Diane Gallois-Wong
Raphaël Rieu-Helft

Janvier 2015

Algorithmes pour les systèmes de recommandation : un comparatif

Introduction

Un algorithme de *collaborative filtering* a pour but de se baser sur le comportement passé des utilisateurs pour proposer des recommandations pertinentes. On va ici se placer dans le cadre d'une matrice de ratings donnés par des utilisateurs à des objets, éventuellement incomplète voire parcimonieuse. Étant donné un sous-ensemble des ratings, le but est d'estimer les autres. On a plusieurs attentes vis-à-vis d'un tel algorithme :

- Requêtes rapides : un utilisateur doit pouvoir obtenir rapidement une recommandation (typiquement, l'objet à qui on estime qu'il attribuera le meilleur score).
- Ajout d'un nouvel utilisateur : on aimerait pouvoir ajouter raisonnablement vite un nouvel utilisateur à la base de données, en mettant à jour à la volée l'estimation des ratings des autres utilisateurs.
- Démarrage à froid : on aimerait qu'un utilisateur ayant donné peu de ratings puisse recevoir des recommendations pertinentes.
- Précision : on cherche à obtenir les recommandations les plus pertinentes possibles.

On peut s'attendre à ce que l'efficacité d'un algorithme varie selon les caractéristiques du jeu de données (forme de la matrice, caractère parcimonieux ou non...)

Dans la suite, nous allons comparer nos implémentations de différents algorithmes de collaborative filtering en terme de précision (au sens des erreurs MSE et MAE) et de temps de calcul (pour fournir une estimation complète des ratings à partir d'une matrice à trous).

Table des matières

Notations

n utilisateurs, m objets.
$r_{i,j}$: note donnée par l'utilisateur i à l'objet j.
$\tilde{r}_{i,j}$: estimation de la note donnée par l'utilisateur i à l'objet j.
$\overline{r}$: moyenne des notes connues.
u_i : vecteur des notes attribuées par l'utilisateur i.
v_i : vecteur des notes attribuées à l'objet j.
S_i : ensemble des objets notés par i.
S^j : ensemble des utilisateurs qui ont noté j.
$S^{j,k}$: ensemble des utilisateurs qui ont noté à la fois j et k.

1 Les algorithmes utilisés

1.1 Algorithmes témoins

Algorithmes simples (voire naïfs pour certains) servant de points de comparaison :
– "Witness" : Estimer tous les ratings inconnus par la moyenne de tous les ratings exprimés
– "PerUserAverage" : Estimer tous les ratings inconnus pour un utilisateur i donné par la moyenne $\overline{u}_i$ des ratings connus qu'il a attribués
– "BiasFromMean" : Tenir compte en plus de l'écart entre les notes attribuées à l'objet et les moyennes des utilisateurs qui l'ont noté :

$$\tilde{r}_{i,j} = \overline{u}_i + \frac{1}{\text{card}S^j} \sum_{i' \in S^j} (r_{i,j} - \overline{u}_{i'})$$

.
– "UnbiasedWitness" : Retirer les biais (cf. 1.3), estimer tous les ratings inconnus par 0, remettre les biais

1.2 Algorithmes Slope One

Prédicteur affine, en imposant une pente de 1. Pour tout couple d'objet, on définit une déviation de l'un par rapport à l'autre, un peu comme la mesure de similarité cosinus, mais en beaucoup plus simple :

$$dev_{j,k} = \sum_{i \in S^{j,k}} \frac{r_{i,j} - r_{i,k}}{\text{card}S^{j,k}}$$

$\tilde{r}_{i,j}$ s'obtient alors comme la moyenne de prédictions $dev_{j,k} + r_{i,k}$ associées aux objets $k \neq j$ qui sont pertinents, i.e. qui ont été notés par i et tels qu'au moins un utilisateur a noté à la fois j et k. En notant $R_{i,j}$ cet ensemble, on peut développer en

$$\tilde{r}_{i,j} = \overline{u}_i + \frac{1}{\text{card}R_{i,j}} \sum_{k \in R_{i,j}} dev_{j,k}$$

D'où le qualificatif affine : le prédicteur a une forme $f(x) = x + b$.

