UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Stochastic Gradient Methods for Matrix Completion

Dissertation presented by
Gauthier MUGUERZA

for obtaining the Master's degree in
Mathematical Engineering

Supervisors
Pierre-Antoine ABSIL, Rodolphe SEPULCHRE

Readers
Estelle MASSART, Shuyu DONG

Academic year 2017-2018

Abstract

A common hurdle for handling high-dimensional data is the fact that the observations are often incomplete. The low-rank matrix completion problem attempts to overcome this hurdle by recovering this missing data in a large data set. The most famous application of low-rank matrix completion are recommender systems, but predicting missing values in a database is used in a diverse set of fields including control, system identification and statistics, just to name a few.

The method used in this thesis is based on optimization on manifolds, since the rank is fixed beforehand, and the error is minimized. The number of data points being very large, we use a stochastic gradient descent method. This method is scaled, i.e., efficient preconditioners are computed to resolve the issue of scale invariance. Two families of sequences of data points are defined, "Smart shuffling" and "Weighted shuffling", in order to improve the slow convergence obtained when the data points are visited in a cyclic order.

Numerical comparisons show that the smart shuffling performs like a random sequence, and the weighted shuffling improves the convergence.

Key words. low-rank matrix completion, Riemannian optimization, fixed-rank manifold, stochastic gradient descent methods, shuffling methods, importance sampling.

AMS subject classifications. 15A83, 65K99.

To my parents.

Acknowledgments

After a member of my family introduced me to Prof. Sepulchre, I knew I wanted to work with him. His sense of excellence and emotional intelligence were clear signs that he would be a great supervisor. When he suggested to work with Prof. Absil, I immediately accepted, since their long-lasting collaboration has been so fruitful that the idea of being a part of it was really an honor. I am grateful to them for making it possible to carry out this work, both in the University of Cambridge and in the Université catholique de Louvain. Without their continuous input and support, this thesis would have hardly been completed.

I sincerely want to thank the PhD students and post-docs with whom I had the chance to work. Cyrus Mostajeran, Guillaume Olikier and Christian Grussler, thank you for your patience and all the insightful discussions we had. Special thanks to Estelle Massart, Bamdev Mishra and Léopold Cambier, three brilliant researchers who generously shared both their work and their time with me.

My fellow students and friends, Alexandre Olikier, Quentin Lété and Harold della Faille, have accepted to review this thesis and to give me precious comments and ideas. Many thanks go to them as well.

Shuyu Dong provided me with interesting data sets for the matrix completion problem, including the one used in Section 5.2.

Last but not least, many thanks to my family and friends for their help and support.

Louvain-la-Neuve, June 2018

Contents

Notation

Variables

$\mathbf{A}$	A matrix, usually of size $d_1 \times d_2$ unless stated otherwise.
$\mathbf{A}^\top$	The transpose of matrix $\mathbf{A}$.
A_{ij}	The (i, j) entry of matrix $\mathbf{A}$.
$d_{\max}$	The maximum between d_1 and d_2: $d_{\max} = \max\{d_1, d_2\}$.
$d_{\min}$	The minimum between d_1 and d_2: $d_{\min} = \min\{d_1, d_2\}$.
$\mathcal{M}$	A Riemannian smooth manifold.

Sets

$\mathbb{R}$	The set of real numbers.
$[m]$	The set of integers from 1 to m: $[m] = \{1, 2, \ldots, m\}$.
Ω	The set of observed entries of a matrix of size $d_1 \times d_2$: $\Omega \subseteq ([d_1] \times [d_2])$.
$\bar{\Omega}$	The set of unknown entries of a matrix of size $d_1 \times d_2$: $\bar{\Omega} = ([d_1] \times [d_2]) \setminus \Omega$.
$\mathbb{R}^{d_1 \times d_2}$	The set of matrices of size $d_1 \times d_2$.
$\mathbb{R}_r^{d_1 \times d_2}$	The set of matrices of size $d_1 \times d_2$ and of rank r. $= \left\{\mathbf{X} \in \mathbb{R}^{d_1 \times d_2} : \operatorname{rank}(\mathbf{X}) = r\right\} = \mathcal{M}_r$.
$\operatorname{St}(r, d)$	The compact Stiefel manifold of orthogonal matrices of size $d \times r$. $= \left\{\mathbf{X} \in \mathbb{R}^{d \times r} : \mathbf{X}^\top \mathbf{X} = \mathbf{I}_{r \times r}\right\}$.
$\mathbb{R}_*^{d \times r}$	The non-compact Stiefel manifold of full-column rank matrices of size $d \times r$. $= \left\{\mathbf{X} \in \mathbb{R}^{d \times r} : \operatorname{rank}(\mathbf{X}) = r\right\} = \mathbb{R}_r^{d \times r}$.
$\operatorname{GL}(r)$	The General Linear group of order r, i.e., the set of all invertible matrices of size $r \times r$. $= \{\mathbf{T} \in \mathbb{R}^{r \times r} : \det(\mathbf{T}) \neq 0\}$.

Operations

$\|\cdot\|_{\ell_p}$	The ℓ_p norm: $\|\mathbf{X}\|_{\ell_p} = \left(\sum_{i=1,j=1}^{d_1,d_2} \lvert X_{ij}\rvert^p\right)^{1/p}$.
$\|\cdot\|_{\ell_2} = \|\cdot\|_F$	The ℓ_2 norm or Frobenius norm: $\|\mathbf{X}\|_F = \left(\sum_{i=1,j=1}^{d_1,d_2} \lvert X_{ij}\rvert^2\right)^{1/2}$.
$\mathcal{P}_\Omega(\mathbf{X})$	The orthogonal projector of $\mathbf{X}$ onto the space of $d_1 \times d_2$ matrices with 0 on $\bar{\Omega}$.
R_x	A retraction on $\mathcal{M}$, that is, a smooth function from $T_x\mathcal{M}$ to $\mathcal{M}$ with $x \in \mathcal{M}$.

Abbreviations

ALS Alternating Least Squares

AUC Area Under the Curve

CV Cross-Validation

GDPR General Data Protection Regulation

GPU Graphics Processing Unit

IRLS Iteratively Reweighted Least Squares

LRMC Low-Rank Matrix Completion

MAE Mean Absolute Error

MSE Mean Squared Error

NMAE Normalized Mean Absolute Error

NPC Non-Positive Curvature

RCGMC Robust Conjugate Gradient Matrix Completion

RMSE Root Mean Squared Error

SGD Stochastic Gradient Descent

SPD Symmetric Positive Definite

SVD Singular Value Decomposition

Introduction

After the famous technology company IBM stated that 90 % of the data in the world today has been created in the last two years, certain questions about good and bad arose. This data comes from everywhere: sensors used to gather shopping habits, interaction on social media, digital pictures and videos, and financial transactions, just to name a few. This data is *big* data. Therefore, using data mining tools has become crucial. *Inferring information from partial data* is one of the most important challenges we are facing. This master's thesis deals with one of the solutions to this problem, namely the low-rank matrix completion problem, which can be stated as *trying to recover a low-rank matrix given only a sampling of its entries.*

In October 2006, the entertainment company Netflix launched its now famous competition: the Netflix Prize. It challenged anyone to achieve a higher accuracy than their current recommendation system, which was called CineMatch. The prize was $1M to be awarded to the team attaining the highest improvement. The winner was declared in June 2009, "BellKor's Pragmatic Chaos" having bettered the root mean squared error (RMSE) by over 10 %, see [BK07]. One of their main insights was to learn how to optimally blend different predictors to yield a more accurate predictor. The Netflix Prize was a prime example of a recommender system: the system tried to predict a rating that a user would give for an item. Recommender systems are just one of the many applications of matrix completion. Indeed, such problems are of considerable interest in a diverse set of fields including control, system identification, statistics and signal processing.

Although the general low-rank matrix completion problem is NP-hard, there exist several heuristic methods that solve the problem approximately by solving the convex relaxation of the original problem. Nowadays, plenty of powerful tools for this problem have been developed. The most important tool used to solve the low-rank matrix completion problem, can be stated as follows: *often, data live in low-dimensional spaces.* Optimization on manifolds, also called Riemannian optimization is described in [AMS08], while papers like [BMAS14, Bon13, Van13] use these concepts to develop powerful algorithms. Stochastic gradient descent is illustrated in [MS16, TFBJ18, RR13]. Low-rank matrix completion is a trendy problem, and solutions include [BA11, Cam15].

All of these tools achieve great results, in terms of personalization and predictions. The advances are so impressive that the European Union has introduced the famous General Data Protection Regulation (GDPR) in order to help all citizens of the EU with data protection and privacy issues. The goal of the GDPR is to enable citizens to control their personal information, and to simplify the regulatory environment for international business. Data-driven companies now have to deal with the consequences of their powerful algorithms: the public is more and more afraid of providing information, and recent papers like [VBK18] provide some guidelines on how to implement algorithms while respecting the rights and obligations laid out in GDPR.

In the recent paper [MHA18], the authors have tried new ways of browsing the data set to perform Riemannian optimization. In particular, the authors took the Karcher function as the

objective function, that is, the least squares mean of matrices, and the search space was the manifold of symmetric positive definite (SPD) matrices, endowed with an affine-invariant metric. In this framework, an order defined by a specific shuffling performed (slightly) better than a random order. The reason for this could be the fact that the last data points receive too much importance, i.e., they are overemphasized. This could be the case for all Hadamard manifolds, that is, all the complete and simply connected manifolds with non-positive sectional curvature. Inversely, the phenomenon of underemphasized data points could be observed on manifolds with non-negative sectional curvature. A reminder about the sectional curvature of manifolds can be found in [Lee03].

This master's thesis aims at answering a simple question: *is it useful to look at the known data in a particular order, to solve the low-rank matrix completion problem?* In particular, the starting point is to try the shuffling of [MHA18] on the low-rank matrix completion problem, and then to develop other ideas to improve the stochastic gradient descent algorithm.

This report is structured in the following way: the problem and the current state-of-the-art are presented in Chapter 1. We cover the main tools of Riemannian optimization in Chapter 2. After that, Chapter 3 states some different aspects of stochastic gradient descent, and Chapter 4 collects our contribution, which looks to improve the stochastic gradient descent method. Finally, in Chapter 5, we test our methods on three different real-world data sets.

Following the discipline of reproducible research, the source code and data files required to reproduce the experimental results of this master's thesis can be downloaded from `https://bitbucket.org/gauthmug/rsgd/src/master/`.

Chapter 1

Low-rank matrix completion

In this first chapter, we start with Section 1.1 by formally defining the problem. A short summary of the state-of-the-art is given in Section 1.2. Finally, Section 1.3 explains why it is interesting to solve the low-rank matrix completion problem.

1.1 Problem definition

The low-rank matrix completion problem, also called the LRMC problem, is the task of filling in the missing entries of a matrix from which only a subset of the entries are observed. Of course, in the sampled entries, we have some noise or even some outliers. If the original matrix was perfectly noiseless, the problem could be stated as recovering a matrix $\mathbf{M} \in \mathbb{R}^{d_1 \times d_2}$, observed only on a subset Ω of its entries, by finding a matrix $\mathbf{X} \in \mathbb{R}^{d_1 \times d_2}$ such that

$$X_{ij} = M_{ij}, \ \forall \ (i,j) \in \Omega, \tag{1.1}$$

where the matrix $\mathbf{X}$ has the lowest possible rank. The reader can find some linear algebra reminders in Appendix A.1, including a basic definition of the rank. Intuitively, the rank of a matrix can be thought of as the *complexity* of a matrix: if a matrix has a low rank, it means that many of its columns (or rows) are linearly dependent, hence these columns (or rows) do not give more information about the matrix; if a matrix has a high rank, it means that its columns (or rows) are very different from one another, hence every column (or row) adds more information about the matrix.

Here we stress the fact that $\mathbf{M}$ is given only on Ω. In other words, the sample set Ω is defined as the set of all (i,j) such that M_{ij} is sampled, i.e., M_{ij} is observed. Rigorously, the entry M_{ij} is sampled if and only if $(i,j) \in \Omega$.

The equality constraint (1.1) is often expressed differently. For this, the *orthogonal sampling operator* first needs to be defined. It is a projection $\mathcal{P}_\Omega : \mathbb{R}^{d_1 \times d_2} \to \mathbb{R}^{d_1 \times d_2}$ defined as

$$\left[\mathcal{P}_\Omega(\mathbf{X})\right]_{ij} = \begin{cases} X_{ij}, & \text{if } (i,j) \in \Omega, \\ 0, & \text{otherwise.} \end{cases}$$

Hence, the constraint (1.1) can be written as

$$\mathcal{P}_\Omega\left(\mathbf{X}\right) = \mathcal{P}_\Omega\left(\mathbf{M}\right). \tag{1.2}$$

A standard way of approaching the problem is to minimize the rank. However, in this thesis we will cast the problem as a fixed-rank optimization problem, assuming that the rank r is known *a priori*, i.e.,

$$\begin{aligned} \min_{\mathbf{X}\in\mathbb{R}^{d_1\times d_2}} \quad & \frac{1}{2}\left\|\mathcal{P}_\Omega\left(\mathbf{X}\right)-\mathcal{P}_\Omega\left(\mathbf{M}\right)\right\|_F^2 \\ \text{subject to} \quad & \operatorname{rank}\left(\mathbf{X}\right)=r. \end{aligned}$$

Example 1.1. *Consider the partially sampled matrix* $\mathbf{M}$ *defined as*

$$\mathbf{M}=\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & \star \\ \star & \star & 9 \end{bmatrix},$$

where each unsampled entry is denoted as $\star$*. We have* $\Omega=\{(1,1);(1,2);(1,3);(2,1);(2,2);(3,3)\}$*. There are an infinite number of completions of* $\mathbf{M}$*. Let us give two examples, namely the matrices*

$$\mathbf{X}_1=\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix}, \quad \mathbf{X}_2=\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 7 & 9 \end{bmatrix}.$$

We see that $\operatorname{rank}\left(\mathbf{X}_1\right)=1$*, while* $\operatorname{rank}\left(\mathbf{X}_2\right)=2$*. Hence, if the rank* r *is fixed to* 1 *for this problem, then* $\mathbf{X}_1$ *is a solution to the LRMC problem, while* $\mathbf{X}_2$ *is not. Of course, this is only a small example: in real applications the matrices that we are trying to fill are much larger.*

1.2 Related work

Formally, the LRMC problem can be regarded as an optimization problem, as follows

$$\begin{aligned} \min_{\mathbf{X}\in\mathbb{R}^{d_1\times d_2}} \quad & \operatorname{rank}\left(\mathbf{X}\right) \\ \text{subject to} \quad & \mathcal{P}_\Omega\left(\mathbf{X}\right)=\mathcal{P}_\Omega\left(\mathbf{M}\right). \end{aligned} \tag{1.3}$$

Looking at this formulation, the difficulties are not apparent. However, rank minimization is (almost always) an intractable problem: it is in fact provably NP-hard and the only known algorithm to solve it is computationally too expensive, as it is shown in [CG84].

The solution suggested by [CR09] was to rewrite the problem as

$$\begin{aligned} \min_{\mathbf{X}\in\mathbb{R}^{d_1\times d_2}} \quad & \left\|\mathbf{X}\right\|_* \\ \text{subject to} \quad & \mathcal{P}_\Omega\left(\mathbf{X}\right)=\mathcal{P}_\Omega\left(\mathbf{M}\right), \end{aligned} \tag{1.4}$$

where $\left\|\mathbf{X}\right\|_*$ is the nuclear norm of $\mathbf{X}$ defined as the sum of the singular values of $\mathbf{X}$. The nuclear norm is in fact the Schatten norm with $p=1$. A reminder about matrix norms is given in Appendix A.2. A family of low-rank inducing norms and regularizers, which includes the nuclear norm as a special case, was recently introduced in [GG16]. The authors of [FHB03] have shown that the nuclear norm is the tightest convex relaxation of the rank. In the context of *exact* LRMC, i.e., if the strict constraint (1.1) is imposed, the authors have shown that this formulation recovers the original underlying matrix under some mild assumptions. Namely, if the number $|\Omega|$ of sampled entries obeys

$$|\Omega|\geq Cd^{\frac{6}{5}}r\log\left(d\right),$$

for some positive numerical constant C, then with very high probability, most $d_1 \times d_2$ matrices of rank r can be perfectly recovered by solving a simple convex optimization program. Here, it is assumed that $d = d_1 = d_2$, but this discussion would apply to arbitrary rectangular matrices as well. The advantage of focusing on square matrices is a simplified exposition and reduction in the number of parameters we need to keep track of.

Exact matrix completion, i.e., with the constraint (1.1), is used in many problems, e.g., in fluid dynamics as in [ZJG17]. Strict equality is imposed, similarly to the case of covariance completion, where precise measurements have to be matched. However, one can see that this constraint is very strict. Indeed, in real applications, the observations are noisy, thus imposing this equality can cause problems. These problems come from the fact that the matrix $\mathbf{X}$ tries to perfectly match $\mathbf{M}$, including the noise. This phenomenon is called *overfitting*; in short it refers to the situation where the model fits the known entries too well, while deviating too much from the mean on the unknown entries. Overfitting and underfitting are important concepts in machine learning, and they will be introduced in Section 4.1. This being said, let us now look at the problem where strict equality must not be imposed. In order to avoid overfitting, the authors of [CP10] have defined the problem

$$\begin{aligned} \min_{\mathbf{X} \in \mathbb{R}^{d_1 \times d_2}} \quad & \|\mathbf{X}\|_* \\ \text{subject to} \quad & \|\mathcal{P}_\Omega(\mathbf{X}) - \mathcal{P}_\Omega(\mathbf{M})\|_F \leq \delta, \end{aligned} \tag{1.5}$$

for some small value of $\delta \in \mathbb{R}$. Of course, since the projection $\mathcal{P}_\Omega$ is linear, it is equivalent to write $\mathcal{P}_\Omega(\mathbf{X}) - \mathcal{P}_\Omega(\mathbf{M})$ or $\mathcal{P}_\Omega(\mathbf{X} - \mathbf{M})$. The second formulation is neater, but one should remember that the matrix $\mathbf{X} - \mathbf{M}$ cannot be computed, since $\mathbf{M}$ is only known on the set Ω.

The methods described above are focused on minimizing the rank, or some relaxation of the rank, and the constraint ensures that the new matrix is close to the original data. Now the tables are turned: the rank is fixed *a priori*, and the error between the new matrix and the original data is minimized. Hence, the difference between the matrices $\mathbf{X}$ and $\mathbf{M}$ on the set Ω is minimized. If the rank r of the target matrix is fixed *a priori*, then the problem can be stated as

$$\begin{aligned} \min_{\mathbf{X} \in \mathbb{R}^{d_1 \times d_2}} \quad & \frac{1}{2} \|\mathcal{P}_\Omega(\mathbf{X} - \mathbf{M})\|_F^2 \\ \text{subject to} \quad & \operatorname{rank}(\mathbf{X}) = r. \end{aligned} \tag{1.6}$$

A regularization term is often added in order to avoid overfitting. Hence, a term $\lambda \|\mathbf{X}\|_F^2$ would be added to the objective function, for some non-negative regularization parameter $\lambda \geq 0$. The optimization algorithm then tries to ensure that $\|\mathbf{X}\|_F^2$ is as small as possible. One can think of the term $\|\mathbf{X}\|_F^2$ as a way to quantify the model complexity. Usually, a factor $\frac{1}{2}$ is added in front such that the gradient is easier to write. The careful reader will readily have noticed that overfitting is already avoided by the constraint $\operatorname{rank}(\mathbf{X}) = r$ with a small value for the rank r. This is a good point, but fixing the rank of $\mathbf{X}$ to a small value does not avoid its Frobenius norm to explode. Hence, some formulations include the term $\frac{\lambda}{2} \|\mathbf{X}\|_F^2$ in the objective function, but we will always set $\lambda = 0$. Indeed, since stochastic gradient descent (SGD) methods are used in this thesis, we are not fully optimizing the cost function, and it is unlikely that $\|\mathbf{X}\|_F^2$ explodes. In real applications however, a non-zero value for λ can be defined, since this usually allows to improve the performance on unseen data.

Regarding the constraint $\operatorname{rank}(\mathbf{X}) = r$, one might ask why we did not write $\operatorname{rank}(\mathbf{X}) \leq r$ instead. In turns out that the search space could be defined as $\mathcal{M}_{\leq r}$, with

$$\mathcal{M}_{\leq r} = \left\{ \mathbf{X} \in \mathbb{R}^{d_1 \times d_2} : \operatorname{rank}(\mathbf{X}) \leq r \right\},$$

but in practice the optimization variable $\mathbf{X}$ always has the highest rank possible. The search space can thus be written as $\mathcal{M}_r$, with

$$\mathcal{M}_r = \left\{ \mathbf{X} \in \mathbb{R}^{d_1 \times d_2} : \operatorname{rank}(\mathbf{X}) = r \right\}.$$

This space, which is a set, can be endowed with a manifold structure, a concept that will be introduced in Chapter 2, see Definition 2.3. Hence, the machinery of optimization on manifolds described in [AMS08] can be used. The general philosophy of optimization on manifolds is to recast a constrained optimization problem in the Euclidean space into an unconstrained optimization on a non-linear search space that encodes the constraint. The idea of a direct optimization over the set of all fixed-rank matrices, with a generalization of classical non-linear conjugate gradients, leads to the famous low-rank geometric conjugate gradient method, often denoted by LRGeomCG, described in [Van13].

Algorithms for low-rank matrix completion based on preconditioned Riemannian optimization over a single Grassmannian manifold are presented in [BA15, BA11]. The authors have defined a method called "RCGMC" for Robust Conjugate Gradient Matrix Completion. Another version called "RCGMC 2" was developed by applying second-order Riemannian trust-region methods.

Now that recent methods have been defined, let us focus on the model. A popular way to tackle the rank-constraint in (1.6) is by using a factorization model. Suppose that one wishes to recover a rectangular $d_1 \times d_2$ matrix $\mathbf{M}$ of rank r. Such a matrix $\mathbf{M}$ can be represented by $d_1 d_2$ numbers, but it only has $(d_1 + d_2 - r)\, r$ degrees of freedom. Indeed, it is stated in [HM94, Proposition 1.14] that the dimension of the manifold of $d_1 \times d_2$ matrices of rank r is $(d_1 + d_2 - r)\, r$, as long as $\max\{d_1, d_2\} > 1$. Among others, this can be revealed by counting the number of parameters in the singular value decomposition (SVD), i.e., the number of degrees of freedom associated with the description of the singular values and of the left and right singular vectors.

Let us talk about the procedure defined in [CCS10] to define synthetic data. A fraction of the entries are randomly removed with uniform probability. Since the dimension of $\mathcal{M}_r$ is $(d_1 + d_2 - r)\, r$, the number of known entries is a multiple of this dimension. This multiple is called the *over-sampling ratio*, often abbreviated by OS ratio. The OS ratio determines the number of known entries. For example, if OS $= 5$, it means that $|\Omega| = 5\,(d_1 + d_2 - r)\, r$ of randomly and uniformly selected entries are known *a priori* out of a total of $d_1 d_2$ entries.

We are reminded in [Bon13] that a gradient descent method could be applied to solve problem (1.6). However, the matrix is potentially of high dimension, $d_1 \approx 10^6$ and $d_2 \approx 10^5$ in the Netflix case. Hence, computing the full gradient is not feasible. Therefore, a standard method to reduce the computational burden is to draw random elements of Ω, say, element (i, j), and perform gradient descent ignoring the remaining entries. The updated matrix

$$\mathbf{X}_+ = \mathbf{X} - \alpha \nabla_{\mathbf{X}} \left(M_{ij} - X_{ij} \right),$$

is obtained, where α is the step-size. Unfortunately, this matrix does not have rank r. Seeking the matrix of rank r which best approximates it can be numerically costly, especially for very large d_1 and d_2. Basically, the idea would be to project $\mathbf{X}_+$ on the manifold of fixed-rank matrices, i.e., $\mathcal{M}_r$. This would involve a SVD. Instead, a more natural way to enforce the rank constraint is to endow the parameter space with a Riemannian metric, and to perform a gradient step *within the manifold* $\mathcal{M}_r$. This approach of Riemannian optimization will be at the core of this thesis.

Finally, two recent PhD theses were very useful to get acquainted with the subject of LRMC. The thesis [Mis14] is primarily motivated by the LRMC problem that is viewed as a least squares problem on a matrix manifold with symmetries, whereas optimization on manifolds is exhaustively discussed in [Bou14].

1.3 Motivations

The problem of imputing missing values of a matrix appears in dozens of applications. In this section, three applications are highlighted, taken from very different fields.

The first application is related to *healthcare*. As it is pointed out in [CMY+16], a medical record is the practitioner's most important tool to analyze and monitor a patient's health status. However, such records are usually incomplete, due to unpunctuality, incomplete electronic health records, incorrect trial data collection, absence of patients, and so on. One can use matrix completion to resolve this missing data problem, and therefore avoid the multiplication of medical measurements, without affecting the quality of the treatments. Another healthcare-related topic from [JMC+16] uses matrix completion for genotype imputation, which could enable genome-wide association studies to increase the chance of a causal SNP being identified. As a reminder, single-nucleotide polymorphism (SNP) is a variation in a single nucleotide that occurs at a specific position in the genome. Hence, this application positions itself in the famous Human Genome Project, often abbreviated to HGP.

The second application is related to *traffic planning*. Many network management, planning, and optimization applications rely on accurate and complete traffic measurement. However, inevitably, traffic flow data are often incomplete or lost. There can be numerous reasons for that, from sensor failure, transmission error, high measurement costs and unreliable network transport protocols. As a result, various matrix completion methods were proposed to estimate missing traffic data, among which those presented in [YCY+17, LLL14].

Finally, probably the most famous application relates to *recommender systems*. Such a system seeks to predict the preference a user would give to an item. In the context of collaborative filtering, the system first collects a large amount of information on the users' activities. Then, by analyzing this data and by looking at similarities to other users, it tries to predict what a particular user will like. Another similar approach is called content-based filtering. It tries to match the description of an item with the preferences of a user by using keywords, and the history of the user. Nowadays, a combination of both is used. Hybrid recommender systems are essential for many companies like the GAFA, i.e., Google, Amazon, Facebook and Apple. The celebrated Netflix prize presented in [BL07] was one of the key events that energized research in recommender systems.

Chapter 2

Riemannian optimization

This chapter describes the essential tools of Riemannian optimization, also called optimization on manifolds. It is greatly inspired from [Cam15, AMS08]. We refer the reader to the latter reference for a more rigorous introduction.

Riemannian optimization is a branch of optimization where the search space is a Riemannian manifold. First, the notion of a Riemannian manifold is defined, which is a set often denoted by $\mathcal{M}$. Therefore, Sections 2.1 and 2.2 review some key elements of Riemannian geometry. Then, the topic of interest is studied, which is the optimization of a differentiable function $f : \mathcal{M} \to \mathbb{R}$. The generic formulation is as follows

$$\min_{x \in \mathcal{M}} f(x). \tag{2.1}$$

Thus, Section 2.3 gives the main differences between Riemannian optimization and classical Euclidean optimization. As it was said earlier, the general idea behind optimization on manifolds is to rewrite a constrained optimization problem in the Euclidean space into an unconstrained optimization problem on a non-linear search space, called a manifold, that encodes the constraint. Finally, more intuition about the manifold of interest for the LRMC problem is given in Section 2.4.

