridiagonale telle que $PTP^T = A$. Dans la deuxième étape, on cherche D diagonale et Q orthogonale telle que $T = QDQ^T$. On connait alors les valeurs propres de A (la diagonale de D), ainsi que ses vecteurs propres : en effet, les vecteurs propres de T sont les colonnes de Q et ceux de A sont obtenus par multiplication avec P.

2.3.2 De symétrique à tridiagonale : les matrices de Householder

Principe Etant donné un vecteur $u \in \mathbb{R}^n$, on définit $P_u = I_n - 2\frac{uu^T}{\|u\|^2}$ la matrice de Householder associée à u. On remarque que P_u est symétrique et orthogonale. Par ailleurs, considérons $x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{R}^n$ et $e_1 = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$. Posons aussi $u = x \mp \|x\|e_1$. Alors on montre que $P_u x = \pm\|x\|e_1$. Pour ce u bien choisi, l'effet de P_u sur x est de mettre toutes ses composantes à 0 sauf la première. L'idée est alors de multiplier à gauche et à droite de A par des P_u bien choisis pour supprimer les coefficients indésirables.

Détaillons les deux premières étapes de l'algorithme. On a une matrice symétrique A qu'on écrit $\left(\begin{array}{c|c} x_{11} & x_{21}\cdots x_{n1} \\ \hline \begin{matrix} x_{21} \\ \vdots \\ x_{n1} \end{matrix} & A' \end{array}\right)$. On considère alors $P_1 = \left(\begin{array}{c|c} 1 & 0\cdots 0 \\ \hline \begin{matrix} 0 \\ \vdots \\ 0 \end{matrix} & P_{u_1} \end{array}\right)$ où P_{u_1} est une matrice de Householder de taille $(n-1)\times(n-1)$ avec u_1 choisi de sorte à annuler tous les composantes de $\begin{pmatrix} x_{21} \\ \vdots \\ x_{n1} \end{pmatrix}$ sauf la première. On a

$$P_1 A P_1 = \left(\begin{array}{c|c} x_{11} & k\,0\cdots 0 \\ \hline \begin{matrix} k \\ 0 \\ \vdots \\ 0 \end{matrix} & P_{u_1} A' P_{u_1} \end{array}\right) = \left(\begin{array}{cc|c} x_{11} & k & 0 \quad \cdots \quad 0 \\ k & x'_{22} & x'_{32}\cdots x'_{n2} \\ \hline 0 & x'_{32} & \\ \vdots & \vdots & A'' \\ 0 & x'_{n2} & \end{array}\right)$$

On considère $P_2 = \left(\begin{array}{cc|c} 1 & 0 & 0\cdots 0 \\ 0 & 1 & 0\cdots 0 \\ \hline 0 & 0 & \\ \vdots & \vdots & P_{u_2} \\ 0 & 0 & \end{array}\right)$ où P_{u_1} est une matrice de Householder de taille $(n-2)\times(n-2)$ avec u_2 choisi de sorte à annuler tous les composantes de

$\begin{pmatrix} x'_{32} \\ \vdots \\ x'_{n2} \end{pmatrix}$ sauf la première, de sorte que $(P_2P_1)A(P_1P_2) = \left(\begin{array}{cc|ccc} x_{11} & k & 0 & \cdots & 0 \\ k & x'_{22} & k' & \cdots & 0 \\ \hline 0 & k' & & & \\ \vdots & \vdots & & P_{u_2}A''P_{u_2} & \\ 0 & 0 & & & \end{array}\right)$

On continue de cette façon jusqu'à ce que la matrice obtenue soit tridiagonale. En posant $P = P_1P_2 \cdots P_{n-2}$ on a $T = P^TAP$, comme voulu.

Implémentation Le produit P_uAP_u est en réalité particulièrement simple à effectuer. En effet, en posant $p = 2\frac{Au}{\|u\|^2}$, $K = \frac{u^Tp}{\|u\|^2}$ et $q = p - Ku$, on montre que $P_uAP_u = A - qu^T - uq^T$. Le membre de droite ne fait pas apparaitre de produit matriciel (qui demande $O(n^3)$ temps) mais simplement deux produits dyadiques qui nécessitent $O(n^2)$ temps.

