

# Mini-projet : un système de recommandation

17 octobre 2014

## 1 Introduction

La plupart des sites web (comme amazon, youtube, Google, ...) adaptent leur contenu à l'utilisateur qui s'y connecte en leur offrant, par exemple, des recommandations personnalisées.

Voici quelques exemples de recommandations typiques issues de `amazon.fr` :


**amazon.fr** Chez Barbara Promotions Chèques-cadeaux Vendre Aide

Parcourir les boutiques ▼ Rechercher Toutes nos boutiques ▼ Go


Chez vous Votre page à votre image Recommandé pour vous Affinez nos conseils personnalisés Votre profil Plus d'informations

### Votre Amazon.fr

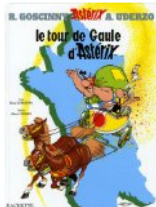
#### Livres




Harry Potter, II ...  
J. K. Rowling  
★★★★☆ (99)  
EUR-8,00 **EUR 7,60**  
Pourquoi est-ce recommandé ?



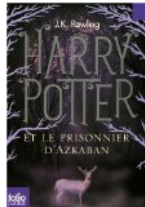
Astérix - Le combat ...  
René Goscinny  
★★★★☆ (11)  
EUR-9,90 **EUR 9,41**  
Pourquoi est-ce recommandé ?



Astérix - Le tour de ...  
René Goscinny  
★★★★☆ (17)  
EUR-9,90 **EUR 9,41**  
Pourquoi est-ce recommandé ?



Astérix - Astérix ...  
René Goscinny  
★★★★☆ (13)  
EUR-9,90 **EUR 9,41**  
Pourquoi est-ce recommandé ?



Harry Potter, III ...  
J. K. Rowling  
★★★★☆ (93)  
EUR-8,70 **EUR 8,27**  
Pourquoi est-ce recommandé ?

► Voir toutes les recommandations dans Livres

Afin de pouvoir faire des recommandations, ces systèmes ont besoin de prévoir les préférences des utilisateurs. Ceci se fait au moyen de ce que l'on appelle des **systèmes de recommandation**.

Ces systèmes ont pour rôle de prédire les "préférences" ou "notes" que l'utilisateur attribuerait à tel ou tel article. Ils sont extrêmement répandus de nos jours et on peut les retrouver dans une large palette d'applications dont les plus emblématiques sont sans doute celles liées au cinéma, à la musique, aux nouvelles, aux livres et articles scientifiques, aux requêtes de recherches sur internet, aux tags sociaux et, de façon plus générale à une pléthore d'articles commerciaux.

On trouvera aussi des systèmes de recommandations ciblant des domaines d'expertise spécifiques, le monde de la finance et des assurances, celui de la restauration, celui des sites de rencontres en ligne ou des abonnés Twitter et même celui des blagues et autres bons mots.

Le but de ce projet est d'implémenter un algorithme, appelé **UV-Decomposition**, qui peut être utilisé comme composant d'un système de recommandation visant à prédire les préférences des utilisateurs.

## 2 Un modèle de système de recommandation – Matrice des préférences

Un système de recommandation manipule deux types d'entités : des **utilisateurs** et des **articles**. Les utilisateurs ont des **préférences** pour des articles. Le but est de parvenir à deviner ces préférences en partant d'un ensemble de données connues.

Les données manipulées par le système de recommandation sont usuellement stockées dans une matrice des préférences, appelée dans la littérature anglophone «**Utility Matrix**», et que nous désignerons sous ce vocable dans le reste de ce document.

La «**Utility Matrix**» stocke pour chaque paire utilisateur-article une valeur représentant ce qui est connu du degré de préférence que l'utilisateur a pour l'article. Les valeurs sont choisies dans un ensemble ordonné exprimant les préférences. Par exemple le nombre d'étoiles associées à un film pourra être codifié au moyen d'entiers allant de 1 à 5. Ces préférences peuvent être données par l'utilisateur lui-même (lorsqu'il décide d'attribuer une note à un film par exemple).

