Projet MDI 343
Systèmes de recommandation

Nicolas Keriven et Jean-Baptiste Alayrac

1er mai 2014

Table des matières

Introduction

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 connait 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*.

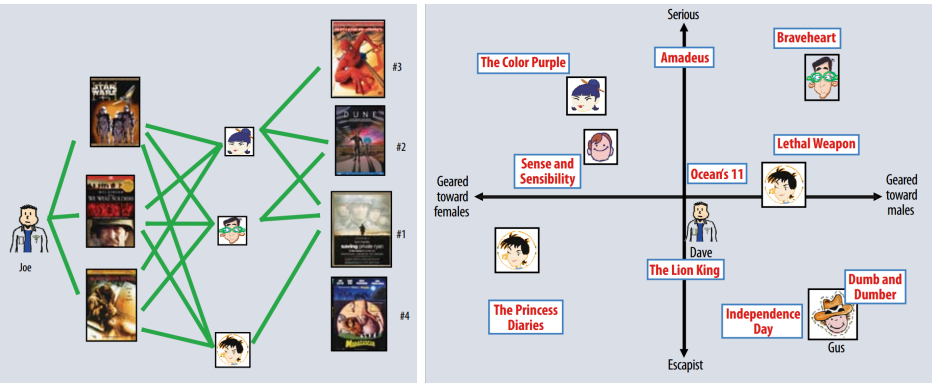

FIGURE 1 – A gauche : *neighborhood methods*, A droite : *Latent factor models*

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 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 représenté 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 probablemnet détester le film *The Princess Diaries*.

1.2 Factorisation de matrice et notations

Nous allons voir qu'une méthode afin de calculer ces facteur latents est de factoriser la partie connue de la matrice de notes $\mathbf{M}$. On rappelle que $\mathbf{M}_{u,i}$ 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 Ω 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 a aimé le produit i.

Maintenant essayons de décrire de manière plus mathématique la notion de *proche dans un certain sens de l'information contenue dans* $\boldsymbol{M}$. 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 conte tenu du biais b_{ui} que nous connaissons sur nos données :

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

où μ 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 :

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

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 [1] [2]. 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) :

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

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 example (donc un couple (u, i)) tiré aléatoirement. L'étape de mise à jour devient alors la suivante :

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

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) :

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

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

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 γ_t décroit 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 lâche.

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 différent quelque peu de la méthode présentée ici. On s'efforcera alors de comparer ces deux méthodes.

2.2 Algorithme parallélisé Jellyfish

Dans [3], 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 [3]. 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" [4]. 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 Ω. 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 π_u sur les utilisateurs et une permutation π_i sur les items, et de sélectionner la permutation sur les couples (π_u, π_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 Ω, il y a $(n_u n_i)!$ permutations possibles sur Ω, tandis qu'il existe seulement $n_u! n_i!$ de la forme (π_u, π_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 π_u sur les utilisateurs et une permutation π_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 $(u, i) \in \Omega$:

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

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 parcoure Ω 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 [3] [2], 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 [4]. 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

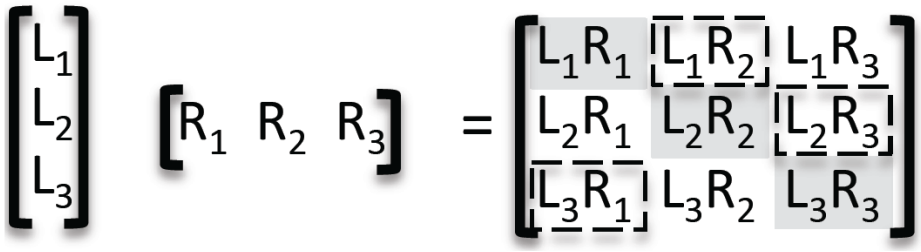

FIGURE 2 – Partition cyclique des blocs (pour question de simplicité, ici π_u et π_i sont les permutations identités). Les blocs surlignés de la même manière sont traités en parallèle. Schéma issu de [3].

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 [2] [4]. 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 [5], où le tirage se fait avec remise à l'intérieur de chaque bloc.

Détails d'implémentation

3 Résultats

Conclusion

Références

[1] 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.

[2] Léon Bottou and Olivier Bousquet. The tradeoffs of large scale learning. In *IN : ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 20*, pages 161–168, 2008.

[3] Benjamin Recht and Christopher Ré. Parallel Stochastic Gradient Algorithms for Large-Scale Matrix Completion. *submitted*, 2011.

[4] Angelia Nedic and Dimitri Bertsekas. Convergence rate of incremental subgradient algorithms. In *Stochastic Optimization : Algorithms and Applications*, pages 263–304. Kluwer, 2000.

[5] 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.