Institut National des Sciences
Apliquées de Lyon

3ème Année Bioinformatique et Modélisation

Professeur : Samuel Bernard

Systèmes de recommandation

Auteur :
Lisa Nicvert
Émilie Mathian

Année 2017 - 2018

Table des matières

Introduction

Le jeu de données MoviesLens a été créé par le groupe de recherche GroupLens travaillant pour l'université du Minnesota. Ils ont créé un site web proposant aux utilisateurs de visionner des films gratuitement, dans le but de construire cette base de données. Les premières données ont été récoltées entre le 19 septembre 1997 et le 22 avril 1998. Les utilisateurs devaient attribuer une note de 1 à 5 pour chaque film regardé. Le premier jeu de données MoviesLens contient ainsi les notes d'environ 1000 utilisateurs parmi un choix d'environ 1700 films, soit 100 000 *ratings*. En outre, d'autres informations sont disponibles sur les utilisateurs (sexe, âge, activités...) et sur les films classés par genre (fantastique, science fiction, drame ...) permettant le *clustering* de ces données.
Cette base de données a été publiée gratuitement, pour le développement de programmes informatiques, offrant ainsi une référence pour mesurer la performance des algorithmes. En constante évolution, elle contient à présent 20 millions de *ratings* sur 27 000 films pour 138 000 utilisateurs !
Cette base de données a été particulièrement importante pour le développement de systèmes de recommandation, de plus en plus utilisés pour les requêtes sur internet. De nombreuses équipes ont travaillé sur la question : comment améliorer la pertinence des réponses d'une requête pouvant rassembler des milliers de réponses connaissant quelques informations sur l'utilisateur ? Autrement dit, comment l'algorithme de Google parvient-il à fournir une liste de sites web qui sont susceptibles de vous intéresser ?

Dans cette étude nous proposerons un système de recommandation, basé sur le premier jeu de donné publié par GroupLens, grâce à un programme développé en Python.
Notre système de recommandation devra permettre :

1. De prédire la note d'un utilisateur donné pour un film donné.
2. Pour un utilisateur donné de fournir une liste de N recommandations

Keywords : Systèmes de recommandation, Décomposition en valeurs singulières

1 Description et importation des données

1.1 Description des données

Le dossier ml-100k regroupe les fichiers suivants :

- u.info : résumé du nombre total d'utilisateurs (943), du nombre total de films (1682) et du nombre total de *ratings* (100 000)
- u.genre : liste du nombre de films par genre
- u.user : informations sur les utilisateurs (*user id* | *age* | *gender* | *occupation* | *zip code*)
- u.item : informations sur chaque film. Pour chaque film, on connaît son ID, son titre, sa date de sortie, son URL et le genre auquel il appartient parmi les 19 proposés
- u.occupation : liste des professions des utilisateurs
- u.data : ensemble des *ratings* pour chaque utilisateur et chaque film

Nous disposons de 5 fichiers u.base et de 5 fichiers u.test qui sont des partitions disjointes du fichier u.data. Les fichiers u.base et u.test sont deux portions de respectivement 80% et 20% du fichier total u.data. Les fichiers u.base sont utilisés comme données d'entraînement et les fichiers u.test sont les données tests.

1.2 Importation des données

Organisation :
Deux programmes constituent notre système de recommandation. La classe *R_matrix* permet d'importer les données et de construire les matrices R et le programme principal de recommandation contient l'ensemble des fonctions permettant d'implémenter le système de recommandation en question.
Les résultats seront affichés selon le choix de l'utilisateur au lancement du programme.

1.3 Classe *R_matrix*

Nous avons créé une classe objet *R_matrix*, permettant notamment d'importer les données issues du fichier texte. On les importe à l'aide de la fonction *open*. Elles sont ensuite stockées sont sous formes listes de dictionnaires. Ainsi, chaque ligne d'un fichier u.base ou u.test devient un dictionnaire possédant les clés : *user, movie, rating, datastamp*. Les données sont importées ainsi :

```
"""
import numpy as np
import csv

class R_matrix:
    #--------------------------------
    # Cette classe définit un objet complexe, contenant une matrice (la matrice users-items)
    #et deux dictionnaires faisant correspondre les coordonnées aux identifiants des films
    # et des utilisateurs (resp nM et nU).
    #--------------------------------

    def __init__(self,datafilename):
        """Construit la matrice Utilisateur-item qui contient les ratings de chaque film
        par l'utilisateur (0 si non noté)."""

        with open(datafilename, 'r') as f:
            fieldnames=['user','movie','rating','datestamp']
            reader = csv.DictReader(f, delimiter = '\t', fieldnames=fieldnames)
            # create a dict out of reader, converting all values to integers
            data = [dict([key, int(value)] for key, value in row.items()) for row in list(reader)]

        nU=self.number_user(data)[0]
        nM=self.number_movie(data)[0]

        self.R=np.zeros( (nU,nM) )

        self.cU=self.correspondance_users(data)
        self.cM=self.correspondance_movies(data)

        for row in data:
            ncol=self.cM[row['movie']]
            nline=self.cU[row['user']]
            self.R[nline,ncol] = row['rating']

    def __str__(self):
        return str(self.R)
```

1.4 Importation des fichiers associés (u.genre)

Les autres jeux de données sont importés simplement sous forme de liste. Par exemple pour le fichier u.genre, nous obtenons :

```
def info_genre ( datafilename):
    """ Importer les donnees sur le genre"""
    with open(datafilename, 'r') as f:
        reader = csv.DictReader(f, delimiter = '|', fieldnames=['genre','number'])
        Genre = list(reader)
    return Genre
```

