Collaborative Filtering using Matrix Factorization

Salma El Alaoui Talibi[1], Olivier Chancé[2], Zhengying Liu[3], Sophia Lazraq[4]

Contents

1. Introduction

The setting for collaborative recommendation systems is typically as follows: the input to the system are user rating on items they have used; and predictions of user preferences on items they have not used yet are based on patterns in the partially observed rating matrix. In content-based approaches, predictions are made using the features of the movie, such as the genre and the actors, as well as features of the the users, such as the age and the gender. Contrarily to this approach, in a collaborative filtering context, users collaborate by sharing their ratings instead of relying on external information [1].

[1] salma.el-alaoui-talibi@polytechnique.edu
[2] olivier.chance@polytechnique.edu
[3] zhengying.liu@polytechnique.edu
[4] sophia.lazraq@gmail.com

In Collaborative Filtering, two main approaches are used, namely, memory-based approach (nearest-neighbor techniques) and model-based approach (latent-factor models techniques).
As the Netflix Prize competition has demonstrated, matrix factorization models are superior to classic nearest-neighbor techniques for producing product recommendations. In its basic form, matrix factorization characterizes both items and users by vectors of factors inferred from item rating patterns [2].

Consider the situation with n_U users and n_I items. For each user $u \in U$, we associate a `row` vector $x_u \in \mathbb{R}^r$ to characterize this user. Similarly, for each item $i \in I$, we use a `row` vector $y_i \in \mathbb{R}^r$ to characterize i. Then the matrix factorization model assumes that r_{ij}, the rating for item i given by user u should be close to

$$\hat{r}_{u,i} := x_u y_i^\top.$$

Let $R \in \mathbb{R}^{n_U \times n_I}$ denote the real rating matrix with manyf missing values and $X \in \mathbb{R}^{n_U \times r}, Y \in \mathbb{R}^{n_I \times r}$ be the matrices with x_u, y_i's as **rows** respectively. Let $E \subset U \times I$ be the set of all user-item pairs having corresponding known ratings in R. Then the loss function to minimize can be given by:

$$F(X, Y) = \sum_{(u,i)\in E} (r_{u,i} - x_u y_i^\top)^2.$$

With a penalization term to avoid over-fitting, this gives:

$$F(X, Y) = \sum_{(u,i)\in E} (r_{u,i} - x_u y_i^\top)^2 + \lambda \Omega(X, Y) \tag{1}$$

To simplify the notation, we can write:

$$F(X, Y) = \|\mathcal{P}_E(R - XY^\top)\|_F^2 + \lambda \Omega(X, Y). \tag{2}$$

This loss function is a particular case of the general loss that various forms of matrix factorization attempt to optimize. The latter can be written as:

Optimize $J =$ [(Regularized) Objective function quantifying matching between R and XY^T] subject to: constraints on X, Y

The difference between various forms of matrix factorization is in the nature of the loss function and the constraints on the matrices X and Y. [3, 4] explore *Maximum Margin Factorization*, which uses the maximum margin regularizer with a hinge loss in the objective function, instead of using the Frobenius norm as in 2. Maximizing the margin is supposed to outperform other methods in the presence of data that is prone to overfitting.

In *Probabilistic Matrix Factorization methods* [5, 6, 7], the objective function learns the parameters of a generative process that creates the ratings matrix, so that its likelihood is as large as possible. Therefore, the objective function takes a maximization form. Some

probabilistic methods minimize the Frobenius norm and optimize the regularization parameters simultaneously. In [8], a Bayesian approach is used to model these parameters, and they are determined by using Gibbs Sampling.

Another popular method is *Non-Negative Matrix Factorization* [9], which adds the constraint that X and Y must be non-negative. This method provides better interpretability of the user-item interaction in the case of implicit feedback data sets, in which ratings indicate positive preferences. An example of such ratings is the frequency of buying or browsing an item online.

In the following, we will introduce several algorithms for solving the minimization problem 1, including stochastic gradient descent (SGD), alternating least squares (ALS) and some of their more advanced variants. We will also describe in depth some Probabilistic and Maximum Margin Factorization methods, then compare their performances on chosen data sets.

2. Unconstrained Matrix Factorization

2.1. Stochastic gradient descent

One efficient method to solve the minimization problem in (1) is stochastic gradient descent. In this case, the penalization is given by

$$\Omega(X, Y) = \sum_{(u,i)\in E} \|x_u\|^2 + \|y_i\|^2$$

and the total loss function is

$$F(X, Y) = \sum_{(u,i)\in E} F_{u,i}(X, Y)$$

where

$$F_{u,i}(X, Y) = (r_{u,i} - x_u y_i^\top)^2 + \lambda(\|x_u\|^2 + \|y_i\|^2)$$

and the stochastic gradient descent algorithm procedes as in algorithm 1:

The convergence of this algorithm is an important issue and is discussed in many papers, e.g. [10]. Many variants of SGD exist trying to obtain parallelism, scalability and efficiency, e.g. SSGD and DSGD in [10].

SGD combines implementation ease with a relatively fast running time. Yet, in some cases, it is beneficial to use ALS optimization [2].

2.2. Alternating Least Squares

When we fix one of X and Y and minimize (1) with respect to the other matrix, the non-convex problem becomes a much easier convex problem, which leads to following alternating least squares algorithm, see Algorithm 2.

Algorithm 1: Stochastic Gradient Descent (SGD)

Input : R, E, λ, η
Output: X, Y

Initialize all x_u, y_i's (the rows in X, Y)
while *not converged* **do**
  Choose $(u, i) \in E$ at random
  $x_u^{new} \leftarrow x_u - \eta(\lambda x_u - (r_{u,i} - x_u y_i^\top) y_i)$
  $y_u^{new} \leftarrow y_u - \eta(\lambda y_i - (r_{u,i} - x_u y_i^\top) x_u)$
  $x_u \leftarrow x_u^{new}$
  $y_i \leftarrow y_i^{new}$
end
return X, Y

Algorithm 2: Alternating Least Squares (ALS)

Input : R, E, λ
Output: X, Y

Initialize Y
while *not converged* **do**
  $X \leftarrow \arg\min_X \|\mathcal{P}_E(R - XY^\top)\|_F^2 + \lambda\Omega(X, Y)$
  $Y \leftarrow \arg\min_Y \|\mathcal{P}_E(R - XY^\top)\|_F^2 + \lambda\Omega(X, Y)$
end
return X, Y

In the case where $\Omega(X, Y)$ is a weighted ridge penalization, [11] describes a parallel algorithm called ALS-WR to solve this problem. In the paper, the loss function is given by

$$F(X, Y) = \sum_{(u,i)\in E} (r_{u,i} - x_u y_i^\top)^2 + \lambda \left(\sum_{u \in U} n_{x_u} \|x_u\|^2 + \sum_{i \in I} n_{y_i} \|y_i\|^2 \right)$$

where n_{x_u} and n_{y_i} denote the number of ratings of user u and item i respectively.

The parallelization consists of distributing two copies of the rating matrix R, one distributed by rows (i.e. by users) and the other by columns (i.e. by items). Then the solution of the two arg min problems (which are equivalent to solving a system of linear equations in this case) can be computed in parallel.

2.3. Maximum-Margin Matrix Factorization

Maximum Margin Matrix Factorization (MMMF) is a matrix factorization technique that relies on low-norm factorization. The sparse matrix R is fitted with a matrix $\hat{R} = XY^\top$ such that the norm of X and Y are constrained. Such factorization leads to a convex optimization problem.