2.1 The manifold structure

Informally, one can think of an n-dimensional manifold $\mathcal{M}$ as a set that can be identified to $\mathbb{R}^n$ *locally* through some one-to-one correspondence. *Formally*, we start by defining $\mathcal{M}$ as a set. Then, the concepts of *charts* and *atlases* need to be defined.

Definition 2.1 (Chart)**.** *An n-dimensional* chart φ *of $\mathcal{M}$ from $\mathcal{U} \subseteq \mathcal{M}$ to an open subset $Y \subseteq \mathbb{R}^n$ is a bijection from $\mathcal{U}$ to Y. It is denoted by $(\mathcal{U}, \varphi)$.*

Intuitively, a chart locally assigns some coordinates in Y to a point of $\mathcal{U}$. However, in order for the algorithms to be well-defined, we need to have charts that give compatible information. Therefore, each point of the set $\mathcal{M}$ must be at least in one chart domain $\mathcal{U}$. Furthermore, if a point $x \in \mathcal{M}$ belongs to the domains of two charts, say $(\mathcal{U}_1, \varphi_1)$ and $(\mathcal{U}_2, \varphi_2)$, then the two charts must "match". The concept of atlas takes these requirements into account.

Definition 2.2 (Atlas)**.** *An* atlas $\mathcal{A}$ *of $\mathcal{M}$ into $\mathbb{R}^n$ is a collection of charts $(\mathcal{U}_\alpha, \varphi_\alpha)$ of $\mathcal{M}$ such that the following two conditions are satisfied.*

1. *The union corresponds to the manifold:* $\bigcup_\alpha \mathcal{U}_\alpha = \mathcal{M}$.

2. *For any pair* α, β *with* $\mathcal{U}_\alpha \cap \mathcal{U}_\beta \neq \emptyset$, *the sets* $\varphi_\alpha(\mathcal{U}_\alpha \cap \mathcal{U}_\beta)$ *and* $\varphi_\beta(\mathcal{U}_\alpha \cap \mathcal{U}_\beta)$ *are open sets in* $\mathbb{R}^n$, *and the change of coordinates*

$$\varphi_{\alpha\beta} = \varphi_\beta \circ \varphi_\alpha^{-1} : \mathbb{R}^n \mapsto \mathbb{R}^n$$

is smooth on its domain $\varphi_\alpha(\mathcal{U}_\alpha \cap \mathcal{U}_\beta)$. *The change of coordinates* $\varphi_{\alpha\beta}$ *is often called the* transition map. *We say that the elements of an atlas* overlap smoothly. *A smooth function is formally defined in [AMS08].*

The careful reader will already have noticed that many atlases can be defined for a given set $\mathcal{M}$. Indeed, two atlases $\mathcal{A}_1$ and $\mathcal{A}_2$ are said to be *equivalent* if $\mathcal{A}_1 \cup \mathcal{A}_2$ is an atlas. To get rid of all ambiguity, $\mathcal{A}^+$ is defined as the *maximal atlas* of $\mathcal{M}$ such that it contains all possible charts of $\mathcal{M}$. The atlas $\mathcal{A}^+$ can also be called the *complete atlas*.

We now have the required tools to define a manifold.

Definition 2.3 (Manifold)**.** *An* n*-dimensional manifold is a couple* $(\mathcal{M}, \mathcal{A}^+)$ *where* $\mathcal{M}$ *is a set and* $\mathcal{A}^+$ *is a maximal atlas from* $\mathcal{M}$ *to* $\mathbb{R}^n$.

We say that the manifold is smooth since all the transition maps are smooth, that is, derivatives of all orders exist. In the present document, unless stated otherwise, all manifolds are considered to be smooth.

The definitions of this section allow us to formalize the idea that a manifold $(\mathcal{M}, \mathcal{A}^+)$ is locally equivalent to $\mathbb{R}^n$. We can also simply talk about "the manifold $\mathcal{M}$" if the maximal atlas is obvious from the context.

2.2 Two types of manifolds

There exist two very different manifold structures of interest, namely *embedded* and *quotient manifolds*. Both are manifolds in the sense of Section 2.1, but they are conceptually quite different.

2.2.1 Embedded manifolds

On the one hand, *embedded manifolds* are manifolds that can easily be described by constraints in the ambient $\mathbb{R}^n$ space. They are easy to handle because it is often straightforward to visualize them as curved shapes in the ambient space. Let us give two examples, respectively Example 5.25 and Example 5.30 from [Lee03]. First, the sphere

$$\mathbb{S}_{n-1} = \{\mathbf{x} \in \mathbb{R}^n : \|\mathbf{x}\|_2 = 1\},$$

is an embedded manifold of $\mathbb{R}^n$. Indeed, it is simply defined by constraints on $\mathbb{R}^n$, and it coincides with the previous definition of a manifold. The set of low-rank matrices

$$\mathbb{R}_r^{d_1 \times d_2} = \mathcal{M}_r = \left\{\mathbf{X} \in \mathbb{R}^{d_1 \times d_2} : \operatorname{rank}(\mathbf{X}) = r\right\}, \tag{2.2}$$

can also be seen as an embedded manifold of $\mathbb{R}^{d_1 \times d_2}$.

2.2.2 Quotient manifolds

On the other hand, *quotient manifolds* are manifolds described by means of equivalence classes. A point on the manifold will be a class, and it will be represented by one element of the class. The conventions of [AMS08, Section 3.6.2] will be followed for the notations. Let $\overline{\mathcal{M}}$ be a manifold equipped with an equivalence relation $\sim$, i.e., a relation that is reflexive, symmetric and transitive. The set of all elements that are equivalent to a point $x \in \overline{\mathcal{M}}$ is called the *equivalence class* containing x, and is defined as

$$[x] = \left\{ y \in \overline{\mathcal{M}} : y \sim x \right\}.$$

The set of all equivalence classes of $\sim$ in $\overline{\mathcal{M}}$ is called the *quotient of* $\overline{\mathcal{M}}$ *by* $\sim$, and is defined as

$$\mathcal{M} = \overline{\mathcal{M}} / \sim = \left\{ [x] : x \in \overline{\mathcal{M}} \right\}.$$

Notice that the points of $\mathcal{M}$ are subsets of $\overline{\mathcal{M}}$. The mapping $\pi : \overline{\mathcal{M}} \mapsto \mathcal{M}$ defined by $x \mapsto [x]$ is called the *natural projection* or *canonical projection*. Clearly, $\pi(x) = \pi(y)$ if and only if $x \sim y$, so we have $[x] = \pi^{-1}(\pi(x))$.

The set $\overline{\mathcal{M}}$ is called the *total space*, or the *product space*, of the quotient $\mathcal{M}$. Now we will show that $\mathcal{M}$ is a *quotient manifold*, or a *quotient space*, by defining it.

Let $\left(\overline{\mathcal{M}}, \mathcal{A}^+\right)$ be a manifold with an equivalence relation $\sim$ and let $\mathcal{B}^+$ be a manifold structure on the set $\mathcal{M}$. The manifold $(\mathcal{M}, \mathcal{B}^+)$ is called a *quotient manifold* of $\left(\overline{\mathcal{M}}, \mathcal{A}^+\right)$ if the natural projection π is a submersion. As a reminder, the mapping π is a submersion if its differential (that is, the best linear approximation of π) is everywhere surjective. We know from [AMS08, Proposition 3.4.1] that if $\overline{\mathcal{M}}$ is a manifold and $\mathcal{M}$ is a quotient of $\overline{\mathcal{M}}$, then $\mathcal{M}$ admits at most one manifold structure that makes it a quotient manifold of $\overline{\mathcal{M}}$. Given a quotient $\mathcal{M}$ of a manifold $\overline{\mathcal{M}}$, the set $\mathcal{M}$ is called a *quotient manifold* if it admits a (unique) quotient manifold structure. In this case, the equivalence relation $\sim$ is said to be *regular*, and the set $\mathcal{M}/\sim$ endowed with this manifold structure is referred to as "the manifold $\mathcal{M}$".

In Section 2.4, these notions are applied to our manifold of interest.

2.3 The steepest descent algorithm

As a reminder, this section is again inspired from [Cam15]. We want to generalize the steepest descent algorithm, from its Euclidean version

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k),$$

to the Riemannian one. To do so, some notions must be introduced. To understand these notions, Figure 2.1, on page 12, might prove useful.

2.3.1 Tangent space and Inner product

A way to visualize smooth manifolds is to view them as curved shapes in the ambient space. The smoothness of the manifold enables us to associate a *tangent space*, denoted by $T_x\mathcal{M}$, to each point $x \in \mathcal{M}$. Locally, the tangent space is a first-order approximation of the manifold, and it is

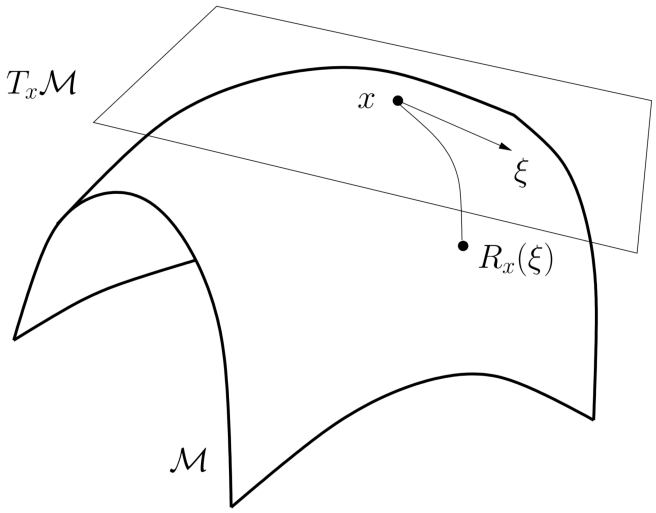


Figure 2.1: Essential tools of Riemannian optimization: tangent space $T_x\mathcal{M}$, tangent vector ξ and retraction $R_x(\xi)$. Courtesy of [AMS08].

a vector space. The elements of a tangent space are called *tangent vectors*, and they generalize the notion of directional derivatives. In order to characterize which direction of motion from x produces the steepest increase in f, we further need a notion of length that applies to tangent vectors.

Given a point $x \in \mathcal{M}$ and a tangent space $T_x\mathcal{M}$, one can define an inner product between two vectors $\xi_x \in T_x\mathcal{M}$ and $\eta_x \in T_x\mathcal{M}$. The subscript x denotes that the tangent vectors are in the tangent space $T_x\mathcal{M}$, however we will sometimes abuse notation and remove this subscript x when the tangent space is obvious from the context. This inner product is defined as

$$\langle \cdot, \cdot \rangle_x : T_x\mathcal{M} \times T_x\mathcal{M} \to \mathbb{R} : (\xi_x, \eta_x) \to \langle \xi_x, \eta_x \rangle_x,$$

with the usual properties of the inner product, i.e., it must be a bilinear, symmetric and positive-definite form. Note that, in general, this inner product depends on the point x, since the tangent space depends on x as well. The norm of a vector $\xi_x \in T_x\mathcal{M}$ is naturally defined as

$$\|\xi_x\|_x = \sqrt{\langle \xi_x, \xi_x \rangle_x}.$$

2.3.2 Riemannian metric and Riemannian distance

A manifold whose tangent spaces are endowed with a smoothly varying inner product is called a *Riemannian manifold*. The smoothly varying inner product g is called the *Riemannian metric*. The following notations

$$g(\xi_x, \eta_x) = g_x(\xi_x, \eta_x) = \langle \xi_x, \eta_x \rangle = \langle \xi_x, \eta_x \rangle_x$$

will be equivalently used to denote the inner product of two elements ξ_x and η_x in $T_x\mathcal{M}$.

The *Riemannian distance* d between two points x and y on a Riemannian manifold $(\mathcal{M}, g)$ is introduced. It can be shown that the Riemannian distance is a metric, because it is a symmetric and positive-definite form, that respects the triangular inequality. We refer the reader to [AMS08, Section 3.6] for a rigorous definition.

2.3.3 Gradient

The most crucial notion in smooth optimization is that of the gradient. With the definition of the norm in the above lines, the normalized direction of steepest ascent is defined as

$$\underset{\xi \in T_x\mathcal{M}:\|\xi\|_x=1}{\operatorname{argmax}} \mathrm{D}f(x)[\xi],$$

where $\mathrm{D}f(x)[\xi]$ is the Fréchet derivative of f at x in the direction ξ. Here, one should write ξ_x instead of ξ but the x was dropped to ease notations. A reminder of the Fréchet derivative can be found in [BN08] or in Appendix A.3.

With this, the *gradient* of f at x, denoted by $\operatorname{grad} f(x)$, is defined as the unique element of $T_x\mathcal{M}$ that satisfies

$$\langle \operatorname{grad} f(x), \xi \rangle_x = \mathrm{D}f(x)[\xi] \ \ \forall\ \xi \in T_x\mathcal{M}.$$

It is no surprise, yet it is a remarkable result that the direction of $\operatorname{grad} f(x)$ is the steepest ascent direction of f at x. In other words, we have

$$\frac{\operatorname{grad} f(x)}{\|\operatorname{grad} f(x)\|} = \underset{\xi \in T_x\mathcal{M}:\|\xi\|_x=1}{\operatorname{argmax}} \mathrm{D}f(x)[\xi].$$

This being said, when working with embedded manifolds, we can find an easy way to compute the gradient. The function f is defined from $\mathcal{M}$ to $\mathbb{R}$. Let us assume that $\mathcal{M}$ is an n-dimensional manifold. Then we define $\bar{f}$ as an extension of f from $\mathcal{M}$ to $\mathbb{R}^n$, i.e., $\bar{f}$ is defined from $\mathbb{R}^n$ to $\mathbb{R}$. The Euclidean gradient of $\bar{f}$ at x is computed, and we use $\nabla_x \bar{f}(x) \in \mathbb{R}^n$ to denote this direction. Finally, this direction is projected onto the tangent space at x. Therefore, we define $\mathcal{P}_x$ as the orthogonal projector from $\mathbb{R}^n$ onto $T_x\mathcal{M}$. We obtain

$$\operatorname{grad} f(x) = \mathcal{P}_x\left(\nabla_x \bar{f}(x)\right).$$

2.3.4 Exponential mapping and Retraction

In order to solve problem (2.1), we need to move while staying on the manifold. Therefore, let us assume that we have a direction $\xi \in T_x\mathcal{M}$. The algorithm starts at a point $x \in \mathcal{M}$, and moves in the direction of the tangent vector ξ.

We define a geodesic as a parameterized curve γ that is locally distance-minimizing with respect to the Riemannian metric d. The exponential map $\operatorname{Exp}_x(\xi) : T_x\mathcal{M} \to \mathcal{M}$ maps $\xi \in T_x\mathcal{M}$ to $y \in \mathcal{M}$ such that there is a geodesic with $\gamma(0) = x$, $\gamma(1) = y$ and $\dfrac{\mathrm{d}\gamma}{\mathrm{d}t}(0) = \xi$.

In many cases, the exponential map is not easy to compute. Instead, it is much easier to use the retraction as a first-order approximation of the exponential. If $R_x : T_x\mathcal{M} \to \mathcal{M}$ is a retraction mapping, then we use $R_x^{-1} : \mathcal{M} \to T_x\mathcal{M}$ to denote its inverse, when it is well-defined. Informally, the retraction from x in a direction $\xi \in T_x\mathcal{M}$, denoted as $R_x(\xi)$, is a point on the manifold $\mathcal{M}$ obtained by coming from x in the direction ξ. Formally, we refer to [AMS08, Definition 4.1.1].

For example, let us take the sphere, $\mathcal{M} = \mathbb{S}^{n-1}$, endowed with the natural metric inherited through immersion in $\mathbb{R}^n$. If we consider a point $x \in \mathcal{M}$ and a tangent vector ξ at this point, it is clear that the new point $x + \xi$ does not belong to the sphere. This is because the sphere is not a vector space. Hence, we define a retraction which simply consists of an addition in the ambient

space $\mathbb{R}^n$, followed by a projection onto the sphere. Numerically, this is a very simple operation compared to the exponential mapping which requires the explicit computation of the geodesic distance.

In that case, one might be tempted to see optimization on manifolds as a special case of a projected gradient descent method. However, this is not the case. On the one hand, with the projected gradient method, we compute a classical gradient in the ambient space and it is projected on the search space. On the other hand, with methods from optimization on an embedded manifold, we start by projecting the classical gradient to the tangent space. Then, when the retraction is a projection, we move in this direction, and we project on the embedded manifold. If the manifold is a sphere, the second method goes a bit further but the results can be vastly different on a "bumpy" manifold. Actually, the constraint defining the manifold is *encoded in the gradient*. In that viewpoint, the direction of the Riemannian gradient is more relevant than the direction of the classical Euclidean gradient because the latter does not take the constraint into account.

2.3.5 Steepest descent

We are now able to define the equivalent of the steepest descent algorithm on a manifold $\mathcal{M}$. As will be explained in Chapter 3, we start by choosing an initial iterate, say x_0, and we set $k = 0$. Until x_k is close enough to the optimum $x^\star$, the iterate is updated with

$$x_{k+1} = R_{x_k}\left(-\alpha_k \operatorname{grad} f\left(x_k\right)\right),$$

where α_k is a positive step-size, and then we set $k \leftarrow k+1$. Regarding the step-size, the Armijo's backtracking procedure is often used, as in [AMS08, Definition 4.2.2]. Different ways to pick the step-size are discussed in Section 3.5. It is worth noticing that $\alpha_k \operatorname{grad} f\left(x_k\right)$ is well-defined. Indeed, we have $\operatorname{grad} f\left(x_k\right) \in T_x\mathcal{M}$, $T_x\mathcal{M}$ is a vector space and $\alpha_k \in \mathbb{R}$.

2.4 The manifold of low-rank matrices

2.4.1 Quotient geometry of the full-rank factorization

In practice, the matrix $\mathbf{X} \in \mathcal{M}_r$ cannot be stored as a $d_1 \times d_2$ matrix. Indeed, it requires storing $d_1 d_2$ numbers, which can be equal to 10^{11} in the Netflix case. We know from Chapter 1 that the manifold dimension is $\left(d_1 + d_2 - r\right) r$. Since we often have $r \ll \min\left\{d_1, d_2\right\}$, it is clear that $d_1 d_2$ is much larger than the manifold dimension. Therefore, as it is suggested in [MMBS14, AO15], a popular way to parameterize fixed-rank matrices is through *matrix factorization*. One popular matrix factorization for fixed-rank non-symmetric matrices is reviewed, and we study the underlying Riemannian geometry of the resulting search space. This fixed-rank matrix factorizations is called the *full-rank factorization*. It arises from the thin singular value decomposition (SVD) of a rank-r matrix $\mathbf{X}$, i.e.

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top, \tag{2.3}$$

where $\mathbf{U}$ is a $d_1 \times r$ matrix with orthogonal columns, $\mathbf{\Sigma}$ is a $r \times r$ diagonal matrix with positive entries and $\mathbf{V}$ is a $d_2 \times r$ matrix with orthogonal columns. As a reminder, the Stiefel manifold is defined as $\operatorname{St}\left(r, d\right) = \left\{\mathbf{U} \in \mathbb{R}^{d \times r} : \mathbf{U}^\top\mathbf{U} = \mathbf{I}_{r \times r}\right\}$. The set $\operatorname{Diag}_{++}\left(r\right)$ is defined as the set of $r \times r$ diagonal matrices with positive entries. Hence, we have $\mathbf{U} \in \operatorname{St}\left(r, d_1\right)$, $\mathbf{\Sigma} \in \operatorname{Diag}_{++}\left(r\right)$

and $\mathbf{V} \in \mathrm{St}(r, d_2)$. The SVD exists for any matrix $\mathbf{X} \in \mathbb{R}_r^{d_1 \times d_2}$, as it is proved in [GV96]. The decomposition (2.3) is illustrated in Figure 2.2.

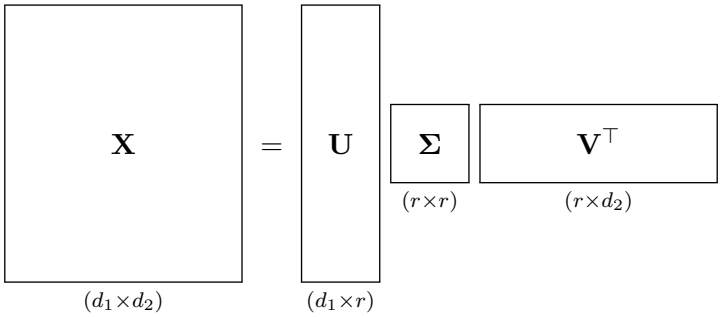


Figure 2.2: The thin singular value decomposition (SVD) of a rank-r matrix of size $d_1 \times d_2$.

Notice that $\mathbf{\Sigma} = \mathbf{\Sigma}^\top$. For the sequel we need to define $\mathbf{\Sigma}^{\frac{1}{2}}$, which is a diagonal matrix composed of the square root of the elements of $\mathbf{\Sigma}$. It is easy to check that $\mathbf{\Sigma} = \mathbf{\Sigma}^{\frac{1}{2}}\mathbf{\Sigma}^{\frac{1}{2}}$, and $\mathbf{\Sigma}^{\frac{1}{2}} = \left(\mathbf{\Sigma}^{\frac{1}{2}}\right)^\top$. The full-rank factorization is obtained when the SVD is rearranged as

$$\mathbf{X} = \left(\mathbf{U}\mathbf{\Sigma}^{\frac{1}{2}}\right)\left(\mathbf{\Sigma}^{\frac{1}{2}}\mathbf{V}^\top\right) = \mathbf{L}\mathbf{R}^\top, \tag{2.4}$$

where $\mathbf{L} = \mathbf{U}\mathbf{\Sigma}^{\frac{1}{2}} \in \mathbb{R}_*^{d_1 \times r}$ and $\mathbf{R} = \mathbf{V}\mathbf{\Sigma}^{\frac{1}{2}} \in \mathbb{R}_*^{d_2 \times r}$. We recall that $\mathbb{R}_*^{d \times r}$ is the set of *full column rank* $d \times r$ matrices. The factorization (2.4) is illustrated in Figure 2.3.

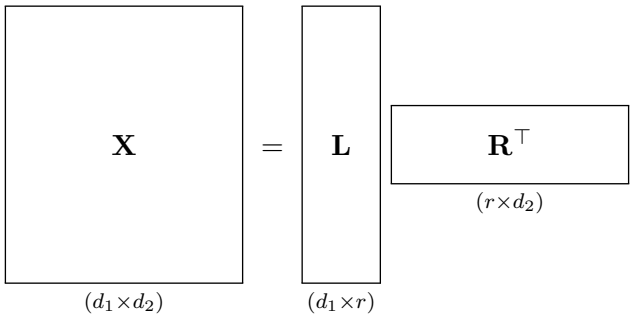


Figure 2.3: The full-rank factorization of a rank-r matrix of size $d_1 \times d_2$.

The resulting factorization is not unique because the transformation

$$(\mathbf{L}, \mathbf{R}) \mapsto \left(\mathbf{L}\mathbf{T}^{-1}, \mathbf{R}\mathbf{T}^\top\right) \tag{2.5}$$

leaves the original matrix $\mathbf{X}$ unchanged, where $\mathbf{T}$ is a regular $r \times r$ matrix, that is, an element of the General Linear group, $\mathbf{T} \in \mathrm{GL}(r) = \{\mathbf{T} \in \mathbb{R}^{r \times r} : \det(\mathbf{T}) \neq 0\}$. This symmetry stems from the fact that the row and column spaces are invariant to the change of coordinates.

The classical remedy to remove this indeterminacy in the case of symmetric positive semi-definite matrices is the Cholesky factorization. In the case of non-symmetric matrices, the LU decomposition is used. However, in the case of optimization on manifolds, these techniques will not be used. Instead, the invariance mapping (2.5) is encoded in an abstract search space by optimizing over a quotient manifold $\mathcal{M}_r$ that will be defined next. In other words, we cannot

simply optimize over the product space $\overline{\mathcal{M}}_r = \mathbb{R}_*^{d_1 \times r} \times \mathbb{R}_*^{d_2 \times r}$. Let us use the same notations as in Subsection 2.2.2. The equivalence relation $\sim$ is defined by

$$(\mathbf{L}, \mathbf{R}) \sim (\mathbf{G}, \mathbf{H}) \Leftrightarrow \exists\ \mathbf{T} \in \mathrm{GL}(r) : \mathbf{G} = \mathbf{L}\mathbf{T}^{-1}, \mathbf{H} = \mathbf{R}\mathbf{T}^{\top}. \tag{2.6}$$

The following proposition verifies that this is indeed an equivalence relation.

Proposition 2.1. *The relation defined in* (2.6) *is an equivalence relation.*

Proof. See Appendix B.1. □

The product space $\overline{\mathcal{M}}_r$ defined above can also be called the total space. The set of all elements that are equivalent to a point $x = (\mathbf{L}, \mathbf{R}) \in \overline{\mathcal{M}}_r$ is called the *equivalence class* containing x, and is defined as

$$[x] = [(\mathbf{L}, \mathbf{R})] = \left\{ \left(\mathbf{L}\mathbf{T}^{-1}, \mathbf{R}\mathbf{T}^{\top} \right) : \mathbf{T} \in \mathrm{GL}(r) \right\}. \tag{2.7}$$

The set of all equivalence classes of $\sim$ in $\overline{\mathcal{M}}_r$ is called the *quotient of* $\overline{\mathcal{M}}_r$ *by* $\sim$, and is defined as

$$\mathcal{M}_r = \overline{\mathcal{M}}_r / \sim = \left\{ [x] : x \in \overline{\mathcal{M}}_r \right\}.$$

We will thus use the notation $\overline{\mathcal{M}}_r / \mathrm{GL}(r)$ for $\overline{\mathcal{M}}_r / \sim$. The set $\mathrm{GL}(r)$ is called the *fiber space*. The set of equivalence classes is called the *quotient space* and it is denoted as

$$\mathcal{M}_r = \overline{\mathcal{M}}_r / \mathrm{GL}(r) = \left(\mathbb{R}_*^{d_1 \times r} \times \mathbb{R}_*^{d_2 \times r} \right) / \mathrm{GL}(r). \tag{2.8}$$

At this point, the reader might be confused about $\mathcal{M}_r$, the set of matrices of fixed rank. Is $\mathcal{M}_r$ an embedded or a quotient manifold? In fact, it can be viewed in both ways. Expressing $\mathcal{M}_r$ in the form of (2.2) shows the embedded structure. However, in the sequel we more often use the quotient formulation (2.8), since it is more useful here.