Programme d'importation des données relatives au genre du film

```
GENRE MOVIE
[OrderedDict([('user', 'unknown'), ('age', '0'), ('sex', None), ('occupation', None), ('zipcode', None)]
), OrderedDict([('user', 'Action'), ('age', '1'), ('sex', None), ('occupation', None), ('zipcode', None)
])]
```

Affichage du fichier u.genre

1.5 Organisation du programme principal

Dans le programme principal, l'ensemble des données est chargé avant l'exécution des questions. D'une part celles-ci sont importées par la création de l'objet *R_matrix*, et d'autre part grâce à boucle itérative suivante :

```
base = ['u1.base', 'u2.base', 'u3.base', 'u4.base', 'u5.base']
baseUserItem = [ 'baseUserItem1', 'baseUserItem2', 'baseUserItem3', 'baseUserItem4', 'baseUserItem5']

for i in range(5):
    with open( base[i] , 'r') as f: # rb for read binary code
        fieldnames=['user','movie','rating','datestamp']
        reader = csv.DictReader(f, delimiter = '\t', fieldnames=fieldnames)
        baseUserItem[i]=  [dict([key, int(value)] for key, value in row.items()) for row in list(reader)]
```

Importations itérative des fichiers u.base dans le programme principal

Ce programme importe les 5 fichiers u.base, il nous suffira alors de choisir `baseUserItem[i]` où i correspond à l'index désiré pour obtenir l'un d'entre eux. Le même algorithme est utilisé pour importer les fichiers u.test.
Les données sont donc chargées avant l'exécution du programme principal, pour optimiser le temps de calcul.

```
R1obj=Rm.R_matrix('u1.base')
Rc1 = R1obj.Construct_Rc()
Rr1 = R1obj.Construct_Rr()
nM=R1obj.number_movie(baseUserItem[0])[0] # Nombre de films du fichier u1.base
nU=R1obj.number_user(baseUserItem[0])[0] # Nombre Utilisateurs du fichier u1.base
```

Importations des données en amont du programme

2 Matrice-Utilisateur Item

La matrice Utilisateur-Item nommée R contient 943 lignes correspondant à chaque utilisateur i du fichier u1.data et 1650 colonnes pour chaque film. Ainsi les coefficients de R[i][j] correspondent à la note d'un utilisateur pour un film.
Procédure :

1. Calcul du nombre de films et d'utilisateurs du fichier traité grâce aux fonctions `number_user` et `number_movie`. Ces fonctions créent une liste de valeurs des clés associées à 'user' et 'movie'. Puis on élimine les doublons grâce à la fonction *set*. Enfin, elles calculent la taille des listes sans doublons correspondant respectivement au nombre d'utilisateurs et au nombre de films. [1]. Les fonctions retournent les listes sans doublons comme `list_unique_user` et `list_unique_movie` ainsi que le nombre d'utilisateurs et de films (respectivement `number_user` et `number_movie`).

```
def number_user(self,datafilename):
    """This function allows to find the users number in a file, deleting duplicated items.
    Cf: set function"""
    user_list=[]
    for x in datafilename:
        user_list.append(x['user'])
        #print(user_list)
    list_unique_user = set(user_list)
    number_user = len(list_unique_user)
    return (number_user,list_unique_user)
```

2. Les listes sans doublons d'utilisateurs et de films sont utilisées par les fonctions `list_users` et `list_movies` qui associent un index i à chaque élément.

```
def list_users(self,datafilename):
    """Construit une liste avec les identifiants uniques des utilisateurs."""
    set_users=self.number_user(datafilename)[1]
    lU=[]
    for i in set_users:
        lU.append(i)
    return lU
```

3. Ces listes sont finalement utilisées par les fonctions `correspondance_users` et `correspondance_movies`. La fonction `correspondance_users` crée un dictionnaire associant l'ID (identifiant) de l'utilisateur à un numéro de ligne de la matrice R. Respectivement la fonction `correspondance_movies` associe l'ID du film à un numéro de colonne de la matrice R.

1. Remarquons que chaque fichier base contient l'ensemble des utilisateurs mais pas nécessairement l'ensemble des films

```
def correspondance_users(self,datafilename):
    """Retourne un dictionnaire.
    Associe un id d'user (clé) à un numero de ligne de la (future) matrice R (valeur)."""
    lU=self.list_users(datafilename)
    c={}
    for i in range(0,len(lU)):
        c[lU[i]]=i
```

Fonction *correspondance_ users*

4. Nous créons ensuite une matrice vide dont la dimension correspond au nombre d'utilisateur par le nombre de films, définie grâce aux fonctions `number_user` et `number_movie`. Pour chaque ligne et colonne dont l'ID est donné par les fonctions `correspondance_users` et `correspondance_movies` nous associons le *rating* R[i][j] donné par les valeurs des clés ['rating'].

```
self.R=np.zeros( (self.nU,self.nM) )
for row in data:
    ncol=self.cM[row['movie']]
    nline=self.cU[row['user']]
    self.R[nline,ncol] = row['rating']
```

2.1 Résultats

Nous illustrons la construction de la matrice R avec le fichier global u.data :

```
MATRICE TAILLE R DU FICHIER U.DATA
(943, 1682)
```

La taille correspond bien aux données du fichiers u.info avec 943 lignes, soit le même nombre d'utilisateurs, et 1682 colonnes correspondant au nombre de films.

```
943 users
1682 items
100000 ratings
~
```