Those constraints are justified when the problem of matrix factorization is interpreted as a feature-learning problem for large-margin linear prediction. If X is fixed and Y is to be learned, predicting each column of R is viewed as a SVM problem where one has to find a row in Y using the feature vectors of X. Therefore, to maximize the margin, one has to minimize the norm of Y: $\|Y\|_F$. In the case of collaborative filtering, X is unknown too and both X and Y plays similar role and has to be learned together. As a consequence, one gets the minimization problem:

$$\min_{\hat{R}=XY^\top} \|X\|_F^2 + \|Y\|_F^2 + C.\text{loss}(R, \hat{R}) \tag{3}$$

with C the trade-off constant.

Using [3, lemma 1]:

$$\|\hat{R}\|_T = \min_{\hat{R}=XY^\top} \|X\|_F.\|Y\|_F = \min_{\hat{R}=XY^\top} \frac{1}{2}(\|X\|_F^2 + \|Y\|_F^2)$$

where $\|\hat{R}\|_T = \sum |\lambda_i| = Tr(\sqrt{\hat{R}\hat{R}^\top})$ is the trace norm of $\hat{R}$, we get:

$$\min_{\hat{R}} \|\hat{R}\|_T + C.\text{loss}(R, \hat{R}) \tag{4}$$

The trace norm is a convex function, so for any loss function that is convex too, 4 is a convex optimization problem.

To get a concrete grasp on the problem, let's consider that $R_{i,j} = \pm 1$ (1 if user$_i$ liked item$_j$, -1 otherwise). In this case, hinge-loss function can be used in 4.

$$\min_{\hat{R}} \|\hat{R}\|_T + C \sum_{i,j \in E} \max(0, 1 - R_{i,j}\hat{R}_{i,j}) \tag{5}$$

[3] shows that 5 can be rewritten as a semi definite optimization program. The dual associated is *strictly feasible*, with *no duality gap*. Once solved, $\hat{R}$ is easily recovered [3].

However, for most problems, R is not a binary target matrix but a rating one. In this case, $R_{i,j} \in \{0, 1, 2, ..., r_{max}\}$ and $\hat{R}_{i,j} \in \mathbb{R}$. Without relaxation, we can then introduce $r_{max} - 1$ threshold values $\theta_1, ..., \theta_{r_{max}-1}$ such that $\forall (i,j)$, $\theta_{R_{i,j}-1} + 1 \leq \hat{R}_{i,j} \leq \theta_{R_{i,j}} - 1$.

With relaxation, we have to penalize both inequalities above. But it's not sufficient, those inequalities imply too that $\hat{R}_{i,j} \geq \theta_r + 1$ for $r < R_{i,j}$ and $\hat{R}_{i,j} \leq \theta_r - 1$ for $r \geq R_{i,j}$, and these inequalities have to be penalized too. We finally get, from 4 and the considerations above:

$$\min \|\hat{R}\|_T + C \sum_{i,j \in E} \Big(\sum_{r=1}^{R_{i,j}-1} max(0, 1 - (\hat{R}_{i,j} - \theta_r)) + \sum_{r=R_{i,j}}^{r_{max}-1} max(0, 1 - (\theta_r - \hat{R}_{i,j})) \Big)$$

Introducting T, such that $T_{i,j}^r = \begin{cases} +1 \text{ for } r \geq R_{i,j} \\ -1 \text{ otherwise} \end{cases}$, we can reformulate this minimization problem:

$$\min \|\hat{R}\|_T + C \sum_{i,j \in E} \sum_{r=1}^{r_{max}-1} \max(0, 1 - T_{i,j}^r(\theta_r - \hat{R}_{i,j})) \tag{6}$$

Even if 6 can be rewritten as a semi-definite program as we did for the first case, when R was a binary matrix, [4] suggests to replace $\hat{R}$ by $XY^\top$ and the penalization term - the trace norm - by the sum of the frobenius norms of X and Y as expressed in 3. This way, we get an objective function J:

$$J(X, Y, \theta) = \frac{1}{2}(\|X\|_F^2 + \|Y\|_F^2) + C \sum_{i,j \in E} \sum_{r=1}^{r_{max}-1} \max(0, 1 - T_{i,j}^r(\theta_r - X_i Y_j^\top)) \tag{7}$$

that is differentiable, when $\|\hat{R}\|_T$ was not. This technique allows using gradient descent methods to get a minimum.

3. Probabilistic Matrix Factorization

General Case with automatic control complexity

[6] introduces a probabilistic approach to the CF problem that takes a Bayesian perspective. That results in representing users as a mixture of preference profiles, hence it attributes item preference to the preference profiles rather than directly to users. This method particularly tackles two problems, it handles very large data sets and it deals with users who have very few ratings. In PMF, the ratings R are modeled as draws from a Gaussian distribution where the mean for R_{ij} is equal to $X_i Y_j^\top$. The paper defines α as the precision, a fixed parameter which reflects the uncertainty of the estimations. As for the complexity, it is controlled by placing a zero-mean spherical Gaussian prior on X and Y. In other words, each row of X is drawn from a multivariate Gaussian with mean $\mu = 0$ and precision which is some multiple of the identity matrix I. The complexity of the model here is measured by the magnitude of latent variables and prevents overfitting. Thus the model is defined as follows:

$$P(R|X, Y, \sigma^2) = \prod_{i=1}^{n_U} \prod_{j=1}^{n_I} [\mathcal{N}(R_{ij}|X_i Y_j^\top, \sigma^2)]^{I_{ij}} \tag{8}$$

$$P(X, \sigma_X^2) = \prod_{i=1}^{n_U} \mathcal{N}(X_i|0, \sigma_X^{-1} I) \tag{9}$$

$$P(Y, \sigma_Y^2) = \prod_{j=1}^{n_I} \mathcal{N}(Y_j|0, \sigma_Y^{-1} I) \tag{10}$$

Therefore, the challenge is to find the appropriate values for σ_X , σ_Y and σ.

The problem consists in maximizing the log of the posterior $P(X, Y|R, \sigma^2, \sigma_X^2, \sigma_Y^2)$. When the observation noise variance σ and the prior variances σ_X and σ_Y are all kept fixed, maximizing the log posterior is equivalent to minimizing the sum-of-squared-errors objective function with quadratic regularization terms as follows.

$$E = \frac{1}{2}\sum_{i=1}^{n_U}\sum_{j=1}^{n_I} I_{ij}(R_{ij} - X_i Y_j^\top)^2 + \frac{\lambda_X}{2}\sum_{i=1}^{n_U}\|X_i\|_2^2 + \frac{\lambda_Y}{2}\sum_{j=1}^{n_I}\|Y_j\|_2^2, \tag{11}$$

where $\lambda_X = \sigma^2/\sigma_X^2$, and $\lambda_Y = \sigma^2/\sigma_Y^2$.

A local minimum of the previous equation can be found by performing gradient descent in X and Y.

The paper explains how to control the capacity of the model in order to generalize well. The complexity of the model is controlled by the hyperparameters: the noise variance σ^2 and the parameters of the priors (σ_X^2 and σ_Y^2). Instead of using the previous equation, the paper suggests to introduce priors for the hyperparameters and maximize the log-posterior over both parameters and hyperparameters according to the following expression:

$$\ln P(X, Y, \sigma^2, \Theta_X, \Theta_Y|R) = \ln P(RX, Y, \sigma^2) + \ln P(X|\Theta_X) + \ln P(Y|\Theta_Y) + \ln P(\Theta_X) + \ln P(\Theta_Y) + C, \tag{12}$$

where Θ_X and Θ_Y are the hyperparameters for the priors over user and item vectors respectively and C is a constant that does not depend on the parameters or hyperparameters.

