The Matrix Completion Problem

Gianmarco Nalin
Registration Number: 591875
Supervisor: Professor Michele Pavon
University of Padua
Faculty of Engineering
Bachelor Degree in Information Engineering

July 21st, 2011

Contents

1 Introduction **5**

2 Useful Mathematical Concepts on Matrices **7**
2.1 Notation and Definitions 7
2.2 Singular Value Decomposition 8
2.2.1 Example on SVD of a matrix 10
2.3 Frobenius Norm 12
2.3.1 Example on Frobenius norm 12

3 The Matrix Completion Problem **13**
3.1 Some Examples about MCP 13
3.2 Motivation 14
3.2.1 The Netflix Problem 14
3.2.2 Triangulation from incomplete data 14
3.3 Basics 15
3.3.1 The kind of matrices 15
3.3.2 The Sampling Set 16
3.3.3 The Optimization Problem 16

4 Mazin Results and an Algorithm **19**
4.1 Main Results 19
4.2 The Algorithm 20
4.3 Open Issues 22

5 Numerical Experiments for MCP **23**
5.1 Some preliminar hypotheses 23
5.2 Main results 24

A A MATLAB approach to MCP **27**
A.1 The Code 27

B Useful MATLAB Functions **33**
B.1 The Code 33

Chapter 1

Introduction

The aim of this thesis is giving a brief introduction to the *matrix completion problem*(MCP).

After recalling some useful mathematical concepts and results such as the *Singular Value Decomposition* (SVD) and the *Frobenius Norm* (FN), we give an intuitive idea about what is meant with the *matrix completion problem*.

As the reader well knows, nowadays the relevant amount of information exchanges force the *experts* to create more reliable, secure and error-less devices through which these can happen.

A possible solution to this problem is to create an algorithm for reconstructing corrupted data based on the knowledge of the devices' physics and the data's structures to exchange. Within this context, the thesis will give a simple introduction to the algorithm proposed by *Montanari*, *Keshevan* and *Oh* in their article [1]. This field of science is an open stage where the story has not been fully written yet. Being an open field, the algorithm proposed works fine with a very small subset of all the possibilities; in fact this works with systems we can model with a *low-rank* matrix.

As we will see in the Chapters 3 and 4 the correct recovering of the entire structure should be done under some additional hypothesis which restrict the class of possibilities in which we can apply this method.

This algorithm will be detailed in Chapter 5 to understand how works and which kind of data it can restore. The algorithm is based on the concepts of *SVD* and *FN* of which we give a definition in the following chapter.

Chapter 2

Useful Mathematical Concepts on Matrices

In this chapter, we state the basic concepts the reader should have to fully understand the *Matrix Completion* strategies. We start with the definition of *singular value* following with *singular value decomposition* of a matrix. After introducing the SVD we give the definition of the *Frobenius norm*.

2.1 Notation and Definitions

In this section, we give the basic notation that will be in used in the entire thesis.

A^T Transpose matrix: Matrix whose rows and columns are the A's columns and rows, respectively.

A^* Conjugate matrix: Matrix whose elements are the complex conjugate of A's elements.

A^H Adjoint matrix: The transpose conjugate of A.

Other important definitions are those of *symmetric* matrix and *Hermitian* matrix.

Definition 2.1 (Symmetric Matrix)**.** A matrix A is said to be *symmetric* if

$$A = A^T \tag{2.1}$$

Example 2.1.1. The matrix A is symmetric

$$A = \begin{bmatrix} 1 & 5 & 3 \\ 5 & 7 & 2 \\ 3 & 2 & 4 \end{bmatrix} = A^T$$

Definition 2.2 (Hermitian Matrix)**.** A matrix H is said to be *Hermitian* if

$$H = \overline{H}^T \tag{2.2}$$

Example 2.1.2. The matrix H is hermitian

$$H = \begin{bmatrix} 2 & -i & -1 \\ i & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} = \overline{H}^T$$

Proposition 2.1.1. *Let H be Hermitian. Then all of its eigenvalues are real. Moreover, there exists a unitary matrix U such that*

$$UHU^H = D \tag{2.3}$$

with D diagonal.

Other important definitions are those of *positive definite* matrix and *positive semidefinite* matrix.

Definition 2.3 (Positive Definite Matrix)**.** An Hermitian matrix P is said to be *positive definite* if

$$x^H P x > 0 \qquad \forall x \neq 0 \in \mathbb{C}^n \tag{2.4}$$

Definition 2.4 (Positive Semidefinite)**.** An Hermitian matrix P is said to be *positive semidefinite* if

$$x^H P x \geq 0 \qquad \forall x \in \mathbb{C}^n \tag{2.5}$$

2.2 Singular Value Decomposition

Before giving the definition of *singular value* and expressing the *singular value decomposition* of a matrix, we formulate the following proposition,

Proposition 2.2.1. *Given two matrices $A \in \mathcal{M}_{m,n}$ and $B \in \mathcal{M}_{n,m}$, the matrices AB and BA have the same non-zero eigenvalues with the same geometric multiplicities.*

We now replace B with A^H in AB. By 2.2.1, the matrices AA^H and A^HA have the same rank of A. If A is in $\mathcal{M}_{m,n}$ and it have rank r, $\lambda = 0$ is eigenvalue of AA^H if and only if $m > r$, and in such case its geometric multiplicity is $m - k$. Likewise, $\lambda = 0$ is eigenvalue of A^HA if and only if $n > r$, and in such case its geometric multiplicity is $n - k$. From this point of view, $\lambda = 0$ could be eigenvalue of A^HA but not of AA^H and vice versa. This cannot happen by the following propositions.

Proposition 2.2.2. *Let $A \in \mathcal{M}_{m,n}$. The following statements are equivalent:*

1. *the matrices $A^H A$ and AA^H are* hermitian*, so they are diagonalizable by unitary matrices;*

2. *the eigenvalues of AA^H are nonnegative real numbers; the positive ones are equal to the eigenvalues of $A^H A$.*

In order to achieve the singular value decomposition of a matrix A, we must study the eigensystem of AA^H, i.e. we have to find the eigenvalues and the eigenvectors of AA^H.