Données du fichier u.info

3 Approximation SVD bas rang

3.1 Construction des matrices *Rr* et *Rc*

Les premières prédictions naïves consistent à compléter la matrice R par les moyennes des utilisateurs ou des films. Ainsi nous allons construire les matrices complétées par

les moyennes des films (Rc) et celles par les moyennes des utilisateurs (Rr). Pour cela nous suivons la procédure décrite dans le programme R_matrix :

1. Calcul des moyennes. Les matrices Rc et Rr sont complétés par les moyennes des utilisateurs et respectivement des films. Elles sont calculées en omettant les coefficients R[i][j] égaux à zéro. Pour ce faire nous avons remplacé chaque valeur nulle par la valeur spéciale NaN (Cf : fonction `replace_zero_to_NAN`). Pour le calcul des moyennes par colonne Rc et par ligne Rr nous avons ainsi écrit les fonction `mean_by_movie` et `mean_by_user` telles que :

```
def mean_by_user (self):
    """This function computes the means by lines of a matrix ignoring NAN values"""
    mU = np.nanmean(self.R, axis=1)
    return mU
```

2. Grâce aux fonctions calculant les moyennes nous construisons les matrices Rc et Rr, via les fonction `Construct_Rc` et `Construct_Rr` telles que :

```
def Construct_Rc(self):
    """This function completes the matrix R by the means of movies' ratings (per movie)"""
    self.replace_zero_to_NAN()
    self.Rc = np.copy(self.R)
    moy=self.mean_by_movie()
    for i in range(0,self.Rc.shape[0]):
        for j in range(0,self.Rc.shape[1]):
            if np.isnan(self.Rc[i][j]) == True:
                self.Rc[i][j] = moy[j]
    return self.Rc

def Construct_Rr(self):
    """This function completes the matrix R by the means users' ratings (per user)"""
    self.replace_zero_to_NAN()
    self.Rr = np.copy(self.R)
    moy=self.mean_by_user()
    for i in range(0,self.Rr.shape[0]):
        for j in range(0,self.Rr.shape[1]):
            if np.isnan(self.Rr[i][j]) == True:
                self.Rr[i][j] = moy[i]
    return self.Rr
```

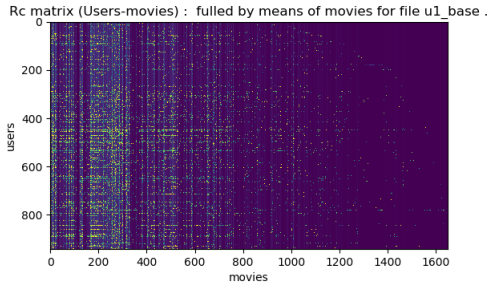
   Pour chacune des deux matrices nous commençons par copier les matrices R puis nous remplaçons chaque valeur nulle par la valeur spéciale NaN. Enfin si le coefficient R[i][j] est égal à NaN, nous le remplaçons par sa moyenne (moyenne calculée pour chaque utilisateur et respectivement pour chaque film par les fonctions `mean_by_user` et `mean_by_movie`).

3.2 Résultats

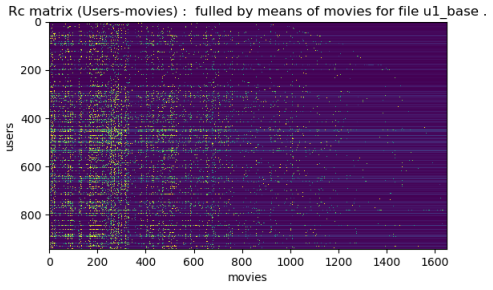
Nous illustrons ci-dessous la construction de la matrice R grâce au fichier total *u.data*.

```
10 first columns and 10 first lines of matrix R built by U.data, after replacing zero value by NAN:      [[ 5.   3.   4.   3.   3.   5.   4.   1.   5.   3.]
 [  4.  nan  nan  nan  nan  nan  nan  nan  nan   2.]
 [ nan  nan  nan  nan  nan  nan  nan  nan  nan  nan]
 [ nan  nan  nan  nan  nan  nan  nan  nan  nan  nan]
 [  4.   3.  nan  nan  nan  nan  nan  nan  nan  nan]
 [  4.  nan  nan  nan  nan  nan   2.   4.   4.  nan]
 [ nan  nan  nan   5.  nan  nan   5.   5.   5.   4.]
 [ nan  nan  nan  nan  nan  nan   3.  nan  nan  nan]
 [ nan  nan  nan  nan  nan   5.   4.  nan  nan  nan]
 [  4.  nan  nan   4.  nan  nan   4.  nan   4.  nan]]

the third first means by movies columns of matrix R built by U.data, after replacing zero value by NAN:  [ 3.87831858  3.20610687  3.03333333]

the third first lines of matrix Rr built by U.data, after replacing zero value by NAN:   [ 3.61029412  3.70967742  2.7962963 ]
number of means by columns in matrix R built by U.data  1682 number of means by lines in matrix R built by U.data  943
```

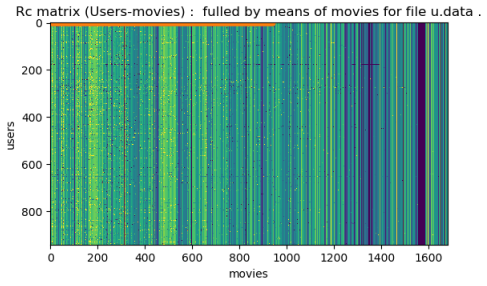
Le premier résultat est obtenu après avoir remplacé les valeurs nulles par la valeur spéciale *NaN*. Le second et le troisième après avoir calculé les moyennes avec le fichier *u.data*. La quatrième ligne nous permet de vérifier la taille des matrices (ici, on vérifie que la dimension est bien 943×1682 car le fichier total compte 943 utilisateurs et 1682 films).



