

Projet MDI 343  
Systèmes de recommandation

Nicolas Keriven et Jean-Baptiste Alayrac

4 mai 2014

---

## Table des matières

<b>Introduction</b>	<b>2</b>
<b>1 Présentation du problème et notations</b>	<b>3</b>
1.1 Différentes méthodes . . . . .	3
1.2 Factorisation de matrice et notations . . . . .	4
<b>2 Algorithmes</b>	<b>5</b>
2.1 Descente de gradient stochastique . . . . .	5
2.2 Algorithme parallélisé JELLYFISH . . . . .	6
<b>3 Résultats</b>	<b>9</b>
3.1 Performance des deux approches . . . . .	9
3.2 Résultats sur la base de données de films . . . . .	11

## Introduction

Le challenge Netflix a mis en avant la problématique de créer de bons systèmes de recommandations. A l'heure où l'on est en mesure d'agréger énormément d'informations sur les utilisateurs, leurs préférences, leurs attitudes, il est légitime de se demander s'il est possible de traiter ces données afin d'améliorer l'expérience de l'utilisateur en lui proposant des produits qui vont lui plaire.

Dans ce contexte beaucoup de méthodes ont été mises à l'épreuve afin de créer le système de recommandation le plus performant. Les techniques de factorisation de matrice se sont démarquées et ont notamment permis à l'équipe *BellKor's Pragmatic Chaos* de remporter le prix d'un million de dollar décerné par Netflix [1].

On s'intéresse ici aux méthodes mathématiques et algorithmiques qui permettent de traiter efficacement un grand ensemble de notations afin de prédire les préférences d'utilisateurs pour des films.

Dans un premier temps nous présenterons le problème et son cadre mathématique. Nous introduirons ensuite les algorithmes que nous avons implémenté afin de résoudre numériquement le problème. Enfin nous commenterons nos résultats en gardant en tête l'objectif de recommandation.

# 1 Présentation du problème et notations

Le cadre général de la problématique de recommandation est de relier des utilisateurs à des produits. Dans cette partie nous présentons tout d'abord les méthodes les plus utilisées actuellement pour les système de recommandations avant de nous concentrer sur la méthode que nous avons approfondie.

## 1.1 Différentes méthodes

Actuellement, il existe deux principales approches afin de créer un système de recommandation. La première approche est appelée *content filtering* alors que la seconde est nommée *collaborative filtering*

### *Content filtering* (ou filtrage par contenu)

Cette méthode vise à caractériser la nature de chaque utilisateur ainsi que de chaque produit afin d'en dégager un profil le plus précis possible. Pour un utilisateur, cela peut être son pays d'origine, son âge... Pour un produit, par exemple un film, cela peut être son genre, son box-office... Les algorithmes ont alors pour tâche d'associer des profils d'utilisateurs avec des profils de produits. Le projet Music Genome Project, utilisé notamment dans la radio Pandora.com utilise de telles méthodes afin de proposer du contenu aux utilisateurs. Chaque musique est alors caractérisée par des centaines d'attributs assimilable à des "gènes".

### *Collaborative filtering* (ou filtrage collaboratif)

L'autre méthode se base essentiellement sur des avis passés (implicites ou explicites) d'utilisateurs à propos de produits. La méthode de *collaborative filtering* analyse les liens qui existent entre utilisateurs et produits via la connaissance de votes passés afin de proposer de nouvelles associations utilisateur/produit. L'avantage principal de cette méthode par rapport à celle de *content filtering* est qu'ici il n'y a nul besoin de connaître le produit ou les utilisateurs à priori. Par contre cette méthode connaît le problème du *cold start*, i.e., elle est incapable de traiter l'arrivée d'un nouveau produit ou d'un nouveau utilisateur. Afin de la faire fonctionner il faut posséder un nombre déjà conséquents d'avis de chaque utilisateurs et pour chaque produits.

Dans ce contexte de *collaborative filtering* il existe à nouveau deux principales méthodes. La première est connue sous le nom de *neighborhood method*, la deuxième se concentre autour de *latent factor models*.

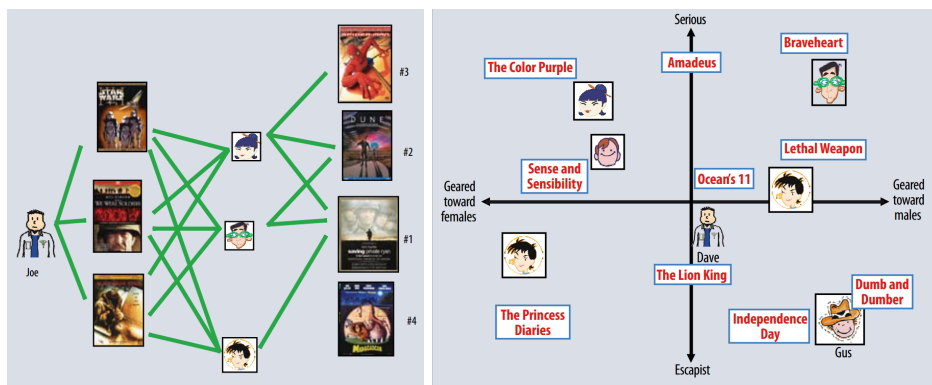


FIGURE 1 – A gauche : *neighborhood methods*, A droite : *Latent factor models*

**Neighborhood methods** La principale idée est d'essayer de modéliser les relations qui existent entre les utilisateurs d'une part ou les produits d'autre part. Partons de l'approche produit. Supposons qu'on veuille prédire la note que va mettre un utilisateur à un nouveau produit A. On va alors regarder comment cet utilisateur a noté les produits identifiés comme étant des produits voisins de A, afin de prédire la note qu'il va mettre sur ce nouvel objet. Sur la gauche de la Figure 1, on a représenté l'approche utilisateur où l'on a d'abord trouvé un voisinage de l'utilisateur Joe afin de lui conseiller de nouveaux films (dans ce cas le premier film conseillé serait *Il faut sauver le soldat Ryan*).