De même, le calcul de $P_1P_2 \cdots P_{n-2}$ ne nécessite aucun produit matriciel : en effet,

$$(P_1P_2 \cdots P_{k-1})P_k = \begin{pmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{pmatrix} \begin{pmatrix} I_k & 0 \\ 0 & P_{u_k} \end{pmatrix} = \begin{pmatrix} P_{11} & P_{12}P_{u_k} \\ P_{21} & P_{22}P_{u_k} \end{pmatrix}$$

En posant $P'_k = \begin{pmatrix} P_{12} \\ P_{22} \end{pmatrix}$, on a $\begin{pmatrix} P_{12}P_{u_k} \\ P_{22}P_{u_k} \end{pmatrix} = P'_kP_{u_k} = P' - yu^T$ avec $y = 2\frac{P'_ku}{\|u\|^2}$

Enfin, on code une matrice tridiagonale symétrique par deux vecteurs : sa diagonale et la bande au-dessus de sa diagonale. A partir de ces remarques nous aboutissons à l'algorithme ci-dessous (les listes sont indexées à partir de 0).

Commentaires D'après [7], cet algorithme a un coût temporel en $O(n^3)$. En pratique, cette méthode nous a paru numériquement très stable : même pour de grandes matrices, nous n'avons pas eu de problème d'overflow. Cependant, elle est plutôt lente : pour une matrice d'ordre 500×500, l'algorithme prend quelques minutes pour s'exécuter.

Algorithme 3 Décomposition d'une matrice symétrique en une matrice tridiagonale symétrique

Entrées : $A \in S_n(\mathbb{R})$

diag=[A[0][0]], updiag = [ ], $P = I_n$, $x = A[0][1...n-1]$
for $i = 1, \ldots, n-1$ **do**
  $A = A[1...n-1][1...n-1]$ // Suppression première ligne, colonne de A
  if $x[0] > 0$ **then**
    $k = \|x\|$
  else
    $k = -\|x\|$
  end if
  $x[0] = k + x[0]$, $H = \frac{\|x\|^2}{2}$, $p = \frac{Ax}{H}$, $K = \frac{u^T p}{2H}$, $q = p - Kx$
  $A = A - qx^T - xq^T$, diag.append($A[0][0]$), updiag.append($-k$)
  B=matrice formée des $n-i$ dernières colonnes de P
  $y = \frac{Bx}{H}$
  $B = B - yx^T$
  Les $n-i$ dernières colonnes de P deviennent B
  $x = A[0][1...n-i-1]$
end for
return (diag, updiag, P)

2.3.3 De tridiagonale à diagonale : la décomposition QR

Principe Il reste à calculer les éléments propres d'une matrice tridiagonale symétrique. On appelle décomposition QR d'une matrice $A \in M_n(\mathbb{R})$ la donnée de Q orthogonale et R triangulaire supérieure telle que $A = QR$. Lorsque A est tridiagonale, cette décomposition se calcule facilement en faisant le produit à gauche de $n-1$ matrices de Givens (des matrices creuses avec des coefficients bien choisis, voir [2]).

Une fois cette décomposition acquise, on peut implémenter l'algorithme QR dont le principe est donné ci-dessous : D'après [7], lorsque T est symétrique semi-

Algorithme 4 Algorithme QR

Entrées : $A \in M_n(\mathbb{R})$, $k \geq 1$

$A_0 = A$
for $j = 1, \ldots, k$ **do**
  Q_j, R_j tels que $Q_j R_j = A_{j-1}$ // Décomposition QR de A_{j-1}
  $A_j = R_j Q_j$
end for
return A_k

définie positive (c'est le cas des matrices tridiagonales qu'on considère), l'algorithme QR converge vers une matrice diagonale D, où les éléments diagonaux sont ordonnés dans l'ordre décroissant de gauche à droite. On peut alors écrire $D \approx (Q_k^T \dots Q_1^T)T(Q_1 \dots Q_k)$ et en posant $Q = Q_1 \dots Q_k$ on a $T \approx QDQ^T$ comme souhaité.