Lorsque l'utilisateur n'exprime aucune préférence explicitement, l'entrée correspondante aura une valeur particulière équivalente à **inconnu** (on pourrait choisir zéro dans le cas des films par exemple). Dans le cas général, ces matrices seront **creuses** : seules quelques valeurs sont différentes de **inconnu**, car l'utilisateur n'exprimera de préférence explicite que pour un faible nombre d'articles.

### Exemple

Voici un exemple de « Utility Matrix » stockant les notes attribuées par cinq utilisateurs, Anton, Berta, César, Dora, et Emil à sept films : X-Men 1, 2, 3 et Pirates des Caraïbes 1, 2, 3, et 4.

Utilisateurs, Films	X-Men 1	X-Men 2	X-Men 3	Pirates 1	Pirates 2	Pirates 3	Pirates 4
Anton	4				2	3	
Berta	5	5	4				
César				5	4		
Dora				4			5
Emil		2			1		

Un système de recommandation a pour but d'inférer les entrées inconnues (ici laissées blanches). Comment César apprécierait-il Pirates de Caraïbes 3 ? Ou encore à quel film qu'il n'a pas encore vu attribuerait-il la meilleure note ?

Il existe fort peu d'informations dans cette matrice. On pourrait donc penser à faire intervenir des propriétés caractérisant les films, comme les acteurs principaux, le réalisateur, le genre du film, voire des similarités dans le titre avec d'autres films. Ceci permettrait par exemple de découvrir des similarités entre les Pirates des Caraïbes 1-4 et de conclure que César, qui a aimé le 1 et le 2, a aussi de fortes chances de donner une bonne note au 3 par exemple.

L'algorithme que vous allez implémenter dans ce projet, appelé algorithme **UV-decomposition**, a pour but d'inférer automatiquement de telles propriétés caractéristiques (« **features** ») à partir des données connues.

Plus précisément, cet algorithme part du point de vue que la « Utility Matrix » est en fait le résultat du produit de deux longues et fines matrices, les matrices  $U$  et  $V$ , dont la première décrit la relation entre l'utilisateur et un certain nombre de propriétés et dont la seconde fait le lien entre les propriétés et les articles. Ce point de vue fait sens si l'on considère qu'il y a un nombre relativement faible de propriétés et qu'il existe des utilisateurs déterminant la réaction de la plupart des autres utilisateurs à la plupart des articles.

### 3 UV-decomposition

L'algorithme UV-decomposition prend en entrée une « Utility Matrix »,  $M_{m \times n}$ , et une dimension  $d$  (nombre de propriétés). Il a pour but de trouver une décomposition de  $M$  en deux matrices  $U_{m \times d}$  et  $V_{d \times n}$  dont le produit  $P_{m \times n} = U_{m \times d} \cdot V_{d \times n}$  est similaire à  $M$  pour toutes les entrées non-nulles de cette matrice.

#### 3.1 Définition de la "similarité"

Idéalement, nous souhaiterions que toutes les entrées non-nulles de  $P$  soient aussi proches/similaires que possible des entrées correspondantes de  $M$ . Pour mesurer concrètement cette similarité nous utiliserons le choix typique du RMSE (**Root-Mean-Square-Error**). Il faudra pour obtenir cette mesure :

1. calculer pour toute entrée non nulle de  $M$ , le carré de sa différence avec l'entrée correspondante de  $P$  ; et cumuler ces valeurs dans une somme ; c'est à dire :

$$S = \sum_i \sum_j \begin{cases} (m_{ij} - p_{ij})^2 & \text{if } m_{ij} \neq 0 \\ 0 & \text{if } m_{ij} = 0 \end{cases}$$

2. calculer la moyenne de ces carrés en divisant la somme précédente par le nombre de termes ayant permis son calcul (c-à-d le nombre d'entrées non-nulles) :

$$S_{\text{mean}} = \frac{S}{\sum_i \sum_j \begin{cases} 1 & \text{if } m_{ij} \neq 0 \\ 0 & \text{if } m_{ij} = 0 \end{cases}}$$

3. et prendre la racine carrée de cette moyenne :

$$RMSE = \sqrt{S_{\text{mean}}}$$

Notre but est désormais de trouver les matrices  $U$  et  $V$  de sorte à ce que le « RMSE » entre  $P$  et  $M$  soit faible. C'est ce que fait l'algorithme « UV-decomposition » en améliorant itérativement chaque élément dans  $U$  et  $V$ .