**Latent factor model** Cette dernière approche est celle que nous avons choisi d'étudier plus en détails. L'idée derrière cette méthode est d'essayer de déterminer des *facteurs latents* qui seront les dimensions d'un espace dans lequel on pourra plonger à la fois les utilisateurs et les produits. Cet espace sera directement déduit de la structure des votes qui est le lien entre utilisateurs et produits. L'idée ensuite est de pouvoir rassembler un utilisateur et un film qui seront proches dans cet espace.

Ces facteurs latents peuvent être parfois interprétables. Par exemple une dimension pourront représenter le fait pour un film d'être une comédie. Les points maximaux seront alors considérés comme des comédies alors que les points minimaux pourront être des drames. La position d'un utilisateur sur cet axe témoignera alors de la propension de celui-ci à aimer ce genre.

Cependant parfois ces facteurs ne sont pas interprétables directement, ce qui donne une certaine richesse au modèle. Cette méthode permet en effet de détecter des tendances qui n'aurait pas été prévisible à la main.

Sur la droite de la Figure 2 on a représenté cette approche pour 2 facteurs latents. De cela on peut prédire que l'utilisateur en haut à droite du graphique va certainement adorer le film *Braveheart* alors qu'il va probablement détester le film *The Princess Diaries*.

## 1.2 Factorisation de matrice et notations

Nous allons voir qu'une méthode afin de calculer ces facteurs latents est de factoriser la partie connue de la matrice de notes  $\mathbf{M}$ . On rappelle que  $M_{u,i}$  est la note qu'a mis l'utilisateur  $u$  au produit  $i$  ( $u$  pour *user* et  $i$  pour *item*). Dans le cas où l'on a  $n_u$  utilisateurs et  $n_i$  produits cette matrice est donc de taille  $n_u \times n_i$ . Le gros obstacle est que cette matrice de notation est parcimonieuse, en effet bien souvent on ne possède qu'un certain nombre de produits pour lequel l'utilisateur a voté. Dans la suite on note  $\Omega$  le set d'indices pour lequel la notation est connue. On a donc  $|\Omega| \ll n_u \times n_i$ .

L'idée va alors être de trouver une matrice  $\mathbf{X}$  qui va être *proche dans un certain sens* de l'information contenue dans la matrice  $\mathbf{M}$  et qui va pouvoir s'écrire de manière factorisé :

$$\mathbf{X} = LR$$

où  $L \in \mathbb{R}^{n_u \times r}$  et  $R \in \mathbb{R}^{r \times n_i}$  où  $r$  est le nombre de facteurs latents que l'on recherche. Ainsi la ligne  $u$  (resp. la colonne  $i$ ) de la matrice  $L$  (resp.  $R$ ) va représenter la position dans l'espace des facteurs latents de l'utilisateur  $u$  (resp. produit  $i$ ). Le produit scalaire de ces deux vecteurs va donc représenter la propension de l'utilisateur  $u$  à aimer le produit  $i$ .

Maintenant essayons de décrire de manière plus mathématique la notion de *proche dans un certain sens de l'information contenue dans  $\mathbf{M}$* . Il va donc falloir déterminer une fonction de coût  $f$  d'attaches aux données sur la matrice  $\mathbf{X}$  en vue de se ramener à un problème classique d'optimisation. Une première idée serait de se dire que l'on pourrait écrire  $f$  comme étant la moyenne empirique des écarts quadratiques entre  $X_{ui}$  et la note  $M_{ui}$ . On aurait alors :

$$f(\mathbf{X}) = \frac{1}{|\Omega|} \sum_{(u,i) \in \Omega} \|X_{ui} - M_{ui}\|^2$$

Cependant, avec cette représentation, on ne prend pas en compte le biais naturel qui existe dans nos données. En effet il est important de pouvoir modéliser le fait que certains utilisateurs sont plus sévères que d'autres pour noter les produits ou encore que certains produits sont parfois beaucoup mieux noté que la moyenne. Sans prendre en compte cela il devient difficile de modéliser les notes par un simple produit scalaire comme nous voulions le faire plus haut. Au lieu de représenter directement la note obtenue, on va préférer que le produit scalaire  $X_{ui}$  représente l'écart à la note obtenue compte tenu du biais  $b_{ui}$  que nous connaissons sur nos données :

$$b_{ui} = \mu + b_u + b_i$$

où  $\mu$  est la moyenne totale des notes,  $b_u$  est la moyenne des notes décernés par l'utilisateur  $u$ , et  $b_i$  est la moyenne des notes obtenues par le produit  $i$ . Tel quel  $b_{ui}$  est déjà une sorte d'estimateur de la note que va mettre l'utilisateur  $u$  au produit  $i$ . On verra dans la suite qu'il peut être intéressant de comparer nos résultats à ce système de recommandation très naïf. Ainsi on va réécrire notre fonction de coût d'attache aux données :

$$f(\mathbf{X}) = \frac{1}{|\Omega|} \sum_{(u,i) \in \Omega} \|X_{ui} + \mu + b_u + b_i - M_{ui}\|^2$$