Implémentation et commentaires Nous avons implémenté avec succès et sans problème particulier la décomposition QR d'une matrice triadiagonale, ainsi que l'algorithme QR. Nous avons tiré parti du caractère creux des matrices de Givens en évitant de calculer entièrement les produits matriciels en question. D'après [2], la décomposition QR dans le cas tridiagonal a un coût temporel en $O(n^2)$ et l'algorithme QR est en $O(kn^3)$

2.3.4 Combinaison des algorithmes

Munis de ces méthodes nous sommes en mesure de revenir au problème initial et calculer tous les vecteurs propres (et les valeurs propres bien qu'elles soient inutiles) d'une matrice symétrique semi-définie positive.

Nos implémentations de toutes les méthodes abordées sont disponibles dans le fichier `QR.py`

L'algorithme obtenu est numériquement stable et précis : avec un nombre suffisant d'itérations on converge toujours vers les éléments propres cherchés, peu importe la taille de la matrice considérée. Cependant, notre algorithme est notablement plus lent que celui de Scipy. Après profilage du code avec cProfile, nous nous sommes rendus compte que la fonction `mult` en charge de la multiplication de deux matrices était la plus chronophage. En remplaçant notre implémentation de la multiplication par celle de Numpy (avec la fonction `mult2`), l'algorithme obtenu est devenu significativement plus rapide (le temps nécessaire pour l'exécution d'une itération de l'algorithme QR est divisé par 8).

2.4 Application de l'ACP à la compression d'images

Afin de mettre en pratique les méthodes que nous avons implémentées, nous avons appliqué l'ACP à une image 512*512 (bien que dans ce cas, le nombre d'individus soit égal au nombre de variables). Nous avons importé l'image en noir et blanc de la Figure 1 dans Python sous la forme d'une matrice 512*512 dont les coefficients compris entre 0 et 255 représentent un niveau de gris. L'image initiale `lena.jpg` pèse 66 kilooctets. Nous avons appliqué l'ACP au nuage correspondant,

FIGURE 1 – Image de départ

puis nous avons calculé les composantes grâce à la méthode QR, ce qui nous a donné $v_1, \dots, v_k$ une famille orthonormée de $\mathbb{R}^{512}$ (avec $k \ll 512$). Nous avons ensuite calculé les projections de chaque individu sur cette famille : $x_i \approx \sum_{j=1}^{k} \langle x_i, v_j \rangle v_j$ ce qui s'écrit encore

$$X^T \approx \begin{pmatrix} | & & | \\ v_1 & \cdots & v_k \\ | & & | \end{pmatrix} \cdot \begin{pmatrix} \langle x_1, v_1 \rangle & & \langle x_m, v_1 \rangle \\ \vdots & \cdots & \vdots \\ \langle x_1, v_k \rangle & & \langle x_m, v_k \rangle \end{pmatrix}$$

Pour k dans $\{2, 5, 10, 20, 30, 40, 50, 60, 70\}$ nous avons enregistré le nuage reconstitué (ie l'image reconstituée par rapport aux k premières composantes principales), nos résultats sont visibles en Figure 2. L'image reconstituée est nette et sans artefacts à partir de $k = 40$. En terme de compression, les images produites sont toutes au moins deux fois plus légères que l'image de départ (33 kilooctets pour $k = 70$).

Le code que nous avons écrit est situé dans le fichier `image.py` et les images sont dans le dossier `/images`.

FIGURE 2 – Images reconstituées de gauche à droite et de haut en bas avec $k = 2, 5, 10, 20, 30, 40, 50, 60, 70$ composantes principales

3 La Non-negative Matrix Factorization

3.1 Principe

La NMF est une méthode de factorisation de matrices ($X^T = WH$) au même titre que l'ACP, à la différence qu'on contraint X, W et H à avoir des coefficients positifs ou nuls. Il s'agit d'un sujet de recherche assez récent (années 90).