From this study, we derive the following definition

Definition 2.5 (Singular Value)**.** We define *singular values* of A as the square root of the non-zero eigenvalues of AA^H

$$\sigma_i(A) := \sqrt{\lambda_i(AA^H)} = \sqrt{\lambda_i(A^H A)}. \tag{2.6}$$

Remark 1. From proposition 2.2.2, we have that

$$\sigma_1(A) \geqslant \sigma_2(A) \geqslant \cdots \geqslant \sigma_r(A) > \sigma_{r+1}(A) = \cdots = \sigma_{\min(m,n)}(A) = 0$$

where $r = rank(A)$.

Now, we give the definition of SVD for an arbitrary $n \times m$ matrix.

Definition 2.6 (Singular Value Decomposition)**.** For each $A \in \mathcal{M}_{m,n}(\mathbb{C})$ with rank r is always possible to find two unitary matrices $U \in \mathcal{M}_m, V \in \mathcal{M}_n$ and a diagonal matrix $D = diag(\sigma_1, \sigma_2, \ldots, \sigma_r)$ such that

$$A = U \begin{bmatrix} D & 0 \\ 0 & 0 \end{bmatrix} V^H \tag{2.7}$$

with $\sigma_1 \geqslant \sigma_2 \geqslant \cdots \geqslant \sigma_r$.

$\sigma_1, \sigma_2, \ldots, \sigma_r$ are called *singular values* of A. The factorizazion in 2.7 is called *singular value decomposition* of A and the columns of U and V are called *left* and *right singular vectors* for A, respectively.

We now proceed with the proof of the existence of the singular value decomposition of a matrix A; this proof is constructive because it shows how to find the matrices U, V and D.

Theorem 2.2.3. *Every complex matrix A with rank r admits a singular value decomposition like 2.7 and such a factorization is unique.*

Proof. **First Step.** From Proposition 2.2.2, AA^H is diagonalizable by a unitary matrix U so that $AA^H = U\Lambda U^H$, where $U = [u_1, u_2, \ldots, u_n]$ and $\Lambda = diag(\lambda_1, \ldots, \lambda_r, 0, \ldots, 0)$. The columns of U satisfy the following properties:

$$\begin{cases} AA^H \cdot u_i = \lambda_i u_i & i \leqslant r \\ AA^H \cdot u_i = 0 & i > r \end{cases}.$$

Let $\sigma_i = \sqrt{\lambda_i}$. From Proposition 2.2.2, if we set

$$v_i = \sigma_i^{-1} A^H u_i, \qquad i \leqslant r$$

the vectors $v_1, v_2, \ldots, v_r$ make up a orthonormal base of $A^H A$.

Second Step. Now, we have to complete the base of $A^H A$ to a $\mathbb{C}^n$ base. To achive this, we consider the *nullspace* of $A^H A$, i.e. the eigenspace of $\lambda = 0$. If $[v_{r+1}, \ldots, v_n]$ denotes a orthonormal base of $A^H A$ nullspace, we can choose the vectors $\{v_1, \ldots v_r, v_{r+1}, \ldots, v_n\}$ as a $\mathbb{C}^n$ base and we set $V = [v_1 \ldots v_n]$.

□

2.2.1 Example on SVD of a matrix

Let A be

$$A = \begin{bmatrix} 3 & 1 & 1 \\ -1 & 3 & 1 \end{bmatrix}.$$

Now, we follow the steps in Theorem 2.2.3 to decompose the matrix A in its SVD.

As we have seen in Definition 2.6, the matrices U, V, Σ have dimensions 2×2, 3×3 and 2×3, respectively.

Step Zero In order to find the singular values of A, we compute AA^H:

$$AA^H = \begin{bmatrix} 3 & 1 & 1 \\ -1 & 3 & 1 \end{bmatrix} \begin{bmatrix} 3 & -1 \\ 1 & 3 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 11 & 1 \\ 1 & 11 \end{bmatrix}$$

Now we find the eigenvalues of AA^H

$$\det(\lambda I - AA^H) = \det \begin{bmatrix} \lambda - 11 & -1 \\ -1 & \lambda - 11 \end{bmatrix} =$$

$$= (\lambda - 11)^2 - 1 =$$

$$= \lambda^2 - 22\lambda + 120 = (\lambda - 12)(\lambda - 10)$$

so

$$\lambda_1 = 12 \qquad \lambda_2 = 10$$

As we have defined before, the singular values of A are the square roots of non-zero eigenvalues of AA^H, so

$$\sigma_1 = \sqrt{12} \qquad \sigma_2 = \sqrt{10}$$

Once we have found the singular values of A, we have that the matrix Σ is uniquely identified and it is equal to

$$\Sigma = \begin{bmatrix} \sqrt{12} & 0 & 0 \\ 0 & \sqrt{10} & 0 \end{bmatrix}$$

First Step The left singular vectors are the columns of the matrix which diagonalize the matrix AA^H. With the usually diagonalization method (see [5] for more details about this topic) we find that

$$U = \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix}$$

Now we can find the first two columns of the matrix V by the formula

$$v_i = \sigma_i^{-1} A^H u_i$$

so

$$v_1 = \sigma_1^{-1} A^H u_1 = \begin{bmatrix} \frac{\sqrt{6}}{6} \\ \frac{\sqrt{6}}{3} \\ \frac{\sqrt{6}}{6} \end{bmatrix}$$

$$v_2 = \sigma_2^{-1} A^H u_2 = \begin{bmatrix} -\frac{2\sqrt{5}}{5} \\ \frac{\sqrt{5}}{5} \\ 0 \end{bmatrix}$$

Second Step in order to complete the $\{v_1, v_2\}$ orthonormal base to a $\mathbb{R}^3$ orthonormal base, we have to find the nullspace of $A^H A$.