Finalement notre problème de détermination de facteur latents peut s'écrire de la manière suivante :

$$\text{minimiser } \frac{1}{|\Omega|} \sum_{(u,i) \in \Omega} \|X_{ui} + \mu + b_u + b_i - M_{ui}\|^2 + P(\mathbf{X}) \quad (1)$$

où  $P$  est une fonction de régularisation qui permet de contrôler la complexité de la matrice  $\mathbf{X}$ . Dans la suite nous allons décrire et comparer les algorithmes que nous avons utilisé afin de résoudre ce problème.

## 2 Algorithmes

### 2.1 Descente de gradient stochastique

Lorsqu'on s'expose à des sets de données de très grande taille, le temps de calcul et la complexité de l'algorithme d'optimisation deviennent des contraintes que l'on ne peut pas négliger [2] [3]. Dans ce contexte, l'algorithme de descente de gradient stochastique (noté SGD dans la suite) présente de nombreux avantages que ce soit du point de vue de la complexité mais aussi du point de vue de la performance.

Dans un souci de généralisation de notation nous remplaçons l'équation (1) par l'équation suivante, en supposant que la fonction de régularisation  $P$  peut se décomposer en une somme sur les éléments de  $\mathbf{X}$  (ce qui sera le cas en pratique) :

$$\text{minimiser } \frac{1}{|\Omega|} \sum_{(u,i) \in \Omega} l(X_{ui}, M_{ui}) + P(X_{ui}) \quad (2)$$

#### Descente de gradient "usuelle"

Les algorithmes de gradient classique calculent à chaque étape  $t$  le gradient complet de la fonction objectif par rapport à  $\mathbf{X}_t$  et mette ensuite à jour  $\mathbf{X}_{t+1}$  en suivant différentes méthodes. L'une des plus efficaces en terme de convergence est la méthode de Newton qui sous certaines hypothèses parvient à une convergence quadratique.

Cependant lorsque  $|\Omega|$  devient très grand (ici 1 million voire 10 millions) il devient très coûteux de faire les étapes de calculs précédentes. On préférera alors une méthode plus simple comme l'algorithme SGD.

### Descente de gradient stochastique

Au lieu de calculer le gradient total à chaque itération de l'algorithme on va simplement calculer le gradient par rapport à un exemple (donc un couple  $(u, i)$ ) tiré aléatoirement. L'étape de mise à jour devient alors la suivante :

$$X_{ui}^{t+1} = X_{ui}^t - \gamma_t (\nabla_X l(X_{ui}, M_{ui}) + \nabla_X P(X_{ui})) \quad (3)$$

En se ramenant au cas particulier de l'équation (1), on voit que l'étape de mise à jour peut s'exprimer comme une mise à jour sur la  $u$ -ème ligne  $L_u$  de la matrice  $L$  et sur la  $i$ -ème colonne  $R_i$  de la matrice  $R$  (en supposant que  $P$  peut se décomposer sur  $L$  et  $R$  ce qui sera le cas en pratique) :

$$L_u \leftarrow L_u + \gamma_t (e_{ui} R_i^* - \nabla_L P_L(L_u)) \quad (4)$$

$$R_i \leftarrow R_i + \gamma_t (e_{ui} L_u^* - \nabla_R P_R(R_i)) \quad (5)$$

où  $e_{ui} = M_{ui} - \mu - b_u - b_i - L_u R_i$  est l'erreur de prédiction.

Le processus stochastique des exemples tirés au cours des itérations de l'algorithme est donc fortement corrélé à la manière dont on tire les exemples aléatoirement. Ici on espère que chacune des contributions (équation (3)) se comportent comme la moyenne (descente de gradient usuelle), et ce malgré le bruit introduit par ces tirages.

### Tirage avec remise

Ici les tirages sont fait avec remise. De la sorte, en espérant que l'ensemble d'apprentissage est assez riche au sens de la distribution qu'il vise à représenter, on peut considérer qu'à chaque itération, les exemples sont tirés en suivant la véritable distribution des données. Intuitivement, on s'attend alors bien à ce que l'agrégation de toutes ces gradients ponctuels (3) se comportent comme la moyenne (gradient total).

Des résultats de convergences ont été prouvés en supposant que le pas d'incrément  $\gamma_t$  décroît suffisamment au cours des itérations. Le théorème de Robbins-Siegmund (1971) permet notamment de prouver la convergence presque sure de la méthode avec des hypothèses assez lâches.

Dans ce qui suit nous présentons une méthode qui permet de paralléliser cet algorithme afin de gagner du temps de calcul. On verra aussi que la méthode de tirage des exemples diffère de la méthode présentée ici. Nous appelons cette première méthode Simple Gradient Stochastique (SGD), et on s'efforcera de comparer ces deux approches.