When the prior is Gaussian, it is better to alternate between optimizing the hyperparameters and updating the feature vectors using steepest ascent with the values of hyperparameters fixed. When the prior is a mixture of Gaussians, the hyperparameters can be updated by performing a single step of EM. In all the experiments, improper priors (not conjugate priors) were used in this study.

Constrained PMF

The problem with the previous approach concern users with very few ratings since these users will have feature vectors that are close to the prior mean. The paper presents an additional way of constraining user-specific feature vectors that has a strong effect on infrequent users.

The article defines the new feature vector for user i as follows:

$$X_i = H_i + \frac{\sum_{k=1}^{n_I} I_{ik} W_k}{\sum_{k=1}^{n_I} I_{ik}}, \tag{13}$$

where $W \in \mathbb{R}^{r \times n_I}$ is the latent similarity constraint matrix. H_i can be seen as the offset added to the mean of the prior distribution to get the feature vector X_i for the user i.

Then , the conditional distribution over the observed ratings is :

$$P(R|H,Y,W,\sigma^2) = \prod_{i=1}^{n_U}\prod_{j=1}^{n_I}[\mathcal{N}(R_{ij}|g([H_i + \frac{\sum_{k=1}^{n_I} I_{ik}W_k}{\sum_{k=1}^{n_I} I_{ik}}]Y_j^\top), \sigma^2)]^{I_ij} \tag{14}$$

In this paper, the regularization of the latent similarity constraint matrix W is done by placing a zero-mean spherical Gaussian prior on it:

$$P(W|\sigma_W) = \prod_{k=1}^{n_I}\mathcal{N}(W_k|0,\sigma_W^2 I) \tag{15}$$

Eventually, as with the general PMF model, maximizing the log-posterior is equivalent to minimizing the sum-of-squared errors function with quadratic regularization terms:

$$E = \frac{1}{2}\sum_{i=1}^{n_U}\sum_{j=1}^{n_I} I_{ij}(R_{ij} - g([H_i + \frac{\sum_{k=1}^{n_I} I_{ik}W_k}{\sum_{k=1}^{n_I} I_{ik}}]Y_j^\top)^2 + \frac{\lambda_H}{2}\sum_{i=1}^{n_U}\|H_i\|_2^2 + \frac{\lambda_Y}{2}\sum_{j=1}^{n_I}\|Y_j\|_2^2 + \frac{\lambda_W}{2}\sum_{k=1}^{n_I}\|W_k\|_2^2, \tag{16}$$

where $\lambda_H = \sigma^2/\sigma_H^2, \lambda_Y = \sigma^2/\sigma_Y^2$ and $\lambda_W = \sigma^2/\sigma_W^2$. In this study, they perform a gradient descent on H, Y and W to minimize the objective function.

3.1. Non-linear Matrix Factorization with Gaussian Processes

In [7], the Probabilistic Matrix Factorization described in 3 is extended by the following contributions:

Relating probabilistic Matrix Factorization to probabilistic PCA

As we have previously seen, least squares fit has a natural probabilistic interpretation. Considering this change of notation:

$$\begin{aligned} X = Y &\in \mathbb{R}^{n_I \times r} \quad \text{(latent matrix)} \\ W = X &\in \mathbb{R}^{n_U \times r} \quad \text{(mapping matrix)} \\ Y = R^T &\in \mathbb{R}^{n_I \times n_U} \quad \text{(observation matrix)} \end{aligned}$$

the likelihood becomes:

$$p(Y|W,X,\sigma^2) = \prod_{i=1}^{n_I}\prod_{j=1}^{n_U}\mathcal{N}(y_{i,:}|Wx_{i,:},\sigma^2 I) \tag{17}$$

We therefore recognize muti-output linear regression. If we place a spherical Gaussian prior over the X:

$$p(X) = \prod_{i=1}^{n_I}\prod_{j=1}^{r}\mathcal{N}x_{i,j}|0,\alpha_x^{-1})$$

and marginalizing over W leads to:

$$p(Y|W, \sigma^2, \alpha_x) = \prod_{i=1}^{N} \mathcal{N}(y_{i,:}|O, \alpha_x^{-1} W W^T + \sigma^2 I) \tag{18}$$

where we recognize the marginal likelihood function associated with probabilistic PCA, which can be optimized over the parameter W and the hyperparameters of the model, and can be brought to an eigenvalue problem that can be solved in closed form.
By taking a spherical Gaussian prior over W, and marginalizing it out in 17, we obtain:

$$p(Y|X, \sigma^2, \alpha_w) = \prod_{j=1}^{D} \mathcal{N}(y_{:,j}|O, \alpha_w^{-1} X X^T + \sigma^2 I) \tag{19}$$

which is an equivalent problem called Dual Probabilistic PCA.

Optimizing with SGD

Instead of using an EM-style algorithm, [7] proposes to present the ratings for a single user j at a time, computing the log-likelihood gradients for this user only at each step. We thus aim to minimize the negative log likelihood given by:

$$E_j(X) = \frac{N_j}{2} \log |C_j| + \frac{1}{2} \left(y_{i_j,j}^T C_j^{-1} y_{i_j,j} \right) \tag{20}$$

where $C_j = a_w^{-1} X_{i_j,:} X_{i_j,:}^T + \sigma^2 I$ and N_j is the number of items rated by user j. The associated gradient w.r.t. X is:

$$\frac{dE_j(X)}{dX_{i_j,:}} = -G X_{i_j,:} \tag{21}$$

with $G = \left(C_j^{-1} y_{i_j,j} y_{i_j,j}^T C_j^{-1} - C_j^{-1} \right)$. The gradients for σ^2 and α_w are computed similarly. The gradients for this update step are computed in $O(N_j^3)$ or $O(r^3)$ in the case $r < N_j$.

Using Gaussian Processes to non-linearize Matrix Factorization

19 is also Gaussian Process with a linear covariance function:

$$p(Y|X, \sigma^2, \theta) = \prod_{i=1}^{n_U} \mathcal{N}(y_{:,j}|0, K) \tag{22}$$

where $K = \alpha_w^{-1} X X^t + \sigma^2 I$. Making the model non-linear can therefore be achieved by using non-linear covariance functions, such as the RBF covariance. This can also be seen as a "kernalisation" of the algorithm. However, this approach is different from kernel PCA, as the latter constructs the kernel in data space, which would make it difficult to deal with missing data.

4. Experimental evaluation

4.1. Data sets

In order to assess our algorithms, we have selected two real datasets, in addition to a toy dataset we have built. The two actual datasets are:
- Movielens: Each row in the set corresponds to a rating given by a user to a movie. There are different versions of this set, each of which has a different number of ratings.
- Jester: Each row in the set correponds to a rating given by a user to a joke.

The toy data set was built according to the process described in [12]. The number of users and items is set. Each user is associated with a user class and each item is associated with an item class such that the rating between two classes is fixed and randomly picked in Uniform(1, 5). The ratings matrix is then easily built and the desired density is achieved by removing ratings at random.