2.4.2 Retraction

Let us take a tangent vector $\xi_x \in T_x \overline{\mathcal{M}}_r$ with $\xi_x = (\xi_{\mathbf{L}}, \xi_{\mathbf{R}})$. Assume that we are at point $x \in \overline{\mathcal{M}}_r$, and we want to move in the direction ξ_x. If $x = (\mathbf{L}, \mathbf{R})$, then this is equivalent to saying two things: we are at $\mathbf{L} \in \mathbb{R}_*^{d_1 \times r}$, and we want to move in the direction $\xi_{\mathbf{L}}$; we are at $\mathbf{R} \in \mathbb{R}_*^{d_2 \times r}$, and we want to move in the direction $\xi_{\mathbf{R}}$. A simple and efficient retraction is provided by the following formulas

$$\begin{cases} R_{\mathbf{L}}(\xi_{\mathbf{L}}) = \mathbf{L} + \xi_{\mathbf{L}}, \\ R_{\mathbf{R}}(\xi_{\mathbf{R}}) = \mathbf{R} + \xi_{\mathbf{R}}. \end{cases} \tag{2.9}$$

This might seem weird, but since the considered embedded manifold has the form (3.16), it makes sense that the simplest retraction is a sum. The inverse retraction is defined as

$$\begin{cases} R_{\mathbf{L}}^{-1}(\xi_{\mathbf{L}}) = \xi_{\mathbf{L}} - \mathbf{L}, \\ R_{\mathbf{R}}^{-1}(\xi_{\mathbf{R}}) = \xi_{\mathbf{R}} - \mathbf{R}. \end{cases}$$

Chapter 3

Stochastic Gradient Descent methods

Given a cost function f, we wish to find $x^\star$ to minimize f. This chapter introduces important tools to achieve this goal. The most famous optimization algorithm for doing this is called the *gradient descent method*. However, this method is only a member of the family of *generic line search methods*. These methods consist in two important steps: determining a *search direction* and a *step-size*. This family is quickly defined in Section 3.1. In the category of generic line search methods, the *gradient descent method* is probably the most intuitive one to understand. Hence, Section 3.2 explains how it works, as well as its variant: the stochastic gradient descent method. Then, the gradients for our specific cost function are computed in Section 3.3. We try to improve the search direction by taking into account some scaling, which is explained in Section 3.4. Finally, Section 3.5 gives some theory about how to choose the step-size.

3.1 Generic line search

As we have already said in the introduction of this chapter, the generic line search method consists in two important steps: determining a *search direction* and a *step-size*. Let us assume that we are minimizing a function $f : \mathcal{M} \mapsto \mathbb{R}$, and that we have access to a retraction $R_x : T_x\mathcal{M} \mapsto \mathcal{M}$. We choose an initial iterate, say x_0, and we set $k = 0$. Until we think that x_k is close enough to the optimum $x^\star$, we go through four steps.

1. Compute a search direction p_k from the iterate x_k. Usually we want to ensure that this direction is a descent direction, such that for a small step away from x_k in the direction p_k, the objective function is reduced.

2. Compute a suitable step-size $\alpha_k > 0$. Usually we want to choose α_k such that

$$f\left(x_k + \alpha_k p_k\right) < f\left(x_k\right). \tag{3.1}$$

   The computation of the step-size is called a line search, and this is usually an inner iterative loop.

3. Update the iterate: $x_{k+1} \leftarrow R_{x_k}\left(\alpha_k p_k\right)$.

4. Update the iteration counter: $k \leftarrow k + 1$.

Note that in the sequel it cannot always be guaranteed that p_k is a descent direction. Furthermore, we will not always be able to state that α_k satisfies (3.1). Finally, the careful reader will have noticed that the third step can be simplified if $\mathcal{M} = \mathbb{R}^n$. Indeed, in this case, it becomes $x_{k+1} \leftarrow x_k + \alpha_k p_k$.

Let us look at a special case to illustrate a well-known example of the generic line search method. Assume that $f : \mathbb{R}^n \mapsto \mathbb{R}$ is a convex and differentiable function. A review of convex functions can be found in [BV04]. Let us further assume that the gradient of f is a Lipschitz function with constant $L > 0$, that is

$$\|\nabla f(\mathbf{x}) - \nabla f(\mathbf{y})\| \leq L \|\mathbf{x} - \mathbf{y}\| \ \ \forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^n.$$

Then, let us take $p_k = -\nabla f(x_k)$ and $\alpha_k = \alpha = \frac{1}{L}$. It is known that these choices satisfy

$$f(x_k) - f(x^\star) \leq \frac{L \|x_0 - x^\star\|_2^2}{2k}.$$

This method is called the *gradient descent method*, because the search direction is the gradient. We see that it has a convergence rate of $\mathcal{O}\left(\frac{1}{k}\right)$, or again that it takes $\mathcal{O}\left(\frac{1}{\epsilon}\right)$ iterations to obtain $f(x_k) - f(x^\star) \leq \epsilon$. This convergence is really slow; it is said to be sublinear.

3.2 From Gradient Descent to Stochastic Gradient Descent

Why does the gradient descent method work so well? A gradient (at a point) is the slope of the tangent to the function (at that point). It points to the direction of the largest *increase* of the function. Hence, to minimize the function, it makes sense to follow the *opposite direction* of the gradient. This means that we choose $p_k = -\nabla f(x_k)$.

Let us define $N = |\Omega|$, i.e., the number of observed entries of the matrix $\mathbf{M}$. We know that N can be very large, hence the sublinear complexity rate is problematic. Therefore, we only compute a *fraction of the gradient*.

As often in machine learning, the cost function of interest is formulated as a sum over the training examples. Let us clarify that: one training example is picked, say $n \in [N]$, where N corresponds to the total number of training examples. For this data point, the true value of the output is known, but we can also predict this value by using our algorithm. This gives a predicted value. A cost function $f_n(x)$ is defined to measure the misfit between the predicted value, for any given value of x, and the true value. In other words, f_n is the cost contributed by the n-th training example. This is done for all the training examples. We obtain

$$f(x) = \frac{1}{N} \sum_{n=1}^{N} f_n(x). \tag{3.2}$$

Now, the idea of stochastic gradient descent, or SGD, can be defined: at each iteration k, one training example is randomly selected, say $n \in [N]$, the direction g_k is computed as

$$g_k = \nabla f_n(x_k),$$

and the search direction is defined as $p_k = -g_k$. The idea is the following: the quantity $\nabla f_n(x_k)$ is cheap to compute and acts as an estimate for the full gradient $\nabla f(x_k)$. Let us define the random variable $\mathcal{N}$ to denote the chosen training example, i.e., $\mathcal{N} \in [N]$.

As usual, we use n to denote a realization of the random variable $\mathcal{N}$. Since the training example n is chosen at random with uniform probability, we have

$$\mathbb{E}\left[g_k\right] = \sum_{n=1}^{N} \mathbb{P}\left[\mathcal{N} = n\right] \nabla f_n\left(x_k\right) = \frac{1}{N} \sum_{n=1}^{N} \nabla f_n\left(x_k\right) = \nabla f\left(x_k\right),$$

therefore, $\nabla f_{\mathcal{N}}\left(x_k\right)$ is said to be an *unbiased* estimator of $\nabla f\left(x_k\right)$. However, it is clear that $\nabla f_{\mathcal{N}}\left(x_k\right)$ is not always a descent direction! Indeed, the direction $\nabla f_{\mathcal{N}}\left(x_k\right)$ is the gradient with respect to only *one term in the full objective function*, therefore it can point in a direction that is far different from $\nabla f\left(x_k\right)$. Hence, we would like to find a better estimate. That is why the notion of *batch* is defined.

At each iteration k, instead of choosing a single training example, we can choose a couple of training examples. These training examples are called the batch. Formally, at each iteration k, a subset $\mathcal{B} \subseteq [N]$ of the training examples is randomly chosen. A new direction is obtained, that is written as

$$g_k = \frac{1}{|\mathcal{B}|} \sum_{n \in \mathcal{B}} \nabla f_n\left(x_k\right),$$

and then again the search direction is defined as $p_k = -g_k$.

We use b to denote the size of the batch, i.e., the batch-size, such that $b = |\mathcal{B}|$. In the above computation, a subset $\mathcal{B} \subseteq [N]$ of the training examples is randomly chosen. For each of these selected examples $n \in \mathcal{B}$, the respective gradients are computed at the same current point, that is, $\nabla f_n\left(x_k\right)$. This computation can be parallelized easily. This is how current deep-learning applications utilize Graphics Processing Units, or GPUs. We run over b threads in parallel. Let us assume that we have access to T machines. In short, we go through three steps.

1. Shuffle the data so as to have a representative subset of the full data-set on each machine.
2. Run SGD on each machine locally and compute f_t, the cost according to the subset assigned to machine t, for $t = 1, \ldots, T$.
3. Aggregate the result from all the machines, i.e., compute the final cost $f = \sum_{t=1}^{T} f_t$.

This enables the famous back-propagation algorithm for deep neural networks, which is used in a wide variety of fields including image and speech recognition. Recent papers like [ZE12, PJY$^+$13] go further into that.

Note that the extreme case $\mathcal{B} = [N]$ corresponds to the above described (full) gradient descent method. The other extreme is $b = 1$, which was described earlier. Most of Chapter 4 deals with $b = 1$. In general, if b is small, then g_k is a bad estimate of $\nabla f\left(x_k\right)$, and if b is large, then g_k is a good estimate of $\nabla f\left(x_k\right)$. Hence, the trade-off already appears. Therefore, we define a different batch at each iteration, which is denoted by $\mathcal{B}_k$. The notation b_k is used to denote the size of the batch at iteration k, i.e., $b_k = |\mathcal{B}_k|$.

At this point, an important remark has to be made regarding this trade-off. On the one hand, if b_k is large, then we only need a few iterations to converge, but each iteration is extremely costly. On the other hand, if b_k is small, then we need a lot of iterations to converge, but each iteration is cheap. Therefore, when the evolution of the error is displayed, the number of iterations cannot be taken as an indicator of the speed of convergence. Instead, the *running time* is used to compare the convergence speed of different algorithms. These aspects and the trade-off are discussed in Section 4.4.

3.3 Computing the gradients

As a reminder, for us, the objective function writes

$$f(\mathbf{X}) = \frac{1}{2} \left\| \mathcal{P}_\Omega(\mathbf{X}) - \mathcal{P}_\Omega(\mathbf{M}) \right\|_F^2 = \frac{1}{2} \sum_{(i,j)\in\Omega} [M_{ij} - X_{ij}]^2, \tag{3.3}$$

but since $\mathbf{X} \in \mathcal{M}_r$, we can write $\mathbf{X} = \mathbf{L}\mathbf{R}^\top$ as it is explained in Subsection 2.4.1. The objective function becomes

$$f(\mathbf{L}, \mathbf{R}) = \frac{1}{2} \sum_{(i,j)\in\Omega} \left[M_{ij} - \left(\mathbf{L}\mathbf{R}^\top \right)_{ij} \right]^2, \tag{3.4}$$

with $\left(\mathbf{L}\mathbf{R}^\top \right)_{ij} = \langle \mathbf{L}_{i:}, \mathbf{R}_{j:} \rangle = \sum_{k=1}^r L_{ik} R_{jk}$. We can define the cost functions corresponding to every individual data point, that is

$$f_{ij}(\mathbf{L}, \mathbf{R}) = \frac{1}{2} \left[M_{ij} - \left(\mathbf{L}\mathbf{R}^\top \right)_{ij} \right]^2,$$

for all $(i, j) \in \Omega$, such that we can write

$$f(\mathbf{L}, \mathbf{R}) = \sum_{(i,j)\in\Omega} f_{ij}(\mathbf{L}, \mathbf{R}).$$

In order to see the similarity with (3.2), the objective function (3.3) can be written as

$$f(\mathbf{X}) = \frac{1}{|\Omega|} \sum_{(i,j)\in\Omega} [M_{ij} - X_{ij}]^2. \tag{3.5}$$

However, it is clear that the objective functions (3.3) and (3.5) are completely equivalent, from an optimization point of view.

Now let us choose one entry of the matrix $\mathbf{M}$. This entry corresponds to a pair (i, j), hence to one term in the sum. For this fixed element (i, j) of the sum, we derive the gradient entry (i', k) for $\mathbf{L}$. Analogously, for the same fixed element (i, j) of the sum, we derive the gradient entry (j', k) for $\mathbf{R}$. We find

$$\frac{\partial f_{ij}}{\partial L_{i',k}}(\mathbf{L}, \mathbf{R}) = \begin{cases} -\left[M_{ij} - \left(\mathbf{L}\mathbf{R}^\top \right)_{ij} \right] R_{jk}, & \text{if } i' = i, \\ 0, & \text{otherwise,} \end{cases}$$

$$\frac{\partial f_{ij}}{\partial R_{k,j'}}(\mathbf{L}, \mathbf{R}) = \begin{cases} -\left[M_{ij} - \left(\mathbf{L}\mathbf{R}^\top \right)_{ij} \right] L_{ik}, & \text{if } j' = j, \\ 0, & \text{otherwise.} \end{cases}$$

Let $\mathbf{S}_{(i,j)}$ be the residual matrix of this subproblem, i.e., $\mathbf{S}_{(i,j)} = \left(\mathbf{L}\mathbf{R}^\top \right)_{ij} - M_{ij} = \mathbf{L}_{i:}\mathbf{R}_{j:}^\top - M_{ij}$. Of course, in this case, $\mathbf{S}_{(i,j)}$ is a scalar, not a matrix. Hence, we can write

$$\nabla_{\mathbf{L}} f_{ij}(\mathbf{L}, \mathbf{R}) = \begin{pmatrix} \mathbf{0}_{1\times r} \\ \vdots \\ \mathbf{0}_{1\times r} \\ \mathbf{S}_{(i,j)}\mathbf{R}_{j:} \\ \mathbf{0}_{1\times r} \\ \vdots \\ \mathbf{0}_{1\times r} \end{pmatrix} \leftarrow \text{row } i \tag{3.6}$$

with $\mathbf{R}_{j:} \in \mathbb{R}^{1\times r}$ which is a row vector, and $\nabla_{\mathbf{L}} f_{ij}(\mathbf{L},\mathbf{R})$ is therefore a matrix of size $d_1 \times r$, as it should be. We also find

$$\nabla_{\mathbf{R}} f_{ij}(\mathbf{L},\mathbf{R}) = \begin{pmatrix} \mathbf{0}_{1\times r} \\ \vdots \\ \mathbf{0}_{1\times r} \\ \mathbf{S}_{(i,j)}\mathbf{L}_{i:} \\ \mathbf{0}_{1\times r} \\ \vdots \\ \mathbf{0}_{1\times r} \end{pmatrix} \leftarrow \text{row } j \tag{3.7}$$

with $\mathbf{L}_{i:} \in \mathbb{R}^{1\times r}$ which is a row vector, and $\nabla_{\mathbf{R}} f_{ij}(\mathbf{L},\mathbf{R})$ is therefore a matrix of size $d_2 \times r$, as it should be. Then, the general SGD updates consist of picking one entry $(i,j) \in \Omega$, and computing

$$\begin{cases} \mathbf{L}_{k+1} = R_{\mathbf{L}_k}\left(-\alpha_k \nabla_{\mathbf{L}} f_{ij}(\mathbf{L}_k,\mathbf{R}_k)\right), \\ \mathbf{R}_{k+1} = R_{\mathbf{R}_k}\left(-\alpha_k \nabla_{\mathbf{R}} f_{ij}(\mathbf{L}_k,\mathbf{R}_k)\right). \end{cases} \tag{3.8}$$

where $R_{\mathbf{L}}$ and $R_{\mathbf{R}}$ are retractions (2.9) defined in Subsection 2.4.2. In order to ease the notations, we often write

$$\begin{cases} \mathbf{L}_+ = R_{\mathbf{L}}\left(-\alpha \nabla_{\mathbf{L}} f_{ij}(\mathbf{L},\mathbf{R})\right), \\ \mathbf{R}_+ = R_{\mathbf{R}}\left(-\alpha \nabla_{\mathbf{R}} f_{ij}(\mathbf{L},\mathbf{R})\right). \end{cases} \tag{3.9}$$

By using the definition of the retractions, this becomes

$$\begin{cases} \mathbf{L}_+ = \mathbf{L} - \alpha \nabla_{\mathbf{L}} f_{ij}(\mathbf{L},\mathbf{R}), \\ \mathbf{R}_+ = \mathbf{R} - \alpha \nabla_{\mathbf{R}} f_{ij}(\mathbf{L},\mathbf{R}). \end{cases} \tag{3.10}$$

The structure of $\nabla_{\mathbf{L}} f_{ij}(\mathbf{L},\mathbf{R})$ enables us to see that the matrix $\mathbf{L}_+$ is equal to the matrix $\mathbf{L}$, except for row i. Hence, the entire matrix $\mathbf{L}$ must not be updated, but only $\mathbf{L}_{i:}$, which represents row i of the matrix $\mathbf{L}$ at the previous iteration. The same can be said for $\mathbf{R}$. Therefore, the system (3.10) is rewritten as

$$\begin{cases} \mathbf{L}_{i:,+} = \mathbf{L}_{i:} - \alpha \mathbf{S}_{(i,j)} \mathbf{R}_{j:}, \\ \mathbf{R}_{j:,+} = \mathbf{R}_{j:} - \alpha \mathbf{S}_{(i,j)} \mathbf{L}_{i:}. \end{cases} \tag{3.11}$$

With this, the SGD updates could be completely defined if the batch-size was equal to one, i.e., $b = 1$. However, we will now consider a general stochastic gradient setup, as it is done in [MS16], where we pick b known entries at a time. Due to the cost function structure, as we have seen, we end up updating only a maximum of b rows of $\mathbf{L}$ and $\mathbf{R}$ at a time. Let $b_{\mathbf{L}}$ rows of $\mathbf{L}$ and $b_{\mathbf{R}}$ rows of $\mathbf{R}$ be updated when b known entries are picked, with $b_{\mathbf{L}} \leq b$ and $b_{\mathbf{R}} \leq b$. At this point, the reader might wonder why we need the inequalities $b_{\mathbf{L}} \leq b$ and $b_{\mathbf{R}} \leq b$. One should remember that different data points might correspond to the same row of $\mathbf{L}$ or $\mathbf{R}$. Indeed, to fix the ideas, let us take $b = 3$. Let us assume that the data points have indices (i, j_1), (i, j_2) and (i, j_3). These three different data points only update one row of $\mathbf{L}$, while they update three rows of $\mathbf{R}$. In this case, we thus have $b_{\mathbf{L}} = 1$ and $b_{\mathbf{R}} = 3$.

Let $\mathbf{L}_b$ be the corresponding submatrix of $\mathbf{L}$ with the $b_{\mathbf{L}}$ rows, i.e., its size is $b_{\mathbf{L}} \times r$. Similarly, let $\mathbf{R}_b$ be the corresponding submatrix of $\mathbf{R}$ with the $b_{\mathbf{R}}$ rows, i.e., its size is $b_{\mathbf{R}} \times r$.

An interpretation is that, each time we pick b known entries, we have a subproblem of completing a matrix $\mathbf{M}_b$ of size $b_{\mathbf{L}} \times b_{\mathbf{R}}$ with b known entries at indices Ω_b. We try to find

a matrix $\mathbf{X}_b = \mathbf{L}_b\mathbf{R}_b^\top$ such that $\mathbf{X}_b$ is as close to $\mathbf{M}_b$ as possible on the set Ω_b. Let $\mathbf{S}_b$ be the residual matrix of this subproblem, i.e., $\mathbf{S}_b = \mathcal{P}_{\Omega_b}\left(\mathbf{L}_b\mathbf{R}_b^\top - \mathbf{M}_b\right)$. The matrix $\mathbf{S}_b$ is of size $b_\mathbf{L} \times b_\mathbf{R}$, just like $\mathbf{M}_b$. Then, by looking at (3.6) and (3.7), the partial derivatives are

$$\begin{cases} \nabla_{\mathbf{L}_b} f(\mathbf{L}, \mathbf{R}) = \mathbf{S}_b\mathbf{R}_b, \\ \nabla_{\mathbf{R}_b} f(\mathbf{L}, \mathbf{R}) = \mathbf{S}_b^\top\mathbf{L}_b. \end{cases}$$

Hence, by looking at (3.11) and the partial derivatives, the classical SGD updates are

$$\begin{cases} \mathbf{L}_{b,+} = \mathbf{L}_b - \alpha\mathbf{S}_b\mathbf{R}_b, \\ \mathbf{R}_{b,+} = \mathbf{R}_b - \alpha\mathbf{S}_b^\top\mathbf{L}_b. \end{cases} \tag{3.12}$$

At this point, the careful reader will notice that the order matters. To fix the ideas, let us assume that we start by computing $\mathbf{L}_{b,+}$. After this first step, we hope that $\mathbf{L}_{b,+}$ is in some sense "better" than the previous $\mathbf{L}_b$. Then, the matrix $\mathbf{R}_{b,+}$ is computed, and we notice that $\mathbf{L}_b$ appears in this computation. Since the matrix $\mathbf{L}_{b,+}$ has already been computed, it would make sense to replace $\mathbf{L}_b$ by $\mathbf{L}_{b,+}$ in the computation of $\mathbf{R}_{b,+}$. This idea comes from the famous forward and backward substitution, in the Gauss–Seidel method, which is explained in [GV96, Section 3.1].

However, this technique cannot be used. Indeed, one must see the pair $(\mathbf{L}, \mathbf{R})$ as a decomposition of the *same variable*. Therefore, we must update the matrices together.

3.4 Scaling

Can we take into account more precise information? The idea is to define scaling factors $s_\mathbf{L}$ and $s_\mathbf{R}$ in order to rewrite (3.12) as

$$\begin{cases} \mathbf{L}_{b,+} = \mathbf{L}_b - \alpha\mathbf{S}_b\mathbf{R}_b s_\mathbf{L}, \\ \mathbf{R}_{b,+} = \mathbf{R}_b - \alpha\mathbf{S}_b^\top\mathbf{L}_b s_\mathbf{R}. \end{cases} \tag{3.13}$$

The scaling factors $s_\mathbf{L}$ and $s_\mathbf{R}$ are small matrices of size $r \times r$. These factors are first defined in order to take into account *global* information, then *local* information is taken into account as well.

3.4.1 Global curvature

Newton's optimization algorithm takes into account second-order information. It can be defined as follows

$$x_{k+1} = x_k - \alpha_k \left[\nabla^2 f(x_k)\right]^{-1} \nabla f(x_k).$$

In many cases, using the full Hessian information is computationally costly. As it is explained in [WN99], a popular compromise between convergence and numerical efficiency is to scale the gradient by the diagonal elements of the Hessian. Hence, Newton's method can be seen as a *scaled* gradient descent algorithm. This is also explained in [Nes03, Section 1.3]. The scaling factors $s_\mathbf{L}$ and $s_\mathbf{R}$ are called *preconditioners*, i.e., approximations of the inverse Hessian.

For a matrix $\mathbf{A}$, the vectorization operator $\mathrm{vec}\left(\mathbf{A}\right)$ stacks the columns of $\mathbf{A}$ on top of each other. This is formally reviewed in Appendix A.4. Hence, we can define $\mathrm{vec}\left(\mathbf{L}\right) \in \mathbb{R}^{d_1 r \times 1}$ and $\mathrm{vec}\left(\mathbf{R}\right) \in \mathbb{R}^{d_2 r \times 1}$, and write the objective function f as a function of these vectorizations. Let us define the variable

$$y = \begin{pmatrix} \mathrm{vec}\left(\mathbf{L}\right) \\ \mathrm{vec}\left(\mathbf{R}\right) \end{pmatrix} \in \mathbb{R}^{r(d_1+d_2)\times 1}.$$

The full Hessian matrix with respect to y is a square matrix of size $r\left(d_1+d_2\right) \times r\left(d_1+d_2\right)$. One can prove graphically that f, i.e., the objective function (3.4), is not jointly convex in $\mathrm{vec}\left(\mathbf{L}\right)$ and $\mathrm{vec}\left(\mathbf{R}\right)$, hence the Hessian is not necessarily positive definite. However, f is strictly convex in $\mathrm{vec}\left(\mathbf{L}\right)$ if $\mathrm{vec}\left(\mathbf{R}\right)$ is fixed, and *vice versa*, hence the diagonal elements of the Hessian are strictly positive. We compute

$$\nabla_y^2 f\left(y\right) = \begin{bmatrix} \nabla^2_{\mathrm{vec}(\mathbf{L})} f\left(\mathbf{L},\mathbf{R}\right) & \star \\ \star & \nabla^2_{\mathrm{vec}(\mathbf{R})} f\left(\mathbf{L},\mathbf{R}\right) \end{bmatrix} \in \mathbb{R}^{r(d_1+d_2)\times r(d_1+d_2)}, \tag{3.14}$$

where the stars $\star$ denote the off-diagonal blocks. Hence, we have

$$\mathrm{diag}\left\{\nabla_y^2 f\left(y\right)\right\} = \begin{bmatrix} \mathrm{diag}\left\{\nabla^2_{\mathrm{vec}(\mathbf{L})} f\left(\mathbf{L},\mathbf{R}\right)\right\} & \mathbf{0}_{d_1 r \times d_2 r} \\ \mathbf{0}_{d_2 r \times d_1 r} & \mathrm{diag}\left\{\nabla^2_{\mathrm{vec}(\mathbf{R})} f\left(\mathbf{L},\mathbf{R}\right)\right\} \end{bmatrix} \in \mathbb{R}^{r(d_1+d_2)\times r(d_1+d_2)}.$$

The Hessian matrices are computed as

$$\begin{cases} \nabla^2_{\mathrm{vec}(\mathbf{L})} f\left(\mathbf{L},\mathbf{R}\right) = \mathbf{R}^\top\mathbf{R} \otimes \mathbf{I}_{d_1\times d_1} \in \mathbb{R}^{rd_1\times rd_1}, \\ \nabla^2_{\mathrm{vec}(\mathbf{R})} f\left(\mathbf{L},\mathbf{R}\right) = \mathbf{L}^\top\mathbf{L} \otimes \mathbf{I}_{d_2\times d_2} \in \mathbb{R}^{rd_2\times rd_2}, \end{cases}$$

where $\otimes$ denotes the Kronecker product, which is also reviewed in Appendix A.4. The diagonalizations are

$$\begin{cases} \mathrm{diag}\left\{\nabla^2_{\mathrm{vec}(\mathbf{L})} f\left(\mathbf{L},\mathbf{R}\right)\right\} = \mathrm{diag}\left\{\mathbf{R}^\top\mathbf{R}\right\} \otimes \mathbf{I}_{d_1\times d_1}, \\ \mathrm{diag}\left\{\nabla^2_{\mathrm{vec}(\mathbf{R})} f\left(\mathbf{L},\mathbf{R}\right)\right\} = \mathrm{diag}\left\{\mathbf{L}^\top\mathbf{L}\right\} \otimes \mathbf{I}_{d_2\times d_2}, \end{cases} \tag{3.15}$$

where $\mathrm{diag}\left\{\mathbf{R}^\top\mathbf{R}\right\}$ is a diagonal matrix extracting the diagonal of $\mathbf{R}^\top\mathbf{R}$, and similarly for $\mathbf{L}^\top\mathbf{L}$.