With the usually methods used to find an eigenspace, we find the third vector that complete the $\mathbb{R}^3$ base that is

$$v_3 = \begin{bmatrix} -\frac{1}{6}\sqrt{\frac{6}{5}} \\ -\frac{1}{3}\sqrt{\frac{6}{5}} \\ \frac{5}{6}\sqrt{\frac{6}{5}} \end{bmatrix}$$

After completed these steps, we have find the singular value decomposition of A that is

$$U = \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix} \qquad \Sigma = \begin{bmatrix} \sqrt{12} & 0 & 0 \\ 0 & \sqrt{10} & 0 \end{bmatrix} \qquad V = \begin{bmatrix} \frac{\sqrt{6}}{6} & -\frac{2\sqrt{5}}{5} & -\frac{1}{6}\sqrt{\frac{6}{5}} \\ \frac{\sqrt{6}}{3} & \frac{\sqrt{5}}{5} & -\frac{1}{3}\sqrt{\frac{6}{5}} \\ \frac{\sqrt{6}}{6} & 0 & \frac{5}{6}\sqrt{\frac{6}{5}} \end{bmatrix}$$

The reader can simply check that $A = U\Sigma V^H$.

2.3 Frobenius Norm

Definition 2.7 (Frobenius Norm)**.** The *Frobenius norm* of a matrix A is defined as

$$\|A\|_F = \sqrt{\text{trace}(AA^H)} \tag{2.8}$$

If we have a singular value decomposition of a matrix A we can simply calculate the *Frobenius norm* of a matrix by the following proposition

Proposition 2.3.1. *If A is a complex matrix with rank r and $A = U\Sigma V^H$ is the singular value decomposition of A then*

$$\|A\|_F = \sqrt{\sum_{i=1}^{r} \sigma_i^2} \tag{2.9}$$

where σ_i's are the singular value of A

2.3.1 Example on Frobenius norm

With respect to the example in 2.2.1, we calculate the *Frobenius Norm* of the matrix A by the Definition 2.7 and by the Proposition 2.3.1 and we will show that the results are the equivalent.

Following the Definition 2.7, the Frobenius norm of A is

$$\|A\|_F = \sqrt{\text{trace}(AA^H)} = \sqrt{\text{trace}\begin{bmatrix} 11 & 1 \\ 1 & 11 \end{bmatrix}} = \sqrt{11+11} = \sqrt{22}$$

Instead, following the Proposition 2.3.1, the Frobenius norm of A is

$$\|A\|_F = \sqrt{\sum_{i=1}^{2} \sigma_i^2} = \sqrt{\sigma_1^2 + \sigma_2^2} = \sqrt{12+10} = \sqrt{22}$$

Chapter 3

The Matrix Completion Problem

In this chapter, we give some important results about the *matrix completion problem* (MCP).

3.1 Some Examples about MCP

In daily life situations we are interested in lost data recover. For instance, if we lost an important document, we use tools for automatic data restore. Sometimes we cannot backup data, e.g. data communications, and we simply loose our information. In other situations, the information leakage is intentional, e.g. someone left a question unanswered. At this point, the reader could think *what is the relation between the MCP and the above examples?* In the next section, we will try to explain this connection giving a model for such situations.

3.2 Motivation

3.2.1 The Netflix Problem

Imagine to have a movie rental site where users can rate movies they watched. The rate consists in a fixed number of stars (e.g. from one star to five stars). Hence, we can model this system with a *matrix* whose rows and columns represent users and movies, respectively, while the generic element r_{ij} represents the rating for the j^{th} movie given by the i^{th} user. Since users can only rate a few moves, we are interested in inferring their preferences for unrated movies. Starting with this situation, can we hope to make a guess on the missing rating? Under which conditions is it possible?

3.2.2 Triangulation from incomplete data

Another important application regards distances between objects. Suppose we obtain partial information from sensors scattered randomly across a region. We would like to reconstruct the low-dimensional geometry describing their location. Suppose each sensor can construct distance estimates based on information retrieved from its nearest sensors. From these estimates, we can form a partially observed distance matrix. Now we can estimate the true distance matrix whose rank is two if these sensors are placed in a plane or three if they are located in space.

3.3 Basics

In general, one can say that it is impossible without other additional information. In practical cases, however, these matrices have a *well-definite structure*, that is they are *low-rank* or *approximately low-rank*. In the previous examples we can suppose to model these systems with low-rank matrices because, in the former, it is commonly believed that the users base their rating only on a few factors while, in the latter, the matrix rank can assume only two value and in general they can be smaller than the number of sensors.

In the following sections, we focus our attention on which additional hypothesis are necessary to infer the entire matrix from a subset of its entries.

3.3.1 The kind of matrices

For simplicity, we want to recover a square $n \times n$ matrix with rank r. As it is apparent, this matrix M can be represented by n^2 elements but it only has $(2n - r)r$ degrees of freedom.This fact can be revealed by counting parameters in the SVD of M, i.e. the number of degrees of freedom associated with the description of its singular values and left and right singular vectors. As seen before, one cannot recover a matrix form a subset of its entries. Consider the following example with a 1-rank matrix

$$M = \mathbf{e}_1\mathbf{e}_n^H = \begin{bmatrix} 1 \\ 0 \\ \cdots \\ 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & \dots & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & \cdots & 0 & 1 \\ 0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & 0 \end{bmatrix} \tag{3.1}$$

As the reader can see, we cannot infer the entire matrix without seeing all of its entries. Indeed, many subsets of its entries contain only zeros so one can guess that M is the all-zero matrix.

From this example, we could think it is impossible to infer all low-rank matrices from a sampling of their entries. Thus, a more reasonable question is: *can one recover most of them?* We can answer this question introducing the *SVD* of a matrix M

$$M = USV^H = \sum_{k=1}^{r} \sigma_k \mathbf{u}_k \mathbf{v}_k^H \tag{3.2}$$

where, as we have seen in chapter 2, $\mathbf{u}_k$'s, $\mathbf{v}_k$'s and σ_k's are the left and right singular vectors and singular values, respectively. With the *SVD* of matrix M, we can restate the definition of a *low-rank* matrix as follow: the $\mathbf{u}_k$ family is selected uniformly random among all families of r orthonormal vectors, and similarly for the $\mathbf{v}_k$ family. We do not make any assumption on the singular values, so the $\mathbf{u}_k$ and $\mathbf{v}_k$ may or may not be independent of each other.