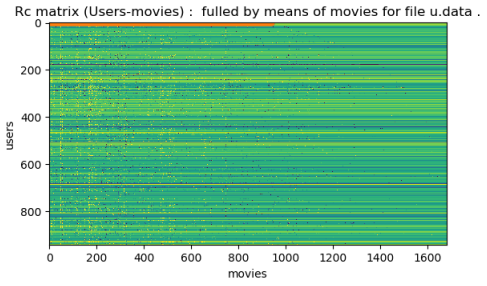
(a) Matrice Rc (films) calculée grâce au fichier u1.base



(b) Matrice Rr (utilisateurs) calculée grâce au fichier u1.base



(a) Matrice Rc (films) calculée grâce au fichier u.data



(b) Matrice Rr (Utilisateurs) calculée grâce au fichier u.data

Sur les figures ci-dessus, nous pouvons observer les notes moyennes attribuées par un utilisateur à un film. En effet grâce à un gradient de couleurs, plus le 'rating', pour un utilisateur et un film donnés est haut (max : 5), plus la couleur est claire et réciproquement. En premier lieu nous pouvons constater que les matrices Rr et Rc des données totales contiennent davantage de données que celles générées par le fichier u1.base du fait de la troncature des données. Ensuite, on complète la matrice Rc par les moyennes des films. On homogénise ainsi les données par colonne. D'après les graphiques, les matrices Rr sont harmonisées par ligne, soit par utilisateur.

3.3 Décomposition en valeurs singulières

Les décompositions en valeur singulières des matrices Rr et Rc de taille (943×1650) sont calculées telles que :

$$R_{[r,c]} = USV'_s$$

Où S est la matrice diagonale des valeurs singulières calculées grâce aux racines du polynôme caractéristique $\mathcal{P}_{R[r,c]} = det(Rr - \lambda I)$. Rappelons que les valeurs singulières équivalent à la racine carrée des valeurs propres, soit $\sigma_j = \sqrt{\lambda_j}$, avec $j \in [0, 943]$. Les matrices V_s dont V'_s est la transposée sont des matrices unitaires, dont les lignes correspondent respectivement aux vecteurs propres de $R^*_{[r,c]}R_{[r,c]}$. On calcul ainsi les 943 vecteurs propres associés aux 943 valeurs propres $(V_{s1}, ..., V_{sn})$ telles que :

$$R^*_{[r,c]}R_{[r,c]}V_{si} = \lambda V_{si}$$

On complète ensuite cette matrice avec des vecteurs orthogonaux et unitaires pour faire la SVD complète.

Pour les vecteurs de la matrice U, on calcule $U_1, ..., U_n$ tels que :

$$U_j = R_{[r,c]}V_j/\sigma_j$$

La matrice U forme une base orthonormée étant donné que $943 < 1650$.

Grâce à la fonction `np.linalg.svd`, nous pouvons calculer la matrice diagonale complétée des valeurs singulières, notée S, et les matrice unitaire U et V. Ce calcul est permis par la fonction `SVD` telle que :

```
def SVD(R_matrix):
    U, s, Vs = np.linalg.svd(R_matrix, full_matrices=True)
    return U , s , Vs
```

Notons que cette fonction renvoie directement la matrice V_s' transposée. Nous pouvons alors déterminer les dimensions des matrices issues de la SVD de Rr d'après le fichier u1.base donnant les résultats :

```
MATRICES UTILISATEURS Rr : TAILLE DES MATRICES Uc , Sc , Vc
SVD_Rc1 size of U matrix 889249 Liste des valeurs de Sc 943 SVD_Rc1 size of Vs matrix 2722500
MATRICES UTILISATEURS Rr : FORMES DES MATRICES Uc , Sc , Vc
SVD_Rc1 size of U matrix (943, 943) Liste des valeurs de Sc (943,) SVD_Rc1 size of Vs matrix (1650, 1650)
```

3.4 Décomposition en valeur singulière réduite

Pour implémenter le système de recommandation, nous prendrons par la suite en compte un nombre limité de valeurs singulières k. Nous construisons une matrice diagonale notée SS qui comprend k valeur singulières (nombre choisi par l'utilisateur). Puis nous réduisons les matrice U et V en ne prenant en compte que les lignes et les colonnes allant de l'indice 0 à $k-10$ (cela revient au même dans notre cas, puisque les valeurs qu'on omet correspondraient à des 0, et permet d'économiser de la mémoire).