#### 3.2 Ajustement d'un élément unique

Étant données les matrices  $M_{n \times m}$ ,  $U_{n \times d}$ , et  $V_{d \times m}$ , nous utiliserons les notations  $m_{ij}$ ,  $u_{ij}$ , et  $v_{ij}$ , respectivement, pour référencer les éléments de ces matrices. En analysant avec soin comment un élément donné influence le RMSE, il est possible de déduire les deux formules suivantes. Elles permettent de calculer des valeurs améliorées de  $u_{rs}$  dans  $U$  et  $v_{rs}$  dans  $V$  réduisant le RMSE entre  $M$  et  $P = U \cdot V$ .

$$u'_{rs} = \frac{\sum_j v_{sj} \cdot (m_{rj} - \sum_{k \neq s} u_{rk} \cdot v_{kj})}{\sum_j v_{sj}^2}$$

$$v'_{rs} = \frac{\sum_i u_{ir} \cdot (m_{is} - \sum_{k \neq r} u_{ik} \cdot v_{ks})}{\sum_i u_{ir}^2}$$

Nous utilisons les notations abrégées suivantes dans les formules ci-dessus :

- $\sum_j$  désigne la somme sur les  $j, j = 1, \dots, m$  telle que  $m_{rj}$  n'est pas inconnue.
- $\sum_i$  désigne la somme sur les  $i, i = 1, \dots, n$  telle que  $m_{is}$  n'est pas inconnue
- $\sum_{k \neq s}$  désigne la somme sur les  $k, k = 1, \dots, d$ , exceptant  $k = s$ .
- $\sum_{k \neq r}$  désigne la somme sur les  $k, k = 1, \dots, d$ , exceptant  $k = r$ .

### 3.3 Application des ajustements

Trouver la décomposition UV ayant le plus faible RMSE implique de commencer par des matrices  $U$  et  $V$  arbitrairement choisies ; puis d'ajuster itérativement les éléments de  $U$  et  $V$  de sorte à réduire le RMSE. Nous ne nous intéresserons qu'à l'ajustement de valeurs uniques, même si en principe il est possible de faire des choix d'ajustement plus complexes. L'ajustement des valeurs se fera au moyen des formules de la Section 3.2.

Quels que soient les ajustements réalisés, il existera typiquement de nombreux minima locaux ; c'est à dire concrètement des matrices  $U$  et  $V$  dont aucun ajustement possible ne réduit le RMSE. Il se trouve que seul un de ces minima locaux sera le minimum global (les matrices  $U$  et  $V$  donnant le plus faible RMSE possible). Une façon possible d'augmenter les chances de trouver un minimum global consiste à prendre plusieurs points de départ différents ; c'est à dire effectuer différents choix pour les matrices initiales  $U$  et  $V$ . **Il est important de noter qu'il ne sera jamais garanti que la meilleure solution trouvée soit effectivement le minimum global.**

**Mettre en oeuvre l'optimisation du RMSE** Afin d'atteindre un minimum local en partant de valeurs de départ pour  $U$  et  $V$ , nous pouvons ajuster les valeurs de ces matrices itérativement jusqu'à atteindre le résultat voulu. A chaque itération (étape/cycle d'optimisation) nous ajusterons l'ensemble des valeurs de  $U$  et  $V$  une par une. Nous devons pour cela définir un ordre selon lequel seront visités les éléments de  $U$  et  $V$  à chaque étape.

La façon la plus simple de procéder consiste à traverser les matrices  $U$  et  $V$  ligne par ligne par exemple ; en visitant les éléments de façon circulaire (« round-robin strategy »). Notez que le fait qu'un élément ait déjà été optimisé ne signifie pas qu'il ne sera plus possible de lui trouver une meilleure valeur par la suite ; c'est à dire une fois que d'autres éléments auront à leur tour été améliorés. Il est par conséquent nécessaire de répéter les ajustements fait à un élément donné jusqu'à ce qu'il y ait des raisons suffisantes de penser qu'il n'est plus possible d'obtenir une nouvelle amélioration.