Les calculs sont simples. De plus, on peut calculer une fois pour toutes les déviations et les garder en mémoire : ainsi traiter une requête, i.e. calculer $\tilde{r}_{i,j}$ pour un seul i et un seul j donnés en connaissant la matrice dev, est très rapide. Ces déviations peuvent même être mises à jour sans tout recalculer quand on ajoute un rating.

Enfin, on verra que ce n'est pas vraiment moins précis que d'autres algorithmes plus sophistiqués.

Variantes :
- Weighted Slope One : donner plus de poids aux déviations des paires d'objets qui ont été notés tous deux à la fois par un grand nombre d'utilisateurs, car elles sont plus fiables. Pour cela on pondère la moyenne des $dev_{j,k} + r_{i,k}$ par le cardinal de $S^{j,k}$.
- Bi-Polar Slope One : l'estimation des notes de l'utilisateur i ne dépendent dans les algos ci-dessus que de l'ensemble des objets notés par i, pas des notes qu'il a attribuées... On ne sait pas tenir compte de ces notes dans le détail en gardant les avantages de Slope One, mais on peut adapter l'algorithme pour que les estimations pour i dépendent de la façon dont les objets qu'il a notés se partitionnent en ceux qu'il a aimés (note supérieure à sa moyenne $\overline{u}_i$) et ceux qu'il n'a pas aimés.

Algorithmes tirés de [2].

1.3 Algorithmes par similarité cosinus

Implémentation des algorithmes vus en cours : on calcule les biais en résolvant le problème d'optimisation

$$\min_{b^i, b_j} \sum_{i,j \in \Omega} (r_{ij} - \overline{r} - b^i - b_j)$$

via la solution matricielle du cours, puis on calcule des scores de similarité entre utilisateurs ou entre objets en utilisant le cosinus de l'angle comme mesure de similarité entre deux vecteurs et on estime les ratings comme une moyenne pondérée par les scores de similarité.

1.4 SVD

Variations sur la question bonus du DM 2. L'algorithme du DM commence par estimer les ratings inconnus à 0 avant d'appliquer la décomposition en valeurs singulières. L'estimation à 0 permet de faire des calculs sur des matrices creuses quand on connaît peu de ratings, donc un temps de calcul plus court. Les variantes qui suivent opèrent différentes transformations sur la matrice à estimer avant d'appliquer l'algorithme du DM, le but étant que l'estimation à 0 soit aussi pertinente que possible.

1.4.1 ShiftSVD

On calcule le rating moyen $\overline{r}$, on applique l'algorithme du DM à $R - \overline{r}$ et on ajoute $\overline{r}$ au résultat.

1.4.2 UnbiasedSVD

Plutôt que de se contenter de retrancher le rating moyen partout, on calcule les biais en résolvant le problème d'optimisation vu en cours comme en section 1.3, on applique l'algorithme du DM à la matrice non biaisée puis on réintroduit les biais. On s'attend à un résultat plus précis, mais le temps de calcul des biais est non négligeable (il faut inverser une matrice).

1.5 Analyse en composantes principales : algorithme Eigentaste

Idée : s'appuyer sur un petit sous-ensemble d'objets notés par tous les utilisateurs (*gauge set*) pour projeter un utilisateur sur un espace de petite dimension puis estimer ses notes à partir de celles de ses voisins au sens d'un algorithme de clustering. Eigentaste ne s'applique donc qu'aux

jeux de données comprenant un tel gauge set, comme le jeu de données Jester. L'algorithme est détaillé dans [1], nous le reprenons ici.
Notons R la matrice $(n \times m)$ des ratings, G le gauge set, $k = |G|$.
On commence par normaliser les ratings des utilisateurs sur les objets du gauge set : la qualité objective d'un objet importe peu ici car on cherche à obtenir des informations sur les ressemblances entre utilisateurs par analyse de composantes principales.
Pour tout objet j, on calcule donc $\mu_j = \sum_{i=1}^{n} r_{ij}$ la note moyenne de cet objet, et