Nous obtenons l'algorithme suivant :

```
def SVD_k_singular(k , U , s, Vs ):
    """Cette fonction crée une sous-matrice carrée
    de U, s et Vs avec les k premières colonnes et lignes.
    Puis retourne UK, SK et VK les matrices de la SVD réduite.
    """
    SS= np.diag(s[:k])
    UK, SK , VK = U[0:k, 0:k], SS , Vs[0:k,0:k]
    return UK, SK, VK
```

```python
def X_Y ( Uk , Sk, Vk) :
    rac_Sk  = np.sqrt(Sk)
    X = np.dot(Uk, rac_Sk)
    Y = np.dot(rac_Sk , Vk)
    return (X , Y)
```

3.5 Résultats

```
DECOMPOSITION EN VALEUR SINGULIERE REDUITE POUR K =20
MATRICES UTILISATEURS (Rc) : TAILLE DES MATRICES UK_c , SK_c , VK_c
Taille de la matrice Uk_c 400 Taille de la matrice des valeurs de Sk_c 400 Taille de la matrice Vs  400

MATRICES UTILISATEURS Rr : FORME DES MATRICES Uk_c , Sk_c , Vk_c
Taille de la matrice Uk_c matrix (20, 20) Liste des valeurs de Sk_c (20, 20) Taille de la matrice Vk_c (20, 20)

DECOMPOSITION EN VALEUR SINGULIERE REDUITE POUR K =20
MATRICES FILMS (Rc) : TAILLE DES MATRICES UK_r , SK_r , VK_r
Taille de la matrice Uk_c 400 Taille de la matrice des valeurs de Sk_c 400 Taille de la matrice Vs  400
```

D'après notre algorithme, la taille des matrices de la SVD réduite dépend de la valeur k choisie. Ainsi nous obtenons 3 matrices carrés $k \times k$.
Avec une approche théorique, et grâce à la formule :

$$U_k S_k V_k' = U S_k V'$$

nous concluons que les tailles et les formes des matrices U_k et S_k et V_k sont respectivement identiques aux matrices U, S et V. En revanche, le rang des matrices varie : il décroit avec k (nombre de valeurs singulières conservées).

3.6 Prédiction

Pour calculer la prédiction, c'est-à-dire une note pour un film et un utilisateur, pour remplacer les coefficients nuls de la matrice R, nous avons réalisé le protocole suivant.
Protocole :

1. Calcul des matrice *features* X et Y grâce à la fonction `X_Y`. La matrice X_k est ainsi calculée telle que $X_k = U_k\sqrt{S_k}$; cette fonction prend comme entrée les matrices réduites U_k et S_k calculées par la fonctions `SVD_k_singular`. De même la matrice Y_k est calculée telle que $Y_k = \sqrt{S_k}V_k'$. Les matrices X et Y sont des matrices carrées $k \times k$. Elles sont remplies de valeurs pouvant être négatives, comme le montre la sortie ci-dessous :

```
----------------------------------------------------------------------
Matrices X et Y pour k= 20
----------------------------------------------------------------------
Première ligne de X : [ -2.05788420e+00   1.29679650e-02  -3.96362242e-04  -1.05656709e-01
  -4.28841383e-03  -5.69585669e-02  -7.88517456e-02  -6.98832076e-02
  -1.63375154e-02  -1.18984536e-01]
Dimension de X :  (10, 10)
Première ligne de Y : [-0.83513453 -0.33624919 -0.19057922 -0.24989876 -0.17285597  0.09007484
 -0.28540715 -0.29521102  0.05874194  0.13837094]
Dimension de Y :  (10, 10)
```

Affichage des matrices X et Y

On peut interpréter les matrices X et Y comme remplies de coefficients caractéristiques d'un utilisateur (matrice X) ou d'un film (matrice Y). La SVD a caractérisé les utilisateurs par une base différente de R donnée par les vecteurs colonne de U, et de même pour les films avec la matrice V'.

2. Calcul de la prédiction. On crée une fonction `predict` qui multiplie une ligne de la matrice X, correspondant aux données d'un utilisateur IDU, et une colonne de la matrice Y (correspondant au film IDM). Cette fonction retourne une valeur p correspondant à la prédiction pour ce couple film-utilisateur. On remarque que ces prédictions sont comprises entre 1 et 5 (comme on pouvait s'y attendre).
3. Ces prédictions seront ensuite calculées pour chaque utilisateur et chaque film lors de la mesure de l'erreur pour calculer la MAE (voir section 4).

```python
def predict (IDU , IDM, X, Y):
    """Calcule la prediction pour un utilisateur IDU et un film IDM grace aux
    matrices X et Y en parametre.
    Retourne cette prediction."""
    p =np.dot(X[IDU,:] , Y[:,IDM])
    return p
```

Fonction de prédiction

```
----------------------------------------------------------------
Prediction Utilisateur , pour k=10
----------------------------------------------------------------
Affichage des valeurs prédites pour k=10
[4.0444778911666477, 3.1713344633628178, 3.0043133855903212, 3.7738035485669168, 3.2586646175457923, 3.399642
9832339, 3.8534212337176266, 3.1731793467037823, 2.9732116876860242, 3.6219438376323367, 3.2903734905388022,
8844431817404188, 3.8317742886574617, 3.2030468091255777, 2.9303383669176992, 3.4694541860553869, 3.256320750
662241, 3.8785888754864644, 3.8870710492103542, 3.1670400693840626, 3.0164632467482093, 3.5691168000465789, 3
69848975079059, 3.8921177776042657, 3.821769174358578, 3.1833926043391179, 3.026892200661599, 3.5611116215016
 3.6775609583882169, 3.8985845259357359, 3.7650359371844724, 3.055756731957692, 2.9402894309413368, 3.398104
764461842, 3.7552248976546854, 3.8348381314321922, 4.0264490718088721, 3.5504383500296015, 2.9925719480598212
 4.464599615645767, 4.2852381425086588, 3.8864869878375878, 3.9091846800457395, 3.2269042862170716, 3.0048577
50355476, 4.0276606120212453, 3.7840929743715161, 3.8372693467544288, 3.9790760464814343, 3.2393790265209814,
.8175070450939899, 4.038434219546704, 3.8270278673467999, 3.8846144368170106, 4.0225723897602652, 3.190628593
65933, 3.9289286286836607, 4.0673670588795989, 3.997657720711671, 3.8910273337022727]
```

Ratings prédits pour $k = 10$

4 Qualité de la prédiction par SVD

Maintenant que nous avons rempli la matrice Utilisateur-Item R, on va chercher à évaluer la justesse de ce remplissage.