Il existe évidemment d'autres stratégies de sélection des éléments à améliorer. Il est possible par exemple de sélectionner ces éléments de façon aléatoire. Afin de garantir dans ce cas que chaque élément sera considéré à chaque étape, on peut partir d'une permutation aléatoire des éléments. Permutation qu'il faudra parcourir en séquence à chaque étape.

**Convergence vers un minimum** Idéalement, le RMSE devrait devenir nul à un moment donné. Nous saurions à ce moment avec certitude qu'il ne pourrait être amélioré davantage.

En pratique, comme il existe généralement beaucoup plus d'éléments connus dans  $M$  qu'il n'y a d'éléments dans  $U$  et  $V$  réunis, il y a peu de chances d'atteindre un RMSE nul. Il nous faut alors détecter la situation où les chances d'améliorer le RMSE, en revisitant les éléments de  $U$  et/ou de  $V$ , deviennent suffisamment faibles pour être ignorées.

Il est possible par exemple de décider de s'arrêter lorsque l'amélioration du RMSE au terme d'un cycle d'optimisation descend au dessous d'un certain seuil. Il est aussi possible de considérer l'amélioration d'éléments particuliers et de décider d'arrêter le processus lorsque l'amélioration maximale atteinte pour ces éléments descend au dessous d'un certain seuil.

### 3.4 Algorithme UV-Decomposition

Dans les sections précédentes nous avons présenté les différentes étapes permettant de chercher une décomposition de la « Utility Matrix ».

L'algorithme UV-Decomposition complet doit inclure les étapes suivantes :

1. **Pre-traitement de la matrice  $M$**  : afin de prendre en compte (i) la différence entre les habitudes de notation des utilisateurs (par exemple certains seront habituellement plus généreux et attribueront de meilleures notes que d'autres) ainsi que (ii) la différence de qualité entre les articles (par exemple, certains films auront une note moyenne bien meilleure que les autres), l'algorithme UV-decomposition incorpore habituellement une étape de pre-traitement des données d'entrée. Dans un souci de simplification, nous ne mettrons pas en oeuvre cette étape dans le mini-projet mais nous référons ceux

d'entre vous que cela intéresse aux sections 9.3.1 et 9.4.5 du livre "Mining of Massive Datasets" <http://www.mmds.org/> (Version 1.0).

2. **Initialisation de  $U$  et  $V$**  : un point de départ simple consiste à donner la même valeur (par exemple  $v = 1$ ) à tous les éléments.

Un choix judicieux pour cette valeur  $v$  sera tel que les éléments de la matrice produit  $U \cdot V$  atteignent la moyenne des éléments non nuls de  $M$ . Cette valeur  $v$  peut être obtenue en calculant d'abord la moyenne de tous les éléments non nuls de  $M$  puis en divisant cette moyenne par  $d$  (nombre d'articles) et en prenant la racine carrée de cette valeur :

$$v = \sqrt{\frac{\sum_{ij} m_{ij}}{d}}, \text{ où } \sum_{ij} \text{ désigne la somme sur tous les } i, j = (1, 1), \dots \text{ telle que } m_{ij} \text{ est non-nulle.}$$

Comme mentionné précédemment, pour augmenter nos chances de converger vers un minimum global, il est bon de calculer la décomposition en partant de différents points de départ.

Il est possible d'obtenir différents points pour  $U$  et  $V$  en perturbant la valeur  $v$  aléatoirement et de façon indépendante pour chacun des éléments. Dans le cadre de ce mini-projet, vous choisirez quelques points de départ différents et retournerez les résultats relatifs au point ayant permis d'atteindre le plus faible RMSE.

S'il vous intéresse d'en connaître d'avantage sur la sélection de bons points de départs, vous pouvez lire la Section 9.4.5 du livre "Mining of Massive Datasets" <http://www.mmds.org/> (Version 1.0).