The following table presents the data sets:

Model	Toy dataset	Movielens			Jester
		S	M	L	
Number of ratings	20K	100K	1M	10M	1.7M
Number of users	100	1K	6K	72K	60K
Number of items	1000	1.7K	4K	10K	150
Density	20%	6.4%	4.3%	0.5%	31.5%
Scale	1 - 5	1 - 5	1 - 5	0.5 - 5	-10 - 10
Type	Discrete	Discrete	Discrete	Discrete	Continuous

Performance measures

Two different performance measures are considered:
- Root Mean Squared Error

$$\text{RMSE} = \sqrt{\frac{\sum_{i,j \in E} (\hat{r}_{i,j} - r_{i,j})^2}{|E|}}$$

- Normalized Mean Absolute Error

$$\text{NMAE} = \frac{\sum_{i,j \in E} |\hat{r}_{i,j} - r_{i,j}|}{E[\text{MAE}]}$$

NMAE is interesting because it allows to compare different data sets that have different scales of ratings. Indeed, NMAE is a normalized MAE where we divide MAE by E[MAE]. When NMAE is less than one, then the model is better than just a random guess, otherwise it performs worse.

Train and test sets

We use the setup described by [13] who defines two types of generalizations:

Weak generalization. One rating is held out for each user to form the test set while other ratings constitute the training set.

Strong generalization. In the weak generalization approach, ratings have to be predicted for users that have been already seen during the training phase, leading to a lack of generalization. To avoid this, the set of users is first divided into two groups (U1 and U2). From U1, we build the training set. The model is then trained using all data available for this set. Once done, the set of ratings corresponding to the rest of users (U2) is divided. For each user, one rating is held out and the rest is kept visible. The model is then tested by predicting the held out values using the model built during the training phase and the observable data that was kept visible.

4.2. Alternating Least Squares with Weighted-λ-Regularization (ALS-WR)

We implemented a special form of Alternating Least Squares (ALS) algorithm called ALS with Weighted-λ-Regularization (ALS-WR) according to [11].

We recall that the loss function in this situation is given by

$$F(X,Y) = \sum_{(u,i)\in E} (r_{u,i} - x_u y_i^\top)^2 + \lambda \left(\sum_{u\in U} n_{x_u} \|x_u\|^2 + \sum_{i\in I} n_{y_i} \|y_i\|^2 \right).$$

With this special form of loss function, we can actually represent the solutions of the optimization problems in Algorithm 2 in explicit forms.

Take X for example.

$$\frac{1}{2}\frac{\partial}{\partial x_{u,k}} F(X,Y) = \sum_{i\in U(u)} (x_u y_i^\top - r_{u,i}) y_{i,k} + \lambda n_{x_u} x_{u,k} = 0, \forall k = 1, ..., r$$

where $U(u)$ stands for those i such that $(u,i) \in E$. Thus

$$\sum_{i\in U(u)} (x_u y_i^\top - r_{u,i}) y_i + \lambda n_{x_u} x_u = 0$$

and

$$x_u \left(\sum_{i\in U(u)} y_i^\top y_i + \lambda n_{x_u} Id_r \right) = \sum_{i\in U(u)} r_{u,i} y_i$$

which means

$$x_u = V_u A_u^{-1}$$

where

$$A_u = \sum_{i\in U(u)} y_i^\top y_i + \lambda n_{x_u} Id_r$$

and

$$V_u = \sum_{i\in U(u)} r_{u,i} y_i.$$

Similarly, for Y we have

$$y_i = V_i A_i^{-1}$$

where

$$A_i = \sum_{u \in I(i)} x_u^\top x_u + \lambda n_{y_i} I d_r$$

and

$$V_i = \sum_{u \in I(i)} r_{u,i} x_u.$$

We notice that to compute x_u and y_i for each u and i, we need to solve a system of linear equations. However, as finding each x_u and y_i is actually independent (e.g. A_u, V_u only depend on u given R), so the good news is, this whole procedure can be done **in parallel**!

To do this, it suffices to send the rating matrix R to each process (distributed by rows and columns separately) to calculate A_u and V_u (or A_i and V_i) and broadcast X or Y after each iteration.

Furthermore, if we use a `pandas.DataFrame` object to store the rating matrix R, all above steps can be done by simply using two `pandas.DataFrame.groupby` objects grouped by `user_id` and `item_id` respectively, according whether the sum is carried out for $i \in U(u)$ or $u \in I(i)$.

In our code (`als_wr.py`), we use the module `multiprocessing` to achieve parallelization, where all processors actually share the memory and the copying and broadcasting are done automatically by generating mirror copies in memory.

Now let's see some experimental results. Let's first take look at the MovieLens dataset 100k (`dataset='movielens'` and `size='S'`). The training and testing RMSE after each iteration is given in Figure 1.

Here we take $r = 10$ and $\lambda = 0.065$. The analysis in the function `perf_weak` in main function shows that after 20 iterations, the average time for each iteration is **10.21 seconds** with a final testing RMSE equal to **0.9860**. Notice that without parallelization, the average time for each iteration was more than 25 seconds. We see that the algorithm converges but there is a gap between training errors and testing errors, which is very common in collaborative filtering.

Then for the toy data set, we take $r = 20$ and use the same λ. The corresponding RMSE curve is shown in Figure 2. After 20 iterations, the average time for each iteration is **2.45 seconds** with a final testing RMSE equal to **0.8204**.

For `dataset='movielens'` and `size='M'`, the RMSE after each iteration is shown in Figure 3. In this case, the algorithm needs **158.68 seconds** on average for each iteration and gives a final testing RMSE equal to **0.8924**.

At last, we run our ALS-WR on the Jester dataset, see Figure 4. As the computation for training errors takes much time, we omitted this in the figure. Each iteration takes **396 seconds** and the algorithms finally gives an RMSE of **4.8066**.

In conclusion, ALS-WR algorithm has a good enough performance on RMSE. And to our knowledge, ALS-WR works relatively fast compared to the approaches that we adopted in the same context.

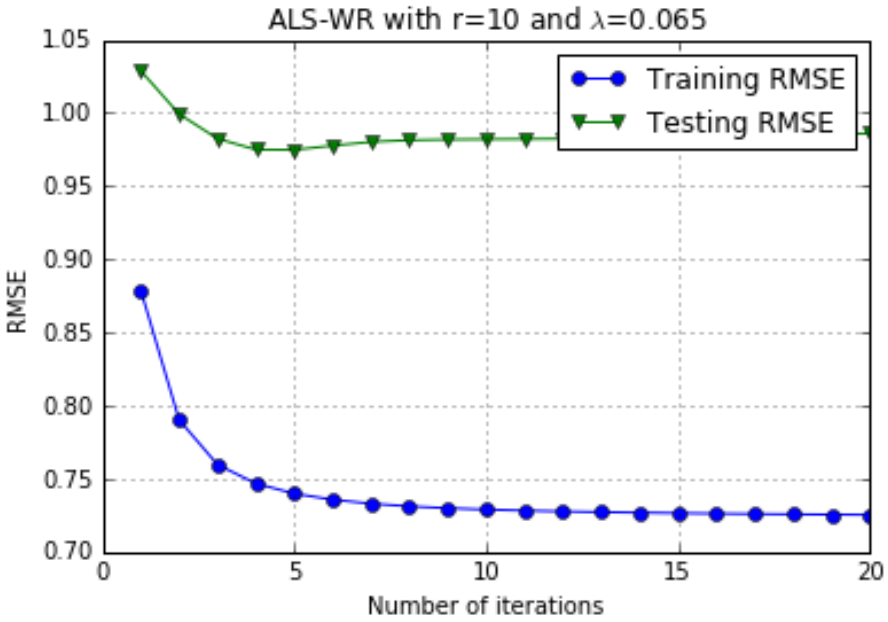

Figure 1: ALS-WR with dataset='movielens', size='S'