$$\sigma_j^2 = \frac{1}{n-1} \sum_{i=1}^{n} (r_{ij} - \mu_j)$$

sa variance.
On obtient alors la matrice A (de taille $k \times k$) des ratings normalisés sur le gauge set : $a_{ij} = \frac{r_{ij} - \mu_j}{\sigma_j}$.
On pose $C = \frac{1}{n-1} A^T A$ (matrice de corrélation de Pearson). Elle est symétrique définie positive, on note Λ la matrice diagonale contenant ses valeurs propres triées par ordre décroissant et E la matrice orthogonale de vecteurs propres associée. On a $EE^T = Id$, $C = E\Lambda E^T$.
On peut alors projeter les vecteurs correspondant aux utilisateurs sur un petit sous-espace (de dimension v) contenant l'essentiel de la variance en posant $X = AE_v^T$. En particulier, pour $v = k$, on a $C_X = \frac{1}{n-1} X^T X = ECE^T = \Lambda$ diagonale : les vecteurs sont décorrélés. On prend $v = 2$ dans la suite.

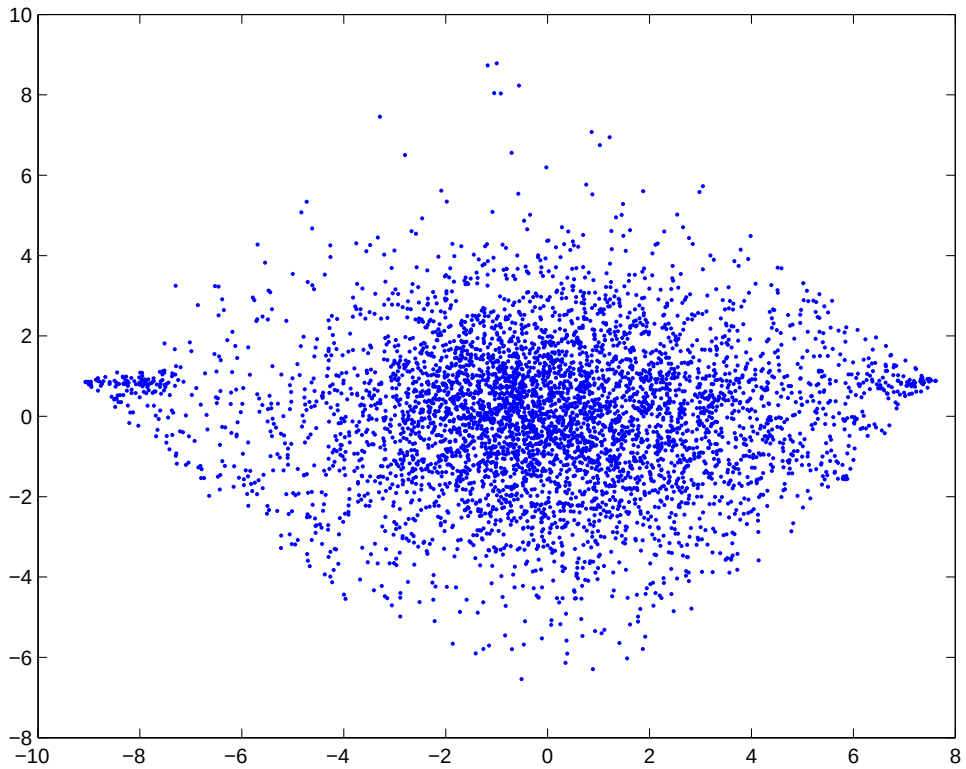

FIGURE 1 – Projection pour $v = 2$ sur un sous-ensemble du jeu de données Jester

On regroupe ensuite les utilisateurs proches par amas en utilisant un algorithme de clustering

([1] utilise un algorithme récursif rectangulaire, nous avons utilisé les fonctions Matlab *linkage* et *cluster* dans notre implémentation).
On estime ensuite un rating de l'utilisateur i sur l'objet j par la moyenne des notes pour j des utilisateurs se trouvant dans le même cluster que i.