3. **Ajustement de  $U$  et  $V$  et décision de quand arrêter** : voir la Section 3.3

## 4 Détails d'implémentation et tâches

Nous vous fournissons dans ce qui suit les entêtes des méthodes principales à implémenter. Vous pouvez bien sûr en créer d'autres afin de rendre votre code lisible et modulaire.

### 4.1 Premiers pas

Nous vous fournissons un fichier source `Recommendation.java` incluant des variables (où stocker vos noms et scipers) ainsi que des entêtes de méthodes à compléter.

Afin de compiler ce fichier dans Eclipse procédez comme suit :

1. **Ouvrir Eclipse**
2. **Créer un nouveau projet** : `New -> Java Project`. Dans la fenêtre de dialogue qui s'ouvre indiquez un nom (par exemple `MiniProject`) et cliquez sur `Finish`.
3. **Créer un paquetage appelé `assignment`** : dans le `Package Explorer` (par défaut sur la gauche) sélectionner le dossier `src` du projet nouvellement créé puis `New -> Package`. Dans la fenêtre de dialogue `New Package` insérez `assignment` en guise de nom de paquetage puis cliquez sur `Finish`.
4. **Ajouter le fichier `Recommendation.java` dans le paquetage `assignment`** : le moyen le plus simple est de faire un «drag&drop» depuis le gestionnaire de fichier dans le `Package Explorer` dans `src`. Sélectionnez alors le bouton `Copy files` et cliquez OK.
5. **Commencer à travailler** vous pouvez alors ouvrir le fichier copié et commencer à travailler dessus.

**IMPORTANT** : n'oubliez pas de mettre à jour (`NAME1`, `SCIPER1`, `NAME2`, `SCIPER2`) avec vos noms et scipers (ces champs permettent l'affichage de données utiles lors des tests automatiques effectués pour contrôler votre code).

Contrôler que le fichier compile après ces changements effectués.

## 4.2 Représentation de M, U, et V.

Pour représenter les matrices  $M$ ,  $U$  et  $V$ , vous utiliserez des tableaux bi-dimensionnels. Comme par exemple :

---

```
public class Recommendation {  
    static double[][] M = {  
        { 11, 0, 9, 8, 7 },  
        { 18, 0, 18, 18, 18 },  
        { 29, 28, 27, 0, 25 },  
        { 6, 6, 0, 6, 6 },  
        { 17, 16, 15, 14, 0 }  
    };  
  
    static double U[][] = { {1,1}, {1,1}, {1,1}, {1,1}, {1,1} };  
    static double V[][] = { {1,1,1,1,1}, {1,1,1,1,1} };  
}
```

---

## 4.3 Représentation de la matrice sous la forme d'une String

Implémentez la méthode `matrixToString` dans le fichier `Recommendation.java`. Cette méthode prendra en paramètre une matrice sous la forme d'un tableau bi-dimensionnel et retournera une représentation de la matrice sous la forme d'une `String` respectant la syntaxe utilisée pour l'initialisation d'un tableau en Java (voir l'exemple ci-dessous).

**Signature :**

---

```
public static String matrixToString(double[][] A)
```

---

**Exemple :**

`matrixToString(M)` devrait créer la chaîne de caractère suivante :

---

```
{  
    {11.0,0.0,9.0,8.0,7.0},  
    {18.0,0.0,18.0,18.0,18.0},  
    {29.0,28.0,27.0,0.0,25.0},  
    {6.0,6.0,0.0,6.0,6.0},  
    {17.0,16.0,15.0,14.0,0.0}  
};
```

---

La chaîne créée par votre méthode peut différer un peu de cet exemple au niveau du nombre de décimale après la virgule (11.000 ou 11.00 seront tout autant valables l'un que l'autre), des espaces ou des sauts de ligne. La seule contrainte à respecter est que le format soit conforme à la syntaxe d'initialisation Java. Utiliser votre `String` pour initialiser un tableau dans un programme Java ne devrait causer aucune erreur de compilation.