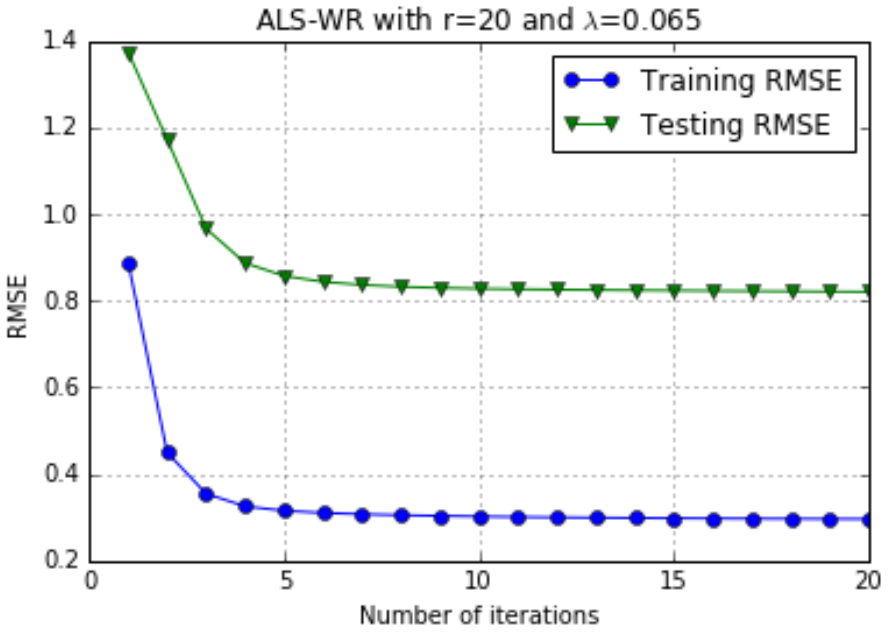

Figure 2: ALS-WR with dataset='toy'

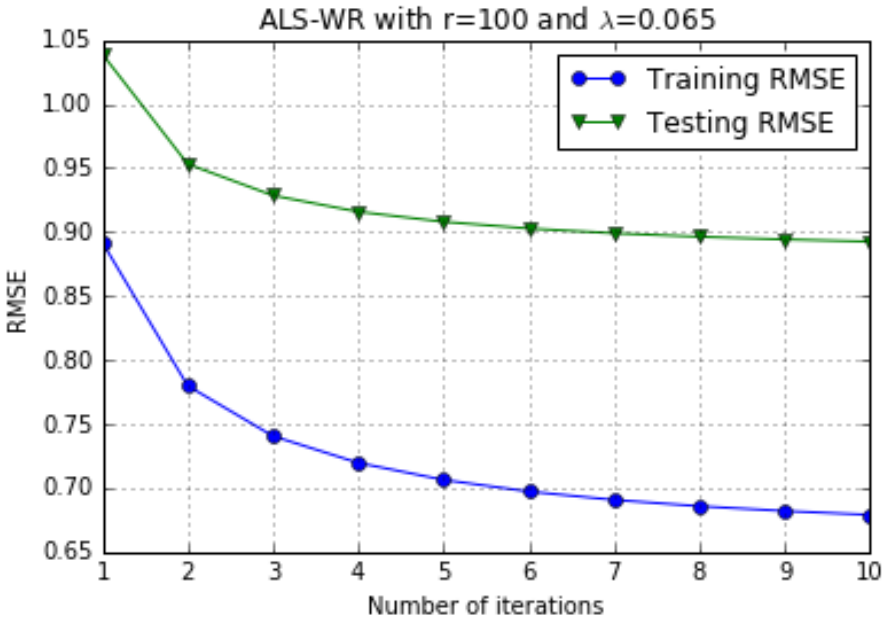

Figure 3: ALS-WR with dataset='movielens', size='M'

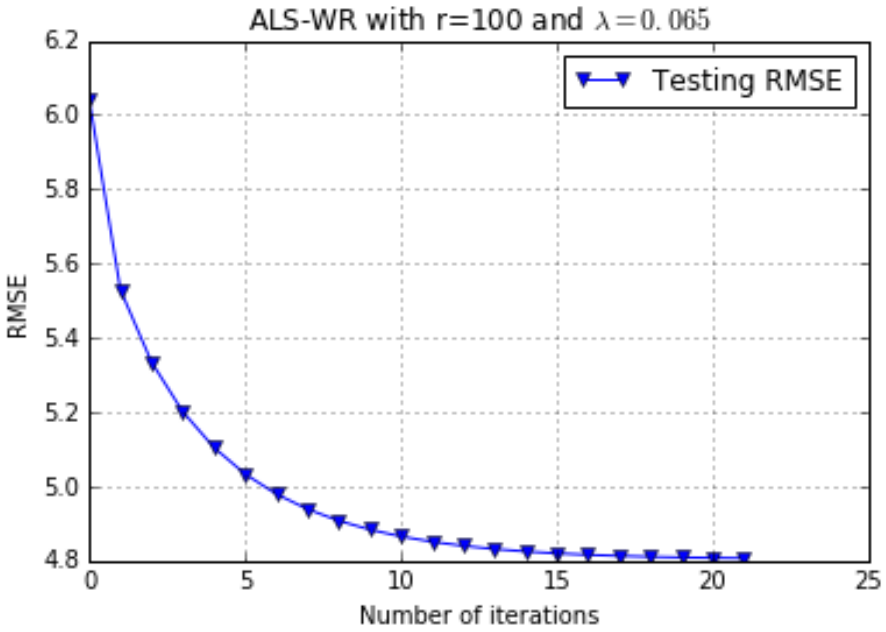

Figure 4: ALS-WR with dataset='jester'

4.3. Probabilistic Matrix Factorization (PMF)

Choice of the optimization algorithm

Computing the cost and gradient for the entire training set can be very slow and sometimes intractable especially for big datasets. SGD tackles this issue by following the negative gradient of the objective after seeing only a single or few training examples. In our case, each parameter update in SGD is computed with respect to a minibatch (instead of a few training examples in the general case). Moreover, SGD is sensitive to the order in which data is presented to the algorithm and can bias the gradient and thus lead to poor convergence. In order to avoid this issue, we randomly shuffle the data prior to each epoch of training. Eventually, if the objective function 11 has the form of a long shallow ravine leading to the optimum and steep walls on the sides, standard SGD will tend to oscillate across the narrow ravine since the negative gradient will point down one of the steep sides rather than along the ravine towards the optimum. Momentum is the method for pushing the objective more quickly along the shallow ravine that we use in our algorithm.

Details of training

We could perform gradient descent (batch learning on the whole dataset) but it would be slower and may not fit the memory with large datasets. That is why, to speed up the training, we subdivise the dataset into mini-batches and update the user and item vectors after each k mini-batches. Basically, SGD computes the gradient using one sample. In the case of our loss (least square loss), the "noisier" gradient computed using few samples tends to jerk the model out of a local minima into a region that could be more optimal. Single samples are really noisy while the mini-batches tend to average a little of the noise out. Thus, the amount of jerk is reduced when using minibatches. A good balance is struck when the minibatch size is small enough to avoid some of the poor local minima, but large enough that it doesn't avoid the global minima or better-performing local minima. Also, another benefit of SGD is that it is computationally faster. Indeed, for large datasets that can't be held in the RAM vectorization is much less efficient and the gradient is hardly tractable. Minibatch SGD is, on the other hand, built small enough to be computationnally tractable.