Now, let us talk about metrics. The same notations as in Section 2.4 are used. The pair $(\mathbf{L},\mathbf{R})$ is denoted by $x \in \overline{\mathcal{M}}_r$. As a reminder, the total space is $\overline{\mathcal{M}}_r = \mathbb{R}_*^{d_1\times r} \times \mathbb{R}_*^{d_2\times r}$. The tangent space of the total space at $x = (\mathbf{L},\mathbf{R}) \in \overline{\mathcal{M}}_r$ has the expression

$$T_x\overline{\mathcal{M}}_r = \mathbb{R}^{d_1\times r} \times \mathbb{R}^{d_2\times r}. \tag{3.16}$$

The metric that is proposed in [MAS12] is defined as g_x and maps $\xi_x, \eta_x \in T_x\overline{\mathcal{M}}_r$ to a scalar value in $\mathbb{R}$. The tangent vector $\xi_x \in T_x\overline{\mathcal{M}}_r$ has $(\xi_\mathbf{L}, \xi_\mathbf{R})$ as matrix representation, i.e., $\xi_\mathbf{L} \in \mathbb{R}^{d_1\times r}$ and $\xi_\mathbf{R} \in \mathbb{R}^{d_2\times r}$, and similarly for $\eta_x \in T_x\overline{\mathcal{M}}_r$. We have

$$g_x\left(\xi_x,\eta_x\right) = \mathrm{vec}\left(\xi_\mathbf{L}\right)^\top \left(\mathbf{R}^\top\mathbf{R} \otimes \mathbf{I}_{d_1\times d_1}\right) \mathrm{vec}\left(\eta_\mathbf{L}\right) + \mathrm{vec}\left(\xi_\mathbf{R}\right)^\top \left(\mathbf{L}^\top\mathbf{L} \otimes \mathbf{I}_{d_2\times d_2}\right) \mathrm{vec}\left(\eta_\mathbf{R}\right),$$

and (A.4) enables us to rewrite this as

$$g_x\left(\xi_x,\eta_x\right) = \mathrm{Trace}\left\{\left(\mathbf{R}^\top\mathbf{R}\right)\xi_\mathbf{L}^\top\eta_\mathbf{L}\right\} + \mathrm{Trace}\left\{\left(\mathbf{L}^\top\mathbf{L}\right)\xi_\mathbf{R}^\top\eta_\mathbf{R}\right\}, \tag{3.17}$$

with $\mathrm{Trace}\left\{\cdot\right\}$ the matrix trace operator.

We end up with the scaling factors

$$\begin{cases} s_\mathbf{L} = \left[\mathbf{R}^\top\mathbf{R}\right]^{-1}, \\ s_\mathbf{R} = \left[\mathbf{L}^\top\mathbf{L}\right]^{-1}, \end{cases} \tag{3.18}$$

that are incorporated in (3.13).

3.4.2 Local and global curvature

Starting from (3.17), let us look at the first of the two traces. The square matrix $\mathbf{R}^\top\mathbf{R}$ can be seen as containing *global* second order information of the cost function, since it captures knowledge about the entire matrix $\mathbf{R}$. However, in a stochastic setup, we only modify $b_{\mathbf{R}}$ rows of $\mathbf{R}$, namely we only update the submatrix $\mathbf{R}_b$. In [MS16], this leads to the argument that more weight should be given to the submatrix $\mathbf{R}_b^\top\mathbf{R}_b$, since it contains *local* second order information of the cost function. Therefore, $\left(\mathbf{R}^\top\mathbf{R}\right)$ in (3.17) is replaced by $\left[\mu\left(\mathbf{R}^\top\mathbf{R}\right)+(1-\mu)\left(\mathbf{R}_b^\top\mathbf{R}_b\right)\right]$, for some non-negative scalar $\mu\in[0,1]$ that weights $\mathbf{R}^\top\mathbf{R}$ and $\mathbf{R}_b^\top\mathbf{R}_b$ differently. A normalization factor $\frac{b}{d_{\max}}$ has to be added to the first term in this sum, to ensure that the Frobenius norm of $\mathbf{R}^\top\mathbf{R}$ and $\mathbf{R}_b^\top\mathbf{R}_b$ are of the same order. Of course, we do the same thing for $\mathbf{L}$. Thus, the metric (3.17) becomes

$$\begin{aligned} g_x\left(\xi_x,\eta_x\right) = &\operatorname{Trace}\left\{\left[\frac{b\mu}{d_{\max}}\left(\mathbf{R}^\top\mathbf{R}\right)+(1-\mu)\left(\mathbf{R}_b^\top\mathbf{R}_b\right)\right]\xi_{\mathbf{L}}^\top\eta_{\mathbf{L}}\right\} \\ &+\operatorname{Trace}\left\{\left[\frac{b\mu}{d_{\max}}\left(\mathbf{L}^\top\mathbf{L}\right)+(1-\mu)\left(\mathbf{L}_b^\top\mathbf{L}_b\right)\right]\xi_{\mathbf{R}}^\top\eta_{\mathbf{R}}\right\}. \end{aligned} \tag{3.19}$$

The scaling factors are defined as

$$\begin{cases} s_{\mathbf{L}} = \left[\frac{b\mu}{d_{\max}}\left(\mathbf{R}^\top\mathbf{R}\right)+(1-\mu)\left(\mathbf{R}_b^\top\mathbf{R}_b\right)\right]^{-1}, \\ s_{\mathbf{R}} = \left[\frac{b\mu}{d_{\max}}\left(\mathbf{L}^\top\mathbf{L}\right)+(1-\mu)\left(\mathbf{L}_b^\top\mathbf{L}_b\right)\right]^{-1}, \end{cases} \tag{3.20}$$

that are incorporated in (3.13). Overall, these scaling factors act as efficient *preconditioners* for the standard SGD updates (3.12), as it was seen above.

Let us analyze the two extreme cases. If on the one hand $\mu=0$, then we only use local curvature information. If on the other hand $\mu=1$, then we only use global curvature information, and we are back at (3.18), except that we have added a normalization factor $\frac{b}{d_{\max}}$.

For any other $\mu\in(0,1)$, we take both the local and the global information into account. A short analysis of the effect of μ can be found in [MS16]. In problem instances where numerous entries are already known, i.e., $|\Omega|$ is large, the influence of μ is minimal. However, for ill-conditioned data, making use of local information is more critical, and a smaller value of μ is more appropriate, e.g., $\mu=0.5$. Since the goal of this master's thesis is not to make this analysis, we will always take $\mu=0.5$, unless stated otherwise.

3.4.3 Scale invariance

The issue of *scale invariance* refers to the behavior of algorithms which behave equivalently when initialized, say, either with $(\mathbf{L}_0,\mathbf{R}_0)$ or with $\left(\mathbf{L}_0\mathbf{T}^{-1},\mathbf{R}_0\mathbf{T}^\top\right)$ for all non-singular matrices $\mathbf{T}\in\mathrm{GL}(r)$. The scaling that has been described resolves the issue of scale invariance that exists in matrix factorization models. Indeed, as it was stated in Subsection 2.4.1, the non-uniqueness of matrix factorization implies that $\mathbf{X}$ remains unchanged under the action (2.5), for all non-singular matrices. It is straightforward to show that the scaled SGD updates (3.13) with (3.20) are scale invariant, whereas the standard SGD updates (3.12) are not: let us use $\mathbf{L}_{b,+}$ and $\tilde{\mathbf{L}}_{b,+}$ to denote the update if the algorithm is initialized with $(\mathbf{L}_0,\mathbf{R}_0)$ and $\left(\mathbf{L}_0\mathbf{T}^{-1},\mathbf{R}_0\mathbf{T}^\top\right)$ respectively. The same notation is used for $\mathbf{R}$. With the scaled SGD, we find

$$\begin{cases} \tilde{\mathbf{L}}_{b,+} = \mathbf{L}_{b,+}\mathbf{T}^{-1}, \\ \tilde{\mathbf{R}}_{b,+} = \mathbf{R}_{b,+}\mathbf{T}^\top, \end{cases}$$

which enables us to prove scale invariance. For the standard SGD, such a result cannot be obtained.

3.5 Step-size

In the machine learning community, the name *learning rate* is often preferred over *step-size*. The condition $\alpha_k \geq 0$ should be verified for all iterations k. Indeed, since the search direction is usually a descent direction, we do not want to reverse the direction. The challenges in finding a good α_k are both in avoiding that the learning rate is too long, or too short. Therefore, a backtracking line search is defined as in Algorithm 1. This is heavily used in Riemannian optimization.

Algorithm 1 Backtracking Line Search

procedure BACKTRACKING LINE SEARCH($\alpha_0 > 0, \tau \in (0,1)$)
  $l \leftarrow 0$
  while $f(x_k + \alpha_l p_k) > f(x_k)$ **do**
    $\alpha_{l+1} \leftarrow \tau \alpha_l$
    $l \leftarrow l + 1$
  end while
  return α_l
end procedure

However, since the idea of SGD is to have very cheap iterations, it would not make sense to compute a cheap search direction, and then look for an optimal step-size. In other words, it is useless to find the perfect distance if the direction is only a rough estimate. Therefore, five cheap ways to choose the step-size are discussed.

1. The easiest choice is to take a constant step-size: $\alpha_k = \alpha$ for all k.

2. We can also define a geometric sequence: $\alpha_k = \rho^k \alpha_0$ for some initial step-size α_0 and common ratio ρ. If $\rho < 1$ then the step-sizes are decreasing; if $\rho > 1$ then the step-sizes are increasing.

3. In stochastic approximation, asking that the step-sizes are *square summable but not summable* is a common rule. This rule states that

$$\sum_{k=1}^{\infty} \alpha_k = \infty, \quad \sum_{k=1}^{\infty} \alpha_k^2 < \infty.$$

   The first one is the so-called *infinite travel condition*, while the second one ensures that the step-sizes do not "explode". The intuition is that the step-sizes have to go to zero, but not too fast. One typical example is

$$\alpha_k = \frac{c_1}{c_2 + k},$$

   where $c_1 > 0$ and $c_2 \geq 0$. In particular, the simplest choice is $c_1 = 1$ and $c_2 = 0$. This is a very simple rule, with a count variable, that ensures that the step-size decreases at every iteration.

4. The *bold driver heuristic* is a famous heuristic in the case of SGD. First, we remind that an epoch is a sweep through the entire training set. The heuristic needs to monitor the error after each epoch. We have two possibilities.

(a) The error decreases, then we can be confident that the direction is good, and the algorithm can move further more aggressively. The step-size is increased: $\alpha_{k+1} = \rho\alpha_k$, with $\rho = 1.1$, unless stated otherwise.

(b) The error increases, then we want the algorithm to move more cautiously. The step-size is decreased: $\alpha_{k+1} = \sigma\alpha_k$, with $\sigma = 0.5$, unless stated otherwise.

The name of this heuristic comes from the analogy with the learning process of a young and inexperienced car driver. Of course, different values of ρ and σ are possible but performance does not depend critically on this choice.

5. One last common schedule is the *exponential decay protocol*. It has the mathematical form

$$\alpha_k = \alpha_0 \cdot \exp\{-ck\}$$

where α_0 and c are hyperparameters and k is the iteration number.

Finally, one last method is called *Adagrad*. It is not included in the list, because it is an expensive method. It comes from [DHS11] and it adapts the learning rate to the parameters. Intuitively, the learning rate is small for parameters associated with frequently occurring features, and it is large for parameters associated with infrequent features. For this reason, it is well-suited for dealing with sparse data. This has been used in a very recent field, namely image recognition. Indeed, the authors of [DC12] have found that Adagrad greatly improved the robustness of SGD. Therefore, it was used for training large-scale neural networks at Google. Among other things, the goal of this particular neural network was to recognize cats in YouTube videos.

Chapter 4

Improving Stochastic Gradient Descent methods

This chapter looks for ways to improve the SGD method: it sums up our contribution in this thesis. When working with an incomplete data set, the first thing that one has to do is guessing the best value for the rank. Indeed, up to now we have often considered synthetic data, hence the true rank was known. In real applications, however, the true rank is not known, or it does not even exist, hence Section 4.1 gives some intuition on how to find the best rank.

Section 4.2 answers the main question of this thesis, namely: is it useful to apply the shuffling of [MHA18] to the LRMC problem? The concept of importance sampling and weighted data points are discussed in Section 4.3. The first two sections only consider epochs, which are not always possible to consider in large problems, hence Section 4.4 looks at different SGD methods with smaller samples. Finally, Section 4.5 summarizes the methods that we suggest.

4.1 Determining the rank

At this point, it is clear that the rank r is a *hyperparameter*, i.e., a parameter that the user is able to choose. Hence, the question arises: *how do we choose this hyperparameter?* Often, a value is chosen by convention, like $r = 5$ for small problems, and $r = 10$ for large problems. In this section, we look for a more custom way of finding the best value of r. This is called the *model selection problem*. In particular, two methods are discussed. The first one is called the *elbow method* and is often used to determine the number of clusters in a data set, i.e., the problem of K-means clustering. This is a similar problem to identifying the number of latent features in a data set. The second one is *cross-validation*, which is very famous in the machine learning community.

4.1.1 The elbow method

Here is a summary inspired from [JU18]: the data is split into one training set S_{training} and one testing set S_{testing}. Let us assume that the sizes are equal. In that case, it means that the first half of the observed values of $\mathbf{M}$ is taken to train the algorithm, and the other half to test it. These two sets can be thought of as having been generated independently and sampled according to an underlying but unknown distribution. On the one hand, the training set is used to *construct*

a prediction function; that is, the learning algorithm runs with the training set. On the other hand, the testing set is used to *compute the validation error*, also called the *testing error*, that is, the error that a particular prediction function makes on unseen data.

We have in addition a set of values for the parameter of the model, i.e., the rank r, that the user can choose. Let us denote these values by r_d, for $d = 1, \ldots, D$. To keep things simple we assume that D is some finite value.

The learning algorithm is run D times on the same training set S_{training} to compute the D prediction functions f_{r_d} for $d = 1, \ldots, D$. For each such prediction function, the validation error is computed with the testing set. We then choose that value of the parameter r which gives the smallest validation error.

Models can be too limited, or they can be too rich. In the first case, we cannot find a function that is a good fit for the data in our model. We *underfit*. In the second case, we have such a rich model family that the prediction function fits not only the underlying function but also the noise in the data. We then talk about an *overfit*. Both of these phenomena are undesirable.

The elbow method works as follows. The training error always decreases as the rank r increases. Hence, we have to find a way to determine the "optimal rank", in a way that the model is rich enough to represent the underlying interactions, but not too rich in order to avoid fitting the noise. This being said, one should know that *there is no perfect rank*, because the true data matrix $\mathbf{M}$ is almost never sampled with a specific rank in mind. However, to fix the ideas, let us use $r^\star$ to denote the optimal rank. We are trying to find the perfect value for r to avoid both underfitting and overfitting. Therefore, we try to identify a *kink* in the curve representing the *training* error as a function of the rank. A kink can be understood as a break, or a fracture in the curve. To fix the ideas, we would identify a line with negative slope $-a$ for $r \in [0, r^\star]$, and a line with negative slope $-b$ for $r \in [r^\star, \infty[$, for $a, b > 0$. Ideally, the first line is very steep, and the second one is flatter, i.e., $a > b$, such that $r^\star$ can easily be identified.

In other words, let us look at the curve representing the training error as a function of the rank: if this line looks like an arm, then the "elbow" on the arm is the best value of r. The elbow method states that the kink happens at the rank $r^\star$.

On the one hand, for all the values of $r < r^\star$, the training error decreases rapidly, since increasing the rank makes the model richer and more able to represent the underlying interactions. In other words, underfitting happens as long as $r < r^\star$, since the model is not complex enough to represent all the interactions in the true matrix, hence both the training error and the testing error are high.

On the other hand, for all the values of $r > r^\star$, the training error decreases more slowly, since increasing the rank only allows the model to fit the underlying noise, which does not decrease the error drastically. In other words, overfitting happens when $r > r^\star$, because the model is too complex, hence it can represent both the interactions in the true matrix, but also the underlying noise. This means that the training error continues to decrease, while the testing error increases, since the model moves too far from the mean of the true data, and it will fail to predict well on unseen data.

Putting everything together, the elbow method is a graphical way of determining the value of $r^\star$. Let us look at Figure 4.1. We are working with the first part of the Jester data set, available at [GRGP01]. It consists of ratings, ranging between -10 and 10, of 100 jokes by 24983 users. Hence, the matrix $\mathbf{M}$ is very thin, with $d_1 = 24983$ and $d_2 = 100$. In order to reduce the size of the problem, 2000 users are randomly selected, and all the other users are neglected. Two ratings

per user are randomly extracted as test data. Using the notations defined above, we have $D = 10$ values for the rank, from 1 to 10. For each rank, the train and the test error are computed using the scaled SGD method described in (3.13), with "Random shuffling", a method described in the upcoming Subsection 4.2.1. The train and test error, or mean squared error (MSE), are computed as the sum of the squared differences between the true values and the predicted values, over the total number of values. The MSE is formally defined in the upcoming equation (4.6). For the initialization of the algorithm, full rank matrices $\mathbf{L}_0$ and $\mathbf{R}_0$ are randomly generated. Regarding the step-size, the bold driver protocol described in Section 3.5 is chosen, with an initial value of 0.01. The number of iterations was limited to 100, and the gradient tolerance for the MSE was set to 10^{-8}. In practice, most of the runs did not reach the gradient tolerance level, because they were interrupted after 100 iterations. This entire process is repeated five times, and we average the results. According to the elbow method, the optimal rank is $r^\star = 2$.

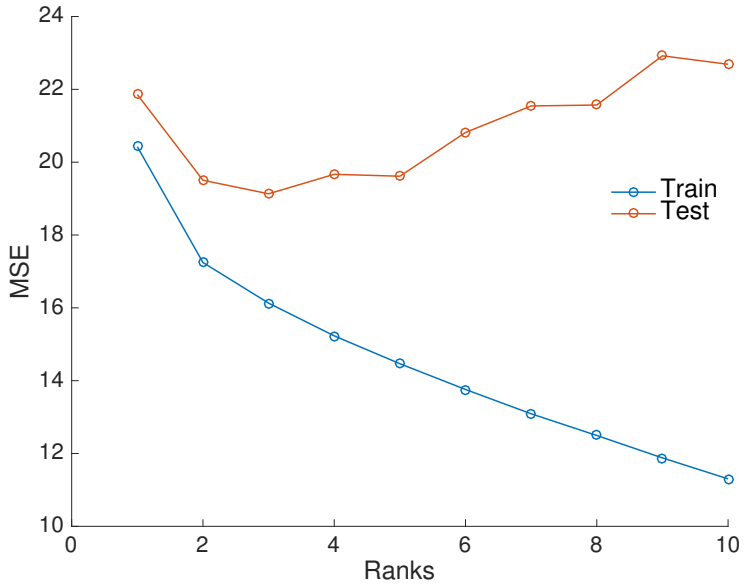


Figure 4.1: Average train and test MSE for different values of the rank r, which is fixed *a priori*, for the Jester data set described in [GRGP01]. In particular, the sampled matrix is rectangular of size 2000×100, and two ratings per user are randomly extracted as test data. The MSE is defined as $\|\mathcal{P}_\Omega(\mathbf{M}) - \mathcal{P}_\Omega(\mathbf{X})\|_F^2 / |\Omega|$. The scaled SGD method described by the equations (3.13) is used. We see that the elbow method is not always easy to apply, because the notion of "true rank" is not always applicable.

The training error is under the testing error, which makes a lot of sense since the training error is always too optimistic. The interpretation is that the training error is not a good representation of the accuracy of the algorithm, since it uses the same data to construct the prediction function and to evaluate the accuracy.

In other applications, we might have a case where the "best" rank is $r = 1$. In such cases, this graphical method does not work well. Indeed, we cannot notice a kink in that case. Therefore, another method, described in [TWH00] is used. However, since it is unlikely that the best rank is $r = 1$ in our situations, it will not be used.

4.1.2 Cross-validation

Splitting the data once into two parts (one for training and one for testing) is not the most efficient way to use the available information. To solve this problem, cross-validation, often abbreviated by CV, is a popular way of making better use of the data. Since there are many versions to date, let us just mention one. For a full review of cross-validation, we refer the reader to [Bis11, Section 1.3]. A popular scheme is called K-fold CV. The data is randomly partitioned into K groups, then we train K times. Each time, exactly one of the K groups is left out for testing and the remaining $K-1$ groups are used for training. Then, the K results are averaged. Note that in this way all the data has been utilized for training and for testing, and each data point has been used the same number of times. One major drawback of K-fold CV is that the number of training runs that must be performed is increased by a factor K, and this can prove problematic for models in which the training is itself computationally expensive.

By trying K-fold CV with $K = 5$ in order to determine the best rank for the same data set, i.e., the Jester data set, we find $r^\star = 5$. This corresponds to the value suggested in [KMO09]. The value for $r^\star$ was set to 2 with the elbow method described above. Hence, we see that finding the best rank can be difficult since such a concept is tedious to define. For a remark on the implementation, we refer the reader to Appendix C.1.

4.2 Shuffling methods

This part is inspired from [MHA18]. The authors have a problem where the objective function is a large sum, as (3.2), and they show that with a particular shuffling, the convergence can be sped up. In particular, the goal is to efficiently average a set of N symmetric positive definite matrices or SPD matrices. The least squares mean of the N-tuple $\mathbf{A} = (\mathbf{A}_1, \ldots, \mathbf{A}_N) \in \mathcal{M}^N$, with corresponding positive weights $\omega = (\omega_1, \ldots, \omega_N)$, is defined as

$$\Lambda(\omega, \mathbf{A}) = \operatorname*{argmin}_{\mathbf{X} \in \mathcal{M}} \sum_{n=1}^{N} \omega_n d^2(\mathbf{X}, \mathbf{A}_n),$$

where $d(\mathbf{G}, \mathbf{H})$ is the affine-invariant distance defined as

$$d(\mathbf{G}, \mathbf{H}) = \left\| \log\left(\mathbf{G}^{-\frac{1}{2}} \mathbf{H} \mathbf{G}^{-\frac{1}{2}}\right) \right\|_F,$$

for $\mathbf{G}, \mathbf{H} \in \mathbb{P}_n$, the cone of $n \times n$ SPD matrices. The matrix exponential and the logarithm of a matrix are defined in Appendix A.5.

Inductive sequences are used to estimate the least squares mean of a set of data points belonging to an arbitrary NPC space $(\mathcal{M}, d)$. As a reminder, a NPC space is a complete metric space with non-positive curvature. These sequences are built by starting from one data point and making successive steps towards other data points. A formal definition for the case of the least squares mean can be found at [MHA18, Definition 2.1].

The authors define a sequence $p = (p_1, \ldots, p_N)$, and two operations that act on that sequence: the *reverse* operation and the *Faro shuffle*. It is easier to think about these two definitions as operations on a deck of cards.

Definition 4.1 (Reverse)**.** *Given a sequence $p = (p_1, \ldots, p_N)$, the* reverse *operation is defined as*

$$\mathbf{reverse}(p) = (p_N, \ldots, p_1). \tag{4.1}$$

Intuitively, we simply swap the order of the deck: we put it upside-down. The last card becomes the first one, the first card becomes the last one, and so on. This operation does not influence the order of one card with respect to its neighbors, which is why the second definition is introduced.

Definition 4.2 (Faro shuffle)**.** *Given a sequence $p = (p_1, \dots, p_N)$ with $N = 2M$, it can be rewritten as $p = (p_1, \dots, p_M, p_{M+1}, \dots, p_{2M})$. Then, the* Faro shuffle*, which is also called the* riffle shuffle*, has two variants: the* in-shuffle *and the* out-shuffle*, defined as*

$$\textbf{in-shuffle}\,(p) = (p_{M+1}, p_1, p_{M+2}, p_2, \dots, p_{2M}, p_M)\,, \tag{4.2}$$

$$\textbf{out-shuffle}\,(p) = (p_1, p_{M+1}, p_2, p_{M+2}, \dots, p_M, p_{2M})\,. \tag{4.3}$$

Let us give a way to interpret this operation. The original sequence can be thought of as a deck of $N = 2M$ cards, which is divided into two halves. The top half of the deck is placed in the left hand, while the bottom half of the deck is placed in the right hand. Then, there are two ways to interleave the cards: for an *in-shuffle*, cards are alternatively interleaved from the left and right hands; for an *out-shuffle*, cards are alternatively interleaved from the right and left hands. Thus, a deck originally arranged as (1 2 3 4 5 6 7 8) would become (5 1 6 2 7 3 8 4) using an in-shuffle, and it would become (1 5 2 6 3 7 4 8) using an out-shuffle. It is easy to understand riffle shuffles with their applications in card tricks but it turns out that they are also used in the theory of parallel processing, as it is explained in [Sto71].

At this point, the careful reader will have noticed that we only consider the case when N is even. When N is odd, one could alternate between an in-shuffle and an out-shuffle, but in practice this will not be necessary. Indeed, for us, the number of data points N corresponds to $N = |\Omega|$, which is usually a really large number. Hence, we have $N \approx N - 1$, and one data point can simply be ignored. Therefore, we are back to the case when N is even.

4.2.1 Different ways to browse the data points

How can we apply this "shuffling" idea to our matrix completion algorithm? At each iteration, we want to go through all the data points. Hence, such an iteration is called an *epoch*. The order in which the data points are scanned through might seem unimportant, but the idea presented in [MHA18, Algorithm 1] gives us a good guess about which order the method should follow. For the reader's convenience, this algorithm is repeated in the following lines, which is called Algorithm 2 in the present document.

Algorithm 2 Smart shuffling algorithm

Data: N, the number of data points.
$p^{(1)} = (p_1, \dots, p_N)$
for $k = 1, 2, \dots$ **do**
    $p^{(2k)} = \textbf{reverse}\left(p^{(2k-1)}\right)$
    $p^{(2k+1)} = \textbf{in-shuffle}\left(p^{(2k-1)}\right)$
end for

All the methods that are described in this section work in the same way, except for *one* detail. For every method, epoch k starts by defining a sequence $p^{(k)}$. Then, the data points are browsed in the order specified by $p^{(k)}$. By browsing, we mean that we perform a scaled SGD step with a batch containing one element: $p_1^{(k)}$, then $p_2^{(k)}$, and so on, until $p_N^{(k)}$. We always have $\left|p^{(k)}\right| = N$.

For example, if $N = 3$, we can have $p^{(k)} = (3, 1, 2)$. In that case, a scaled SGD step is performed with sample $n = 3$, then with sample $n = 1$ and finally with sample $n = 2$.

Of course, the small detail that differentiates the four methods described next is in the way the sequence $p^{(k)}$ is defined.

1. "No shuffling". The sequence $p^{(k)}$ is defined as $p^{(k)} = (1, \ldots, N)$ for every k. This is also called the "Cyclic order" or the "Circular order". This method is deterministic.
2. "Random shuffling without replacement". The sequence $p^{(k)}$ is defined as a random permutation without repetition of the N data points. All the data points are visited. We can use "repl." to shorten "replacement". We also assume that "Random shuffling" denotes a shuffling without replacement in the sequel. This method is stochastic.
3. "Random shuffling with replacement". The sequence $p^{(k)}$ is defined as a random permutation with repetition of the N data points. This means that at epoch k, some data points are visited more than once, while others might not be visited. This method is stochastic.
4. "Smart shuffling". The sequence $p^{(k)}$ is defined by Algorithm 2. This method is deterministic.

4.2.2 Numerical experiments

In the upcoming experiments, synthetic data are created in the following way: two full-rank matrices $\mathbf{A} \in \mathbb{R}_*^{d_1 \times r}$ and $\mathbf{B} \in \mathbb{R}_*^{d_2 \times r}$ are generated according to a standard Gaussian distribution. Of course, all the elements of these two matrices are independently and identically distributed. The product $\mathbf{M} = \mathbf{A}\mathbf{B}^\top$ is a matrix of size $d_1 \times d_2$ and of rank r. Indeed, we have $r \ll \min\{d_1, d_2\}$, and we use Proposition A.1 stated in Appendix A.1. Then, a fraction of the entries of $\mathbf{M}$ are randomly removed with uniform probability. The number of known entries is determined by the OS ratio, since we have $|\Omega| = N = \text{OS} \times r \times (d_1 + d_2 - r)$.