4.1 Approche naïve

On va tout d'abord comparer les données réelles au remplissage direct par les moyennes dans les matrices *Rc* et *Rr*. On crée la fonction `compute_MAE`, qui se sert du jeu de données test : pour chaque ligne du fichier original u1.test, cette fonction retrouve l'utilisateur correspondant et le film noté dans la matrice. Puis, en fonction du choix de l'utilisateur, elle recherche les *ratings* dans *Rr* ou *Rc* et les compare aux données réelles. Cette approche nous donne deux résultats pour la MAE qui sont notés `MAEc = 0.826` et `MAEr = 0.850`. On constate que la matrice qui remplit avec les moyennes par film donne une meilleure prédiction.

On peut en déduire qu'un film considéré comme mauvais par beaucoup d'utilisateurs a plus de chances d'être considéré comme mauvais par d'autres ; alors qu'un utilisateur qui donnerait beaucoup de mauvaises notes ne donnera pas forcément des mauvaises notes à tous les films.

```python
def compute_MAE(Robj,testUserItem,choix):
    """Calcule la qualite de la prediction (MAE entre la valeur prédite par R
    (Rc ou Rr) et la vraie valeur de ui.test).
    Prend en arguments : Robj (objet R_matrix),
    testUserItem (dictionnaire des données test)
    et choix (si 'c', on calcule la MAE sur Rc et si 'r', sur Rr).
    Renvoie la MAE."""

    #Initialisation des listes de ratings
    trueRating = []
    prediction=[]

    # Dictionnaires de correspondance coordonnées-identifiants
    cM=Robj.cM
    cU=Robj.cU
    # On remplit les ratings : en faisant attention qu'on fasse
    # référence à un même user et un même film.
    for r in testUserItem:
        mov=r['movie']
        usr=r['user']
        if mov in cM.keys(): # on fait attention que le film soit bien dans
        # notre jeu de donné moyenné (les users y sont tous mais pas les films)
            trueRating.append(r['rating'])
            usr=cU[usr] # on convertit l'id en le no de ligne correspondant
            mov=cM[mov] # pareil
            if choix=="c":
                prediction.append(Robj.Rc[usr,mov])
            elif choix=="r":
                prediction.append(Robj.Rr[usr,mov])
    errorRating = np.asmatrix(prediction) - np.asmatrix(trueRating)
    MAE = np.mean(np.abs(errorRating))
    return MAE
```

Code de la fonction de calcul de la MAE naïve

4.2 Prédictions basées sur la SVD

Ensuite, on évalue les prédictions définies grâce à la SVD par

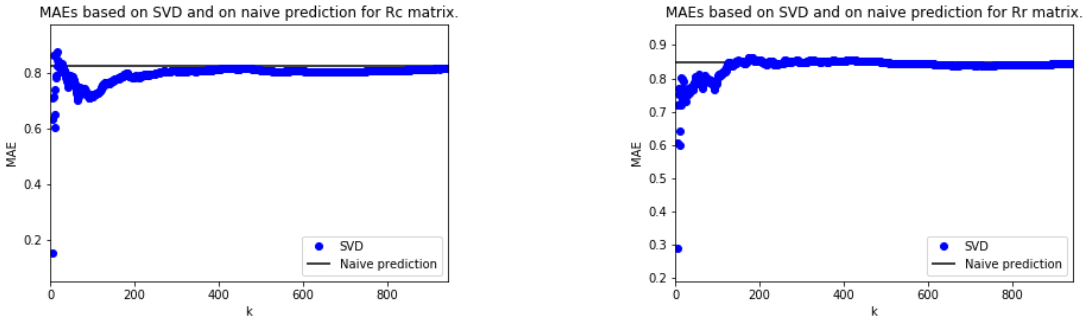
$$p_{ui} = U_k[u,:]\sqrt{S_k}\sqrt{S_k}V'[:,i]$$
$$p_{ui} = X_k[u,:]Y_k'[:,i]$$

On crée une seconde fonction `compute_MAE_SVD`. Cette fonction est basée sur le même principe que la précédente, mais on ne définit alors la prédiction que pour les k premiers films et utilisateurs de la matrice R d'entrée qui sont de plus présents dans le fichier test. Avec l'approche précédente, on calculait une prédiction (et donc une MAE) pour tous les couples Films-Utilisateurs. De plus, pour chaque valeur de k nous obtenons des prédictions différentes et donc des MAE différentes.

On affiche le résultat de cet algorithme sur un graphique grâce à Python pour les matrices Rr et Rc.

```python
def compute_MAE_SVD(Robj ,testUserItem,  X, Y , k):
    """Calcule la MAE de la prediction entre la vraie valeur et ui.test.
    C'est la MAE de la valeur prédite par Rc ou Rr selon les matrices X et Y
    données.
    Prend en arguments : Robj (objet R_matrix), testUserItem (dictionnaire des
    données test), X et Y (matrices de features calculées pour Rc ou Rr) et
    k (nombre de valeurs singulières à garder).
    Renvoie la MAE."""

    #Initialisation des listes de ratings
    trueRating = []
    prediction=[]

    # Dictionnaires de correspondance coordonnées-identifiants
    cM=Robj.cM
    cU=Robj.cU

    # On remplit les ratings
    for r in testUserItem:
        mov=r['movie']
        usr=r['user']
        #print(usr)

        if mov in cM.keys() and (usr < k +1) and (mov <k +1) :
            trueRating.append(r['rating'])
            usr=cU[usr] # on convertit l'id en le no de ligne correspondant
            mov=cM[mov] # pareil
            pred = predict(usr , mov, X, Y)
            prediction.append(pred)

    errorRating = np.ravel(prediction) - np.ravel(trueRating)
    MAE = np.mean(np.abs(errorRating))
    return MAE
```

Code de la fonction de calcul de la MAE basée sur la SVD



Evolution de la MAE en fonction de k pour les SVD basées sur Rc etRr.