How does the algorithm works

The version implemented is the one with fixed priors. As explained previously, PMF is a generative model where X and Y are introduced to estimate the matrix of ratings R. Since X and Y follow normal distributions, they are initialized such that they follow random normal distributions. The minimization of the sum-of-squared-errors objective 11 is performed using SGD on both X 9 and Y 10. For each epoch, we first shuffle the data in case it presents a particular order , then we perform a mini-batch SGD to get the predicted X 9 and Y 10 and compute a first predicted rating. Notice that the epoch doesn't imply a full pass on the data, which doesn't really worsen the result since we have shuffled the data. Once we get X and Y, we use the momentum method and recompute the predicted rating after the momentum step. We can eventually compute the RMSE. More precisely, let us take the example of Movielens, size 'M'. We start by initializing X and Y, then, for each epoch (not necessarily a full pass on data), we perform mini-batch SGD using the 1000 batches. Each

batch contains 10000 samples on which gradient desecent is performed. Then, we apply the momentum method and finally recompute the predicted rating using the new values of X and Y obtained by the momentum method. For each epoch, we compute the RMSE for both the training and the test sets.

Results

We did not implement the version with automatic control complexity that allows us to automatically get the parameters. Thus, we had to find the suitable parameters to use for each dataset. We could perform cross-validation on the the small datasets but not on the large ones since it is computationally very expensive. Moreover, since we have 8 parameters to tune with our algorithm, cross-validation was hard to perform. We used a dictionary for each parameters and we vary a single parameter, the other being fixed and then plot the test and training RMSE to select visually the best parameter (where both derivatives for train and test graphs tend towards 0). Sometimes, it was almost impossible to decide (see Figure 5). For example, this happens for the dataset Movielens S, having all parameters fixed except the number of latent features that we want to tune, we plot the RMSE by number of latent features for both test and train datasets. However, it is visually hard to select a good number of latent feature since the derivatives are not obvious as we can see below.

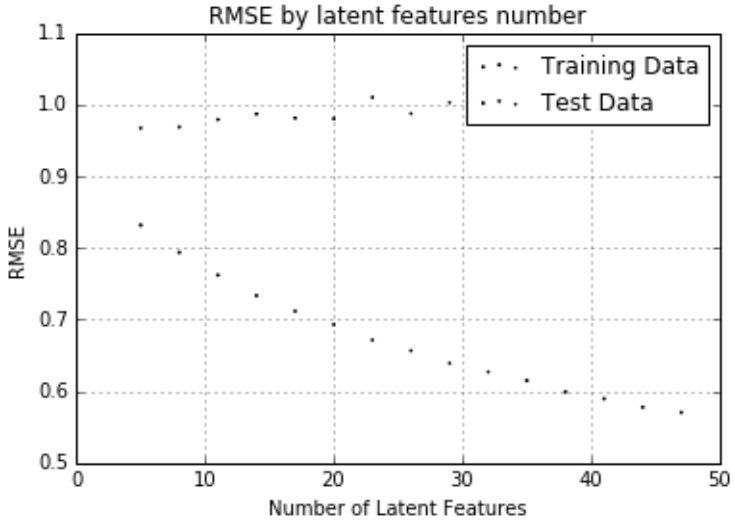

Figure 5: Movielens 100K: Case when it is hard to select visually the best number of latent features given all other parameters fixed.

Generally, as far as possible, for toy and movielens S datasets, we tune parameters by varying them individually. For Jester and movielens M which are much larger, we proceeded visually to select the number of epochs and we use dictionaries for batch-size and number of batches given that it should intuitively be higher (but not too high) for large datasets. We particularly focused to select the satisfying number of latent features (see Figure 6) and batch-size since they highly influence the performance of our algorithm.

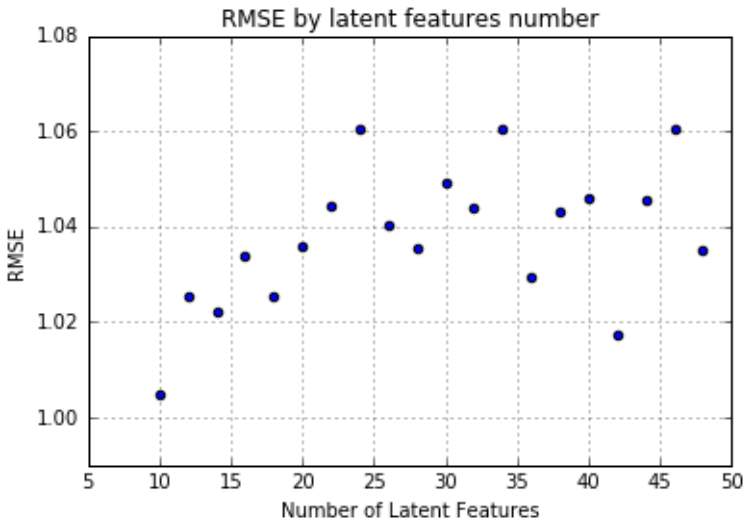

Figure 6: Movielens dataset, 100K : Influence of the number of latent features given all parameters fixed and taking the RMSE of the 10^{th} epoch.

After many trials, we took the parameters which gave us the best results. Once we fixed the parameters, we could compute the RMSEs.

We compute RMSE for both the training set and the test set. As expected the test RMSE is higher. The difference between the two gives some idea of how much we have over-fit. Some difference is always expected, but a very low RMSE on the training set with a high RMSE on the test set is a definite sign of over-fitting.

The following table gathers the results (RMSE and the total time of execution) for each dataset as well as the parameters used to get the mentioned results.

As we can read, the time execution is very low for small datasets. If we compare movienlens S (100K) which is 10 times smaller than movielens M (1M), we have to wait more than 150 times for the large dataset. The parameters of samll datsets,namely toy and movielens S present almost the same parameters chosen such that we stop before it starts to over-fit. The number of batches as well the size of each batch are relatively small (approximately 10 times smaller) compared to movielens M and Jester.

In conclusion, PMF works pretty well and is teh most efficient if we take into account the time needed to compute. It is true that it is less accurate for large datasets but at the opposite of the two other models, it fits the memory and is fast. Eventually, it would have been more accurate if we could find a way to tune simultaneously 8 parameters.

4.4. Non-linear Matrix Factorization with Gaussian Processes (NLMFGP)

Experimental setup

Following the setup of [7] and [13] to allow for direct comparison, we use SGD for all experiments with a learning rate of 1×10^{-4} and a momentum of 0.9. Regarding the Gaussian Process, latent variables are initialized at random using a spherical Gaussian $\mathcal{N}(0, 1\times10^{-3}\cdot I)$;

Parameters	Toy dataset	Movielens		Jester
		100K	1M	
Number of latent features	20	20	100	100
Learning rate	1	1	1	1
Penalization	0.1	0.1	0.1	0.1
Momentum	0.95	0.95	0.9	0.9
Max-epochs	20	20	10	10
Number of batches	100	100	1000	100
Batch size	1000	1000	10000	10000
RMSE	**0.460**	**0.940**	**0.884**	**4.601**
Total time of execution (s)	8	10	1432	313

Table 1: Performance of the PMF algorithm (RMSE, total execution time) for the four datasets using weak generalization

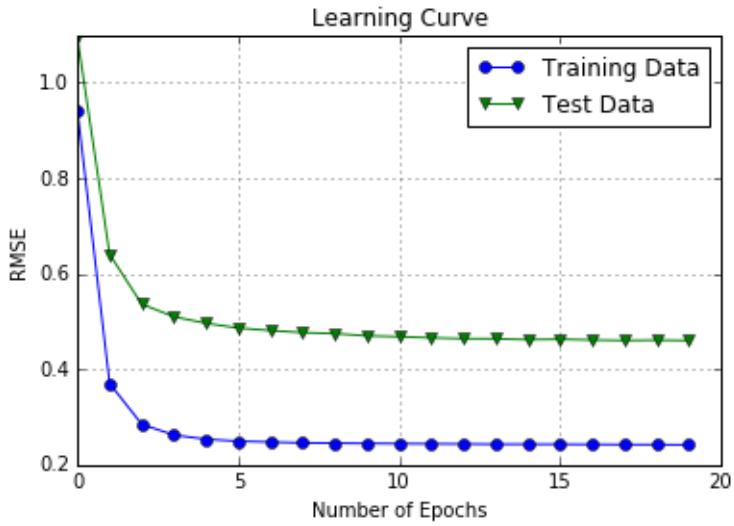

Figure 7: Toy dataset : Performance of PMF on both train and test data. The y-axis displays RMSE and the x-axis shows the number of epochs (in this case not necessarily a full pass on the entire training dataset).