Cet algorithme a l'inconvénient de ne s'appliquer qu'à certains jeux de données : pour ajouter un nouvel utilisateur, il faut d'abord lui faire noter l'intégralité du gauge set (typiquement une dizaine d'objets). De plus, on ne peut rien dire sur un utilisateur seul dans son amas : le démarrage à froid pose problème.
En revanche, ajouter un utilisateur qui a noté le gauge set est peu coûteux en termes de temps de calcul : on peut lui recommander rapidement des objets dès qu'on a choisi son cluster, l'idée étant de ne pas modifier immédiatement les projections des anciens utilisateurs mais plutôt de tout remettre à jour périodiquement.
Dans la suite, on se propose de comparer Eigentaste avec d'autres algorithmes de recommandation en termes de précision et de temps de calcul, sur le jeu de données Jester (pour lequel Eigentaste a été mis au point).

2 Observations expérimentales

2.1 Jeux de données

2.1.1 La matrice de la question bonus du DM

Il s'agit d'une matrice pleine, de taille 100×1000 et de rang 10, dont les coefficients sont des réels entre 0 et 1. Pour évaluer un algorithme, on fixe une probabilité p, on fournit à l'algorithme une sous-matrice dans laquelle chaque rating est présent avec probabilité p (de façon indépendante des autres), on s'en sert pour calculer une matrice pleine qu'on compare avec la matrice de départ.
On peut tester sur cette matrice tous les algorithmes présentés sauf Eigentaste (pas de gauge set).

2.1.2 Le jeu de données Jester

Jeu de données mis à disposition par Ken Goldberg (Berkeley). Il s'agit de ratings d'une centaine de blagues par de nombreux utilisateurs (plus de 50000). Huit blagues ont été notées par tous les utilisateurs : on s'en sert comme gauge set pour implémenter Eigentaste. Les coefficients varient continûment de -10 à 10 : on les ramène à $[0, 1]$ par une transformation affine pour mieux comparer avec la matrice du DM.
La densité d de la matrice est d'environ 0.24. Pour $p < d$, on conserve une proportion p/d des coefficients aléatoirement : on obtient à nouveau une matrice de densité p pour pouvoir comparer avec le jeu de données précédent.
Comme le rang de la matrice cible est inconnu (elle n'est pas pleine) les algorithmes à base de SVD donnent d'assez mauvais résultats (en ne tronquant pas du tout la décomposition en valeurs singulières).
Contrairement à la matrice du DM, les objets notés par les utilisateurs ne sont pas choisis au hasard : il s'agit de recommandations qu'on leur a faites après qu'ils ont noté le gauge set, ils ont donc plutôt tendance à noter des objets qu'ils apprécient.
Par ailleurs, la forme de la matrice est inversée par rapport à celle du DM : il y a beaucoup d'utilisateurs et très peu d'objets.
Pour avoir des temps de calcul et des utilisations mémoire raisonnables sur nos machines personnelles, nous nous sommes restreints aux 5000 premiers utilisateurs.

2.2 Mesures d'erreur

Algorithme	bonus, p=0.05	Jester, p=0.05	bonus, p=0.2	Jester, p=0.2	bonus, p=0.5
Witness	0.0176	0.0673	0.0149	0.0145	0.0094
UnbiasedWitness	0.0126	0.0414	0.0085	0.0081	0.0051
PerUserAverage	0.0180	0.0471	0.0149	0.0089	0.0093
BiasFromMean	NaN	0.0417	0.0086	0.0081	0.0051
PlainSVD	0.1618	0.1839	0.0943	0.1737	0.0338
ShiftSVD	0.0191	0.0850	0.0106	0.0748	0.0050
UnbiasedSVD	0.0128	0.0683	0.0086	0.0511	0.0048
UserCosSim	0.0125	0.0505	0.0093	0.0426	0.0079
ItemCosSim	NaN	0.0456	0.0092	0.0307	0.0069
SlopeOne	NaN	0.0425	0.0086	0.0081	0.0052
WeightedSlopeOne	NaN	0.0499	0.0085	0.0083	0.0052
BiPolarSlopeOne	NaN	0.0479	0.0093	0.0084	0.0058