---

```
double[][] testMatrix = {  
    {11.0,0.0,9.0,8.0,7.0},  
    {18.0,0.0,18.0,18.0,18.0},  
    {29.0,28.0,27.0,0.0,25.0},  
    {6.0,6.0,0.0,6.0,6.0},  
    {17.0,16.0,15.0,14.0,0.0}  
};
```

---

---

## 4.4 Contrôle de la matrice

Implémentez la méthode `isMatrix` dans le fichier `Recommendation.java`. Cette méthode doit contrôler si la matrice fournie en entrée est valide ; c'est à dire qu'elle n'est pas vide (de taille nulle ou valant `null`), qu'aucune de ses lignes n'est vide et que toutes ses lignes sont de même longueur. Si la matrice est valide, la méthode `isMatrix` retournera `true`, sinon, elle retournera `false`.

Signature :

---

```
public static boolean isMatrix( double[] [] A )
```

---

Exemples :

`isMatrix(M)` devrait retourner `true` pour la matrice `M` donnée en exemple pour la tâche précédente.  
`isMatrix(T)` devrait retourner `false` pour `T = {{ 1, 0, 2},{ 0, 1}}`

## 4.5 Multiplication de matrices

Implémentez la méthode `multiplyMatrix` dans le fichier `Recommendation.java`. Cette méthode retourne le produit des deux matrices fournies en paramètre. On rappelle qu'étant données deux matrices  $A_{m \times d}$  et  $B_{d \times n}$ , l'élément  $p_{ij}$  (ligne  $i$  et colonne  $j$ ) de leur produit se calcule comme suit :

$$p_{ij} = \sum_{x=1}^d a_{ix} \cdot b_{xj}$$

Signature :

---

```
public static double[] [] multiplyMatrix(double[] [] A, double[] [] B)
```

---

Exemple :

`multiplyMatrix(A,B)` devrait retourner `C={{ 3, 2},{ 1, 1}}`; pour

---

```
A = {{ 1, 0, 2},
      { 0, 1, 1}};
B = {{ 1, 2},
      { 0, 1},
      { 1, 0}};
```

---

## 4.6 Création d'une matrice test

Implémentez la méthode `createMatrix` dans le fichier `Recommendation.java`. Cette méthode prend en entrée (i) deux entiers  $n$  et  $m$  représentant les dimensions d'une matrice et deux entiers  $k$  et  $l$  représentant une plage de valeurs. Cette méthode doit retourner une matrice avec  $n$  lignes et  $m$  colonnes et dont chaque élément doit être une valeur générée aléatoirement entre les bornes  $k$  et  $l$ , valeur des bornes comprises (voir indications plus bas). Appeler la méthode deux fois à la suite avec les mêmes arguments doit donc produire

deux matrices différentes (pour peu qu'il existe plus d'une matrice avec les plages de données spécifiées). Si  $m = 0$ ,  $n = 0$ , ou  $k > l$ , alors la méthode retournera `null`.

**Signature :**

---

```
public static double[][] createMatrix( int n, int m, int k, int l)
```

---

**Exemple :**

`createMatrix(2,3,0,5)` pourrait retourner la matrice :  
`{{1.5363025048384755, 3.7091406870511334, 3.124007650197829},`  
`{2.7009435847554135, 4.648070599408801, 0.251046533936522}}`. Le résultat n'est évidemment pas unique puisque les valeurs auront été générées aléatoirement.

### Méthodes Java pour la génération de nombres aléatoires

Exemple	Fonctionnalité
<code>Random random = new Random();</code>	Initialiser un générateur aléatoire
<code>random.nextInt(100);</code>	Une valeur entière entre 0 (inclu) et 100 (exclu)
<code>random.nextDouble();</code>	Une valeur "double" entre 0.0 (inclus) et 1.0 (exclu)

Notez que vous pouvez utiliser `createMatrix` pour créer vos propres jeux de données

## 4.7 RMSE

Implémentez la méthode `rmse` dans le fichier `Recommendation.java`. Cette méthode prendra en entrées deux matrices `M` et `P` (représentées au moyen de tableaux bi-dimensionnels) et calculera le RMSE entre ces matrices. Le calcul se fera pour tous les éléments non nuls de `M` et se fera tel que décrit dans la Section 3.1). Si les matrices sont de dimensions différentes, la méthode `rmse` retournera `-1`.