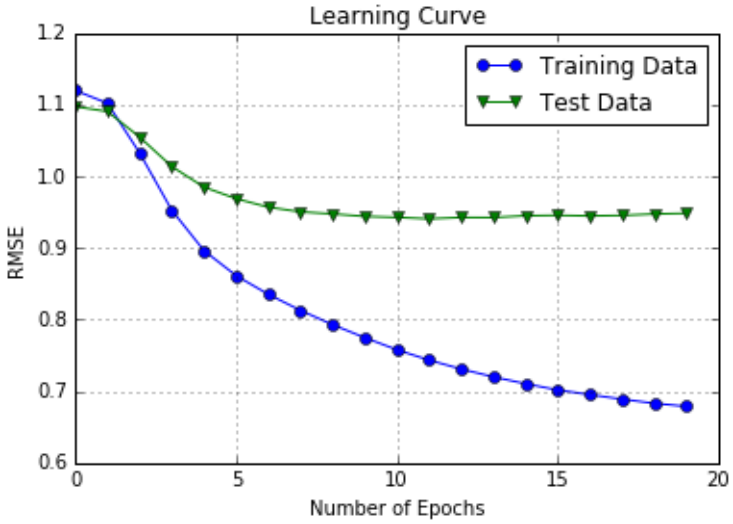

Figure 8: Movielens dataset, 100K : Performance of PMF on both train and test data. The y-axis displays RMSE and the x-axis shows the number of epochs (in this case not necessarily a full pass on the entire training dataset).

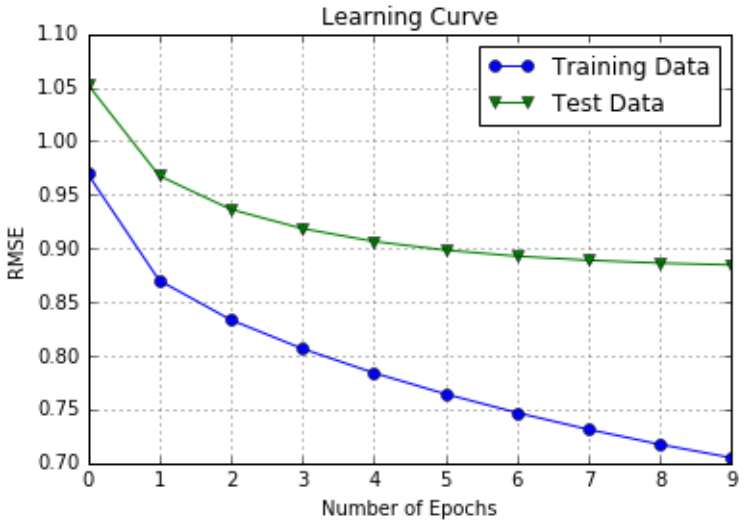

Figure 9: Movielens dataset, 1M : Performance of PMF on both train and test data. The y-axis displays RMSE and the x-axis shows the number of epochs (in this case not necessarily a full pass on the entire training dataset).

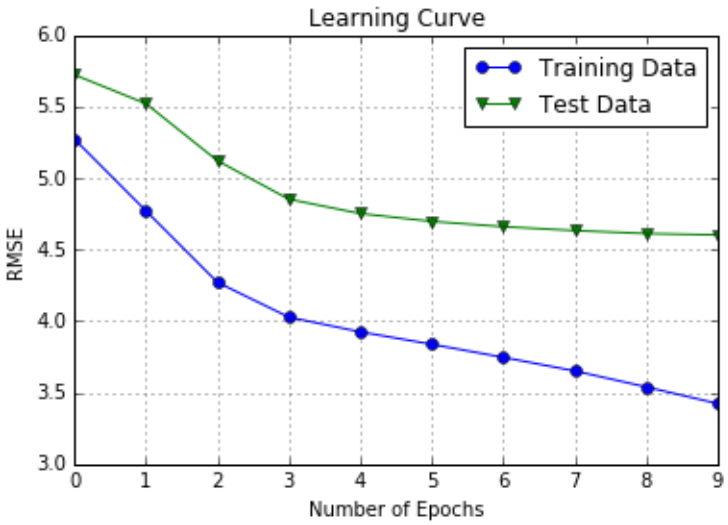

Figure 10: Jester dataset : Performance of PMF on both train and test data. The y-axis displays RMSE and the x-axis shows the number of epochs (in this case not necessarily a full pass on the entire training dataset).

covariance parameters are set to 1 except for the noise and bias variance set to 5 and 0.11 respectively.

Results

We present in Table 2 this method's results on three datasets. Note that the Jester dataset was excluded from the results because of its large runtime.

Dataset	Latent dimensions	Weak NMAE	Weak RMSE	Avg runtime per epoch (s)
Toy	11	0.0547007	0.59248172	0.51s
100k MovieLens	6	0.4491699	0.91917628	6.06s
1M MovieLens	11	0.4284306	0.87452705	490s

Table 2: Results of the NLMFGP method

We also study the evolution of the errors (RMSE and NMAE) with respect to the number of latent dimensions r in Figure 11. It is interesting to note that using more dimensions does not necessarily improve the results, as it may lead to overfitting. For the small dataset used in Figure 11, a small r is best, but more dimensions are needed for larger datasets.

We compare our results to baselines, the linear variant of [7] (which is the one we implemented), and to competing approaches in Table 3. This is made possible by the NMAE metric, which enables direct comparison.

We first note a good agreement between our implementation's and the paper's results, especially on the Weak RMSE metric. The RBF variant of NLMFGP outperforms other methods, while only using a small latent dimension $r = 14$ (compared to e.g. $r = 100$ for MMMF). We also remark the performance improvements brought to MMMF by ensemble

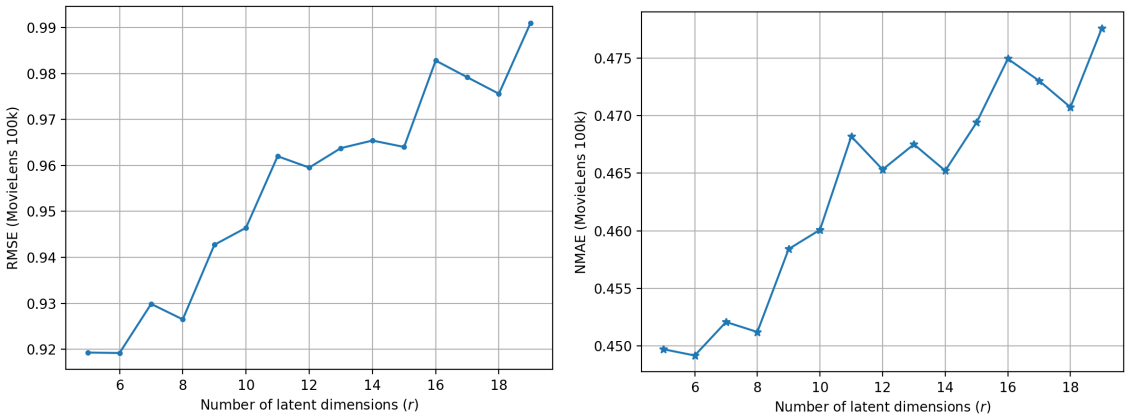

Figure 11: Evolution of RMSE (left) and NMAE (right) with respect to the number of latent dimensions (dataset: MovieLens 100k).

methods (bagging, multiple random initializations of the weights, combining the models with various voting schemes). Similar aggregation can be carried out with the NLMFGP method, and the authors indeed report in [7] an improvement to 0.3987 NMAE when ensembling 11 models with $r = 5 \ldots 15$.