FIGURE 2 – Erreur MSE pour certaines valeurs de p

Sur la matrice du DM, on obtient ce tableau d'erreurs MSE pour différentes valeurs de p :
Les algorithmes simples "BiasFromMean" et "UnbiasedWitness" obtiennent d'excellents résultats !
Les variantes de Slope One obtiennent aussi des résultats très comparables à l'algorithme UnbiasedSVD plus complexe à mettre en œuvre.
ItemCosSim est plus performant que UserCosSim sur les deux jeux de données, mais la différence est bien plus marquée sur le jeu de données Jester : c'est sans doute lié à la forme particulière de la matrice (il y a très peu d'objets par rapport au nombre d'utilisateurs).

Algorithme	p=0.1	p=0.2
Witness	1.0992e-05	2.7715e-06
UnbiasedWitness	6.6646e-06	1.8463e-06
PerUserAverage	9.1804e-06	2.3797e-06
BiasFromMean	8.7602e-06	1.4253e-06
ShiftSVD	7.1533e-06	1.5175e-06
UnbiasedSVD	5.4096e-06	1.5922e-06
UserCosSim	4.7402e-06	1.4364e-06
ItemCosSim	6.6190e-06	1.6920e-06
SlopeOne	7.1207e-06	1.5497e-06
WeightedSlopeOne	5.9259e-06	1.8653e-06
BiPolarSlopeOne	6.6666e-06	2.0102e-06

FIGURE 3 – Erreur MAE sur la matrice du DM

Nous avons aussi évalué l'erreur MAE (Mean Absolute Error) sur la matrice du DM. Il s'agit d'un protocole différent : il consiste à estimer successivement un seul rating à partir de tous les autres, et à prendre la moyenne de l'erreur absolue commise. Comme il s'agit d'une norme 1 plutôt que d'une norme 2, de grosses erreurs ponctuelles nuisent moins à la précision, tandis que de petites erreurs consistentes nuisent plus.
Ainsi, les algorithmes performants au sens des erreurs MSE et MAE diffèrent sensiblement : les algorithmes témoins comme UnbiasedWitness et BiasFromMean sont cette fois nettement moins performants que les variantes de SVD et SlopeOne.
Enfin, UserCosSim semble être le meilleur algorithme pour MAE, très nettement devant ItemCosSim alors que c'était l'inverse pour MSE. Peut-être est-ce dû à la forme des données : peu d'utilisateurs et beaucoup d'objets. UserCosSim estime le vecteur d'un utilisateur à partir des autres vecteurs d'utilisateurs : il y en a peu, mais ils ont une très grande dimension. On va donc

souvent s'inspirer d'utilisateurs qui ont des goûts extrêment proches de celui qui nous intéresse, d'où des bons résultats en moyenne ; mais à cause du peu d'utilisateurs disponibles en tout, parfois il n'y en a pas qui ressemble vraiment, auquel cas on a quelques grosses erreurs, ce qui explique que cet algorithme ne soit pas aussi bon quand on considère une norme 2. Au contraire, ItemCosSim s'intéresse à de plus nombreux vecteurs (liés aux objets) de plus petite dimension. Le grand nombre entraîne une constance : on va toujours avoir plusieurs objets qui ressemblent à celui qui nous intéresse, et en les moyennant on trouve systématiquement quelque chose de pas trop loin de la réalité, ce qui est bien pour la norme 2. En revanche on est un peu "noyé" par la masse des vecteurs disponibles (les vecteurs plus distants ont un petit coefficient, mais il existe néaanmoins) et on ne peut pas, comme UserCosSim, recopier à peu près exactement un vecteur qui nous ressemble beaucoup ; on n'est donc jamais vraiment très précis, ce qui n'est pas efficace pour une norme 1.