**Signature :**

---

```
public static double rmse(double[][] M, double[][] P)
```

---

**Exemples :**

`rmse(M,P)` avec `M = {{ 1, 0, 0},{ 0, 1, 1}}` et `P = {{ 1, 0, 2},{ 0, 1, 1}}` devrait retourner 0.0.

`rmse(M,P)` avec `M = {{ 1, 0, 1},{ 0, 1, 1}}` et `P = {{ 1, 0, 2},{ 3, 1, 1}}` devrait retourner 0.5.

`rmse(M,P)` avec `M = {{ 1, 0, 1},{ 0, 1, 1}}` et `P = {{ 1, 0},{ 3, 1}}` devrait retourner `-1`.

## 4.8 Mise à jour d'un élément

Implémenter les méthodes `updateUElem` et `updateVElem` dans le fichier `Recommendation.java`. La méthode `updateUElem` (ou `updateVElem`) prend en paramètre : (i) les trois matrices `M`, `U`, et `V` et (ii) deux indices `r` et `s` et retourne une nouvelle valeur pour l'élément  $u_{rs}$  (ou  $v_{rs}$ , respectivement) tel que le RMSE entre `M` et le produit de `U` et `V` décroît (voir la Section 3.2 pour le calcul d'une valeur améliorée pour  $u_{rs}$  et  $v_{rs}$ ).



### Signatures :

---

```
public static double updateUElem( double[][] M, double[][] U, double[][] V, int r, int s )
public static double updateVElem( double[][] M, double[][] U, double[][] V, int r, int s )
```

---

### Exemples :

Étant données les instances suivantes de M, M1, U, et V,

- la valeur de retour de `updateUElem(M, U, V, 0, 0)` devrait être entre 6.749 et 6.751,
- la valeur de retour de `updateVElem(M, U, V, 0, 0)` devrait être entre 7.099 et 7.101,
- la valeur de retour de `updateUElem(M1, U, V, 0, 0)` devrait être entre 5.999 et 6.001, et
- la valeur de retour de `updateVElem(M1, U, V, 0, 0)` devrait être entre 7.749 et 7.751.

---

```
double[][] M = {
    { 11, 0, 9, 8, 7 },
    { 18, 0, 18, 18, 18 },
    { 29, 28, 27, 0, 25 },
    { 6, 6, 0, 6, 6 },
    { 17, 16, 15, 14, 0 }
};
double[][] M1 = {
    { 0, 8, 9, 8, 7 },
    { 18, 0, 18, 18, 18 },
    { 29, 28, 27, 0, 25 },
    { 6, 6, 0, 6, 6 },
    { 17, 16, 15, 14, 0 }
};
double[][] U = { {2,2}, {2,2}, {2,2}, {2,2}, {2,2} };
double[][] V = { {1,1,1,1,1}, {1,1,1,1,1} };
```

---

## 4.9 Optimisation de la matrice

Implémentez les méthodes `optimizeU` et `optimizeV` dans le fichier `Recommandation.java`. La méthode `optimizeU` (ou `optimizeV`) prend en paramètre les trois matrices M, U, et V et retourne une nouvelle matrice U (ou V, respectivement) minimisant le RMSE entre M et P. Voir la Section 3.3 pour une discussion sur (i) l'ordre de sélection et de mise à jour des éléments de U (ou V, respectivement) et sur (ii) quand l'optimisation doit être stoppée. **Notez qu'aucune de ces questions n'attend comme réponse une solution optimale.** Par conséquent nous évaluerons votre implémentation en testant si nous pouvons améliorer le RMSE entre M et  $U \cdot V$  de plus de  $10e^{-6}$  en appliquant `updateUElem` (resp. `updateVElem`) à tous les éléments de U (ou V, respectivement).