Method	Average weak NMAE	Weak RMSE
URP [13]	0.4341	
Attitude [13]	0.4320	
MMMF [3]	0.4156	
Item [14]	0.4096	
E-MMMF [15]	0.4029	
NLMFGP Linear [7]	0.4052	0.8801
Our implementation	0.4284	0.8745
NLMFGP RBF [7]	0.4026	

Table 3: Comparison to state of the art methods on the MovieLens 1M dataset.

4.5. Comparison of the three models

In order to compare the three different algorithms, we will focus on the performance based on the RMSE which is the common evaluation metric. Besides, we all used weak generalization since ALS-WR does not lend itself to strong generalization. We used the same number of latent-features and epochs to be able to compare the algorithm. However, one should be cautious when comparing in terms of number of iterations or number of epochs since PMF does not perform a full pass on the training set per iteration since it uses a mini-batch SGD whereas ALS-WR and NLMFGP perform a full pass on users.

In terms of RMSE, the table suggests following facts. PMF has relatively good performance for small dataset, e.g. toy dataset. On the contrary, ALS-WR doesn't work well for small datasets, compared to other two models.

We notice that all three algorithms have similar performance for MovieLens 1M dataset, which is probably due to the fact that the number of users and number of items are almost of the same order in this dataset, thus making the model more robust.

As for the largest dataset (Jester), PMF slightly outperforms ALS-WR in terms of accurracy but it really stands out in terms of time. Indeed, it only takes 312 s for PMF to get a better result than ALS-WR which takes more than two hours. This is mostly due to the mini-batch SGD used to find the point estimate. NLMFGP would give a similar result but is computationally very expensive.

According to the different results, we can highlight the fact that PMF and NLMFGP have more similar results compared to ALS-WR which may due to the fact that NLMFGP is an extension of PMF.

Model	Toy dataset	Movielens 100K	Movielens 1M	Jester
ALS-WR	0.8204	0.9860	0.8569	4.8066
PMF	0.460	0.940	0 .884	4.601
NLMFGP	0.5925	0.9192	0.8745	–

Table 4: RMSE for different dataset - Comparison of the three implemented methods on the tested datasets.

5. Summary and discussion

In this work, we have explored the state-of-the-art of Collaborative Filtering through Matrix Factorization and have studied in-depth three important methods in this field, namely Alternating least square with weighted regularization (ALS-WR), Probabilistic Matrix Factorization with fixed priors (PMF) and Non Linear Matrix Factorization with Gaussian Process (NLMFGP). These three models have been tested on 4 different size datasets. In our case, it turned out each model performs better on a given dataset. Though, PMF with mini-batch SGD is the fastest one and still gives satisfying predictions.

In a nutshell, ALS-WR is an approach slightly adapted from the classical ALS algorithm. The idea is very simple and has good performance in general. ALS algorithm outweighs other approaches especially when the rating matrix is relatively sparse. In addition, the algorithm can be very fast with a parallel implementation.

PMF is a probabilistic approach that scales linearly with the number of observations and more importantly, performs well on the large and sparse datasets. We implemented the version with fixed priors for which we selected the suitable parameters. We demonstrated that PMF can be efficiently trained and successfully applied to large datasets such as Movielens 1M. PMF with fixed priors gives satisfying results with an excellent computation time cost thanks to the mini-batch SGD optimization. However, it would be interesting to implement teh version with automatic control capacity which improves the performance since the choice of parameters is not fully optimized. As an improvement path, it should be worth wile to take a fully Bayesian approach using MCMC methods to perform inference. This approach may be computationally expensive but leads to a significant gain in predictive accuracy.

NLMFGP is an extension of the PMF approach, but introduces a different viewpoint as it shows the equivalence of PMF with Probabilistic Principal Components Analysis. In the framework of Gaussian Processes, it becomes easier to integrate non-linear kernels, which improves the results [7]. Furthermore, GPs output variance estimates for each prediction. In future work, it should be worthwhile to experiment with ensemble methods, including several aggregation schemes. Our implementation only made use of linear kernels, but non-linear ones such as RBF are known to achieve better results [7]. However, those typically come with more hyper-parameters that are challenging to tune when working with full-size datasets, considering our limited computational resources.

References

[1] C. C. Aggarwal, Recommender Systems: The Textbook, Springer, 2016.

[2] Y. Koren, R. Bell, C. Volinsky, et al., Matrix factorization techniques for recommender systems, Computer 42 (8) (2009) 30–37.

[3] N. Srebro, J. D. Rennie, T. S. Jaakkola, Maximum-margin matrix factorization., in: NIPS, Vol. 17, 2004, pp. 1329–1336.

[4] J. D. Rennie, N. Srebro, Fast maximum margin matrix factorization for collaborative prediction, in: Proceedings of the 22nd international conference on Machine learning, ACM, 2005, pp. 713–719.

[5] T. Hofmann, Latent semantic models for collaborative filtering, ACM Transactions on Information Systems (TOIS) 22 (1) (2004) 89–115.

[6] R. Salakhutdinov, A. Mnih, Probabilistic matrix factorization, in: NIPS, Vol. 20, 2011, pp. 1–8.

[7] N. D. Lawrence, R. Urtasun, Non-linear matrix factorization with gaussian processes, in: Proceedings of the 26th Annual International Conference on Machine Learning, ACM, 2009, pp. 601–608.

[8] R. Salakhutdinov, A. Mnih, Bayesian probabilistic matrix factorization using markov chain monte carlo, in: Proceedings of the 25th international conference on Machine learning, ACM, 2008, pp. 880–887.

[9] S. Zhang, W. Wang, J. Ford, F. Makedon, Learning from incomplete ratings using non-negative matrix factorization., in: SDM, Vol. 6, SIAM, 2006, pp. 548–552.

[10] R. Gemulla, E. Nijkamp, P. J. Haas, Y. Sismanis, Large-scale matrix factorization with distributed stochastic gradient descent, in: Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2011, pp. 69–77.

[11] Y. Zhou, D. Wilkinson, R. Schreiber, R. Pan, Large-scale parallel collaborative filtering for the netflix prize, in: International Conference on Algorithmic Applications in Management, Springer, 2008, pp. 337–348.

[12] F. Guillou, R. Gaudel, P. Preux, Collaborative filtering as a multi-armed bandit, in: NIPS'15 Workshop: Machine Learning for eCommerce, 2015, pp. 1–9.

[13] B. Marlin, Collaborative filtering: A machine learning perspective. master's thesis, university of toronto, Computer Science Department.

[14] S.-T. Park, D. M. Pennock, Applying collaborative filtering techniques to movie search for better ranking and browsing, in: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2007, pp. 550–559.

[15] D. DeCoste, Collaborative prediction using ensembles of maximum margin matrix factorizations, in: Proceedings of the 23rd international conference on Machine learning, ACM, 2006, pp. 249–256.