## 2.2 Algorithme parallélisé JELLYFISH

Dans [4], les auteurs introduisent une version parallélisée de l'algorithme SGD, en remarquant qu'un grand nombre d'opérations sont indépendantes et peuvent être réalisées simultanément sans affecter le déroulement de l'algorithme et sans conflit. Cet algorithme, baptisé JELLYFISH, est essentiellement similaire à l'algorithme SGD et réalise successivement des opérations atomiques de descente de gradient selon une suite d'indices  $(u_k, i_k)$ . Outre l'implémentation, la seule différence entre les deux algorithmes repose dans la manière dont ces indices sont tirés.

### Tirage sans remise

Les équations de mise à jour (4) et (5) pour un couple d'indices  $(u, i)$  affectent uniquement la  $u$ -ème ligne de  $L$  et la  $i$ -ème colonne de  $R$ . Ainsi, deux de ces opérations pour des couples  $(u, i)$  et  $(u', i')$  tels que  $u \neq u'$  et  $i \neq i'$  sont totalement indépendantes : elles n'affectent pas les mêmes coefficients et peuvent être réalisées dans n'importe quel ordre, voire simultanément. C'est sur ce principe que repose l'algorithme JELLYFISH [4]. Pour cela, le tirage des indices se fait *sans remise*, contrairement à l'algorithme du gradient stochastique simple, et nous appellerons *époque* le fait de parcourir une fois l'ensemble des données. Cet algorithme, similaire au gradient stochastique avec remise, a déjà été introduit sous le nom de descente de (sous-)gradient "incrémental" [5]. Dans notre contexte, le tirage sans remise est plus une nécessité pratique qu'un véritable choix théorique : il est nécessaire de connaître à chacune de ces époques la séquence complète des indices à traiter afin de pouvoir paralléliser les opérations qui peuvent l'être.

Un véritable tirage sans remise se ferait via la sélection, avec une loi uniforme, d'une permutation sur  $\Omega$ . Cependant, une telle permutation peut rapidement devenir encombrante en mémoire et gloutonne en temps de calcul. Les auteurs de JELLYFISH choisissent donc de tirer une permutation  $\pi_u$  sur les utilisateurs et une permutation  $\pi_i$  sur les items, et de sélectionner la permutation sur les couples  $(\pi_u, \pi_i)$ . Il est évident que l'on est loin de la loi uniforme : par exemple, en admettant que tous les utilisateurs et tous les items sont présents dans  $\Omega$ , il y a  $(n_u n_i)!$  permutations possibles sur  $\Omega$ , tandis qu'il existe seulement  $n_u! n_i!$  de la forme  $(\pi_u, \pi_i)$ , ce qui est bien moindre. Si en pratique on s'attend à ce que cela ne fasse pas une grande différence, cela rend l'éventuelle analyse théorique d'un tel algorithme relativement subtile.

### Description de l'algorithme

Comme décrit dans la section précédente, à chaque époque l'algorithme tire uniformément une permutation  $\pi_u$  sur les utilisateurs et une permutation  $\pi_i$  sur les items. Suivant ces permutations, les données  $(u, i) \in \Omega$  sont ensuite rangées dans des *blocs*  $C_{a,b}$ ,  $1 \leq a, b \leq p$  de manière à ce qu'entre deux blocs  $C_{a,b}$  et  $C_{a',b'}$  tels que  $a \neq a'$  et  $b \neq b'$  toutes les opérations soient indépendantes, c'est-à-dire que pour tout  $(u, i) \in C_{a,b}$  et  $(u', i') \in C_{a',b'}$ , on a  $u \neq u'$  et  $i \neq i'$ . Pour cela, on pose simplement pour chaque couple  $(u, i) \in \Omega$  :

$$a = \left\lfloor \frac{p}{n_u} (\pi_u(u) - 1) \right\rfloor + 1 \quad b = \left\lfloor \frac{p}{n_i} (\pi_i(i) - 1) \right\rfloor + 1 \quad (6)$$

Au sein de chaque bloc, une étape de descente de gradient (3) est réalisée sur chaque coordonnée. Le tirage sans remise est induit par les permutations : il est possible de traiter les données directement dans l'ordre dans lequel elles ont été "rangées" dans les blocs lorsque l'on parcourt  $\Omega$  et que l'on applique (6).

Il est possible de traiter en parallèle jusqu'à  $p$  blocs ne s'intersectant pas en une *étape*, puis de réaliser  $p$  étapes afin de traiter toutes les données. Les blocs sont par exemple groupés selon les  $p$  diagonales cycliques (Fig. 2). Ainsi, il est naturel de fixer  $p$  au nombre maximum de processus pouvant être lancé (nombre de cœurs d'un processeur pour un ordinateur seul).