Rappelons que l'égalité $X^T = WH$ peut être interprétée de la façon suivante : les colonnes de X^T (les individus) sont obtenus comme combinaisons linéaires (à coefficients dans H) des colonnes de W (qui sont peu nombreuses). La NMF, contrairement à l'ACP, est intéressante dans la mesure où tous les coefficients

considérés dans les combinaisons linéaires sont ≥ 0 : il y a un phénomène de superposition (au sens d'addition), sans soustraction possible. On s'attend donc à ce que les colonnes de W fassent apparaitre des caractéristiques ("features") cachés du nuage de données. C'est de cette manière qu'est introduite la NMF dans l'article fondateur [8].

3.2 Application

Par manque de temps, nous n'avons pas pu étudier la théorie permettant de déterminer la NMF d'une matrice et nous avons plutôt souhaité reproduire les résultats de [8] en nous servant de l'implémentation de la NMF disponible dans `sklearn.decomposition`.

Nous nous sommes appuyés sur un dataset composé de 30991 articles de l'encyclopédie Grolier et des 15276 mots les plus courants de la langue anglaise (à l'exclusion de mots qui reviennent très souvent comme the, a, here, ...). On dispose alors d'un nuage de hautes dimensions, où chaque colonne correspond à un article et chaque ligne compte les occurences du mot associé la ligne dans chaque article. La matrice X^T est donc de taille 15276×30991.

Nous avons calculé la NMF de X^T avec $k = 400$ (400 colonnes dans W). Sur une machine disposant de 8 GO de RAM et d'un Intel Core i5 6500 le calcul a pris une quarantaine de minutes. Les 4 coeurs du processeur ont été sollicités (signe que l'implémentation de `sklearn` est parallélisée). Nous avons ensuite examiné les colonnes de W, en observant pour chacune quels étaient les mots avec les occurences (empiriques) les plus élevés. Après un léger traitement, nous avons exporté les résultats vers le fichier `columns.txt`. Dans ce fichier, chaque ligne est associée à une colonne de W : elle contient les 20 mots ayant les plus fortes occurrences dans la colonne considérée. On observe que chaque ligne du fichier regroupe des mots portant sur un même thème : par exemple (cities, urban, city, towns, planning,...), (government, majority, section, supreme, court, representatives,...). On est donc tenté de dire que la NMF a regroupé les mots en différents thèmes (des thèmes qui étaient latents au départ).

Le code associé est situé dans le fichier `NMF.py` et les datasets dans le dossier `/nmf`

4 Conclusion

Ce projet a été très enrichissant : nous avons appris beaucoup en algèbre linéaire, en analyse numérique, en analyse de données et en programmation. Nous avons été confrontés à des difficultés d'ordre algorithmique et matériel (algorithmes qui convergent trop lentement, overflows). L'étude de l'ACP et de la NMF ont enrichi ce que nous avions appris en cours de statistique et d'analyse de données. Nous avons eu le plaisir d'étudier les rouages d'algorithmes d'algèbre linéaire que nous avions utilisé couramment sans nous poser de questions. Ce projet a avant tout été un travail de recherche personnelle et nous avons certainement gagné en autonomie.

Références

[1] Are list-comprehensions and functional functions faster than "for loops"? Stack Overflow, 2014. https://stackoverflow.com/questions/22108488/are-list-comprehensions-and-functional-functions-faster-than-for-loops.

[2] Gene H Golub et Charles F Van Loan : *Matrix computations*, volume 3. JHU Press, 2012.

[3] James F Epperson : *An introduction to numerical methods and analysis*. John Wiley & Sons, 2013.

[4] scipy.linalg.eigh source code, 2017. https://github.com/scipy/scipy/blob/v0.14.0/scipy/linalg/decomp.py#L205.

[5] dsyev lapack documentation, 2017. http://www.netlib.org/lapack/explore-html/d2/d8a/group__double_s_yeigen_ga442c43fca5493590f8f26cf42fed4044.html#ga442c43fca5493590f8f26cf42fed4044.

[6] syevr intel documentation, 2017. https://software.intel.com/en-us/node/469188.

[7] BP Flannery, WH Press, SA Teukolsky et WT Vetterling : Numerical recipes 3rd edition : The art of scientific computing, 2007.

[8] Daniel D Lee et H Sebastian Seung : Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(6755):788–791, 1999.