At this point, the main question is whether or not one can recover such a matrix from a subset of its entries.

3.3.2 The Sampling Set

As we have seen in the above example, there is no hope to recover a matrix M if the sampling set leaves out a column or a row of it. For the sake of simplicity, suppose that M is a 1-rank matrix obtained by the product of $x^T = \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix}^T$ and $y = \begin{bmatrix} y_1 & y_2 & \dots & y_m \end{bmatrix}$. Then, the generic element M_{ij} is given by

$$M_{ij} = x_i y_j \ ,$$

and the matrix M is of the form

$$\begin{bmatrix} x_1y_1 & x_1y_2 & \cdots & x_1y_{m-1} & x_1y_m \\ x_2y_1 & x_2y_2 & \cdots & x_2y_{m-1} & x_2y_m \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_ny_1 & x_ny_2 & \cdots & x_ny_{m-1} & x_ny_m \end{bmatrix} \tag{3.3}$$

It is easy to see that if we do not have samples from the first row, there is no hope to guess the first component x_1. This argument can be extended to any row or column.

This simple example suggests us that if we want to recover the matrix M the sampling set must contain at least one sample per row and one sample per column.

What happens for most sampling set? Can one recover a matrix from almost sampling set of cardinality m? Let Ω be the set of known entries (i.e. $(i, j) \in \Omega$ if M_{ij} is observed) of cardinality m whose elements are picked uniformly at random. Then, can one recover a generic *low-rank* matrix M, with very large probability, from the knowledge of the value of its entries in the set Ω?

3.3.3 The Optimization Problem

If the number of observed entries is sufficiently large and if these ones are sufficiently uniformly distributed, one can hope that there is only one *low-rank* matrix with these entries. If the above hypothesis are satisfied, the matrix completion problem reduces to the following optimization problem

$$\begin{aligned} & \text{minimize} && \operatorname{rank}(X) \\ \text{subject to} \quad & X_{ij} = M_{ij} && (i,j) \in \Omega \end{aligned} \tag{3.4}$$

The problem formulate in (3.4) is a common sense approach that try to find the simplest solution fitting the observed data. This approach is unfortunately of little practical use because these algorithms provide the right solutions in time doubly exponential in the dimension n of the matrix.

If the matrix has rank r, then it has exactly r non-zero singular values so the rank in (3.4) is simply the number of non-vanishing singular values. As in [3], we introduce the *nuclear norm* of X as the sum of X's singular values

$$\|X\|_* = \sum_{k=1}^{r} \sigma_k(X) \tag{3.5}$$

where $\sigma_k(X)$ is the k^{th} singular value of X. With the definition of nuclear norm, we can restate a new optimization problem as

$$\begin{aligned} &\text{minimize} && \|X\|_* \\ &\text{subject to} && X_{ij} = M_{ij} \quad (i,j) \in \Omega \end{aligned} \tag{3.6}$$

In order to state the most important theorem about the *MCP*, we give the definition of the *coherence* of a matrix U in the following:

Definition 3.1. Let U be a subspace of $\mathbb{R}^n$ of dimension r and P_U be the orthogonal projection onto U. Then the *coherence* of U (with respect to standard basis (e_i)) is defined to be

$$\mu(U) = \frac{n}{r} \max_{1 \le i \le n} \|P_U e_i\|^2. \tag{3.7}$$

To state the main results we will give in the next chapter, we introduce two assumptions about the matrix M whose SVD is given by 2.7.

A0 The coherences obey $\max(\mu(U), \mu(V)) \le \mu_0$ for some positive μ_0.

A1 The $n_1 \times n_2$ matrix $\sum_{1 \le k \le r} u_k v_k^*$ has a maximum entry bounded by $\mu_1 \sqrt{r/(n_1 n_2)}$ in absolute value for some positive μ_1.

The μ's above may depend on r and n_1, n_2. Morover, the **A1** always holds with $\mu_1 = \mu_0 \sqrt{r}$ since the (i,j)th entry of the matrix $\sum_{1 \le k \le r} u_k v_k^*$ is given by $\sum_{1 \le k \le r} u_{ik} v_{jk}^*$ and by the Cauchy-Schwarz inequality,

$$\Big| \sum_{1 \le k \le r} u_{ik} v_{jk}^* \Big| \le \sqrt{\sum_{1 \le k \le r} |u_{ik}|^2} \sqrt{\sum_{1 \le k \le r} |v_{ik}|^2} \le \frac{\mu_0 r}{\sqrt{n_1 n_2}}.$$

In this position, we can say that (as we'll state in the next chapter) if a matrix has row and column incoherent with the standard basis, the nuclear norm minimizer in 3.6 can recover the entire matrix from a small sampling set of its entries.

Chapter 4

Mazin Results and an Algorithm

In this chapter, we state the theorems which constitute the foundations of the MCP and give the first main results about the algorithm.

4.1 Main Results

Theorem 4.1.1. *Let M be an $n_1 \times n_2$ matrix of rank r sampled from the random orthogonal model, and put $n = \max(n_1, n_2)$. Suppose we observe m entries of M with locations sampled uniformly at random. Then, there are numerical constants C and c such that if*

$$m \geq Cn^{5/4} r \log n, \tag{4.1}$$

the minimizer to the problem 3.6 is unique and equal to M with probability at least $1 - cn^{-3}$. In addition, if $r \leq n^{1/5}$, then the recovery is exact with probability at least $1 - cn^{-3}$ provided that

$$m \geq Cn^{6/5} r \log n. \tag{4.2}$$

This first theorem states that a very small number of entries are necessary in order to complete the entire low-rank matrix. For small values of the rank r, one only needs to see of the order of $n^{5/6}$ entries. This is considerably smaller than the n^2 entries of a square matrix. The content of this recovery algorithm is that it is tractable and very concrete. Hence the the contribution is twofold:

- Under the hypotheses of Theorem 4.1.1, there is a unique low-rank matrix that matches the observed entries;
- The matrix can be recovered by solving the optimization problem in 3.6.