2.3 Choix des paramètres pour Eigentaste

L'algorithme de clustering utilisé dans le calcul de Eigentaste prend un paramètre qu'il faut optimiser pour augmenter la précision : le nombre maximal de clusters. On observe que le nombre optimal de clusters varie fortement avec p : pour $p = 0.2$, il faut en prendre environ 600 (pour classifier 5000 utilisateurs), alors que pour $p = 0.1$, il est préférable de n'en prendre qu'une vingtaine (voir Figure 4). Ceci est probablement lié à un phénomène de surapprentissage : quand p est petit, on n'a pas assez d'informations pour justifier de séparer certains utilisateurs.

2.4 Temps d'exécution

Une bonne partie du temps de calcul est passé à inverser des matrices presque singulières pour calculer les biais... Ainsi, l'algorithme UnbiasedWitness, qui est très simple conceptuellement et parmi les plus précis sur les deux jeux de données, est loin d'être le moins coûteux en terme de temps de calcul. Les algorithmes par voisinage (CosSimilarity et Eigentaste) sont aussi plutôt longs à exécuter.
Il faut cependant tenir compte du fait que certains algorithmes (tels que Slope One ou Eigentaste, mais pas les variantes de SVD) font beaucoup de calculs "une fois pour toutes" (*off-line*). Une fois ces calculs faits, les requêtes et l'ajout d'un nouvel utilisateur sont rapides. On pourrait envisager une implémentation des algorithmes par similarité cosinus ayant cette propriété, mais il faut alors stocker les scores de similarité : le compromis se fait cette fois au niveau de la mémoire nécessaire...

Références

[1] Ken Goldberg, Theresa Roeder, Dhruv Gupta, and Chris Perkins. Eigentaste : A constant time collaborative filtering algorithm. *Inf. Retr.*, 4(2) :133–151, July 2001.

[2] Daniel Lemire and Anna Maclachlan. Slope one predictors for online rating-based collaborative filtering. *CoRR*, abs/cs/0702144, 2007.