Pour les premières valeurs de k, les MAE varient beaucoup : avec peu de données (puisqu'on ne conserve que les k premiers couples), le résultat de la prédiction fluctue beaucoup. En outre, pour les quatre premières valeurs, il n'y a pas de donnée de référence correspondant aux données test (ce qui correspond à des NaN pour les MAE correspondantes).

On considère les MAE pertinentes pour $k \geq 20$ afin de corriger ce problème. Dans ce cas, le code nous donne les résultats suivants :

— Une MAE de 0,703 pour $k = 64$ pour la SVD sur Rc.
— Une MAE de 0,732 pour $k = 26$ pour la SVD sur Rr.

On voit que c'est la décomposition sur Rc (par film) qui donne, là encore, de meilleures prédictions. Mais la meilleure prédiction est obtenue pour un rang k plus élevé. On peut cependant critiquer ce résultat : en effet, les premières MAE calculées pour Rr fluctuent beaucoup.

4.3 Comparaison des deux méthodes pour la MAE

On constate que pour des valeurs de k choisies, la MAE calculée par la SVD donne des meilleures prédictions que la MAE basée sur la prédiction naïve.

	MAE minimale pour la SVD	Rang k	MAE naïve
Rc	0,703	64	0,826
Rr	0,732	26	0,850

Mesure de la qualité des prédictions pour les deux méthodes

La MAE basée sur la SVD est aussi coûteuse en mémoire, car on crée alors deux matrices de *features* en plus de la matrice R pour pouvoir calculer la MAE.

On mesure les temps de calcul avec la fonction `profile.run()`. Pour les MAE naïves, il se situent autour de 0,230 s. Pour les MAE basées sur la SVD, le temps de calcul augmente avec k (il varie de 0,082 s pour k=1 à 0,470 s pour $k = 943$.) A ce temps de calcul s'ajoute le temps pour calculer la SVD (4,925 s pour Rc à 5,099 s pour Rr). De plus, pour trouver la valeur de k optimale, on a dû appeler la fonction de calcul de la MAE de nombreuses fois. L'exécution prend alors plusieurs minutes (environ 6 min).

5 Question théorique

Soit une matrice A à coefficients réels. On veut montrer que les matrices U et V de la SVD associée sont à coefficients réels.

Par définition, on a :

$$U = \begin{bmatrix} \vdots & \vdots & & \vdots \\ u_1 & u_2 & \dots & u_n \\ \vdots & \vdots & & \vdots \end{bmatrix} \quad \text{et} \quad V = \begin{bmatrix} \vdots & \vdots & & \vdots \\ v_1 & v_2 & \dots & v_p \\ \vdots & \vdots & & \vdots \end{bmatrix}$$

Où u_i, v_i sont les vecteurs propres respectifs des matrices AA^* et A^*A.

Démonstration pour U
Soit u un vecteur propre de AA^* et de dimension égale à n . Soit λ la valeur propre de AA^* associée au vecteur u. Comme u est un vecteur propre, il vérifie :

$$\begin{gathered} AA^*u = \lambda u \\ AA^*u - \lambda u = 0_E \\ (AA^* - \lambda I)u = 0_E \\ u \in ker(AA^* - \lambda I) \end{gathered}$$

Où 0_E désigne le vecteur colonne nul de dimension n.

Montrons que les coefficients de u sont réels, c'est-à-dire que ${}^t u = u^*$.

Calculons $(AA^* - \lambda I)^*$.

On sait que AA^* est hermitienne, donc par définition $AA^* = (AA^*)^*$. De plus, ses valeurs propres sont réelles. En outre, la matrice identité est diagonale et à coefficients réels. On a donc $I = I^*$.

Alors

$$(AA^* - \lambda I)^* = (AA^*)^* - (\lambda I)^* = AA^* - \bar{\lambda} I^* = AA^* - \lambda I$$

$AA^* - \lambda I$ est donc hermitienne.

On sait que $(AA^* - \lambda I)u = 0_E$. Calculons $((AA^* - \lambda I)u)^*$.

$$\begin{aligned}((AA^* - \lambda I)u)^* &= 0_E^* \\ u^*(AA^* - \lambda I)^* &= 0_E^* \\ u^*(AA^* - \lambda I) &= 0_E^*\end{aligned}$$

Montrons que ce produit commute. Pour une matrice carrée M et un vecteur v,

$$\begin{aligned}Mv &= \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ \vdots & a_{22} & \dots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \begin{pmatrix} v_1 & \dots & v_n \end{pmatrix} \\ Mv &= \begin{pmatrix} (a_{11}v_1 + \dots + a_{n1}v_n) & \dots & (a_{1n}v_1 + \dots + a_{nn}v_n) \end{pmatrix}\end{aligned}$$

D'autre part,

$$\begin{aligned}vM &= \begin{pmatrix} v_1 & \dots & v_n \end{pmatrix} \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ \vdots & a_{22} & \dots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \\ vM &= \begin{pmatrix} (v_1a_{11} + \dots + v_na_{n1}) & \dots & (v_1a_{1n} + \dots + v_na_{nn}) \end{pmatrix}\end{aligned}$$

Ainsi, pour une matrice carrée M et un vecteur v, $Mv = vM$.