Since the experiments are random, representative simulations were implemented. The same experiment is run five times, and then the results are averaged. The evolution of the train RMSE or test RMSE is analyzed as a function of the number of epochs. As a reminder, the two most famous metrics used to measure accuracy are the Mean Absolute Error (MAE) and the Root Mean Squared Error (RMSE). We only give the definitions for the training set, since it suffices to replace "Train" by "Test" in the following definitions in order to have them for the testing set.

1. The MAE is defined as
$$\text{MAE}_{\text{Train}} = \frac{\sum_{(i,j)\in\text{TrainingSet}} |M_{ij} - X_{ij}|}{|\text{TrainingSet}|}, \tag{4.4}$$
and the normalized version of the MAE, called the Normalized Mean Absolute Error or NMAE is often considered as well. It is defined as
$$\text{NMAE}_{\text{Train}} = \frac{\text{MAE}_{\text{Train}}}{\text{range}}, \tag{4.5}$$
where the range is the distance between the largest and the smallest value in the data set.
2. For the RMSE, we start by defining the Mean Squared Error or MSE as
$$\text{MSE}_{\text{Train}} = \frac{\sum_{(i,j)\in\text{TrainingSet}} (M_{ij} - X_{ij})^2}{|\text{TrainingSet}|}. \tag{4.6}$$

Then, the RMSE is simply defined as

$$\mathrm{RMSE}_{\mathrm{Train}} = \sqrt{\mathrm{MSE}_{\mathrm{Train}}}. \tag{4.7}$$

Here, since the computation times for the different methods are similar, the number of iterations, or epochs, can be used to truthfully represent the complexity of the algorithm. In Figure 4.2, we see that the four methods behave similarly, "Smart shuffling" behaving slightly worse, for a particular set of parameters. Varying the seed in MATLAB does not change the results, i.e., the experiment is rather stable. Specifically, a small square matrix of size 1000×1000 and of rank 10 is chosen. The OS ratio is set to 4. The MSE tolerance is set to 10^{-8}, and we are considering a maximum of 100 epochs. In practice, the algorithm always converges in less than 100 epochs. Some noise is introduced according to a parameter `noiseFac` $= 10^{-6}$ to a well-conditioned matrix. Starting the algorithm with randomly initialized matrices $\mathbf{L}_0$ and $\mathbf{R}_0$ does not yield good results. Hence, the initial matrices $\mathbf{L}_0$ and $\mathbf{R}_0$ are defined by computing the r dominant SVD, that is, the r largest singular values of the sparse matrix containing the known entries. In other words, we first compute the matrices $\mathbf{U}$, $\mathbf{\Sigma}$ and $\mathbf{V}$ from the sparse matrix containing the known entries. Then the initial matrices $\mathbf{L}_0 = \mathbf{U}\mathbf{\Sigma}^{\frac{1}{2}}$ and $\mathbf{R}_0 = \mathbf{V}\mathbf{\Sigma}^{\frac{1}{2}}$ are defined. Finally, regarding the step-size, the bold driver protocol described at item 4 on page 25 was chosen, with an initial value of 0.05. Since the test RMSE is very similar to the train RMSE, we only display the latter.

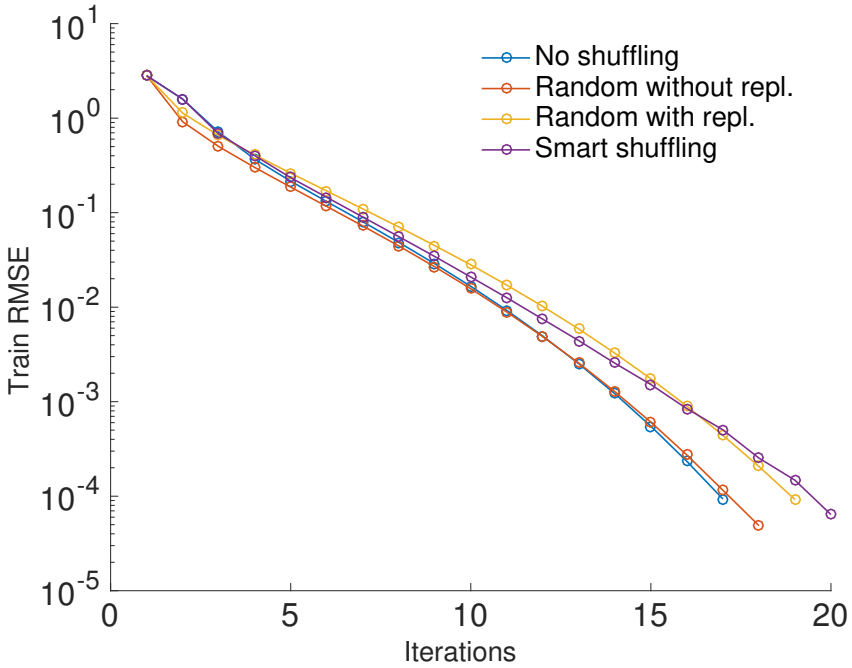


Figure 4.2: Evolution of the train RMSE for scaled SGD, using the four different shuffling methods described in Subsection 4.2.1. A synthetic matrix of size 1000×1000 is constructed, for which the rank is known to be $r = 10$. The OS ratio is set to 4. Initial matrices $\mathbf{L}_0$ and $\mathbf{R}_0$ are computed using r dominant SVD. The bold driver protocol described at item 4 on page 25 is used, with an initial value of 0.05. The algorithm is stopped when either the MSE (defined in (4.6)) is less than 10^{-8} or the number of epochs exceeds 100. The same experiment is run five times, and the results are averaged.

At this point, one should note that even if the "No shuffling" method described above (item 1 on page 32) is deterministic, it will not produce the same results for two identical experiments. Indeed, the true data matrix $\mathbf{M}$ is stochastic, hence the initial matrices $\mathbf{L}_0$ and $\mathbf{R}_0$ are stochastic as well; therefore the sequences of step-sizes defined by the bold driver protocol are different,

and so on. Of course, the same can be said about the "Smart shuffling" method described above (item 4 on page 32), since it also deterministic. This being said, the careful reader will have noticed that for a given experiment, after the first epoch, these two methods move in the same direction and at the same speed, hence they increase or decrease the cost *in the same way*. This stems from the fact that the sequence $p^{(1)}$ is the same for the two methods at the first epoch. Since the initial step-size is also the same for the two methods at the first epoch, the observation makes sense.

All simulations are performed on version R2015b of MATLAB, on a 2.0 GHz Intel Core i7 machine with 8 GB of RAM. For a remark on the implementation, we refer the reader to Appendix C.2.

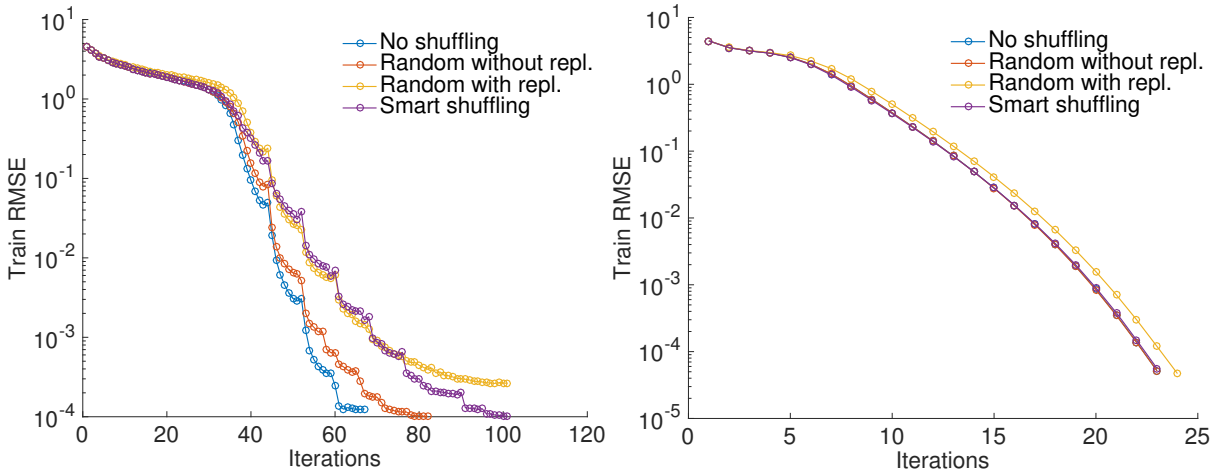
Surprisingly, in Figure 4.2, we observe that the "No shuffling" method performs slightly better than the "Random shuffling without replacement". Indeed, it is shown in [MHA18] that visiting the data points in a cyclic order usually results in a slow convergence. It can be that we need larger tests to observe this phenomenon. Equivalently, the fact that synthetic data is used can also be a problem. We observe that "Smart shuffling" is the worst of the four methods.

Hence, it seems that looking at the data points in a "smart" way does not really influence the speed of convergence compared to a cyclic order, nor a random order. Let us vary some parameters to see if this is always the case.

Step-size All the step-size methods described in Section 3.5 have been tested. We have tried the constant step-size $\alpha_k = \alpha$ for different values of α, it did not yield good results. The same can be said about the geometric step-size, and the exponential decay protocol. The counter method gave similar results than the ones obtained with the bold driver.

Over-sampling ratio In Figure 4.3, page 35, we can see the difference between a small and a large OS ratio on the convergence. Specifically, Figure 4.3a corresponds to OS = 2 and Figure 4.3b corresponds to OS = 8. All the other parameters are unchanged, *ceteris paribus*, which means that we are still working with a small matrix of size 1000×1000, of rank 10, and so on. The only exception to this is that the initial step-size was set to 0.01, otherwise the results were too noisy. As we had predicted, we observe that a large OS ratio enables a faster convergence, i.e., fewer epochs, since we have more data points. In fact, taking a small value for OS can even cause the algorithm to reach the limit of 100 epochs, before having attained convergence. However, the faster convergence guaranteed by a larger OS ratio comes at a cost: every epoch is more expensive when the OS ratio is larger, since more data points have to be browsed. Again, since the train RMSE behaves similarly to the test RMSE, we only display the former. On the one hand, when only a few entries are sampled, the "Smart shuffling" method performs worse than "Random shuffling without repl." or "No shuffling". Surprisingly, again, the cyclic order performs the best. Changing the seed in order to control the random number generation did not give very different results. On the other hand, when many entries are sampled, the "Random shuffling with replacement" method is the worst, as we had expected, and the three others behave similarly.

In or out? One might wonder if it matters to consider an *in-shuffle* or an *out-shuffle*. If N is large, then simulations show that there is no difference. However, if N is small, we can see that the in-shuffle performs better: this is due to the fact that with an out-shuffle, the extreme elements do not change. Indeed, the first element of the sequence $p^{(1)}$ either stays the first



(a) OS = 2 (b) OS = 8

Figure 4.3: Influence of the OS ratio on the evolution of the train RMSE for scaled SGD, using the four shuffling methods described in Subsection 4.2.1. A synthetic matrix of size 1000×1000 is used, for which the rank is known to be $r = 10$. Initial matrices $\mathbf{L}_0$ and $\mathbf{R}_0$ are computed using r dominant SVD. The bold driver protocol described at item 4 on page 25 is used, with an initial value of 0.01. The algorithm is stopped when either the MSE is less than 10^{-8} or the number of epochs exceeds 100. The same experiment is run five times, and the results are averaged.

element of $p^{(k)}$ or becomes the last element of $p^{(k)}$, for all following values of k. Similarly, for the last element of $p^{(1)}$. This is easy to see by looking at Definitions 4.1 and 4.2. Therefore, unless stated otherwise, we always consider an in-shuffle in the sequel.

Rectangular matrices We consider rectangular matrices of size 1000×100, and rank known to be 10. The OS ratio is set to 4, *ceteris paribus*. The results depicted in Figure 4.4, page 36, show that "Smart shuffling" performs worse than "Random shuffling" with or without replacement. However, the results are very noisy, i.e., changing the seed in MATLAB can yield very different curves for the train RMSE. Therefore, no conclusion can be taken, since more tests need to be done.

Other parameters Changing other parameters like the noise ratio, or considering ill-conditioned matrices did not yield interesting results: the four curves overlap most of the time. In short, it seems that the "Smart shuffling" does not give better results, neither for the training error nor for the testing error.

Real data set Here, we are again working with the first part of the Jester data set, taken from [GRGP01], as in Section 4.1. The four methods are tested for 2000 users. The results are depicted in Figure 4.5, page 36. As usual, the training error (4.5a) is lower than the testing error (4.5b), because the training error is overly optimistic. It is observed that "No shuffling" performs worse than the "Random shuffling". Furthermore, "Random shuffling" performs better without replacement than with replacement. Finally, "Smart shuffling" gives the lowest test RMSE. We can see that the initial step-size is obviously too large in these tests, since the cost starts to increase for most of the shuffling methods.

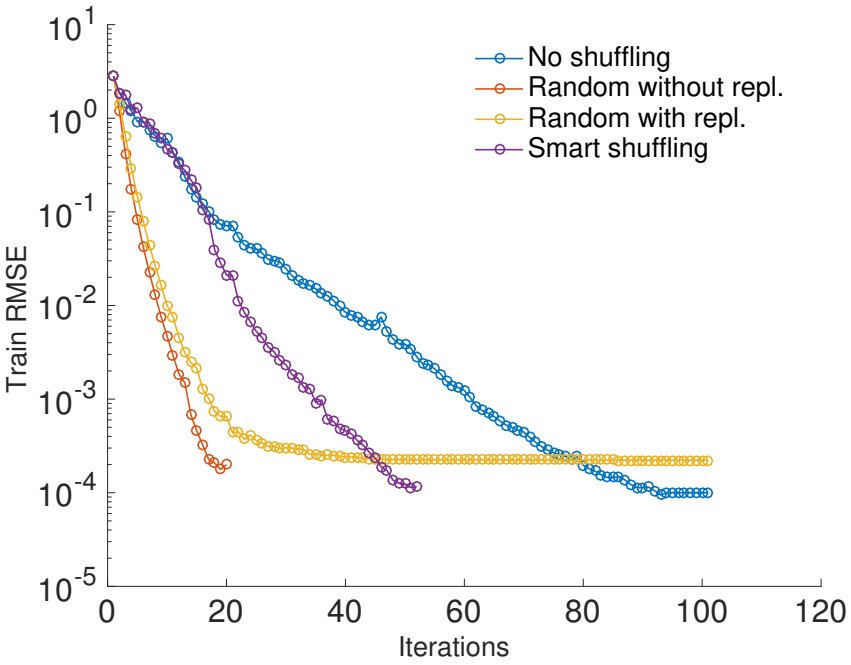
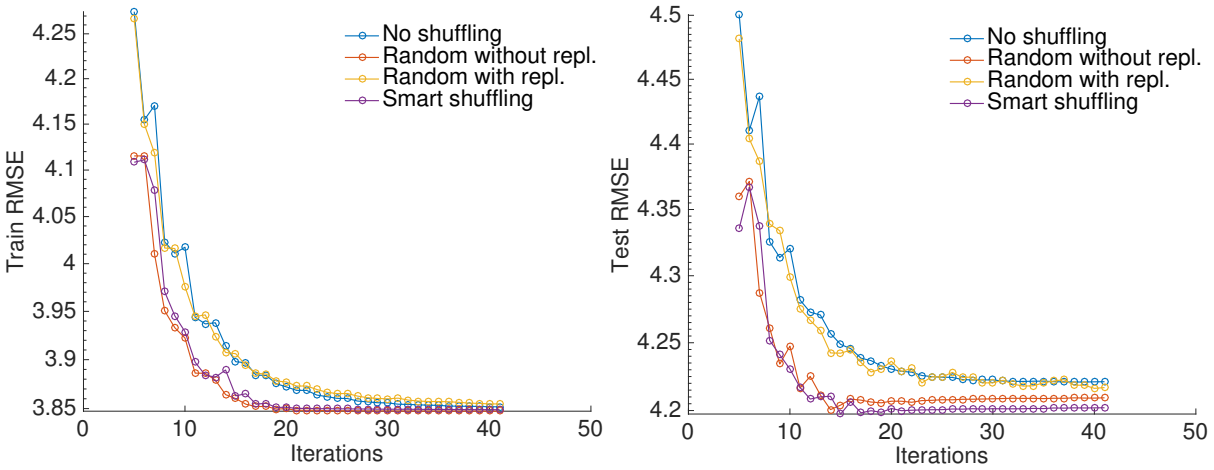


Figure 4.4: Evolution of the train RMSE for scaled SGD, using the four different shuffling methods described in Subsection 4.2.1. A synthetic rectangular matrix of size 1000×100 is used, for which the rank is known to be $r = 10$. The OS ratio is set to 5. Initial matrices $\mathbf{L}_0$ and $\mathbf{R}_0$ are randomly generated according to a standard Gaussian distribution. The bold driver protocol described at item 4 on page 25 is used, with an initial value of 0.1. The algorithm is stopped when either the MSE is less than 10^{-8} or the number of epochs exceeds 100. The same experiment is run five times, and the results are averaged.



(a) Train RMSE

(b) Test RMSE

Figure 4.5: Evolution of the train and test RMSE for scaled SGD, using the four different shuffling methods described in Subsection 4.2.1. The real-world Jester data set is used from [GRGP01], i.e., a matrix of size 2000×100. The rank is fixed to $r = 5$ according to Subsection 4.1.2. Initial matrices $\mathbf{L}_0$ and $\mathbf{R}_0$ are randomly generated according to a standard Gaussian distribution. The bold driver protocol described at item 4 on page 25 is used, with an initial value of 0.1. The algorithm is stopped when either the MSE (defined in (4.6)) is less than 10^{-8} or the number of epochs exceeds 40. Note that the first five epochs are not shown, otherwise the graph would have been less convenient to scale for the vertical axis.

Up to now, we have always considered that all the data points had to be visited at every epoch. This does not have to be the case. Therefore, another shuffling idea, called *importance sampling*, is described in the next section.

4.3 Importance sampling

This section is inspired from [NSW16, Section 3]. The idea is the following: up to now, we were looking at N data points at every epoch. This will not be changed. Instead, we can ask: do we have to browse *all* the data points? In the previous section, three of the four shuffling methods are considering all the data points at every epoch. The only analysis is about the order: is there a specific order to browse all the data points such that the convergence is better? However, "Random shuffling with replacement" does not specifically take all the data points into account: one data point can be taken more than once, while another might not be taken at all. The methods that are discussed in this section have a similar behavior, hence they can be called "Weighted shuffling" methods.

We want to show that some data points are more important than others, and that these data points should be visited more often. Hence, N data points are still considered at every epoch, but they are not necessarily N *different* data points. This can be seen as a shuffling where some data points are left behind, while others are duplicated to make sure that N data points are visited in total at every epoch. E.g., if $N = 3$, we could define the sequence $p^{(k)}$ as $(2, 1, 2)$.

We start by redefining the objective function of the original LRMC formulation as a least squares function. For the reader's convenience, the objective function (3.3) of page 20 is repeated here

$$f(\mathbf{X}) = \frac{1}{2} \left\| \mathcal{P}_\Omega(\mathbf{X}) - \mathcal{P}_\Omega(\mathbf{M}) \right\|_F^2 = \frac{1}{2} \sum_{(i,j)\in\Omega} [M_{ij} - X_{ij}]^2 .$$

Therefore, we have the following proposition.

Proposition 4.1. *The objective function* (3.3) *of page 20 can be written as*

$$f(\mathbf{x}) = \frac{1}{2} \left\| \mathbf{A}\mathbf{x} - \mathbf{b} \right\|_2^2 , \tag{4.8}$$

for some vectors $\mathbf{x}, \mathbf{b}$ *and a matrix* $\mathbf{A}$*.*

Proof. See Appendix B.2. □

The authors of [NSW16] consider SGD methods, where gradient estimates are sampled from a weighted distribution. As previously, the random variable $\mathcal{N}$ is one of the N training examples, i.e., $\mathcal{N} \in [N]$. We use n to denote a realization of $\mathcal{N}$. For a weight function w which assigns a non-negative weight $w(n) \geq 0$ to each index n, the weighted distribution $\mathcal{D}^{(w)}$ is defined as the distribution such that

$$\mathbb{P}_{\mathcal{D}^{(w)}}[I] \propto \mathbb{E}_{\mathcal{N}\sim\mathcal{D}} \left[\mathbf{1}_I(\mathcal{N}) \, w(\mathcal{N}) \right],$$

where I is an event (subset of indices) and $\mathbf{1}_I(\cdot)$ its indicator function. We have a discrete distribution $\mathcal{D}$ since the sample $\mathcal{N}$ is chosen from $[N]$. Hence, the probabilities of the original probability mass function $p(n)$ are weighted in order to obtain a new probability mass function, written as

$$p^{(w)}(n) \propto w(n) \, p(n) .$$

At this point it may be useful to mention that w can be seen as a weight function, but also as a weight vector of size $N \times 1$. Indeed, entry $w_n = w(n)$ simply denotes the weight assigned to data point n. Therefore, we use $\mathbf{w}$ to denote this vector, and we interchangeably write $w(n)$ and w_n to denote the weight of data point n.

The authors use rejection sampling to construct $\mathcal{D}^{(w)}$. The procedure is as follows: first, define a constant $W \geq \sup_n w(n)$. We choose $W = 1$. Then, sample $\mathcal{N} \sim \mathcal{D}$, say we select data point n, and accept that sample with probability $w(n)/W$. Otherwise, reject and continue to resample until a suggestion n is accepted. The accepted samples are then distributed according to $\mathcal{D}^{(w)}$.

How can we define the weight function w ? We propose two heuristics.

4.3.1 A really expensive method

This method could work with small data sets. We start by computing the cost f_0 corresponding to the initial matrices $\mathbf{L}_0$ and $\mathbf{R}_0$. If on the one hand, after the first epoch it seems that sample n does not decrease the cost function, then this sample is given a low weight $w(n)$. If on the other hand, it seems that sample n greatly decreases the cost function, then this sample is given a high weight $w(n)$. Concretely, for each sample $n \in [N]$, corresponding to item $(i, j) \in \Omega$, the rows $\mathbf{L}_{i:}$ and $\mathbf{R}_{j:}$ are updated, and the new cost f_n is computed. We use $\delta_n = f_n - f_0$ to denote the difference, and we hope that $\delta_n < 0$. If indeed $\delta_n < 0$, then it means that data point n allowed us to decrease the cost, and this data point is given a high weight, proportional to $|\delta_n|$. If however $\delta_n > 0$, then it means that data point n did not allow us to decrease the cost, and this data point is given a zero weight. Hence, a weight vector $\mathbf{w}$ is obtained. Of course, when the cost corresponding to the next sample $n + 1$ is computed, one should first undo the updates of the rows $\mathbf{L}_{i:}$ and $\mathbf{R}_{j:}$, otherwise the update of sample $n + 1$ will benefit from the updates of all the preceding samples. We can already mention that this algorithm is not feasible in practice, since evaluating the cost function for every data point is impossible in real-life applications. Furthermore, we would have to compute δ_n for every $n \in [N]$, which can be very expensive.

One could think that this should be done only at the first iteration, and then keep the weights as they are for all the following epochs. However, if the weights are optimal for the first epoch, it does not necessarily mean that they are optimal for the next ones. Furthermore, it is even hard to argue that the weights are optimal *for the first epoch at all*. Indeed, it may be that data point n is "good" when we start with the initial random matrices $\mathbf{L}_0$ and $\mathbf{R}_0$. However, we first visit some other data points before looking at n, and it might be the case that n is not good anymore at that time.

Since we suggest recomputing the weights after every iteration, it is clear that this method cannot work for the Netflix problem, for instance. For this reason, this expensive method is tried on a small synthetic data set: a small matrix of size 200×200 is randomly generated, with a known rank $r = 4$. Then, some entries are removed with uniform probability. As usual, the total number of known entries is specified by the OS ratio, set to 4. In this experiment, the weights are updated *after every epoch*: this means that convergence is really slow, as it can be seen in Figure 4.6. The algorithm is initialized with matrices $\mathbf{L}_0$ and $\mathbf{R}_0$ that are populated with random entries drawn from the standard Gaussian distribution. The initial step-size is set to 0.01 and the bold driver protocol described at item 4 on page 25 is used. As always, the regularization and scaling parameters are set to $\lambda = 0$ and $\mu = 0.5$ respectively. One experiment corresponds to the number of epochs needed to reach the tolerance level for the train MSE, namely 10^{-8}. Five such experiments are run, which gives five different outputs, and the results are averaged. Indeed,

the given outputs are different because each experiment depends on the random initializations, the random permutations, hence the sequences of step-sizes are not the same, and so on. For comparison, we take the "Random shuffling" method described in item 2 on page 32 above where all the data points are browsed in a random order. As expected, the weighted shuffling version is a lot slower, because each epoch starts by computing the weights, which is a very costly process. However, we can observe that fewer epochs are needed, which reinforces our intuition. Indeed, "Random shuffling" needs 17 epochs on average to converge, whereas "Weighted shuffling" only needs 14 epochs on average.

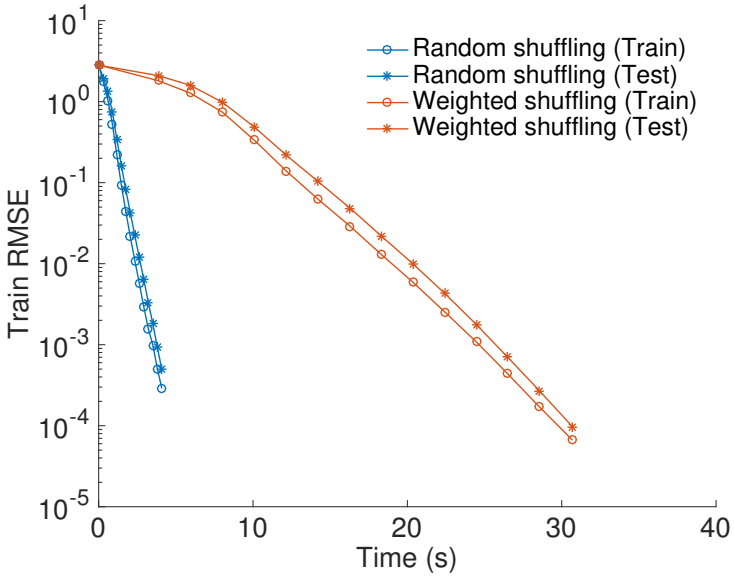


Figure 4.6: Evolution of the train and test RMSE for the "Random shuffling" and "Weighted shuffling" methods, described in Subsections 4.2.1 and 4.3.1 respectively. The synthetic data consists of a small matrix of size 200×200, which is known to be of rank $r = 4$. The OS ratio is set to 4. The bold driver protocol described at item 4 on page 25 is used, with an initial value of 0.01. The weights are updated after every epoch. The algorithm is stopped when either the MSE is less than 10^{-8} or the number of epochs exceeds 100. The experiment is repeated five times, and the results are averaged. The weighted method is a lot slower than the random method, but it needs fewer epochs: the epochs are more expensive but also more accurate.

Since this method is very expensive, we will not consider it, and we focus on the cheaper alternative described next.