Theorem 4.1.1 is in fact a special instance of a more general theorem that covers a larger set of matrices M. If the matrices U and V (U and V are the M's singular value decomposition matrices) satisfied the assumptions in 3.3.3, the following theorem holds:

Theorem 4.1.2. *Let M be an $n_1 \times n_2$ matrix of rank r obeying the assumptions in 3.3.3 and put $n = \max(n_1, n_2)$. Suppose we observe m entries of M with locations sampled uniformly at random. Then there exist constants C, c such that if*

$$m \geq C \max(\mu_1^2, \mu_0^{1/2}\mu_1, \mu_0 n^{1/4}) nr(\beta \log n) \tag{4.3}$$

for some $\beta > 2$, then the minimizer problem in 3.6 is unique and equal to M with probability at least $1 - cn^{-\beta}$. For $r \leq \mu_0^{-1} n^{1/5}$this estimate can be improved to

$$m \geq C\mu_0 n^{6/5} r(\beta \log n) \tag{4.4}$$

with the same probability of success.

Theorem 4.1.2 suggests that if the coherence is low (i.e. the matrices U and V are incoherent), few samples are necessary to recover the matrix M. In order to better explain the meaning of Theorem 4.1.2, we proceed with an example. As we have seen in our first example 3.1, we have no hope to recover a matrix that is equal to zero in nearly all of its entries unless we observe all the entries. To recover a low-rank matrix, this one cannot be in the null space of the sampling operator giving the values of a subset of the entries. If we consider the following 2-rank matrix

$$M = \sum_{k=1}^{2} \sigma_k u_k u_k^* \qquad \text{with} \qquad u_1 = \frac{e_1 + e_2}{\sqrt{2}} \qquad u_2 = \frac{e_1 - e_2}{\sqrt{2}}. \tag{4.5}$$

The matrix vanishes everywhere except in the top-left corner and, as the reader should have understood, one would need to see all the entries for perfect recovery. Hence, we can say that if the singular vectors are somehow sufficiently spread (i.e. uncorrelated with the standard basis), we can minimize the number of observations needed to recover the matrix. Given the two above theorems, we can explain the algorithm proposed by *Keshavan*, *Montanari*, *Oh* in their article [1].

4.2 The Algorithm

The algorithm proposed by *Keshavan*, *Montanari* and *Oh* is based on the theorems in Section 4.1. For more details about the algorithm we refer to the Appendix A where the MATLAB implementation is provided.

The algorithm expects that the unknown entries of the matrix M are set to zero. In other words, defining as Ω the set of pairs (i, j) such that

$$(i, j) \in \Omega \qquad \text{if } M_{ij} \text{ is revealed,} \tag{4.6}$$

we construct a new matrix M^Ω in this way

$$M^\Omega_{ij} = \begin{cases} M_{ij} & \text{if } (i, j) \in \Omega \\ 0 & \text{otherwise} \end{cases}. \tag{4.7}$$

We recall that n and m are the matrix dimensions. In the following the real case is considered.

Now, we are ready to use the algorithm. This one consists of only three steps:

1. Projecting
2. Trimming
3. Cleaning

Projecting Compute the *SVD* of M^Ω (with $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$)

$$M^\Omega = \sum_{i=1}^{\min(m,n)} \sigma_i u_i v_i^T \tag{4.8}$$

and return the matrix

$$\mathcal{T}_r(M^\Omega) = \frac{mn}{|\Omega|} \sum_{i=1}^{r} \sigma_i u_i v_i^T \tag{4.9}$$

obtained by setting to 0 all but the r largest singular values. Notice that, omitting the factor $\frac{mn}{|\Omega|}$, $\mathcal{T}_r(M^\Omega)$ is the projection of M^Ω onto the set of rank-r matrices.

Trimming

- Set to zero all columns in M^Ω with degree[1] larger than $\frac{2|\Omega|}{n}$
- Set to zero all rows in M^Ω with degree larger than $\frac{2|\Omega|}{m}$

From this step we obtain a new matrix which we denote by $\widetilde{M}^\Omega$.

Cleaning Given $U \in \mathbb{R}^{m\times r}$, $V \in \mathbb{R}^{n\times r}$ with $UU^T = I_m$, $VV^T = I_n$, we define

$$F(U, V) \equiv \min_{S\in\mathbb{R}^{r\times r}} \mathcal{F}(U, V, S), \tag{4.10}$$

$$\mathcal{F}(U, V, S) \equiv \frac{1}{2} \sum_{(i,j)\in\Omega} (M_{ij} - (USV^T)_{ij})^2. \tag{4.11}$$

The cleaning step consists in writing $\mathcal{T}_r(\widetilde{M}^\Omega) = U_0 S_0 V_0^T$ and minimizing $F(U, V)$ locally with initial condition $U = U_0, V = V_0$

[1] In this context we define as the *degree* of a row, the number of its non-zero entries.

In order to evaluate the goodness of this algorithm it is necessary to introduce a new index which is called *relative root mean square error* and it is defined as

$$RMSE = \left[\frac{1}{mnr}\|M - \mathcal{T}_r(\widetilde{M}^{\Omega})\|_F^2\right]^{1/2} \tag{4.12}$$

where $\|A\|_F$ is the *Frobenius norm*which was recalled in Chapter 2. With the definition 4.12, we can introduce another important result about the algorithm just presented. This theorem concernes the maximum value that RMSE can reach, under the assumptions in 3.3.3.

Proposition 4.2.1. *Assume M to be a rank $r \leq n^{1/2}$ matrix that satisfies the incoherence assumptions in 3.3.3. Then, with very high probability*

$$RMSE^2 \leq C(\alpha)\frac{nr}{|\Omega|} \tag{4.13}$$

for some $C(\alpha)$.