Dans notre cas, la matrice $AA^* - \lambda I$ étant carrée et u^* étant un vecteur, on a $u^*(AA^* - \lambda I) = (AA^* - \lambda I)u^*$.

D'autre part, calculons ${}^t((AA^* - \lambda I)u)$.

$$\begin{aligned}{}^t((AA^* - \lambda I)u) &= {}^t0_E \\ {}^t(AA^* - \lambda I)\,{}^tu &= {}^t0_E\end{aligned}$$

On sait que les valeurs propres de AA^* sont réelles, (en particulier λ). Les coefficients de $AA^* - \lambda I$ sont donc réels par combinaison linéaire de nombres réels. Donc

$$^t(AA^* - \lambda I) = (AA^* - \lambda I)^* = AA^* - \lambda I$$

et

$$\begin{aligned} ^t(AA^* - \lambda I)\,^t u &= {}^t 0_E \\ (AA^* - \lambda I)\,^t u &= {}^t 0_E \end{aligned}$$

De plus, les coefficients (nuls) de 0_E sont réels, donc $^t 0_E = 0_E^*$. On a donc finalement :

$$(AA^* - \lambda I)\,^t u = 0_E^*$$

Donc finalement,

$$\begin{gathered} (AA^* - \lambda I)u^* = (AA^* - \lambda I)\,^t u \\ (AA^* - \lambda I)(u^* - {}^t u) = 0_E^* \\ u^* - {}^t u \in ker(AA^* - \lambda I) \end{gathered}$$

Cela s'écrit en termes matriciels :

$$\begin{gathered} (AA^* - \lambda I)(u^* - {}^t u) = 0_E^* \\ \begin{bmatrix} d_{11} & d_{12} & \dots & d_{1n} \\ \vdots & \vdots & \vdots & \vdots \\ d_{n1} & \dots & \dots & d_{nn} \end{bmatrix} \begin{pmatrix} (\bar{u_1} - u_1) & \dots & (\bar{u_n} - u_n) \end{pmatrix} = 0 \\ \Leftrightarrow \begin{cases} (\bar{u_1} - u_1)d_{11} + (\bar{u_2} - u_2)d_{21} + \dots + d_{n1}(\bar{u_n} - u_n) = 0 \\ \vdots \\ (\bar{u_1} - u_1)d_{1n} + (\bar{u_2} - u_2)d_{2n} + \dots + d_{nn}(\bar{u_n} - u_n) = 0 \end{cases} \end{gathered}$$

Or $AA^* - \lambda I$ est hermitienne : donc elle est inversible. Cela implique que ses vecteurs colonnes sont linéairement indépendants. De plus, comme elle est auto-adjointe, ses vecteurs lignes sont aussi linéairement indépendants. D'où

$$(\bar{u_1} - u_1) = 0, (\bar{u_2} - u_2) = 0, \dots (\bar{u_n} - u_n) = 0$$

On a donc montré que $\bar{u_i} - u_i$ pour tout coefficient de u. Donc u est un vecteur à coefficients réels.

Démonstration pour V

Soit v un vecteur propre de A^*A et la dimension de A^*A égale à p . De même, v est un vecteur propre, il vérifie :

$$\begin{aligned}(A^*A - \lambda I)v = 0_F \\ v \in ker(A^*A - \lambda I)\end{aligned}$$

Où 0_F désigne le vecteur colonne nul de dimension p.

Montrons que ${}^t v = v^*$.

On sait que A^*A est hermitienne. Donc de même, $A^*A - \lambda I$ est hermitienne.

Calculons $((A^*A - \lambda I)v)^*$. On a montré précédemment qu'un vecteur et une matrice carrée commutent, donc :

$$\begin{aligned}((A^*A - \lambda I)v)^* &= 0_F^* \\ v^*(A^*A - \lambda I) &= 0_F^* \\ (A^*A - \lambda I)v^* &= 0_F^*\end{aligned}$$

D'autre part, calculons ${}^t((A^*A - \lambda I)v)$.

$$\begin{aligned}{}^t((A^*A - \lambda I)v) &= {}^t 0_F \\ {}^t(A^*A - \lambda I)\,{}^t v &= {}^t 0_F \\ (A^*A - \lambda I)^*\,{}^t v &= {}^t 0_F \\ (A^*A - \lambda I)\,{}^t v &= {}^t 0_F\end{aligned}$$

De plus, comme précédemment, ${}^t 0_F = 0_F^*$.

On a donc finalement :

$$(A^*A - \lambda I)v^* = (A^*A - \lambda I)\,{}^t v$$

Donc

$$\begin{aligned}(A^*A - \lambda I)(v^* - {}^t v) = 0_F^* \\ v^* - {}^t v \in ker(A^*A - \lambda I)\end{aligned}$$

$A^*A - \lambda I$ étant hermitienne, comme montré pour AA^* avec u, ses vecteurs ligne sont aussi linéairement indépendants. D'où

$$(\bar{v_1} - v_1) = 0, (\bar{v_2} - v_2) = 0, \dots (\bar{v_p} - v_p) = 0$$

On a donc montré que v est de même un vecteur à coefficients réels.