### Signatures :

---

```
public static double[][] optimizeU( double[][] M, double[][] U, double[][] V)
public static double[][] optimizeV( double[][] M, double[][] U, double[][] V)
```

---

### Exemples :

Comme la sortie dépend des stratégies investiguées, il n'y a pas d'exemples fournis pour cette tâche.

## 4.10 Recommandation

Implémentez la méthode `recommend` dans le fichier `Recommendation.java`. Cette méthode prendra en paramètre la matrice `M`, une dimension  $d$  (définissant les tailles de  $U$ ,  $V$ ; c'est à dire que si  $M$  est de taille  $n \times m$ , alors  $U$  sera de taille  $n \times d$  et  $V$  de taille  $d \times m$ ). Elle retournera un tableau d'entiers indiquant à la position  $i$ , la meilleure recommandation de l'utilisateur  $i$ . Un article ne sera recommandé que si (i) il n'était pas noté par  $i$  (l'entrée correspondante dans  $M$  valait zéro au départ) et (ii) qu'il est recommandé par l'algorithme UV-decomposition utilisant la dimension  $d$  pour  $M$  (cet article a le plus haut score parmi ceux qui n'était pas notés au départ). S'il n'y a pas de tel article la valeur retournée sera  $-1$  pour l'utilisateur  $i$ .

Voir la Section 3.4 pour plus de détails sur comment implémenter l'algorithme générant les recommandations.

**Signature :**

---

```
public static int[] recommend(double[][] M, int d)
```

---

Vous noterez que les méthodes `createMatrix` (voir la tâche 4.6) et `multiplyMatrix` (voir la tâche 4.5) vous permettront de créer autant de jeux de données que souhaité pour tester votre système de recommandation. Rappelons que le but est de trouver les matrices  $U$  et  $V$ , de sorte à ce que leur produit  $P$  soit similaire à une matrice  $M$  (pour les entrées non-nulles). Nous pouvons donc créer deux matrices  $U_T$  et  $V_T$ , et les multiplier pour obtenir  $P_T$ , qui sera notre solution de référence. Nous pouvons ensuite créer une nouvelle matrice  $M_T$ , identique à  $P_T$  mais avec certaines entrées mises à zéro (équivalent de "inconnu").  $M_T$  sera alors la matrice prise en entrée de notre algorithme de recommandation, qui, dans le cas idéal, recommanderait les éléments de la même façon que  $P_T$ . Rappelez-vous à ce sujet que notre algorithme de décomposition peut être bloqué au niveau d'un minimum local et que le cas idéal ne se présentera pas dans le cas général.  $M_T$  se rapprochera donc de  $P_T$  sans toutefois l'atteindre exactement,

**Bonus :**

Lisez le chapitre 9.4 ("Dimensionality Reduction") du livre en ligne "Mining of Massive Datasets" <http://www.mmids.org/> et proposez des améliorations de votre système de recommandation.

## 5 Le Challenge Netflix

En 2006 Netflix, le plus grand fournisseur américain de diffusion en flux continu de films et de séries sur Internet a offert un prix de \$1'000'000 à la première personne ou équipe capable de faire des recommandations 10% meilleures que leur propre système de recommandation.

Le 21 Septembre 2009 le prix a été attribué à l'équipe « BellKor's Pragmatic Chaos », une collaboration de chercheurs dirigée par Chris Volinsky et Robert Bell (deux chercheurs des laboratoires AT&T). Pour plus de détails sur ce challenge et l'équipe gagnante voir <http://www.netflixprize.com/>.

Pour ceux d'entre vous que cela intéresse, nous pouvons fournir une partie des données de test utilisées par le challenge.

**Remarque importante :** Il existe évidemment de nombreuses façons de créer un bon système de recommandation. La plupart des systèmes existants combinent plusieurs approches complémentaires. Nous ne faisons ici qu'explorer une petite facette de cet univers. L'un des grands défis pour les systèmes de recommandation est notamment la gestion de données massives (« Big Data »). C'est un point que nous n'aborderons pas du tout dans le cadre de ce mini-projet.