4.3 Open Issues

The above results are really great but they have some limitations. For instance, as we have said before, this algorithm works with a very small set of system models, i.e. the *low-rank* matrices. It could be of interest to be able to extend these results to unknown matrices which are *approximately low-rank*. Suppose we write the SVD of a matrix M as

$$M = \sum_{1\leq k\leq n} \sigma_k u_k v_k^H$$

where $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$ and assume, for the sake of simplicity, that $\sigma_i \neq 0, \quad i = 1, \ldots, n$. It is clear that, in general, it is impossible to recover the entire matrix from a partial subset of its entries. However, one can hope to complete a good *approximation* of it if, for example, most of the singular values are small or negligible. Consider, for instance, the truncated SVD of M,

$$M_r = \sum_{1\leq k\leq r} \sigma_k u_k v_k^H$$

where the sum is extended to the r largest singular values of M and let M_* be the solution to 3.6. In what follows, $\|\cdot\|$ denotes any matrix norm. It is obvious that it is impossible that $M = M_*$ but if $\|M - M_r\|$ is comparable with $\|M_* - M\|$ then we can say that approximately low-rank matrices can be recover from a small set of sampled entries.

Chapter 5

Numerical Experiments for MCP

In this chapter we test the algorithm implementation proposed by *Montanari*, *Oh* and *Keshavan* in their article [1].

5.1 Some preliminar hypotheses

In order to do a statistically significant test, we have given as input to the algorithm different kinds of matrices with different quantities of revealed entries. To achieve this, we have created *ad-hoc* functions that we report in Appendix B.

In these experiments, we used square matrices of different dimensions and we tried to vary the number of revealed entries with a given fixed rank. To create a matrix with a given rank, we have generated its singular value decomposition, i.e. we have created the matrices U, V (the matrices of left and write singular vectors, respectively) and Σ (the matrix of singular values). In particular, fixing the rank r is equivalent to fixing the dimension of the matrix Σ which is diagonal with all elements grater than zero. For instance if we want to create a random square matrix A with rank $r = 3$ and dimension $n = 100$, we have to choose the above matrices in this way

- $U \in \mathcal{M}_{100,3}$
- $V \in \mathcal{M}_{100,3}$
- $\Sigma = \begin{bmatrix} x & 0 & 0 \\ 0 & y & 0 \\ 0 & 0 & z \end{bmatrix} \quad x, y, z \geq 0$

and then $A = U\Sigma V^* \in \mathcal{M}_{100,100}$ (the notation $\mathcal{M}_{n,m}$ is declare in Section 2.1). After creating the matrix, we picked uniformly at random a fixed

number of entries. When this was done, we tested the algorithm on these matrices and we repeated this procedure 50 times for every single matrix.

5.2 Main results

To manage the informations of the algorithm execution, we introduced an index which shows how much the recovered matrix is *close to* the real one. This index is defined as

$$\mu = \frac{\|\hat{M} - M\|_F}{\|M\|_F} \tag{5.1}$$

where M the real matrix and $\hat{M}$ is the recovered matrix. For every iteration, we store the related μ and we report the mean and the variance of the set of μ's calculated during the execution. We consider a matrix M_i as correctly recovered if $\mu_i < 10^{-3}$. Under this assumption, we can give to the reader the fraction of correctly recovered matrices.

In the following table, we report the results of the experiment with $\Sigma = I_r$ while we denote with m the number of revealed entries.

r = 3, n = 100			
m	**Mean**	**Variance**	**% Successes**
1000	8.78×10^{-1}	2.04×10^{-1}	0.0
2000	2.06×10^{-1}	1.06×10^{-1}	70.0
3000	1.88×10^{-2}	8.65×10^{-3}	96.0
4000	8.06×10^{-9}	2.78×10^{-18}	100.0
5000	6.80×10^{-9}	2.50×10^{-2}	100.0
6000	6.10×10^{-9}	2.30×10^{-18}	100.0

r = 5, n = 100			
m	**Mean**	**Variance**	**% Successes**
1000	1.51×10^{0}	1.05×10^{-1}	0.0
2000	5.91×10^{-1}	1.66×10^{-1}	24.0
3000	2.66×10^{-1}	1.32×10^{-1}	62.0
4000	6.54×10^{-2}	5.03×10^{-2}	92.0
5000	4.50×10^{-2}	3.30×10^{-2}	94.0
6000	5.60×10^{-9}	7.80×10^{-19}	100.0

r = 10, n = 100			
m	**Mean**	**Variance**	**% Successes**
1000	1.79×10^{0}	1.03×10^{-1}	0.0
2000	1.13×10^{0}	1.93×10^{-2}	0.0
3000	7.99×10^{-1}	5.91×10^{-2}	0.0
4000	5.98×10^{-1}	1.46×10^{-1}	12.0
5000	3.80×10^{-1}	1.90×10^{-1}	56.0
6000	1.60×10^{-1}	1.10×10^{-1}	80.0
7000	1.76×10^{-2}	1.54×10^{-2}	98.0
8000	3.65×10^{-9}	4.64×10^{-19}	100.0

With respect to the lower bound on the number of revealed entries in 4.1.1, we have that for $r = 3$, $m \geq C1900$: the experiment confirms this bound. The reader should not be surprised by the fact that, when the rank of the unknown matrix increases, the algorithm needs more entries in order to recover correctly the entire matrix. The tables show this fact: in order to achieve a success rate grater than 50%, the minimum number of entries are 2000, 3000 and 5000 for rank values of 3, 5 and 10, respectively.

The tests also confirm that when the number of revealed entries increase, the algorithm works better and it is able to recover the matrix more precisely. From the tests, it seems that the value of the constant C in 4.1.1 is affected by the value of the singular values. Indeed, as the reader can see in the following tables, the success rates decreases if the singular values of the matrix are not equal to 1.