### Tirage sans remise, le retour

Théoriquement, le tirage sans remise n'est pas à préférer au tirage avec remise. Ce dernier possède une vitesse théorique de convergence optimale [4] [3], et lors d'un tirage sans remise une époque entière ne garantit nullement une plus grande proximité de la solution qu'après une seule étape de gradient [5]. Cependant, plusieurs raisons portent à croire qu'un tirage sans remise est préférable pour des problèmes de taille importante. La plupart des analyses théoriques de l'algorithme de descente de

$$\begin{bmatrix} L_1 \\ L_2 \\ L_3 \end{bmatrix} \begin{bmatrix} R_1 & R_2 & R_3 \end{bmatrix} = \begin{bmatrix} L_1 R_1 & \overline{L_1 R_2} & \overline{L_1 R_3} \\ L_2 R_1 & L_2 R_2 & \overline{L_2 R_3} \\ \overline{L_3 R_1} & L_3 R_2 & L_3 R_3 \end{bmatrix}$$

FIGURE 2 – Partition cyclique des blocs (pour question de simplicité, ici  $\pi_u$  et  $\pi_i$  sont les permutations identités). Les blocs surlignés de la même manière sont traités en parallèle. Schéma issu de [4].

gradient stochastique (avec remise) montrent que les bornes d’optimalité ne font pas intervenir la taille des données, ce qui est avantageux car adapté aux problèmes de grande taille, mais également limitant, le bruit résultant des choix aléatoires réalisés ne pouvant être contourné [3] [5]. Il est possible qu’en pratique un tirage sans remise évite certains de ces écueils. Intuitivement, une passe sur l’intégralité des données requiert en moyenne  $n \log n$  tirages avec remise. Sur un problème de petite taille, le facteur  $\log n$  est tout à fait négligeable, mais lorsque  $n$  se compte en millions, cela peut résulter en un algorithme des dizaines de fois plus lent ; le temps d’exécution s’accroît d’un ordre de grandeur non-négligeable.

Nous notons toutefois qu’il existe une version de l’algorithme extrêmement similaire [6], où le tirage se fait avec remise à l’intérieur de chaque bloc.

### Détails d’implémentation

De manière similaire au papier original, nous dévouons un processus entier à la permutation et l’ordonnancement des données. Il est lancé en parallèle des processus calculant le gradient à l’époque courante, et créé une copie des données permutées pour l’époque suivante (afin d’éviter les conflits de lecture). Ainsi, à tout instant deux copies des données sont gardées en mémoire.

Le papier original [4] incite à dévouer plusieurs processus à cette permutation, quitte à accroître la mémoire utilisée. Cependant cela demande un niveau de contrôle de cette mémoire et de granularité quelque peu en dehors du cadre de ce projet – la fusion des résultats des différents processus doit se faire intelligemment et rapidement, or de manière générale la mémoire partagée entre processus doit être pré-allouée et ne supporte pas les structures mutables comme les listes, etc. De plus, les auteurs dévouent deux processus sur douze à la permutation, et en pratique nous n’avons pas accès à d’aussi gros processeurs. Nous gardons donc dans la plupart des cas un seul processus alloué à cette permutation.

Cependant, dû à la non-mutabilité des données en mémoire partagée et au fait que la taille de chaque bloc dépend des permutations à chaque époque, l’ordonnancement des données se fait en deux étapes de temps de calcul à peu près égaux ; pour calculer la taille de chaque blocs puis pour ranger les données. Ainsi, il est possible de diviser cet ordonnancement en deux processus : à l’époque  $n$ , un processus calcule les permutations et la taille des blocs pour l’époque  $n + 2$ , un processus ordonne les données pour l’époque  $n + 1$ , et les autres processus calculent les gradients. Selon le nombre de cœurs de la machine, il peut être possible de préférer affecter deux cœurs à l’ordonnancement des données, cependant lors de nos expériences sur nos machines relativement limitées, nous avons constaté que le calcul des gradients était systématiquement plus long pour des  $r$  raisonnables, ainsi nous n’utilisons pas cette configuration.



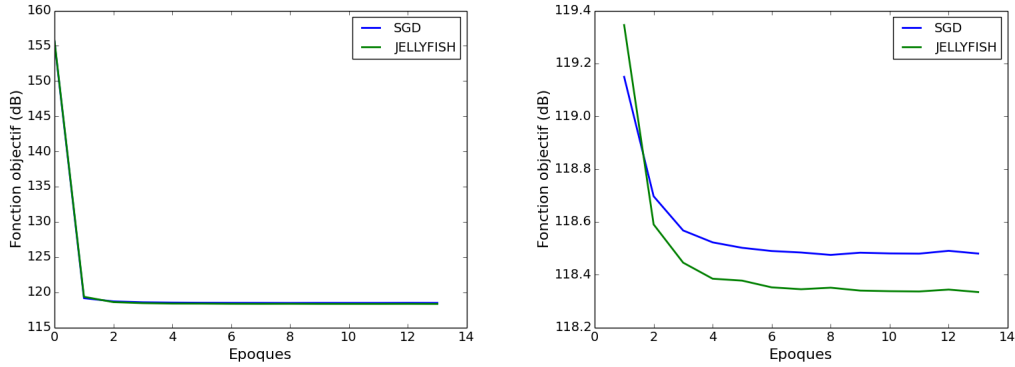


FIGURE 3 – Fonction objectif pour les deux algorithmes. A droite, la première valeur (avant le déroulement des algorithmes) a été supprimée, permettant de mieux apprécier les différences entre les deux courbes, qui restent toutefois minimes.

### 3 Résultats

Nous présenterons nos résultats sous deux aspects. La première partie concernera la comparaison entre les deux algorithmes précédents en termes de précision par rapport à la fonction objectif et en termes de temps de calcul. On reviendra aussi sur la différence entre le tirage avec remise et sans remise.