4.3.2 A cheaper method

This method can work for larger data sets. As always, let us consider one data point, i.e., one pair $(i, j) \in \Omega$, and let us use $n \in [N]$ to denote this data point. At epoch k, we use $\mathbf{X}_k = \mathbf{L}_k \mathbf{R}_k^\top$ to denote the current iterate, and X_{ij}^k to denote the prediction for entry (i, j). We define a new variable $\delta_{ij}^k = \left| M_{ij} - X_{ij}^k \right|$ to measure the distance between the true value and the predicted value of data point n after epoch k. If δ_{ij}^k is small, then we guess that data point n has "done its job", and it will not enable us to decrease the cost function anymore. If however δ_{ij}^k is large, then the heuristic tells us that data point n "still has some job to do", and it will surely enable us to decrease the cost function. Hence, the N-dimensional vector δ^k is computed to see which data points are most likely to decrease the cost function.

Let us fix one iteration k. If one was trying to define the optimal weights, the following

problem would need to be solved

$$\begin{aligned} \mathbf{w}^\star(\mathbf{X}) = \underset{\mathbf{w}\in\mathbb{R}^N}{\operatorname{argmax}} \quad & \frac{1}{2}\sum_{(i,j)\in\Omega} w_{ij}\left(\delta_{ij}^k\right)^2 \\ \text{subject to} \quad & \sum_{(i,j)\in\Omega} w_{ij} = 1, \\ & w_{ij} \geq 0 \ \forall (i,j) \in \Omega. \end{aligned}$$

First, notice that the search space $\mathbb{R}^N$ can be simplified to $[0,1]^N$. It is well-known that the solution to this problem is very sparse. Indeed, let us define the pair $(i',j') \in \Omega$ such that $\delta_{i'j'}^k \geq \delta_{ij}^k \ \forall (i,j) \in \Omega$. In other words, the pair (i',j') denotes the data point such that the predicted value $X_{i'j'}$ and the true value $M_{i'j'}$ are the furthest apart. Then, the solution has the form

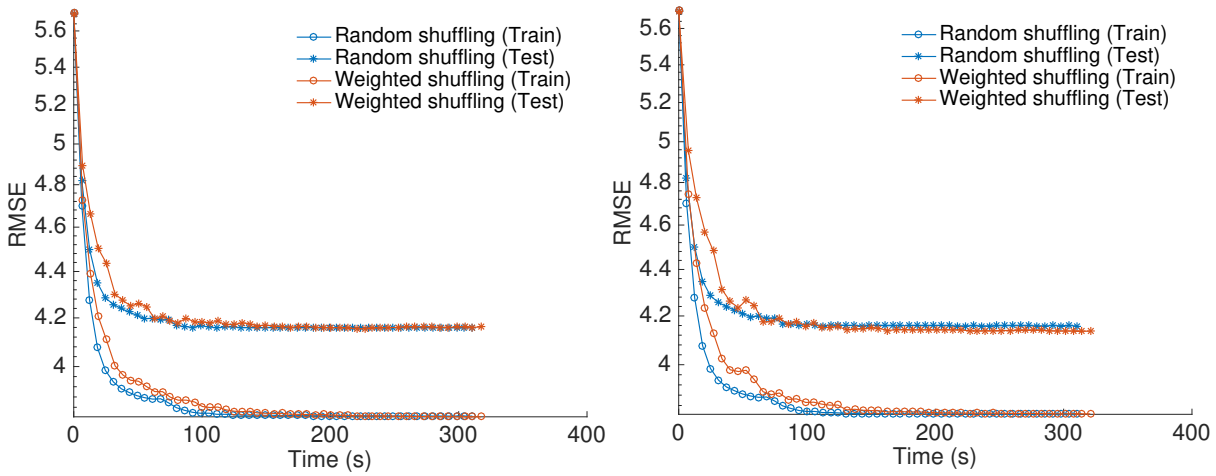
$$w_{ij}^\star(\mathbf{X}) = \begin{cases} 1, & \text{if } (i,j) = (i',j'), \\ 0, & \text{otherwise.} \end{cases}$$

If we effectively use this weight vector, we end up choosing only one data point, which is rather extreme. Hence, the vector δ is sorted in *decreasing* order, and the β first data points are selected in this list. These data points correspond to the ones that have the largest distance between the predicted value and the true value. Therefore, they correspond to the data points that "still have some job to do" according to the heuristic that we propose. A non-zero weight w_{ij} is given to these points. This weight can be attributed in several ways, let us simply choose two: either we set $w_{ij} = 1$, or we give a weight proportional to δ_{ij}, i.e., $w_{ij} \sim \delta_{ij}$. In the sequel, unless stated otherwise, we consider $w_{ij} = (\delta_{ij})^{1/2}$. Of course, after the computation of the weights, vector $\mathbf{w}$ is rescaled such that $\sum_{n=1}^N w_n = 1$. With this vector, epoch k starts by constructing a vector of size N with random data points, but knowing that data point n has a probability w_n of being chosen. Of course, the weights need not be updated after every epoch.

In Figure 4.7, the Jester data set is considered. We decide to update the weights after five iterations, and we use $\beta = \frac{3N}{4}$. This means that only 75% of the vector $\mathbf{w}$ has a non-zero value. As stated above, we suggest two ways of defining the weights: Figures 4.7a and 4.7b show the results when $w_{ij} = 1$ and $w_{ij} = (\delta_{ij})^{1/2}$, respectively. The results are similar: for the two ways of defining w_{ij}, the weighted shuffling method and the random shuffling method perform equally well on the test set and on the train set. If we take 200 users instead of 2000, then it appears that the weighted shuffling performs better than the random shuffling for the test set, and the inverse is true for the train set. However, we can say that with the Jester data set, the cheap weighted method does not improve the convergence rate: the curve of the weighted shuffling, i.e., the red curve, decreases more slowly at the beginning, but joins the curve of the random shuffling, i.e., the blue curve, after some epochs.

The next thing that we want to analyze is the size of β. In theory, the optimal choice would be $\beta = 1$, i.e., all the weights are equal to zero, except $w_{i'j'}$, where the pair (i',j') has been defined above. However, simulations have shown that this does not work in practice. Hence, we want to know the best value for β, i.e., the best number of data points that have to be given a non-zero weight. A large value of β means that the selection is rather lax, i.e., a lot of data points are assigned a non-zero weight, whereas a small value of β indicates a more selective process.

We know from Figure 4.7 that it is hard to achieve a RMSE lower than 3.5 for the training error on the Jester data set. Therefore, the tolerance for the MSE is set to $15 > 12.25 = 3.5^2$. The time needed for several methods to attain this tolerance level is recorded. The procedure is repeated five times, the results are averaged and depicted in Table 4.1. Of course, in order to



(a) $w_{ij} = 1$ (b) $w_{ij} = (\delta_{ij})^{1/2}$

Figure 4.7: Evolution of the train and test RMSE for the "Random shuffling" and "Weighted shuffling" methods, described in Subsections 4.2.1 and 4.3.2 respectively. The Jester data set comes from [GRGP01], and 2000 users are randomly selected. The rank is fixed to $r = 5$ according to Subsection 4.1.2. The bold driver protocol described at item 4 on page 25 is used, with an initial value of 0.01. The weights are updated after every five epochs, with $\beta = \frac{3N}{4}$. The algorithm is stopped when either the MSE is less than 10^{-8} or the number of epochs exceeds 50. The experiment is repeated five times, and the results are averaged. Marginally, the weighted shuffling attains a lower test RMSE than the random shuffling, but the inverse is true for the train RMSE. Therefore, it is fair to say that with larger matrices, the two curves coincide, both for $w_{ij} = 1$ and for $w_{ij} = (\delta_{ij})^{1/2}$.

compare with previously defined methods, the timing is also given for the "Random shuffling" method: 94.7267 seconds, with the specifications given in the caption of Table 4.1. These results are not surprising: indeed, we have seen in Figure 4.7 that "Random shuffling" is usually better than "Weighted shuffling" on the training set, but the latter is better on the testing set.

β	$w_{ij} = 1$	$w_{ij} \sim \delta_{ij}$
$N/4$	∞	∞
$N/2$	183.2642	352.8762
$3N/4$	**97.7030**	123.1840
N	102.1952	104.3330

Table 4.1: Timing in seconds to reach 15 for the train MSE, defined in (4.6), for scaled SGD with "Weighted shuffling" method described in Subsection 4.3.1. The Jester data set comes from [GRGP01], and 2000 users are randomly selected. The rank is fixed to $r = 5$ according to Subsection 4.1.2. The initial matrices $\mathbf{L}_0$ and $\mathbf{R}_0$ are populated with random entries drawn from the standard Gaussian distribution. The bold driver protocol described at item 4 on page 25 is used, with an initial value of 0.01. The weights are updated after every five epochs. We define $w_{ij} \sim \delta_{ij}$ as $w_{ij} = (\delta_{ij})^{1/2}$. The algorithm is stopped when either the MSE is less than 15 or the number of epochs exceeds 5000. The experiment is repeated five times, and the results are averaged. It appears that the fastest method corresponds to $w_{ij} = 1$ with $\beta = \frac{3N}{4}$. The "Random shuffling" described in Subsection 4.2.1 converges in 94.7267 seconds, which is slightly better than "Weighted shuffling". We see that when $\beta = N/4$, the algorithm never converges, i.e., the selection is too selective, hence we do not consider enough data points.

As a side note, the curious reader might see the equivalence of this weighted method with the Iteratively Reweighted Least Squares (IRLS) method. Indeed, a weight has been assigned to

each observation $(i,j) \in \Omega$. This weight depends on the matrix $\mathbf{X}$. We use $w_{ij}(\mathbf{X})$ to denote the weight of data point (i,j). Thus, the problem becomes

$$\min_{\mathbf{X} \in \mathcal{M}_r} \frac{1}{2} \sum_{(i,j) \in \Omega} w_{ij}(\mathbf{X}) (M_{ij} - X_{ij})^2,$$

and the algorithm creates a sequence of iterates $\left\{\mathbf{X}^{(k)}\right\}_{k=1}^{K}$ by solving

$$\mathbf{X}^{(k+1)} = \operatorname*{argmin}_{\mathbf{X} \in \mathcal{M}_r} \frac{1}{2} \sum_{(i,j) \in \Omega} w_{ij}\left(\mathbf{X}^{(k)}\right) (M_{ij} - X_{ij})^2.$$

Up to now, both in Section 4.2 and in this section, we have always considered iterations of N data points. This does not have to be the case. Therefore, a last shuffling idea, called *batch SGD*, is described in the next section.

4.4 Batch Stochastic Gradient Descent

Up to now, the batch $\mathcal{B}$ contained a single data point. In Section 4.2, four sequences of N data points are discussed, but only epochs were considered, i.e., passes through all the data points. At every epoch, one data point is visited at a time, in some specific order. Section 4.3 introduces two new methods where N data points are chosen with repetition among N data points, hence some data points are chosen more than once, while others are not chosen at all. We cannot really talk about an epoch in that case since *all* the data points are not visited, but the algorithm still browses N data points.

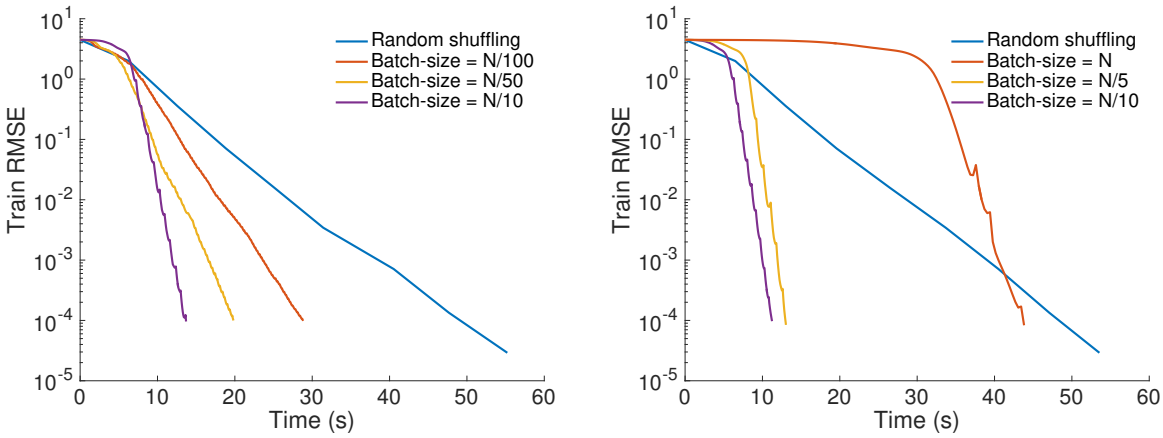
In this section, we do not consider epochs anymore. Instead, we consider iterations such that at iteration k, b_k data points are visited, with $b_k = |\mathcal{B}_k| \leq N$. It has been shown previously that, as the batch-size increases, the error in the sampled gradient decreases, and so the batch-size can be used to implicitly control the error in the gradient. Formally, using the notation of the introduction of Chapter 3, the direction g_k is defined as

$$g_k = \nabla f(x_k) + e_k,$$

where e_k is the error, and the search direction is defined as $p_k = -g_k$. It is clear that if $\mathcal{B}_k = [N]$, then $e_k = 0$. At this point, it is important to mention that the number of iterations cannot be used unequivocally anymore to show the evolution of the error, because one iteration for method A can be very different from one iteration for method B. Indeed, if method A takes $b_k = N$ and method B takes $b_k = 1$, then iteration k is cheaper for method B, but it is less precise.

If only b_k data points are considered at every iteration, the easiest idea is to fix the batch-size $b_k = b$ for every iteration k. Then, one might ask how to determine the best value for b. In order to determine this, a small experiment is run, from which the results are displayed on Figure 4.8. We work with a square matrix of size 1000×1000, and rank known to be $r = 10$. The OS ratio is set to 6. The "Constant batch-size" method is compared with "Random shuffling" described on item 2 in Section 4.2. In subfigure 4.8a it can be seen that when the value of b is increased, convergence is achieved faster. Hence, it would make sense to take the largest possible value for b, i.e., $b = N$, where N is the number of data points. However, subfigure 4.8b shows that when b increases too much, the convergence is slower. Hence, taking $b = \frac{N}{10}$ seems to be a sweet spot.

Of course, other methods can also be thought of, since $b_k = b$ for every k is not a very rich model. Another choice is to let b_k take a random integer value between 1 and N at every iteration. This gives rise to a method called "Random batch-size".



(a) Small values of b.

(b) Large values of b.

Figure 4.8: Evolution of the training error as a function of computation time. The "Constant batch-size" defined in Section 4.4 is compared to "Random shuffling" described on item 2 in Section 4.2, for the scaled SGD method. We consider a matrix of size 1000×1000, rank known to be 10 and OS ratio set to 6. Initial matrices $\mathbf{L}_0$ and $\mathbf{R}_0$ are randomly generated according to a standard Gaussian distribution. The bold driver protocol described at item 4 on page 25 is used, with an initial value of 0.1. It appears that a trade-off in the size of the batch has to be found.

Finally, a choice inspired from [FS12] is based on an increasing batch-size strategy. The name "Increasing batch-size" is used to denote this method. The idea is to bridge the gap between two ends of a spectrum, where $b_k = 1$ is at the end of "cheap iterations with slow convergence", and $b_k = N$ is "expensive iterations with fast convergence".

Let us summarize: we have defined three new methods that do not consider epochs anymore. At iteration k, an integer b_k between 1 and N is defined. Then, b_k data points are chosen, and only the rows of $\mathbf{L}$ and $\mathbf{R}$ corresponding to these data points are updated. In Figure 4.9, page 44, these three methods are compared to "Random shuffling" described on item 2 in Section 4.2. We see that the hybrid batch methods are faster, and the "Increasing batch-size" has the fastest convergence.

4.5 Conclusion

Two families of sequences for the data points have been defined. The first family (Section 4.2) is based on the way one would shuffle a deck of cards: either in a cyclic fashion, a random order or in some deterministic order based on interleaving the data points. The second family (Section 4.3) considers that some data points are more important than others, hence they should be visited more often.

The goal of these sequences is to improve the slow convergence obtained when the data points are visited in a cyclic order. We see that in most cases, the "Random shuffling (without replacement)" performs better than the "Cyclic shuffling". The "Smart shuffling" is comparable to the "Random shuffling" for synthetic data. For the "Weighted shuffling", we have defined an expensive method that works well in theory but not in practice. Another method, based on a cheap heuristic, performs slightly better than "Random shuffling".

Finally, we show that when it is not required to visit N data points at every epoch, hybrid

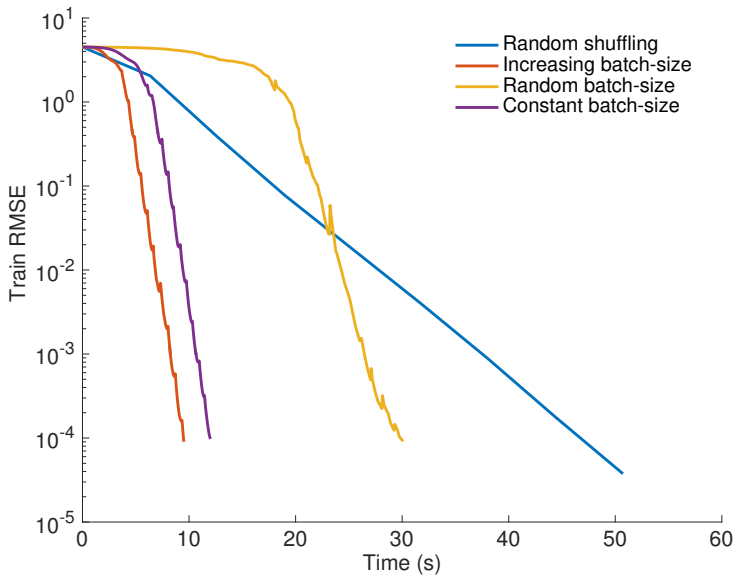


Figure 4.9: Evolution of the training error as a function of computation time. The scaled SGD method is applied with four different methods for the batch-size. The "Random shuffling" is described on item 2 in Section 4.2. The "Constant batch-size" takes $b_k = b = \frac{N}{10}$ for every k. The "Random batch-size" takes b_k at random between 1 and N for every k. The "Increasing batch-size" starts with $b_1 = 1$, then linearly adds some factor Δ, which defines an arithmetic sequence: $b_k = b_{k-1} + \Delta$. The factor Δ is called the common difference, and is set to $\Delta = \frac{M}{N}$ where M is the maximal number of iterations, set to 100. Initial matrices $\mathbf{L}_0$ and $\mathbf{R}_0$ are randomly generated according to a standard Gaussian distribution. We consider a matrix of size 1000×1000, rank known to be 10 and OS ratio set to 6. The bold driver protocol described at item 4 on page 25 is used, with an initial value of 0.1. The algorithm is stopped when either the MSE is less than 10^{-8} or the number of epochs exceeds 100.

methods can be defined that perform very well (Section 4.4).

This chapter puts together all of our contributions. Numerical comparisons can show that the proposed methods compete favorably with the state-of-the-art, like the standard alternating least squares (ALS) algorithm described in Appendix D.

Chapter 5

Applications

This last chapter tests the methods that we have discussed so far, both the already existing methods as the ones we have suggested, on three real-world data sets. The structure of this chapter will follow the one of Section 1.3. Namely, we will start with a data set coming from the medical world in Section 5.1. In Section 5.2 we solve a problem that often occurs in traffic management. Finally, Section 5.3 deals with the most famous application of the LRMC problem, namely the recommender systems.

5.1 Alzheimer's Disease

The authors of [TWYS13] have identified patients suffering from Alzheimer's Disease (AD) from normal controls. Therefore, they have used the Alzheimer's Disease Neuroimaging Initiative (ADNI) data set, which is available at [Wei13]. In this section, we run the scaled SGD method on this data set with three different shuffling techniques: the "Random shuffling" and the "Smart shuffling", presented in items 2 and 4 respectively, in Subsection 4.2.1, and the "Weighted shuffling" presented in Subsection 4.3.2. The ADNI data set contains $d_1 = 10762$ patients and $d_2 = 20$ descriptive measurements. By using ten-fold CV with, the rank was set to $r = 5$. We compute that 34.21% of the entries are missing.

Among other things, the measurements are derived from neuroimaging data, e.g., magnetic resonance imaging (MRI), and from biological data like the cerebruspinal fluid. The data set is represented by a small matrix, and one can readily see that it is a perfect example of LRMC, although the entries do not represent continuous variables. We attempt to recover a large amount of missing values in a matrix, i.e., to impute the missing data. However, the famous LRMC formulation of [CR09] relies on the assumption that the missing data is distributed randomly and uniformly. Unfortunately, this assumption does not hold in our case, since the data is missing in blocks, i.e., the entire data from a single modality might be missing.

In order to identify patients suffering from Alzheimer's Disease from normal controls, we build a classifier that assigns a label to a patient. The problem is that we need to make as few errors as possible in the imputation process, otherwise the errors introduced during this process will affect the performance of the classifier. Since the goal of this thesis is not to study classifiers, we focus on recovering the missing values.

Initially, the results were decent, although the assumption mentioned above was not satisfied. By shuffling the measurements and the patients such that the empty blocks are dispersed over

the data set, we have been able to improve the results significantly. In other words, the missing entries are uniformly distributed again. The results are illustrated in Appendix E.1, and the details of the algorithm are mentioned in the caption of Figure E.1 on page 63. The famous 80/20 rule is chosen: 80% of the data is used for training and 20% for testing. The evolution of the train RMSE defined in (4.7) is displayed as a function of the number of iterations. The regularization parameter λ is set to 10^{-3}, since this is a real-world data set, hence we know that adding a regularization term allows to improve the test RMSE. It appears that with these parameters, the weighted shuffling method attains the lowest train RMSE, while browsing the data points in a "smart" way, rather than in a "random" fashion, does not significantly improve the accuracy.

For the test RMSE, instead of showing the full result, we focus on one measurement. One of the tests is designed to determine how well-oriented the patient is with regard to time and place. In this test, there are a series of questions, among which the month of the year. If the patient answers the question correctly, the entry is filled with a 1, otherwise with a 0. However, some practicians go over the questions quickly, hence this question is sometimes overlooked. Therefore, for this particular column, the LRMC becomes a classification problem: given a patient for which we do not know whether he is well-oriented with regard to time and place, we want to predict if the entry is filled with a 1 or a 0. The Area Under the Curve or AUC is a scalar between 0 and 1, often used to measure the quality a classifier, see [Faw06]. We recall that the maximum AUC is 1, which corresponds to a perfect classifier. In our case, we find the AUC to be 0.7419 for this measurement, by using the "Weighted shuffling" method. All the other implementation details are specified in Appendix E.1.

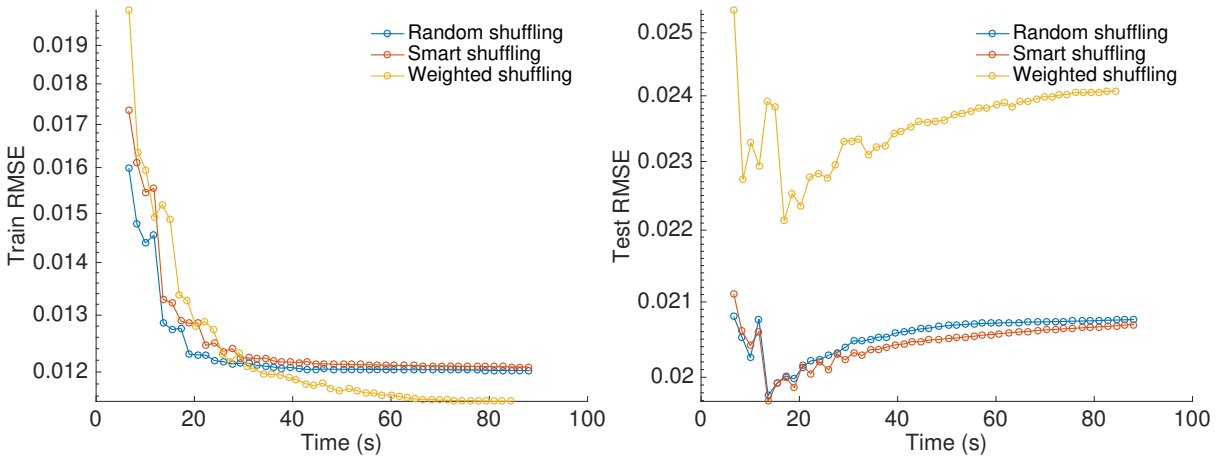
5.2 Traffic

The database available at [DKT17, Cut11] collects 15 months of data from the California Department of Transportation PEMS website, where PEMS stands for Performance Measurement System. The data describes the occupancy rate, between 0 and 1, of different car lanes of San Francisco freeways. The measurements cover roughly 450 days and are sampled every sixty minutes. Public holidays, as well as two days with anomalies, were removed from the data set, hence we only consider 440 days. We have a large matrix of size $d_1 \times d_2$, with $d_1 = 963$, the number of sensors which functioned consistently throughout the studied period, and $d_2 = 10560 = 24 \times 440$, the number of samples, i.e., one sample every hour during 440 days.

With this data set, one could construct a classifier to predict the day of the week as a function of the activity on the highway. Therefore, it makes sense to take $r = 7$, since there are seven days in a week. With the elbow method, it is checked that this is indeed a good choice.

By applying scaled SGD with the "Random without replacement" shuffling method presented in item 2 on page 32 to this data set, we have tried five different ways to define the step-size. The details of this experiment are available at Appendix E.2. It results that we choose the counter method described at item 3 on page 25, that is $\alpha_k = \frac{c_1}{c_2+k}$ with $c_1 = c_2 = 1$.

The given matrix has no missing entry. Therefore, a fraction of the entries are randomly removed with uniform probability. We refer the reader to Subsection 4.2.2 for a detailed explanation. The OS ratio is set to 4, hence the number of known entries is 82768. The results are depicted in Figure 5.1.



(a) Train RMSE

(b) Test RMSE

Figure 5.1: Evolution of the train and test RMSE for scaled SGD, using the three different shuffling methods: the "Random shuffling" and the "Smart shuffling", presented in items 2 and 4 respectively, in Subsection 4.2.1, and the "Weighted shuffling", presented in Subsection 4.3.2. For the experiments, the matrix is of size 963×1000. The rank and the OS ratio are set to $r = 7$ and $\text{OS} = 4$ respectively. Initial matrices $\mathbf{L}_0$ and $\mathbf{R}_0$ are computed using r dominant SVD. For the step-size, the counter method described at item 3 on page 25 is used, with $c_1 = c_2 = 1$, and with an initial value of 0.1. The algorithm is stopped when either the MSE defined as (4.6) is less than 10^{-8}, or the number of iterations exceeds 100. The same experiment is run five times, and the results are averaged. Note that the first five epochs are not shown, otherwise the graph would have been less convenient to scale for the vertical axis. Since it is clear that the weighted method overfits, different values of λ have been tried. Indeed, for real-world data sets, taking a non-zero value for the regularization parameter can improve the test RMSE. With the regularization, the three shuffling methods give similar results.

5.3 Recommender systems

As the last application, we look at a recommender system. The MovieLens data set, taken from [HK15], contains 20 million ratings applied to 27000 movies by 138000 users. This means that only 0.5368% of the entries or known, i.e., less than 1%. Out of the 20M possible ratings, we sample 1M of them, randomly.