The following tables are the results of a test where the singular value matrix is a *random diagonal matrix* with singular values such that $1 \leq \sigma_i \leq 10, \quad i = 1, \ldots, r$.

r = 3, n = 100			
m	**Mean**	**Variance**	**% Successes**
1000	6.57×10^{-1}	1.43×10^{-1}	0.0
2000	3.09×10^{-1}	5.55×10^{-2}	12.0
3000	2.46×10^{-1}	4.82×10^{-2}	26.0
4000	1.51×10^{-1}	2.49×10^{-2}	42.0
5000	7.24×10^{-2}	1.14×10^{-2}	60.0
6000	6.86×10^{-2}	1.10×10^{-2}	58.0
7000	4.10×10^{-2}	4.33×10^{-3}	68.0
8000	2.11×10^{-2}	1.74×10^{-3}	76.0
9000	4.24×10^{-3}	4.40×10^{-4}	96.0

r = 5, n = 100			
m	**Mean**	**Variance**	**% Successes**
1000	1.06×10^{0}	7.93×10^{-2}	0.0
2000	6.34×10^{-1}	4.52×10^{-2}	0.0
3000	4.04×10^{-1}	4.62×10^{-2}	0.0
4000	2.81×10^{-1}	2.87×10^{-2}	4.0
5000	2.15×10^{-1}	2.57×10^{-2}	10.0
6000	1.76×10^{-1}	3.27×10^{-2}	18.0
7000	8.27×10^{-2}	6.91×10^{-3}	40.0
8000	6.45×10^{-2}	3.52×10^{-3}	38.0
9000	3.38×10^{-2}	1.81×10^{-3}	54.0

r = 10, n = 100			
m	**Mean**	**Variance**	**% Successes**
1000	1.54×10^{0}	1.18×10^{-1}	0.0
2000	8.75×10^{-1}	1.32×10^{-2}	0.0
3000	6.44×10^{-1}	4.41×10^{-2}	0.0
4000	5.53×10^{-1}	5.39×10^{-2}	0.0
5000	6.02×10^{-1}	5.74×10^{-2}	0.0
6000	4.47×10^{-1}	7.49×10^{-2}	0.0
7000	3.85×10^{-1}	7.83×10^{-2}	2.0
8000	2.93×10^{-1}	8.04×10^{-2}	8.0
9000	1.20×10^{-1}	2.45×10^{-2}	14.0

In conclusion, the experiments confirm that in order to achieve a valid recover of a matrix this algorithm works fine when the rank is small, as said in 4.1.1 about the value of the rank.

Appendix A

A MATLAB approach to MCP

In the following sections, we give the MATLAB code implementation of the algorithm proposed by *Keshavan*, *Montanari* and *Oh* in [1] and discussed in Chapter 4.

A.1 The Code

```
function [X S Y dist] = OptSpace(M_E,r,niter,tol)
if(nargin==1)

        M_E = sparse(M_E);
        [n m] = size(M_E);
        E = spones(M_E);
        eps = nnz(E)/sqrt(m*n) ;

        tol = 1e-6;

        fprintf(1,'Rank not specified. Trying to guess ...\n');
        r = guessRank(M_E) ;
        fprintf(1,'Using Rank : %d\n',r);

        m0 = 10000 ;
        rho = 0;

        niter = 50;
elseif(nargin==4)

        M_E = sparse(M_E);
        [n m] = size(M_E);
        E = spones(M_E);
        eps = nnz(E)/sqrt(m*n) ;
```

```
        if( length(tol) == 0 )
                tol = 1e-6;
        end

        if( length(r) == 0 )

                fprintf(1,'Rank not specified. Trying to guess ...\n');
                r = guessRank(M_E) ;
                fprintf(1,'Using Rank : %d\n',r);
        end

        m0 = 10000 ;
        rho = 0;

        if( length(niter) == 0 )
                niter = 50 ;
        end
else
        fprintf(1,'Improper arguments (See "help OptSpace")\n');
        fprintf(1,'Usage :\n[X S Y dist] = OptSpace(A,r,niter,tol) \n') ;
        fprintf(1,'[X S Y dist] = OptSpace(A)\n');
        return;
end

rescal_param = sqrt( nnz(E) * r / norm(M_E,'fro')^2 ) ;
M_E = M_E * rescal_param ;

fprintf(1,'Trimming ...\n');
% Trimming

M_Et = M_E ;
d = sum(E);
d_=mean(full(d));
for col=1:m
    if ( sum(E(:,col))>2*d_ )
        list = find( E(:,col) > 0 );
        p = randperm(length(list));
        M_Et( list( p(ceil(2*d_):end) ) , col ) = 0;
    end
end

d = sum(E');
d_= mean(full(d));
for row=1:n
    if ( sum(E(row,:))>2*d_ )
        list = find( E(row,:) > 0 );
        p = randperm(length(list));
        M_Et(row,list( p(ceil(2*d_):end) ) ) = 0;
    end
```

```
end

fprintf(1,'Sparse SVD ...\n');
% Sparse SVD
[X0 S0 Y0] = svds(M_Et,r) ;

clear M_Et;

% Initial Guess
X0 = X0*sqrt(n) ; Y0 = Y0*sqrt(m) ;
S0 = S0 / eps ;


fprintf(1,'Iteration\tFit Error\n');

% Gradient Descent
X = X0;Y=Y0;
S = getoptS(X,Y,M_E,E);


dist(1) = norm( (M_E - X*S*Y').*E ,'fro')/sqrt(nnz(E) )  ;
fprintf(1,'0\t\t%e\n',dist(1) ) ;

for i = 1:niter

% Compute the Gradient
        [W Z] = gradF_t(X,Y,S,M_E,E,m0,rho);

% Line search for the optimum jump length
        t = getoptT(X,W,Y,Z,S,M_E,E,m0,rho) ;
        X = X + t*W;Y = Y + t*Z;S = getoptS(X,Y,M_E,E) ;

% Compute the distortion
        dist(i+1) = norm( (M_E - X*S*Y').*E,'fro' )/sqrt(nnz(E));
        fprintf(1,'%d\t\t%e\n',i,dist(i+1) ) ;
        if( dist(i+1) < tol )
                break ;
        end
end

S = S /rescal_param ;

% Function to Guess the Rank of the input Matrix
function r = guessRank(M_E);
        [n m] = size(M_E);
        epsilon = nnz(M_E)/sqrt(m*n);
    S0 = svds(M_E,100) ;

    S1=S0(1:end-1)-S0(2:end);
```

```
    S1_ = S1./mean(S1(end-10:end));
    r1=0;
    lam=0.05;
    while(r1<=0)
        for idx=1:length(S1_)
            cost(idx) = lam*max(S1_(idx:end)) + idx;
        end
        [v2 i2] = min(cost);
        r1 = max(i2-1);
        lam=lam+0.05;
    end

        clear cost;
    for idx=1:length(S0)-1
        cost(idx) = (S0(idx+1)+sqrt(idx*epsilon)*S0(1)/epsilon  )/S0(idx);
    end
    [v2 i2] = min(cost);
    r2 = max(i2);

        r = max([r1 r2]);

% * * * * * * * * * * * * * * * * * * * * *
% Function to compute the distortion
function out = F_t(X,Y,S,M_E,E,m0,rho)
[n r] = size(X) ;

out1 = sum( sum( ( (X*S*Y' - M_E).*E ).^2 ) )/2 ;

out2 =  rho*G(Y,m0,r) ;
out3 =  rho*G(X,m0,r) ;
out = out1+out2+out3 ;

function out = G(X,m0,r)

z = sum(X.^2,2)/(2*m0*r) ;
y = exp( (z-1).^2 ) - 1 ;
y( find(z < 1) ) = 0 ;
out = sum(y) ;
% * * * * * * * * * * * * * * * * * * * * *

% Function to compute the gradient
function [W Z] = gradF_t(X,Y,S,M_E,E,m0,rho)
[n r] = size(X);
[m r] = size(Y);

XS = X*S ;
YS = Y*S' ;
XSY = XS*Y' ;
```

```
Qx = X'* ( (M_E - XSY).*E )*YS /n;
Qy = Y'* ( (M_E - XSY).*E )'*XS /m;

W = ( (XSY - M_E).*E )*YS + X*Qx + rho*Gp(X,m0,r);
Z = ( (XSY - M_E).*E )'*XS + Y*Qy + rho*Gp(Y,m0,r);

function out = Gp(X,m0,r)
z = sum(X.^2,2) /(2*m0*r) ;
z = 2*exp( (z-1).^2 ).*(z-1) ;
z( find(z<0) ) = 0;

out = X.*repmat(z,1,r) / (m0*r) ;
% * * * * * * * * * * * * * * * * * * * * *

% * * * * * * * * * * * * * * * * * * * * *
% Function to find Sopt given X, Y
function out = getoptS(X,Y,M_E,E)

[n r] = size(X);
C = X' * ( M_E ) * Y ; C = C(:) ;

for i = 1:r
        for j = 1:r
                ind = (j-1)*r + i ;
                temp = X' * (  (X(:,i) * Y(:,j)').*E ) * Y ;
                A(:,ind) = temp(:) ;
        end
end

S = A\C ;
out = reshape(S,r,r) ;
% * * * * * * * * * * * * * * * * * * * * *

% * * * * * * * * * * * * * * * * * * * * *
% Function to perform line search
function out = getoptT(X,W,Y,Z,S,M_E,E,m0,rho)
norm2WZ = norm(W,'fro')^2 + norm(Z,'fro')^2;
f(1) = F_t(X, Y,S,M_E,E,m0,rho) ;

t = -1e-1 ;
for i = 1:20
        f(i+1) = F_t(X+t*W,Y+t*Z,S,M_E,E,m0,rho) ;

        if( f(i+1) - f(1) <= .5*(t)*norm2WZ )
            out = t ;
            return;
        end
        t = t/2 ;
end
```

```
out = t ;
```