Dans la deuxième partie nous commenterons en détails les résultats obtenus sur la base de données du GroupLens Research<sup>1</sup>. La plupart des résultats seront présentés sur la base de données qui comprend 1 million de notations de 6000 utilisateurs pour 4000 films, tandis que quelques derniers tests seront réalisés sur la plus grande base de données, comprenant 10 millions de notations pour 72000 utilisateurs et 10000 films.

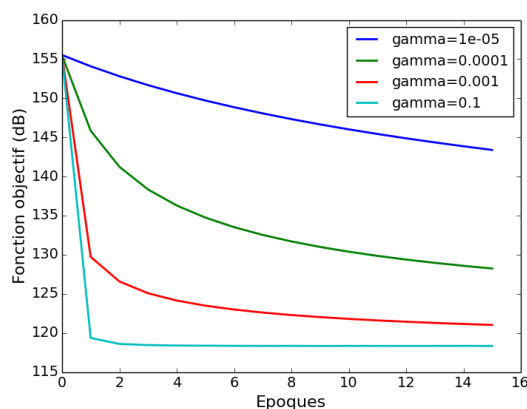
#### 3.1 Performance des deux approches

##### Fonction objectif

Dans un premier temps, nous comparons la convergence des deux algorithmes en examinant la fonction objectif (1). Puisque JELLYFISH ne permet de calculer cette fonction qu'entre chaque époque, nous n'enregistrons également cette valeur qu'aux itérations multiples de  $|\Omega|$  pour l'algorithme simple SGD. Rappelons que la seule différence entre les deux algorithmes est alors le tirage des indices, avec ou sans remise. Ces résultats sont présentés Figure 3. On constate que, toutes proportions gardées, il y a extrêmement peu de différence entre les deux approches, même en échelle logarithmique. Néanmoins, si l'on supprime la toute première valeur de la fonction objectif (avant l'exécution des algorithmes) et que l'on agrandit les courbes, JELLYFISH obtient des performances légèrement meilleures ; dans le cas présent il parvient même à rattraper son retard après une époque. Ce comportement a été systématiquement observé dans toutes les simulations, et contredit empiriquement l'avantage théorique donné au tirage avec remise.

Il est également intéressant d'examiner la vitesse de convergence en fonction du pas de gradient  $\gamma$  (Fig. 4). Comme attendu, plus celui-ci est grand, plus la convergence est rapide. Néanmoins, il faut bien entendu être prudent : passé une certaine valeur (environ 0.4 dans nos simulations), l'algorithme ne converge plus et souvent les valeurs explosent à l'infini.

1. <http://grouplens.org/datasets/movielens/>

FIGURE 4 – Convergence de JELLYFISH en fonction de  $\gamma$ .

### Temps de calcul

Dans la figure 5, nous comparons le temps d'exécution par époque des deux algorithmes, en fonction du nombre de processus  $p$  pour JELLYFISH. Les simulations ont été réalisées sur un processeur Intel i7 49-30K possédant 6 cœurs (12 en *hyper-threading*). Pour comparaison, nous indiquons également le temps d'exécution du processus d'ordonnancement des données. Nous constatons qu'en effet sur nos simulations il n'est pas nécessaire de dévouer plus d'un processus à cette tâche, ce n'est pas le facteur limitant. Concernant JELLYFISH, un minimum est atteint pour  $p = 5$ , correspondant à 6 processus parallèles (en comptant le processus de permutation), le nombre de cœurs réels du processeur. De manière surprenante, l'*hyper-threading* ne permet pas de réduire le temps de calcul. Nous supposons que le gain de temps que ce procédé apporterait est plutôt minime à ces échelles de temps, et compensé par les opérations auxiliaires : création du processus, assignation des variables etc. Enfin, on constate que même pour  $p = 1$  JELLYFISH est bien plus rapide que SGD ; cela est dû au fait qu'à chaque itération SGD doit tirer un indice aléatoire puis aller chercher en mémoire les données correspondantes, tandis que JELLYFISH parcourt linéairement les données permutées, ce qui permet des accès mémoire bien plus rapides (utilisation du cache, etc.). Cela est probablement également dû au langage utilisé, PYTHON, et à ses diverses optimisations lors de parcours simples de tableaux, listes, etc. Un langage plus bas-niveau permettrait de mieux quantifier ces différences.

Les tests ont également été réalisés sur la base à 10 millions de notations. Le temps d'exécution est tout de même supérieur à une simple multiplication par 10 du temps précédent, la quantité de mémoire à gérer étant plus importante les accès mémoire sont moins rapides. En particulier, le temps que demande l'ordonnancement des données ne s'accroît pas linéairement, et il peut devenir avantageux de dédier deux processus à cette tâche, selon les différents  $r$ ... Nous n'avons pas eu accès suffisamment longtemps à la machine capable de gérer cette base de données importante pour pouvoir conclure définitivement.

En résumé, JELLYFISH est de 5 à 6 fois plus rapide que SGD pour des résultats équivalents, sur la base à 1 million d'entrées. Pour 20 époques et  $r = 50$ , il demande environ 4-5 minutes sur la base à 1 million d'entrée et un peu plus d'une heure et 15 minutes sur la base à 10 millions. Nous n'avons pas testé SGD sur la grande base de données, cependant puisque le temps dédié aux accès mémoire ne s'accroît pas linéairement et que SGD en possède un grand nombre non-optimisés, nous supposons que l'algorithme SGD est encore considérablement plus lent.