With the scaled SGD method, we apply three shuffling methods: the "Random shuffling" and the "Smart shuffling", presented in items 2 and 4 respectively, in Subsection 4.2.1, and the "Weighted shuffling" presented in Subsection 4.3.2. The rank $r = 5$ was found using the elbow method, see Appendix E.3. The evolution of the train RMSE as a function of the number of iterations is displayed in Figure 5.2, and the details are mentioned in the caption of the Figure. It seems that the "Weighted shuffling" has the fastest convergence.

The weighted shuffling consistently outperforms random shuffling, as shown in Figure 5.2.

Table 5.1 shows the final NMAE defined in (4.5) obtained by different algorithms on the test data set averaged over five runs, for three different values of the rank. As a reminder, to define the NMAE we need the spread of the entries. In our case, the entries are the ratings, ranging from 0.5 to 5.0, hence the range is 4.5. The standard deviation of the scores in Table 5.1 is $8 \cdot 10^{-3}$.

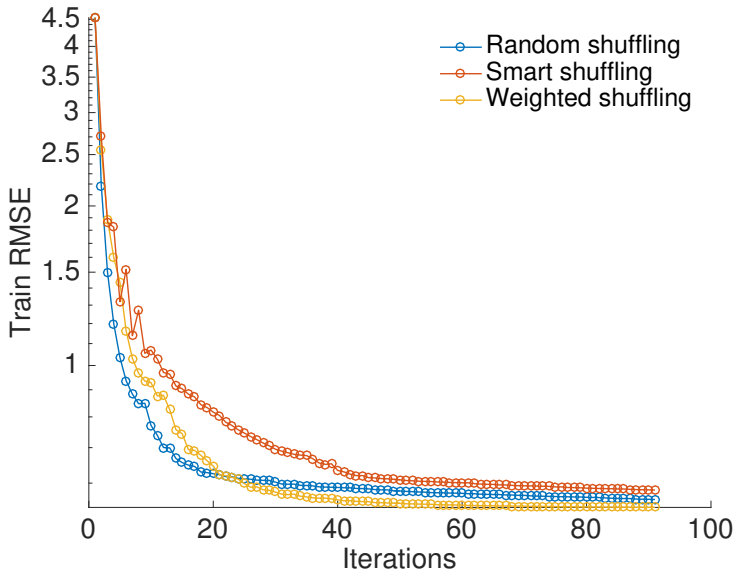


Figure 5.2: Evolution of the train RMSE as a function of the number of iterations for scaled SGD with the three shuffling methods: the "Random shuffling" and the "Smart shuffling", presented in items 2 and 4 respectively, in Subsection 4.2.1, and the "Weighted shuffling" presented in Subsection 4.3.2. The bold driver protocol described at item 4 on page 25 is chosen, with an initial value of 0.01. The MovieLens data set is used from [HK15], i.e., a matrix of size $d_1 \times d_2$, where $d_1 \leq 27000$ and $d_2 \leq 138000$ are determined by the randomly sampled entries. Initial matrices $\mathbf{L}_0$ and $\mathbf{R}_0$ are created with a r dominant SVD procedure. The algorithm is stopped either when the MSE defined in (4.6) is less than 10^{-8}, or when the maximal number of epochs is reached, namely 90.

	Random shuffling	Smart shuffling	Weighted shuffling
$r = 3$	0.245	0.248	0.242
$r = 5$	0.234	0.237	0.235
$r = 7$	0.226	0.228	**0.225**

Table 5.1: Normalized Mean Absolute Errors (NMAE, defined in (4.5)) obtained on the test set of the MovieLens 20M data set, taken from [HK15]. The "Random shuffling" and the "Smart shuffling" methods are presented in items 2 and 4 respectively, in Subsection 4.2.1, and the "Weighted shuffling" is presented in Subsection 4.3.2. The bold driver protocol described at item 4 on page 25 is chosen, with an initial value of 0.01. Initial matrices $\mathbf{L}_0$ and $\mathbf{R}_0$ are created with a r dominant SVD procedure. The algorithm is stopped either when the MSE defined in (4.6) is less than 10^{-8}, or when the maximal number of epochs is reached, namely 90. The same experiment is run five times, and the results are averaged.

Conclusion

In this master's thesis, we were concerned with the low-rank matrix completion problem. In other words, we studied the problem of recovering a low-rank matrix of which only a few entries are observed, possibly with noise. This appears in recommender systems, i.e., systems attempting to guess which items (movies, music, restaurants, and so on) different users might appreciate, based on *partial knowledge* of their preferences. Other applications include image processing, computer vision, the sensor network localization problem, and three more described in Section 1.3.

As often in machine learning, the objective function can be written as a large sum of individual functions, and the number of variables is very large. Therefore, the essential tool is *stochastic gradient descent*. If one decides to take into account all the data points at every epoch, then the question about the order arises: is there a particular order to browse the data points that would be better than the others?

After having stated the problem and defined the most important aspects of Riemannian optimization, we looked at different aspects of the SGD method, and attempted to tune it. The scaled SGD idea is taken from [MS16], but some intuition and a rigorous development are given to explain how it was obtained. Then we applied two methods to find the optimal rank, before the algorithm starts. For the sequences of the scaled SGD method, two families are defined. On the one hand, when all the data points must be visited at every epoch, we see that there is no particular order that works better than the others. In other words, the "Smart shuffling" method that was inspired from [MHA18] does not achieve better results than randomly browsing all the data points. On the other hand, it is argued that all the data points must not be visited at every epoch. Hence, the "Weighted shuffling" methods that we have defined have proven themselves to be more efficient than random walks through the data points, although the weighting process can be slow for large data sets.

As it is stated in [OAD18], the basic concept of rank can be generalized from matrices to tensors. Hence, the next steps after low-rank matrix completion is *low-rank tensor completion*. Recent papers like [KM16, NJM18, SGCH18] are already handling this subject. Two interesting ideas that we have applied to the LRMC problem can be generalized to tensors, namely the concepts of weighted sampling, and scaled SGD with a preconditioner.

Furthermore, the idea of shuffling the data points like a magician would shuffle a deck of cards is a promising concept. It can be applied to other problems where the objective function is a large sum of individual cost functions. In the very recent paper [TFBJ18], the authors construct a sequence of iterates produced from the first-order optimization of the cost function that are converging to a strict local minimum. Then, they consider and analyze the convergence of a streaming average of iterates defined in [TFBJ18, Eq. (2)]. This idea could be tested to see if the "Smart shuffling" method can be made *even smarter*.

Finally, we know that tuning the LRMC algorithm is crucial: effective recommender systems

require many application-specific insight in order to improve their accuracy. For example, in the Netflix case, temporal effects can improve the RMSE, because the popularity of some movies evolves with time, as well as users' preferences as they get older. In other words, some heuristics are necessary to design the best recommender systems; matrix completion is just one way to design such a system.

In conclusion, the low-rank matrix completion problem is far from being an easy task. Nowadays, many companies are massively collecting data, and are also wondering how to predict what they cannot measure. Therefore, finding ways to solve it efficiently is becoming more and more important. One could argue that predicting a preference, or the future, requires some intuition. If you are looking for a good song, a music connoisseur among your friends, who knows you and your tastes, might suggest you a better piece than the one Spotify will recommend. Or again, an experienced family physician might be more accurate in predicting a severe disease in your family than a computer. This being said, it is impossible not to refer to the enormous progress that algorithms have achieved in the last ten years. Suitable methods have been developed and could be used for crazy ideas as this one: adding face-recognition sensors to your Kindle e-reader, enabling Amazon to record what makes you laugh, and what makes you cry. Based on these measurements, it will recommend you other books, but even general products as well. *Your books will be reading you, as you are reading them. How long before machines can predict all of our preferences better than ourselves?*

Appendix A

Elements of Linear Algebra and Calculus

A.1 Rank of a matrix

Let us use $\mathbf{A}$ to denote a real matrix of size $d_1 \times d_2$. If $\mathbf{A}$ is decomposed as columns, we write

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{:1} & \dots & \mathbf{A}_{:d_2} \end{pmatrix}, \tag{A.1}$$

with $\mathbf{A}_{:j} \in \mathbb{R}^{d_1 \times 1}$ for $j = 1, \dots, d_2$. Similarly, if $\mathbf{A}$ is decomposed as rows, we write

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{1:}^\top \\ \vdots \\ \mathbf{A}_{d_1:}^\top \end{pmatrix},$$

with $\mathbf{A}_{i:} \in \mathbb{R}^{d_2 \times 1}$ for $i = 1, \dots, d_1$.

The column space of $\mathbf{A}$ is defined by

$$\mathcal{C}(\mathbf{A}) = \operatorname{span}\{\mathbf{A}_{:1}, \dots, \mathbf{A}_{:d_2}\},$$

where the span of a collection of vectors is the set of all possible linear combinations of these vectors. Similarly, the row space of $\mathbf{A}$ is defined by

$$\mathcal{R}(\mathbf{A}) = \operatorname{span}\{\mathbf{A}_{1:}, \dots, \mathbf{A}_{d_1:}\}.$$

The *range* of $\mathbf{A}$, also called the *image*, is defined by

$$\operatorname{ran}(\mathbf{A}) = \left\{ y \in \mathbb{R}^{d_1 \times 1} : \exists x \in \mathbb{R}^{d_2 \times 1}, y = \mathbf{A}x \right\} = \mathcal{C}(\mathbf{A}).$$

The *null space* of $\mathbf{A}$, also called the *kernel*, is defined by

$$\operatorname{null}(\mathbf{A}) = \left\{ x \in \mathbb{R}^{d_2 \times 1} : \mathbf{A}x = 0 \right\}.$$

The *column rank* is defined as the dimension of the column space, and the *row rank* as the dimension of the row space. A fundamental result in linear algebra is that these two ranks are always equal. Therefore, this number is simply called the *rank* of the matrix $\mathbf{A}$.

The matrix $\mathbf{A}$ is said to be full rank if $\text{rank}(\mathbf{A}) = \min\{d_1, d_2\}$, and $\mathbf{A}$ is said to be rank deficient if $\text{rank}(\mathbf{A}) < \min\{d_1, d_2\}$.

The famous rank-nullity theorem states that

$$\text{nullity}(\mathbf{A}) + \text{rank}(\mathbf{A}) = d_2, \tag{A.2}$$

where the nullity of $\mathbf{A}$ is simply defined as $\text{nullity}(\mathbf{A}) = \dim(\text{null}(\mathbf{A}))$.

We end this section with an important proposition.

Proposition A.1. *Let $\mathbf{M} \in \mathbb{R}^{d_1 \times d_2}$ for some positive integers d_1 and d_2. Let $r \in \mathbb{R}$ such that $r \leq \min\{d_1, d_2\}$. The following statements are equivalent:*

(1) $\text{rank}(\mathbf{M}) = r$;

(2) $\exists \mathbf{A} \in \mathbb{R}_*^{d_1 \times r}, \mathbf{B} \in \mathbb{R}_*^{d_2 \times r}$ *such that* $\text{rank}(\mathbf{A}) = \text{rank}(\mathbf{B}) = r$ *and* $\mathbf{M} = \mathbf{A}\mathbf{B}^\top$.

Proof. We have to prove that the two statements are equivalent.

(1) $\Rightarrow$ (2)
Let $\mathbf{A} \in \mathbb{R}_*^{d_1 \times r}$ be a basis of $\mathbf{M}$. In that case, the columns of $\mathbf{A}$ form a basis of $\mathcal{C}(\mathbf{M})$. Thus $\text{rank}(\mathbf{A}) = r$, and there exists $\mathbf{B} \in \mathbb{R}^{d_2 \times r}$ such that

$$\mathbf{M} = \mathbf{A}\mathbf{B}^\top \Leftrightarrow \begin{pmatrix} \mathbf{M}_{:1} & \dots & \mathbf{M}_{:d_2} \end{pmatrix} = \begin{pmatrix} \mathbf{A}_{:1} & \dots & \mathbf{A}_{:r} \end{pmatrix} \begin{pmatrix} B_{11} & \dots & B_{d_2 1} \\ \vdots & & \vdots \\ B_{1r} & \dots & B_{d_2 r} \end{pmatrix},$$

$$\Leftrightarrow \mathbf{M}_{:j} = \sum_{k=1}^{r} B_{jk} \mathbf{A}_{:k}, \quad j = 1, \dots, d_2.$$

It remains to prove that $\text{rank}(\mathbf{B}) = r$. We already know that $\text{rank}(\mathbf{B}) \leq r$, because $\mathbf{B} \in \mathbb{R}^{d_2 \times r}$ and $r \leq d_2$. We also know that

$$r = \text{rank}(\mathbf{M}) \leq \min\left\{\text{rank}(\mathbf{A}), \text{rank}\left(\mathbf{B}^\top\right)\right\} = \min\{r, \text{rank}(\mathbf{B})\},$$

or again

$$r \leq \min\{r, \text{rank}(\mathbf{B})\}.$$

From this, it is clear that $\text{rank}(\mathbf{B}) = r$.

(2) $\Rightarrow$ (1)
As $\text{rank}(\mathbf{M}) \leq \min\{\text{rank}(\mathbf{A}), \text{rank}(\mathbf{B})\} = r$, we have to prove that $\text{rank}(\mathbf{M}) = r$. By using (A.2), this is equivalent to proving that $\text{nullity}(\mathbf{M}) = d_2 - r$. We have

$$\begin{aligned} &\mathbf{M}\mathbf{x} = \mathbf{0} \\ &\Leftrightarrow \mathbf{A}\mathbf{B}^\top\mathbf{x} = \mathbf{0} \\ &\Leftrightarrow \mathbf{B}^\top\mathbf{x} \in \ker(\mathbf{A}) \\ &\Leftrightarrow \mathbf{B}^\top\mathbf{x} = \mathbf{0} \\ &\Leftrightarrow \mathbf{x} \in \ker\left(\mathbf{B}^\top\right), \end{aligned}$$

where the third implication stems from $\ker(\mathbf{A}) = \{0\}$, since $\mathbf{A}$ has full rank. We know that $\text{nullity}\left(\mathbf{B}^\top\right) = \dim\left(\ker\left(\mathbf{B}^\top\right)\right) = d_2 - r$. Therefore, $\text{nullity}(\mathbf{M}) = d_2 - r$. Hence, $\text{rank}(\mathbf{M}) = r$.

The proof is complete. □

A.2 Matrix norms

The theory about matrix norms is vast. As a reminder, a function $\|\cdot\| : \mathbb{R}^{d_1 \times d_2} \mapsto \mathbb{R}$ is called a *matrix norm* on $\mathbb{R}^{d_1 \times d_2}$ if for all $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{d_1 \times d_2}$ and all $\alpha \in \mathbb{R}$ we have

1. Positivity: $\|\mathbf{A}\| \geq 0$, with equality if and only if $\mathbf{A} = \mathbf{0}$.
2. Homogeneity: $\|\alpha\mathbf{A}\| = |\alpha| \, \|\mathbf{A}\|$.
3. Subadditivity: $\|\mathbf{A} + \mathbf{B}\| \leq \|\mathbf{A}\| + \|\mathbf{B}\|$.

Three types of norms can be defined, with some special cases.

1. Matrix norms induced by vector norms, or operator norms:

$$\|\mathbf{A}\|_{(a,b)} = \sup_{\mathbf{x} \neq 0} \frac{\|\mathbf{A}\mathbf{x}\|_a}{\|\mathbf{x}\|_b},$$

   where $\mathbf{x}$ is a vector of size $d_2 \times 1$, hence $\|\cdot\|_a$ denotes a vector norm for vectors of size $d_1 \times 1$ and $\|\cdot\|_b$ denotes a vector norm for vectors of size $d_2 \times 1$. Often one takes $a = b = p$, hence p can denote a norm for an object of size $d_1 \times d_2$, of size $d_1 \times 1$ and of size $d_2 \times 1$.

2. Entry-wise matrix norms:

$$\|\mathbf{A}\|_{\ell_p} = \left(\sum_{i=1,j=1}^{d_1,d_2} |A_{ij}|^p \right)^{1/p}.$$

   The famous Frobenius norm is a special case of the entry-wise matrix norm with $p = 2$. We have

$$\|\mathbf{A}\|_{\ell_2} = \|\mathbf{A}\|_F = \|\mathbf{A}\|_2 = \left(\sum_{i=1,j=1}^{d_1,d_2} |A_{ij}|^2 \right)^{1/2}.$$

3. Schatten norms:

$$\|\mathbf{A}\|_{S_p} = \left(\sum_{i=1}^{d_{\min}} \sigma_i^p \right)^{1/p},$$

   where $\sigma_1, \ldots, \sigma_{d_{\min}}$ are the singular values of $\mathbf{A}$ and $d_{\min}$ is the minimum between d_1 and d_2. The nuclear norm is a special case of the Schatten norm with $p = 1$. We have

$$\|\mathbf{A}\|_{S_1} = \|\mathbf{A}\|_* = \sum_{i=1}^{d_{\min}} \sigma_i.$$

A.3 Derivatives

This corresponds to [AMS08, Appendix A.5]. The concept of a derivative for functions between two finite-dimensional normed vector spaces is presented.

Let $\mathcal{E}$ and $\mathcal{F}$ be two finite-dimensional vector spaces over $\mathbb{R}$. A particular case is $\mathcal{E} = \mathbb{R}^m$ and $\mathcal{F} = \mathbb{R}^n$. A function $f : \mathcal{E} \mapsto \mathcal{F}$ is *Fréchet-differentiable* at a point $x \in \mathcal{E}$ if there exists a linear operator

$$\mathrm{D}f(x) : \mathcal{E} \to \mathcal{F} : h \mapsto \mathrm{D}f(x)[h],$$

called the *Fréchet differential* or the *Fréchet derivative* of f at x, such that

$$f(x+h) = f(x) + \mathrm{D}f(x)[h] + o(\|h\|);$$

in other words,

$$\lim_{y \to x} \frac{\|f(y) - f(x) - \mathrm{D}f(x)[y-x]\|}{\|y-x\|} = 0.$$

The element $\mathrm{D}f(x)[h] \in \mathcal{F}$ is called the *directional derivative* of f at x along h. We use the same notation $\mathrm{D}f(x)$ for the differential of a function f between two manifolds $\mathcal{M}_1$ and $\mathcal{M}_2$; then $\mathrm{D}f(x)$ is a linear operator from the vector space $T_x\mathcal{M}_1$ to the vector space $T_{f(x)}\mathcal{M}_2$.

A.4 Kronecker product and matrix vectorization

Let us take two matrices $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{B} \in \mathbb{R}^{p \times q}$. The *Kronecker product* of $\mathbf{A}$ and $\mathbf{B}$ is defined as

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} A_{11}\mathbf{B} & \cdots & A_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ A_{m1}\mathbf{B} & \cdots & A_{mn}\mathbf{B} \end{bmatrix} \in \mathbb{R}^{mp \times nq}.$$

More compactly, we have

$$(A \otimes B)_{p(i-1)+k,q(j-1)+l} = A_{ij}B_{kl},$$

for every $(i,j) \in \{1,\ldots,m\} \times \{1,\ldots,n\}$ and $(k,l) \in \{1,\ldots,p\} \times \{1,\ldots,q\}$.

Let us go back to the case where $\mathbf{A}$ denotes a real matrix of size $d_1 \times d_2$. If we decompose $\mathbf{A}$ as columns just like in (A.1), then the $\mathrm{vec}(\cdot)$ operator creates a column vector from matrix $\mathbf{A}$ by stacking the columns below one another. We have

$$\mathrm{vec}(\mathbf{A}) = \begin{bmatrix} \mathbf{A}_{:1} \\ \vdots \\ \mathbf{A}_{:n} \end{bmatrix} \in \mathbb{R}^{d_1 d_2 \times 1}.$$

With this definition, the most important result linking the Kronecker product and the $\mathrm{vec}(\cdot)$ operator can be stated. Given three matrices of compatible sizes $\mathbf{A} \in \mathbb{R}^{d_1 \times d_2}$, $\mathbf{X} \in \mathbb{R}^{d_2 \times d_3}$ and $\mathbf{B} \in \mathbb{R}^{d_3 \times d_4}$, we have

$$\mathrm{vec}(\mathbf{AXB}) = \left(\mathbf{B}^\top \otimes \mathbf{A}\right) \mathrm{vec}(\mathbf{X}).$$

The most important property of the vec $(\cdot)$ operator is

$$\text{Trace}\left\{\mathbf{A}^\top \mathbf{B}\mathbf{C}\mathbf{D}^\top\right\} = \text{vec}\,(\mathbf{A})^\top\,(\mathbf{D}\otimes\mathbf{B})\,\text{vec}\,(\mathbf{C})\,.$$

If we take $\mathbf{B} = \mathbf{I}$ and relabel $\mathbf{C}$ and $\mathbf{D}$, we obtain

$$\text{Trace}\left\{\mathbf{A}^\top \mathbf{B}\mathbf{C}^\top\right\} = \text{vec}\,(\mathbf{A})^\top\,(\mathbf{C}\otimes\mathbf{I})\,\text{vec}\,(\mathbf{B})\,. \tag{A.3}$$

For compatible matrices $\mathbf{X}$ and $\mathbf{Y}$, we have

$$\text{Trace}\,\{\mathbf{X}\mathbf{Y}\} = \text{Trace}\,\{\mathbf{Y}\mathbf{X}\} = \text{Trace}\left\{(\mathbf{X}\mathbf{Y})^\top\right\},$$

and the full version with three compatible matrices $\mathbf{X}$, $\mathbf{Y}$ and $\mathbf{Z}$ is

$$\text{Trace}\,\{\mathbf{X}\mathbf{Y}\mathbf{Z}\} = \text{Trace}\,\{\mathbf{Y}\mathbf{Z}\mathbf{X}\} = \text{Trace}\,\{\mathbf{Z}\mathbf{X}\mathbf{Y}\}\,.$$

More generally, the trace is *invariant under cyclic permutations*. This can be used to rewrite (A.3) as

$$\text{Trace}\,\{\mathbf{A}\mathbf{B}\mathbf{C}\} = \text{vec}\left(\mathbf{B}^\top\right)^\top\left(\mathbf{A}^\top\otimes\mathbf{I}\right)\text{vec}\,(\mathbf{C})\,. \tag{A.4}$$

A.5 Matrix exponential and logarithm of a matrix

Let us use $\mathbb{P}_n$ to denote the cone of $n\times n$ symmetric positive definite matrices. We recall that a matrix $\mathbf{G}\in\mathbb{P}_n$ has the following eigen-decomposition

$$\mathbf{G} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^\top,$$

where $\mathbf{U}$ is an orthogonal matrix containing the eigenvectors, and $\boldsymbol{\Lambda}$ is a diagonal matrix containing the eigenvalues.

Then the matrix $\mathbf{G}^a$ is defined for each $a\in\mathbb{R}$ as

$$\mathbf{G}^a = \mathbf{U}\boldsymbol{\Lambda}^a\mathbf{U}^\top,$$

where $\boldsymbol{\Lambda}^a$ is a diagonal matrix with the diagonal elements of $\boldsymbol{\Lambda}$ raised to the power a. In particular if $a = -\frac{1}{2}$, we have

$$\mathbf{G}^{-\frac{1}{2}} = \mathbf{U}\boldsymbol{\Lambda}^{-\frac{1}{2}}\mathbf{U}^\top,$$

where $\boldsymbol{\Lambda}^{-\frac{1}{2}}$ is computed as the inverse of the square root of the diagonal elements of $\boldsymbol{\Lambda}$.

The definition of the logarithm of a scalar can also be extended to a matrix. Let us define the diagonal matrix $\log(\boldsymbol{\Lambda})$, constructed as the element-wise natural logarithm of $\boldsymbol{\Lambda}$. Then we have

$$\log(\mathbf{G}) = \mathbf{U}\log(\boldsymbol{\Lambda})\,\mathbf{U}^\top.$$

Appendix B

Proofs

B.1 Proof of Proposition 2.1

Proof. Three properties need to be verified.

1. Reflexive: $(\mathbf{L}, \mathbf{R}) \sim (\mathbf{L}, \mathbf{R})$ for all $(\mathbf{L}, \mathbf{R}) \in \mathbb{R}_*^{d_1 \times r} \times \mathbb{R}_*^{d_2 \times r}$.
   It suffices to take $\mathbf{T} = \mathbf{I}_{r \times r}$.

2. Symmetric: $(\mathbf{L}, \mathbf{R}) \sim (\mathbf{G}, \mathbf{H}) \Leftrightarrow (\mathbf{G}, \mathbf{H}) \sim (\mathbf{L}, \mathbf{R})$ for all $(\mathbf{L}, \mathbf{R}) \in \mathbb{R}_*^{d_1 \times r} \times \mathbb{R}_*^{d_2 \times r}$ and $(\mathbf{G}, \mathbf{H}) \in \mathbb{R}_*^{d_1 \times r} \times \mathbb{R}_*^{d_2 \times r}$.
   This has to be proved in two ways: ($\Rightarrow$) and ($\Leftarrow$). We will only prove the first part ($\Rightarrow$), since the second part follows by symmetry: a simple relabeling suffices.
   If $(\mathbf{L}, \mathbf{R}) \sim (\mathbf{G}, \mathbf{H})$, then there exists a matrix $\mathbf{T} \in \mathrm{GL}(r)$ such that $\mathbf{G} = \mathbf{L}\mathbf{T}^{-1}$ and $\mathbf{H} = \mathbf{R}\mathbf{T}^\top$. Let us define the matrix $\mathbf{U}$ as $\mathbf{U} = \mathbf{T}^{-1}$. It is clear that $\mathbf{U} \in \mathrm{GL}(r)$. We have $\mathbf{L} = \mathbf{G}\mathbf{U}^{-1}$ and $\mathbf{R} = \mathbf{H}\mathbf{U}^\top$. This gives $(\mathbf{G}, \mathbf{H}) \sim (\mathbf{L}, \mathbf{R})$.

3. Transitive: if $(\mathbf{L}, \mathbf{R}) \sim (\mathbf{G}, \mathbf{H})$ and $(\mathbf{G}, \mathbf{H}) \sim (\mathbf{A}, \mathbf{B})$ then $(\mathbf{L}, \mathbf{R}) \sim (\mathbf{A}, \mathbf{B})$ for all $\mathbf{L}, \mathbf{G}, \mathbf{A} \in \mathbb{R}_*^{d_1 \times r}$ and $\mathbf{R}, \mathbf{H}, \mathbf{B} \in \mathbb{R}_*^{d_2 \times r}$.
   There exist matrices $\mathbf{T}$ and $\mathbf{U}$, both in $\mathrm{GL}(r)$, such that $\mathbf{G} = \mathbf{L}\mathbf{T}^{-1}, \mathbf{H} = \mathbf{R}\mathbf{T}^\top$ and $\mathbf{A} = \mathbf{G}\mathbf{U}^{-1}, \mathbf{B} = \mathbf{H}\mathbf{U}^\top$. Let us define the matrix $\mathbf{V}$ as $\mathbf{V} = \mathbf{U}\mathbf{T}$. We see that $\mathbf{V} \in \mathrm{GL}(r)$. Furthermore, we have $\mathbf{V}^\top = \mathbf{T}^\top\mathbf{U}^\top$ and $\mathbf{V}^{-1} = \mathbf{T}^{-1}\mathbf{U}^{-1}$. Hence, we have $\mathbf{A} = \mathbf{L}\mathbf{T}^{-1}\mathbf{U}^{-1} = \mathbf{L}\mathbf{V}^{-1}$ and $\mathbf{B} = \mathbf{R}\mathbf{T}^\top\mathbf{U}^\top = \mathbf{R}\mathbf{V}^\top$. Therefore, we can write $(\mathbf{L}, \mathbf{R}) \sim (\mathbf{A}, \mathbf{B})$.