Appendix B

Useful MATLAB Functions

In this section, we give the MATLAB code used to generate the partially revealed matrices.

B.1 The Code

```
%
% Function to pick uniformly at random a fixed number of entries from a matrix
%
function [M_E] = PickRandomlyFixedNumberOfEntries(M, m, n)
    E = zeros(n,n);
    k = 1;
    while k <= m
        r = randi(n);
        c = randi(n);
        if (E (r,c) == 0)
            E(r,c) = 1;
            k = k + 1;
        end
    end
    M_E = sparse(M.*E);
end


%
% Sequence of operation used to make the experiments
% Modify the values of r, n and m to understand how the method works
%
clear;
n = 100;
r = 10;
m = 8000;
times = 50;
mu = [1:50];
alpha = [1:50];
```

```
for i = 1 : times

U = randn(n,r);
V = randn(n,r);
Sigma = eye(r);

M0 = U*Sigma*V';

M_E = PickRandomlyFixedNumberOfEntries(M0, m, n);

[X S Y dist] = OptSpace(sparse(M_E), [], 200, 1e-8);
mu(i) = norm(X*S*Y' - M0,'fro')/norm(M0,'fro');
if( mu(i) < 1e-6)
    alpha(i) = 1;
else
    alpha(i) = 0;
end
fprintf(1,'Mu (without noise)           : %e\n\n\n\n',mu(i));
end

fprintf(1,'Mu (MEAN)           : %e\n',mean(mu));
fprintf(1,'Mu (VARIANCE)           : %e\n',var(mu));
success = sum(alpha) / times;
fprintf(1,'success           : %f\n',success*100);
```

Bibliography

[1] Raghundan H. Keshavan, Andrea Montanari, Sewoong Oh, *Matrix Completion from a Few Entries*, *IEEE Trans. Inform. Theory*, **56**, Issue 6, June 2010, 2980-2998.

[2] A. Ferrante and M. Pavon, Matrix Completion *à la* Dempster by the Principle of Parsimony, *IEEE Trans. Information Theory*, **57**, Issue 6, June 2011, 3925-3931.

[3] Emmanuel J. Cands, Benjamin Recht, *Exact Matrix Completion via Convex Optimization*, Found. of Comput. Math., 9 717-772, 2008.

[4] Luigi Salce, *Lezioni sulle matrici, Teoria degli autosistemi e sue applicazioni con argomenti avanzati di teoria delle matrici*, Zanichelli Editore, 1993.

[5] Cristina Ronconi, *Appunti di Geometria*, Univer Editrice, 2009