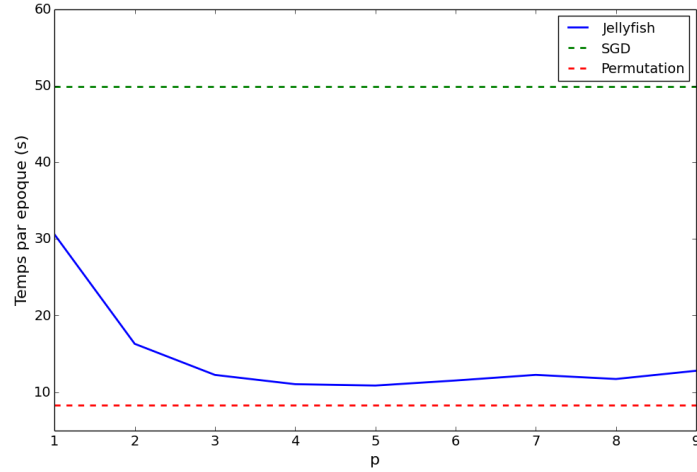


FIGURE 5 – Temps d'exécution par époque en fonction de  $p$ , pour  $r = 30$ . Le temps d'exécution d'une époque de SGD est indiqué, ainsi que le temps pris par le processus de permutation des données.

### 3.2 Résultats sur la base de données de films

Le premier point important est de constituer une base de test afin d'évaluer notre algorithme. Lors de la constitution de cette base de tests, il faut prendre soin de ne pas retirer toutes les notes d'un même utilisateur ou toutes les notes d'un même film. Dans le cas contraire on perd toute l'information sur cet utilisateur ou sur ce film. La stratégie que nous avons adopté est de retirer  $n$  notes par utilisateurs en espérant que toutes les notes d'un même film ne soient pas retirées (une vérification avant chaque test est nécessaire). En pratique  $n = 1, 2$ , ce qui nous permet d'avoir une base de test assez conséquente.

#### Critère d'évaluation

**RMSE** Afin d'évaluer la pertinence de notre algorithme nous avons regardé plusieurs facteurs. Le premier critère chiffré est le RMSE (pour *Root Mean Square Error*) :

$$\text{RMSE} = \sqrt{\frac{1}{|\Omega_{\text{test}}|} \sum_{(u,i) \in \Omega_{\text{test}}} |X_{ui} + b_{ui} - M_{ui}|^2}$$

Par rapport à la simple erreur moyenne, ce critère permet de pénaliser plus les erreurs qui sont vraiment loin de la valeur attendue. On accepte avec une bonne tolérance qu'un algorithme se trompe d'un point sur une note de 5, par contre on sera totalement insatisfait si l'algorithme se trompe de 3 points. Le rapport de mécontentement n'étant pas linéaire, ce critère est plus pertinent que la simple erreur moyenne.

Nous avons aussi choisi ce critère parce qu'il est le critère d'évaluation du challenge Netflix. Pour information sur la base de test Netflix le RMSE avec le prédicteur naïf par la moyenne est de 1.02. Avant la compétition Netflix parvenait à un RMSE de 0.95. Après plusieurs années de compétition le RMSE est passée à 0.85. Il est important de garder cela en tête lorsque nous comparerons les résultats de notre algorithme avec ceux du prédicteur naïf. Ce critère sera donc utile pour comparer différents paramètres, ou la performance des deux algorithmes étudiés ici.

Sur notre base de test, le RMSE "naïf" est de 0.9598 sur la base à 1 million d'entrées, et 0.9062 sur la base à 10 millions d'entrées.

**Histogramme d'erreur** Afin de visualiser mieux ce que peut représenter le chiffre précédent, nous avons affiché l'histogramme des erreurs en comparant un résultat de notre algorithme avec le prédicteur naïf par la moyenne. Un exemple d'un tel histogramme est donnée en Figure 6.

**Affichage des facteurs latents** Enfin un dernier critère d'évaluation a été d'essayer de visualiser la pertinence attendue des facteur latents. Pour cela nous avons reproduits des figures similaires à la Figure 1, en représentant des films en deux dimensions répartis aux extrêmes dans l'espace des facteurs latents. Cependant il faut d'ores et déjà savoir que tout les facteurs latents ne seront pas facilement interprétables en termes de genre cinématographique, de box office...

### Résultats

Dans le Tableau 1 on donne le RMSE obtenu pour différentes valeurs de  $r$ . On rappelle que  $r$  est le rang de la matrice  $X$ , ou encore la dimension de l'espace des facteurs latents. On rappelle aussi que ce résultat est à comparer au RMSE naïf qui vaut 0.9598 pour la base à 1 million d'entrées, et 0.9062 sur la base à 10 millions d'entrées.

Dimension $r$	10		50		100		
Algorithme	SGD	JELLYFISH	SGD	JELLYFISH	SGD	JELLYFISH	Moyenne (naïf)
RMSE (1M)	0.9146	0.9123	0.9114	0.9099	0.9042	0.8997	0.9598
RMSE (10M)	X	0.8698	X	0.8652	X	0.8574	0.9062

TABLEAU 1 – Valeurs du RMSE pour différentes valeurs de  $r$

En Figure 6, on donne l'histogramme des erreurs obtenu pour le plus bas RMSE. En bleu, on visualise les prévisions de notre algorithme d'optimisation et en vert on peut voir les prédictions de la moyenne simple ( $\mathbf{X} = 0$ ).