Since the three properties of an equivalence relation are verified, the proof is complete. □

B.2 Proof of Proposition 4.1

Proof. The objective variable is a matrix $\mathbf{X}$ of size $d_1 \times d_2$ and of rank r. Hence, this matrix can, in theory only, be represented by a vector $\mathbf{x}$ of size $d_1 d_2 \times 1$. Let us use $N = |\Omega|$ to denote the number of known entries, as usual. We define a vector $\mathbf{b}$ that contains all the sampled elements of $\mathbf{M}$, column-wise. Thus, $\mathbf{b}$ is a vector of size $N \times 1$. We also define a matrix $\mathbf{A}$ of size $N \times d_1 d_2$. The entries of $\mathbf{A}$ are only binary, i.e., they are either equal to 0 or 1. In order to find out which entries are equal to 1, let us pick an element $(i, j) \in \Omega$, and let us assume that the corresponding entry M_{ij} is represented at index n of vector $\mathbf{b}$, i.e., $M_{ij} = b_n$. It must be that, using MATLAB

notation, $\mathbf{A}_{n:}\mathbf{x} = X_{ij}$. In other words, on row n, the matrix $\mathbf{A}$ is zero everywhere, except at the column corresponding to the entry X_{ij}.

- On every row of $\mathbf{A}$, there is *exactly* one entry that is equal to 1, while all the other entries are equal to 0.
- On every column of $\mathbf{A}$, there is *at most* one entry that is equal to 1, while all the other entries are equal to 0:
  - If, on the one hand, the column corresponds to an entry X_{ij} such that $(i, j) \in \Omega$, then this column has *exactly* one entry that is equal to 1.
  - If, on the other hand, the column corresponds to an entry X_{ij} such that $(i, j) \notin \Omega$, then this column has only zero entries.

With these definitions of $\mathbf{A}$ and $\mathbf{b}$, the objective function (3.3) can be written as

$$f(\mathbf{x}) = \frac{1}{2}\sum_{n=1}^{N}(\mathbf{A}_{n:}\mathbf{x} - b_n)^2 = \frac{1}{2}\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2 .$$

Since the quantity $\mathbf{A}\mathbf{x} - \mathbf{b}$ is a vector, the Frobenius norm is not needed anymore, but of course it is just the vector version of the ℓ_2 norm. This concludes the proof. □

Appendix C

Implementation details

C.1 MATLAB or Python?

In the machine learning community, Python dominates MATLAB, and `scikit-learn` is the most famous software machine learning library for Python. At this time, it may be useful to mention Surprise, a Python SciPy toolkit for building and analyzing recommender systems, and PyManopt, a Python toolbox for optimization on manifolds that builds upon Manopt [BMAS14]. In the long term, it looks clear that even if Python is a general-purpose programming language that requires add-on libraries, it will become more and more popular in the scientific community. In particular, implementing K-fold CV only requires a couple of lines of code with `scikit-learn`, while the MATLAB implementation is more intricate.

C.2 MATLAB or C++?

At some point, the idea of writing some code in C++ arose. We decided not to follow that path, because the observations would have been clear: it all depends on the Basic Linear Algebra Subprograms, also called BLAS. It is well-known that MATLAB cannot efficiently handle `for` loops. Hence, on the one hand, for small matrices, MATLAB and C++ will give similar results. On the other hand, for large matrices, C++ will be a lot faster than MATLAB. In short, since MATLAB is a script language but C++ is a compiled language, the former will lose time due to internal overhead.

Appendix D

Alternating Least Squares

For simplicity, let us first assume that all the entries of $\mathbf{M}$ are sampled, i.e., $\Omega = [d_1] \times [d_2]$. Then the objective function (3.4) becomes

$$f(\mathbf{L}, \mathbf{R}) = \frac{1}{2} \sum_{i=1}^{d_1} \sum_{j=1}^{d_2} (M_{ij} - \mathbf{L}_{i:}\mathbf{R}_{j:})^2 = \frac{1}{2} \left\| \mathbf{M} - \mathbf{L}\mathbf{R}^\top \right\|_F^2 .$$

The idea of the alternating least squares (ALS) method is to use coordinate descent to minimize the cost. The approach is thus to minimize with respect to $\mathbf{L}$, while $\mathbf{R}$ is fixed; then to minimize with respect to $\mathbf{R}$, while $\mathbf{L}$ is fixed: this is the *alternating* part. This two-step dance is repeated until convergence is attained. In order to find the update rules, the following system is solved

$$\begin{cases} \nabla_{\mathbf{L}} f(\mathbf{L}^*, \mathbf{R}) = 0, \\ \nabla_{\mathbf{R}} f(\mathbf{L}, \mathbf{R}^*) = 0. \end{cases} \tag{D.1}$$

We find

$$\begin{cases} \mathbf{L}^* = \mathbf{M}\mathbf{R} \left(\mathbf{R}^\top \mathbf{R} \right)^{-1}, \\ \mathbf{R}^* = \mathbf{M}^\top \mathbf{L} \left(\mathbf{L}^\top \mathbf{L} \right)^{-1}. \end{cases} \tag{D.2}$$

Of course, we usually do not have $\Omega = [d_1] \times [d_2]$. Thus, the objective function is (3.4) and the system (D.1) needs to be solved again. The solution will not be as elegant as (D.2) since all the entries are not sampled in real situations. Therefore, some notations are introduced.

First, let us fix an index i. We say that $\sum_{(\cdot,j)\in\Omega} M_{ij}$ is the sum of all the entries M_{ij} such that $(i,j) \in \Omega$. Hence, the dot in $(\cdot, j)$ means that the first index is fixed, and that the sum is taken over the second index. Similarly, when an index j is fixed, the notation $\sum_{(i,\cdot)\in\Omega} M_{ij}$ has the same interpretation: the second index is fixed, and the sum is taken over the first index. Solving (D.1) gives

$$\begin{cases} \mathbf{L}^*{}_{i:} = \sum_{(\cdot,j)\in\Omega} M_{ij}\mathbf{R}_{j:} \left[\sum_{(\cdot,j)\in\Omega} \mathbf{R}_{j:}^\top \mathbf{R}_{j:} \right]^{-1}, \\ \mathbf{R}^*{}_{j:} = \sum_{(i,\cdot)\in\Omega} M_{ij}\mathbf{L}_{i:} \left[\sum_{(i,\cdot)\in\Omega} \mathbf{L}_{i:}^\top \mathbf{L}_{i:} \right]^{-1}. \end{cases}$$

The advantage of ALS is that there are several ways to distribute its computation, as it is stated in [ZWSP08].

Appendix E

Details for Chapter 5

E.1 Alzheimer's Disease

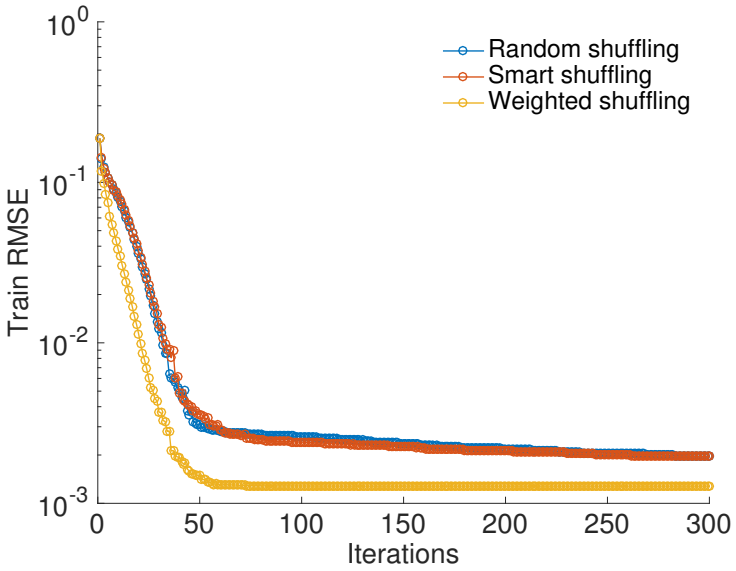


Figure E.1: Evolution of the train RMSE on the "Alzheimer's Disease Neuroimaging Initiative" data set [Wei13] for scaled SGD defined by (3.20) with three different methods: the "Random shuffling" and the "Smart shuffling" presented in items 2 and 4 respectively, in Subsection 4.2.1, and the "Weighted shuffling" presented in Subsection 4.3.2. For the "Weighted shuffling" we take $w_{ij} = 1$ and $\beta = \frac{3N}{4}$. The original matrix $\mathbf{M}$ has size 10762×20, from which 34.21% of the entries are missing, and the rank is fixed to $r = 5$ with ten-fold CV. The algorithm is initialized based on the r dominant SVD of $\mathcal{P}_\Omega(\mathbf{M})$. The maximal number of epochs is set to 300, while the tolerance level for the MSE defined in (4.6) is 10^{-8}. The bold driver protocol described at item 4 on page 25 is chosen, with an initial value of 0.01. The regularization parameter λ is set to 10^{-3}. The same experiment is run five times, and we average the results.

E.2 Traffic

The scaled SGD method with the "Random shuffling" method presented in item 2 on page 32 is applied to the data set described in the first paragraph of Section 5.2. Regarding the step-size,

the five methods presented in Section 3.5 are tested.

A subset of 2000 samples is selected randomly, in order to reduce the size of the problem. The maximal number of epochs is set to 50 and the tolerance level for the train MSE is 10^{-8}. In practice, we will never reach the tolerance, and the algorithm will stop after 50 epochs. The initial matrices $(\mathbf{L}_0, \mathbf{R}_0)$ are defined with a r dominant SVD procedure. The results are illustrated in Figure E.2. We see that the bold driver and the counter methods are performing similarly. Therefore, for once, let us use the counter method instead of the bold driver.

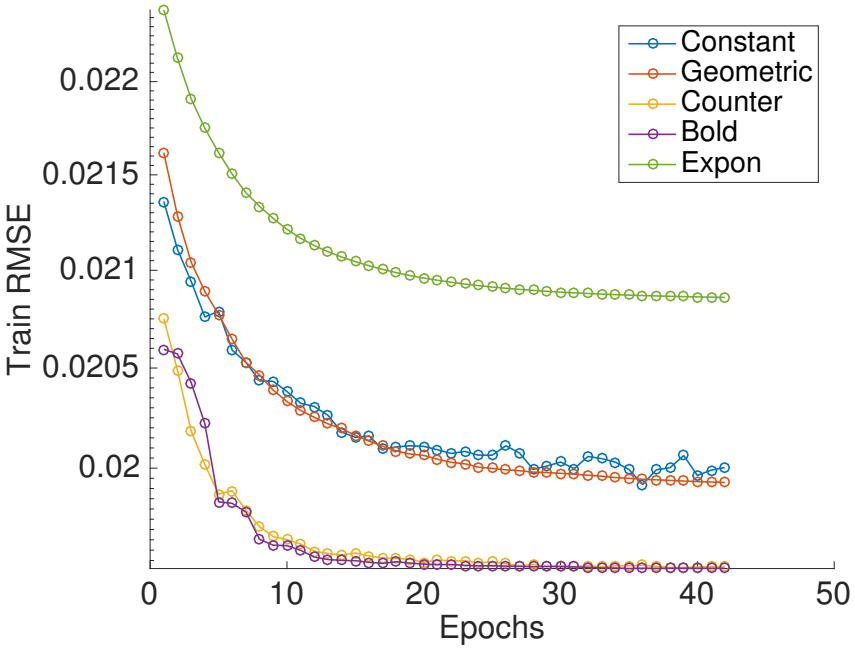


Figure E.2: Evolution of the train RMSE for scaled SGD method with the random shuffling method presented in item 2 applied to the traffic data set [DKT17, Cut11]. Five different step-size methods described in Section 3.5 are tested. The initial step-size is set to 0.1. For the geometric case, the parameter ρ is set to 0.95. For the counter idea, the constants are set to $c_1 = 1$ and $c_2 = 2$. For the exponential case, the parameters are $\alpha_0 = 1$ and $c = 0.1$.

E.3 Recommender systems

The rank r is a *hyperparameter*, i.e., a parameter that the user will be able to choose. Often, a value is chosen by default, however we have discussed two ways to choose a good value of r is Section 4.1. Let us give a quick summary: for the elbow method, the test and train RMSE are displayed as a function of the rank. We try to identify a kink in the train RMSE, i.e., a value of r such that the slopes change. Another graphical method looks at the test RMSE, and finds where it is minimized, such that both underfitting and overfitting are avoided.

The scaled SGD method with the "Random shuffling" method presented in item 2 on page 32 is applied to the MovieLens data set described in the first paragraph of Section 5.3. The bold driver protocol described at item 4 on page 25 is chosen, with an initial value of 0.01. For the rank, a total of 15 ranks are considered, and the best rank seems to be 5. In Figure E.3, the kink is not very clear, hence we rely on the second method: the lowest point for the test RMSE appears to be at $r = 5$.

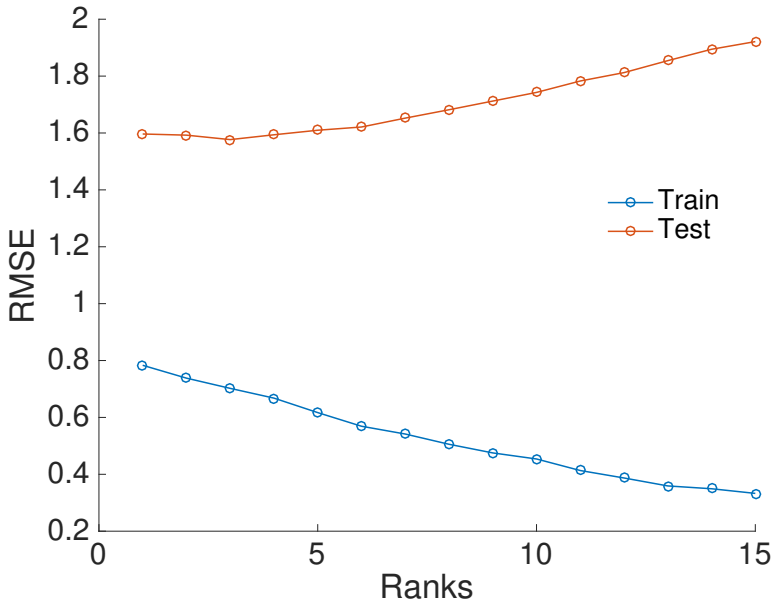


Figure E.3: Evolution of the train and test RMSE defined in (4.7) as a function of the rank for the MovieLens 20M data set, taken from [HK15]. The scaled SGD method (3.20) is used with the "Random shuffling" technique presented in items 2 of Subsection 4.2.1. The bold driver protocol described at item 4 on page 25 is chosen, with an initial value of 0.01. Initial matrices $\mathbf{L}_0$ and $\mathbf{R}_0$ are created with a r dominant SVD procedure. The algorithm is stopped either when the MSE defined in (4.6) is less than 10^{-8}, or when the maximal number of epochs is reached, namely 100. The same experiment is run five times, and the results are averaged.

Bibliography

[AMS08] P.-A. Absil, R. Mahony, and R. Sepulchre. *Optimization Algorithms on Matrix Manifolds*. Princeton University Press, Princeton, NJ, 2008. URL: `https://press.princeton.edu/titles/8586.html`.

[AO15] P.-A. Absil and V. Oseledets. Low-rank retractions: a survey and new results. *Computational Optimization and Applications*, 62(1):5–29, 2015. `doi:10.1007/s10589-014-9714-4`.

[BA11] N. Boumal and P.-A. Absil. RTRMC: A Riemannian trust-region method for low-rank matrix completion. In *Advances in neural information processing systems*, pages 406–414, 2011.

[BA15] N. Boumal and P.-A. Absil. Low-rank matrix completion via preconditioned optimization on the Grassmann manifold. *Linear Algebra and its Applications*, 475(0):200 – 239, 2015. `doi:10.1016/j.laa.2015.02.027`.

[Bis11] C. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag New York, 2011. URL: `https://dl.acm.org/citation.cfm?id=1162264`.

[BK07] R. Bell and Y. Koren. Scalable collaborative filtering with jointly derived neighborhood interpolation weights. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)*, pages 43–52, 2007. `doi:10.1109/ICDM.2007.90`.

[BL07] J. Bennett and S. Lanning. The Netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35, 2007.

[BMAS14] N. Boumal, B. Mishra, P.-A. Absil, and R. Sepulchre. Manopt, a Matlab toolbox for optimization on manifolds. *Journal of Machine Learning Research*, 15:1455–1459, 2014. URL: `http://www.manopt.org`.

[BN08] D. Behmardi and E. Nayer. Introduction of Fréchet and Gâteaux derivative. *Applied Mathematical Sciences*, 2(20):975 – 980, 2008.

[Bon13] S. Bonnabel. Stochastic gradient descent on Riemannian manifolds. *IEEE Transactions on Automatic Control*, 58(9):2217–2229, 2013. `doi:10.1109/TAC.2013.2254619`.

[Bou14] N. Boumal. *Optimization and estimation on manifolds*. PhD thesis, Université catholique de Louvain, 2014. URL: `http://hdl.handle.net/2078.1/142831`.

[BV04] S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge University Press, 2004. URL: `https://dl.acm.org/citation.cfm?id=993483`.

[Cam15] L. Cambier. Robust low-rank matrix completion. Master's thesis, Université catholique de Louvain, 2015. URL: `https://people.stanford.edu/lcambier/publications`.

[CCS10] J.-F. Cai, E. Candès, and Z. Shen. A singular value thresholding algorithm for matrix completion. *SIAM Journal on Optimization*, 20(4):1956–1982, 2010. doi:10.1137/080738970.

[CG84] A. Chistov and D. Grigor'ev. Complexity of quantifier elimination in the theory of algebraically closed fields. In *Mathematical Foundations of Computer Science 1984*, pages 17–31. Springer, 1984. doi:10.1007/BFb0030287.

[CMY+16] F. Chan, A. Ma, P. Yuen, T. Yip, Y. Tse, V. Wong, and G. Wong. Temporal matrix completion with locally linear latent factors for medical applications. *CoRR*, abs/1611.00800, 2016. arXiv:1611.00800.

[CP10] E. Candès and Y. Plan. Matrix completion with noise. *Proceedings of the IEEE*, 98(6):925–936, 2010. doi:10.1109/JPROC.2009.2035722.

[CR09] E. Candès and B. Recht. Exact matrix completion via convex optimization. *Foundations of Computational mathematics*, 9(6):717–772, 2009.

[Cut11] M. Cuturi. Fast global alignment kernels. In *Proceedings of the International Conference on Machine Learning*, 2011. URL: https://dl.acm.org/citation.cfm?id=3104482.3104599.

[DC12] J. Dean and G. Corrado. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1223–1231, USA, 2012. Curran Associates Inc. URL: http://dl.acm.org/citation.cfm?id=2999134.2999271.

[DHS11] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(7):2121–2159, 2011.

[DKT17] D. Dheeru and E. Karra Taniskidou. UCI machine learning repository, 2017. URL: http://archive.ics.uci.edu/ml/datasets/PEMS-SF.

[Faw06] T. Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861 – 874, 2006. doi:10.1016/j.patrec.2005.10.010.

[FHB03] M. Fazel, H. Hindi, and S. Boyd. Log-det heuristic for matrix rank minimization with applications to Hankel and Euclidean distance matrices. *Proc. Am. Control Conf*, 3:2156–2162, 2003. doi:10.1109/ACC.2003.1243393.

[FS12] M. Friedlander and M. Schmidt. Hybrid deterministic-stochastic methods for data fitting. *SIAM J. Scientific Computing*, 34(3):A1380–A1405, 2012. doi:10.1137/110830629.

[GG16] C. Grussler and P. Giselsson. Low-Rank Inducing Norms with Optimality Interpretations. *ArXiv e-prints*, 2016. arXiv:1612.03186.

[GRGP01] K. Goldberg, T. Roeder, D. Gupta, and C. Perkins. Eigentaste: A constant time collaborative filtering algorithm. *Inf. Retr.*, 4(2):133–151, 2001. doi:10.1023/A:1011419012209.

[GV96] G. Golub and C. Van Loan. *Matrix computations*. The Johns Hopkins University Press, 1996.

[HK15] F. Harper and J. Konstan. The MovieLens datasets: history and context. *ACM Trans. Interact. Intell. Syst.*, 5(4):19:1–19:19, 2015. doi:10.1145/2827872.

[HM94] U. Helmke and J. Moore. *Optimization and Dynamical Systems*. Springer-Verlag London, 1994. doi:10.1007/978-1-4471-3467-1.

[JMC+16] B. Jiang, S. Ma, J. Causey, L. Qiao, M. Hardin, I. Bitts, D. Johnson, S. Zhang, and X. Huang. SparRec: An effective matrix completion framework of missing data imputation for GWAS. *Scientific Reports*, 6, 2016. doi:10.1038/srep35534.

[JU18] M. Jaggi and R. Urbanke. CS433 Machine Learning (EPFL), 2017 – 2018. Available at http://edu.epfl.ch/coursebook/en/machine-learning-CS-433.

[KM16] H. Kasai and B. Mishra. Low-rank tensor completion: a Riemannian manifold preconditioning approach. *ArXiv e-prints*, 2016. arXiv:1605.08257.

[KMO09] R. Keshavan, A. Montanari, and S. Oh. Low-rank matrix completion with noisy observations: A quantitative comparison. In *2009 47th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 1216–1222, 2009. doi:10.1109/ALLERTON.2009.5394534.

[Lee03] J. Lee. *Introduction to smooth manifolds*, volume 218. Springer Gradate Texts in Mathematics, 2003. doi:10.1007/978-0-387-21752-9.

[LLL14] Y. Li, Z. Li, and L. Li. Missing traffic data: comparison of imputation methods. *IET Intelligent Transport Systems*, 8(1):51–57, 2014. doi:10.1049/iet-its.2013.0052.

[MAS12] B. Mishra, K. Apuroop, and R. Sepulchre. A Riemannian geometry for low-rank matrix completion. *ArXiv e-prints*, 2012. arXiv:1211.1550.

[MHA18] E. Massart, J. Hendrickx, and P.-A. Absil. Matrix geometric means based on shuffled inductive sequences. *Linear Algebra and its Applications*, 542:334 – 359, 2018. Proceedings of the 20th ILAS Conference, Leuven, Belgium 2016. doi:10.1016/j.laa.2017.05.036.

[Mis14] B. Mishra. *A Riemannian approach to large-scale constrained least-squares with symmetries*. PhD thesis, Université de Liège, 2014. URL: http://hdl.handle.net/2268/173257.

[MMBS14] B. Mishra, G. Meyer, S. Bonnabel, and R. Sepulchre. Fixed-rank matrix factorizations and Riemannian low-rank optimization. *Computational Statistics*, 29(3):591–621, 2014. doi:10.1007/s00180-013-0464-z.

[MS16] B. Mishra and R. Sepulchre. Scaled stochastic gradient descent for low-rank matrix completion. In *2016 IEEE 55th Conference on Decision and Control (CDC)*, pages 2820–2825, 2016. doi:10.1109/CDC.2016.7798689.

[Nes03] Y. Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer Science & Business Media, 2003. doi:10.1007/978-1-4419-8853-9.

[NJM18] M. Nimishakavi, P. Jawanpuria, and B. Mishra. A Riemannian approach to trace norm regularized low-rank tensor completion. *ArXiv e-prints*, 2018. arXiv:1712.01193.

[NSW16] D. Needell, N. Srebro, and R. Ward. Stochastic gradient descent, weighted sampling, and the randomized Kaczmarz algorithm. *Mathematical Programming*, 155(1):549–573, 2016. doi:10.1007/s10107-015-0864-7.

[OAD18] G. Olikier, P.-A. Absil, and L. De Lathauwer. Variable projection applied to block term decomposition of higher-order tensors. *Proceedings of the 14th International Conference on Latent Variable Analysis and Signal Separation*, 2018. accepted for publication. URL: https://sites.uclouvain.be/absil/2018.01.

[PJY+13] T. Paine, H. Jin, J. Yang, Z. Lin, and T. Huang. GPU asynchronous stochastic gradient descent to speed up neural network training. *ArXiv e-prints*, 2013. arXiv:1312.6186.

[RR13] B. Recht and C. Ré. Parallel stochastic gradient algorithms for large-scale matrix completion. *Mathematical Programming Computation*, 5(2):201–226, 2013. doi:10.1007/s12532-013-0053-8.

[SGCH18] Q. Song, H. Ge, J. Caverlee, and X. Hu. Tensor Completion Algorithms in Big Data Analytics. *ArXiv e-prints*, 2018. arXiv:1711.10105.

[Sto71] H. Stone. Parallel processing with the perfect shuffle. *IEEE Transactions on Computers*, C-20(2):153–161, 1971. doi:10.1109/T-C.1971.223205.

[TFBJ18] N. Tripuraneni, N. Flammarion, F. Bach, and M. Jordan. Averaging Stochastic Gradient Descent on Riemannian Manifolds. *ArXiv e-prints*, 2018. arXiv:1802.09128.

[TWH00] R. Tibshirani, G. Walther, and T. Hastie. Estimating the number of clusters in a data set via the gap statistic. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 63(2):411–423, 2000. doi:10.1111/1467-9868.00293.

[TWYS13] K. Thung, C. Wee, P. Yap, and D. Shen. Identification of Alzheimer's disease using incomplete multimodal dataset via matrix shrinkage and completion. In *Machine Learning in Medical Imaging*, pages 163–170, Cham, 2013. Springer International Publishing. doi:10.1007/978-3-319-02267-3_21.

[Van13] B. Vandereycken. Low-rank matrix completion by Riemannian optimization. *SIAM Journal on Optimization*, 23(2):1214–1236, 2013. doi:10.1137/110845768.

[VBK18] M. Veale, R. Binns, and M. Van Kleek. Some HCI Priorities for GDPR-Compliant Machine Learning. *ArXiv e-prints*, 2018. arXiv:1803.06174.

[Wei13] M. Weiner. The Alzheimer's Disease Neuroimaging Initiative: A review of papers published since its inception. *Alzheimer's & Dementia*, 9(5):111–194, 2013. doi:10.1016/j.jalz.2013.05.1769.

[WN99] S. Wright and J. Nocedal. *Numerical optimization*, volume 2. Springer New York, 1999. doi:10.1007/978-0-387-40065-5.

[YCY+17] W. Ye, L. Chen, G. Yang, H. Dai, and F. Xiao. Anomaly-tolerant traffic matrix estimation via prior information guided matrix completion. *IEEE Access*, 5:3172–3182, 2017. doi:10.1109/ACCESS.2017.2671860.

[ZE12] D. Zastrau and S. Edelkamp. Stochastic gradient descent with GPGPU. In *KI 2012: Advances in Artificial Intelligence*, pages 193–204, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-33347-7_17.

[ZJG17] A. Zare, M. Jovanović, and T. Georgiou. Colour of turbulence. *Journal of Fluid Mechanics*, 812:636–680, 2017. doi:10.1017/jfm.2016.682.

[ZWSP08] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. In *Algorithmic Aspects in Information and Management*, pages 337–348. Springer Berlin Heidelberg, 2008. doi:10.1007/978-3-540-68880-8_32.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**