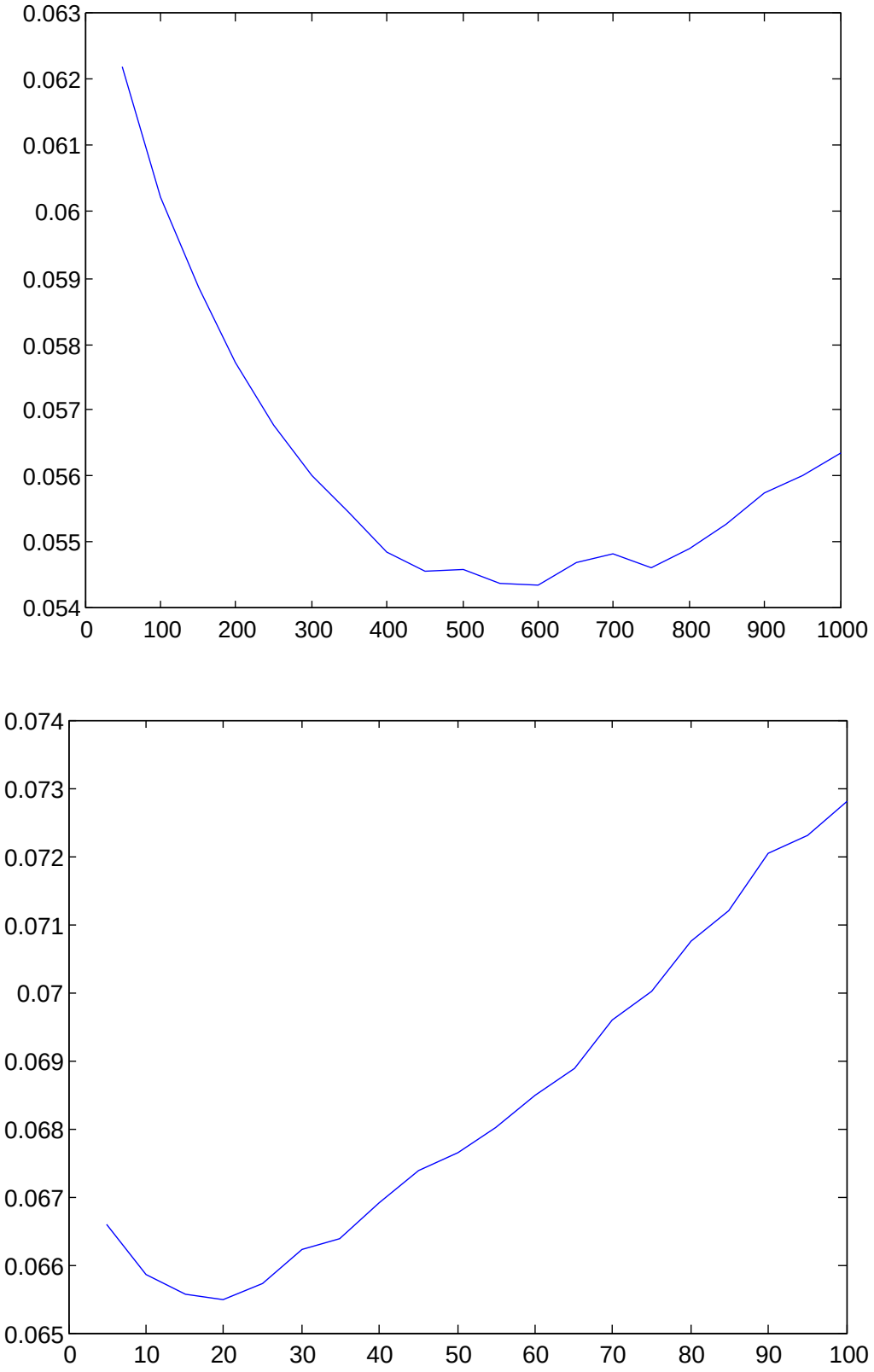

FIGURE 4 – Erreur MSE en fonction du nombre maximal de clusters pour $p = 0.2$ et $p = 0.1$

Function Name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
tests	1	11.315 s	0.248 s	
MSE_forJester	24	7.653 s	0.365 s	
tests>MSEsJester	2	5.848 s	0.001 s	
...Jester(Jh,J,p,iter,algo,default_rank)	22	5.846 s	0.003 s	
tests>Ext/algoExt	35	4.225 s	0.004 s	
unbias	20	2.899 s	2.895 s	
userCosSimilarity	5	2.036 s	1.286 s	
readjester	1	1.853 s	0.170 s	
eigentaste	2	1.729 s	1.291 s	
dlmread	1	1.683 s	1.668 s	
csvread	1	1.683 s	0.000 s	
tests>MSEsBonus	3	1.419 s	-0.000 s	
tests>@(algo)MSE(M,p,iter,algo,rankM)	33	1.411 s	0.002 s	
MSE	33	1.409 s	0.237 s	
unbiasedSVD	5	0.880 s	0.006 s	
itemCosSimilarity	5	0.832 s	0.094 s	
UnbiasedWitness	5	0.750 s	0.006 s	
BiPolarSlopeOne	5	0.647 s	0.382 s	
Dev	20	0.486 s	0.486 s	
randObserve	57	0.445 s	0.445 s	
linkage	2	0.371 s	0.007 s	
stats\private\linkagemex (MEX-file)	2	0.362 s	0.362 s	
WeightedSlopeOne	5	0.297 s	0.177 s	
truncateSVD	10	0.245 s	0.245 s	
SlopeOne	5	0.244 s	0.064 s	
nansum	24	0.242 s	0.242 s	
shiftSVD	5	0.171 s	0.027 s	
nanmean	30	0.147 s	0.147 s	
addpath	6	0.121 s	0.004 s	
path	6	0.116 s	0.086 s	

FIGURE 5 – Profiling fourni par Matlab