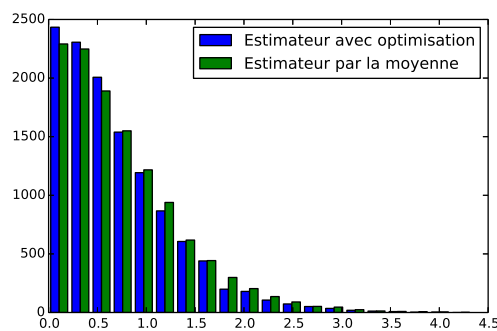


FIGURE 6 – Histogramme d'erreur

En Figure 7 on donne pour chaque note (comprise entre 1 et 5) la proportion de notes où notre algorithme ne s'est pas trompé pour le meilleur RMSE. En comparaison on donne aussi la prédiction

de l'algorithme naïf. Ce graphique permet une visualisation un peu plus parlante des résultats de notre algorithme en vue de créer un système de recommandation.

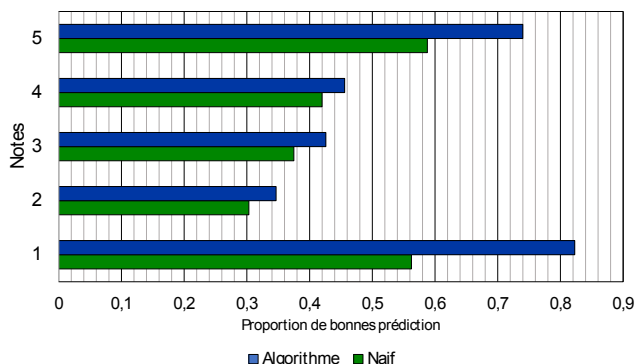


FIGURE 7 – Proportion de bonnes prédiction en fonction de chaque note

### Espace des facteur latents

Nous voyons dans le tableau 1, que la performance de notre algorithme augmente avec  $r$ , du moins au début (nous n'avons pas testé avec des  $r$  trop grand mais le risque ensuite devient l'overfitting). Il est alors intéressant de se pencher sur les espaces engendrés par notre algorithme.

### Commentaires

Dans cette partie nous apportons un regard critique sur les résultats présentés précédemment en essayant de replacer cela dans le contexte de la recommandation.

Comme nous pouvons le voir dans le tableau 1, notre algorithme parvient à battre la méthode naïve qui reviendrait à prédire n'importe quelle note par la moyenne des notes passées. De plus dans le but de réaliser un algorithme de recommandation on a envie que notre algorithme prédise de manière efficace surtout les notes extrêmes afin, d'une part de proposer des produits à l'utilisateur dont on est sûr qu'ils vont plaire, et d'autre part d'être en mesure de bien prévoir ce que l'utilisateur va détester. En ce sens le graphique de la Figure 7 est intéressant parce qu'on voit que notre algorithme est bien plus efficace que la méthode naïve pour ce qui est de prédire les notes extrêmes. Pour la note 5, nous avons 74 % de bonnes prédictions avec notre méthode contre 59 % pour la méthode naïve. Pour la note 1 nous avons 82 % de bonnes prédictions avec notre approche contre 56 % pour la prédiction par la moyenne. En ce sens nous pouvons dire que l'approche étudiée ici fonctionne assez bien pour faire de la recommandation. Il semblerait donc que l'on réussisse à capter de l'information sur les goûts et les préférence des utilisateurs.

Cependant on est aussi en droit de se demander si, du point de vue de l'utilisateur, l'amélioration de prédiction est réellement intéressante. En reprenant l'exemple du projet Netflix, on peut noter que la bonne prédiction de notes au dessus de 3 étoiles est de 98.9 % pour l'algorithme qui a décroché le prix de 1 million de dollar contre 94.4 % pour l'algorithme naïf. On peut alors se demander si l'amélioration de l'expérience de l'utilisateur est vraiment intéressante. Beaucoup de progrès reste encore à faire afin de comprendre et de modéliser les goûts humains. Cette idée permet de relativiser quant aux résultats modestes de nos algorithmes.

## Conclusion

## Références

- [1] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8) :30–37, August 2009.
- [2] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In Yves Lechevallier and Gilbert Saporta, editors, *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'2010)*, pages 177–187, Paris, France, August 2010. Springer.
- [3] Léon Bottou and Olivier Bousquet. The tradeoffs of large scale learning. In *IN : ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 20*, pages 161–168, 2008.
- [4] Benjamin Recht and Christopher Ré. Parallel Stochastic Gradient Algorithms for Large-Scale Matrix Completion. *submitted*, 2011.
- [5] Angelia Nedic and Dimitri Bertsekas. Convergence rate of incremental subgradient algorithms. In *Stochastic Optimization : Algorithms and Applications*, pages 263–304. Kluwer, 2000.
- [6] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In Chid Apté, Joydeep Ghosh, and Padhraic Smyth, editors, *KDD*, pages 69–77. ACM, 2011.