Convex Optimization Project 1

The Matrix Completion Problem: Review and Experiments

Felipe González Casabianca

October 2016

Note

This work corresponds to the first project in the class of: Introduction to Convex Optimization, given in the fall of 2016 at Universidad de los Andes.

Chapter 1

Introduction

The problem to be studied in this paper is known as the Matrix Completion Problem, a very well known and studied problem. Even though it poses an interest from the purely mathematical approach (the study of features that uniquely determine the entries of a given matrix), this problem is known to have vast real world applications [1] including the famous Netflix Prize.

In this short project, our aim is to introduce a popular relaxed version of this problem, motivating the reasons for this relaxation and explaining how it can be solved algorithmically. Finally, we will study the behaviour and performance of the solution procedure over a set of different example matrices. Since this project is mainly based on the paper: "*Exact Matrix Completion via Convex Optimization*" by Emmanuel J. Canès and Benjamin Recht [2], it seemed fitting to use the library: *fancyimpute 0.0.19*[3] to conduct our experiments, since it offers a simple implementation of the algorithm proposed by the mentioned authors.

Chapter 2

The Problem

In this chapter we will motivate and discuss the Matrix Completion Problem (MCP). The reader who is already familiarized with this problem can skip to the next chapter.

Simply stated the matrix completion problem is:

Given a partially observed matrix A, one wishes to find complete matrix B that is equal to the former in the observed coordinates

From a mathematical perspective, this problem poses several questions surrounding the attributes that uniquely determine the entries of a matrix or even how the number of given coordinates and a certain constraint over the matrix can determine the original one.

But from an application point of view, this problem has appeared in many real life scenarios. The first and probably most common is the missing survey problem. When conducting a survey it seems possible that some questions go unanswered, because of the distribution of the survey or other limitations such as time, difficulty of the questions etc... So the matrix completion problem can be understood as trying to fill this missing values with only the answered questions of the survey. This scenario became particularly popular with the *Netflix Prize*[1], a contest conducted by *Netflix*, to find the best algorithm to complete the user rating matrix and improve their recommendation system. [4]

Another real world scenario consists of re-constructing a complete weighted graph by only getting to see the local neighbor's distances of some nodes. This becomes particularly useful when one tries to reconstruct a network of any kind of sensors, where an obvious limitation is the sensor's reach capacity, therefore introducing an incomplete distance matrix that needs to be found.[5]

[1] http://www.netflixprize.com/index.html

Finally, this problem has also appeared in text analysis, surrounding the term-document matrix. When studying the number of times a certain word or phrase appears inside a particular document, the result can be viewed as a matrix where the columns correspond to the different words that are being studied and the rows to the different documents considered in the research. But it is possible that parts of a certain document have been compromised or that an interest over a new word has surfaced and the researches don't want to analyse the whole document set. So again, this scenarios translate to the matrix completion problem.[6]

The popularity of this problem is evident, but the problem surrounding this challenge is that (in its most simple formulation) it's undetermined. If no restriction is given about the studied matrix, then in reality we can choose the entries randomly and still solve the challenge. Of course this notion has led to all kinds of limitations imposed on the structure of the given matrix and they usually adopt quite well to the underlying context. In the following chapter we will study the assumption we will make, how it can be translated into an application's context and how it sufficiently "*relaxes*" the problem, so it can be solved via convex optimization.

Chapter 3

The Approach

As we saw in the previous chapter, the general and unrestricted formulation of the MCP[1] yields an undetermined problem, therefore it's necessary to include some sort of restriction over the given matrix structure. One of the most popular restrictions accepted is that the matrix in question should have the lowest rank possible, so that the MCE can be reformulated as follows:

Given a partial observation: $K \subseteq [1, \ldots, n] \times [1, \ldots, m]$ *of the coordinates of a matrix* $A_{n \times m}$ *we wish to:*

$$\textbf{minimize: } rank(X)$$
$$\textbf{subject to: } X_{ij} = A_{ij} \ \forall \ (i, j) \in K$$

There are other ways to "*relax*" this problem, enforcing other conditions on the given matrix, like assuming the given matrix is sparse (or the product of two sparse matrices)[7] or that the matrix's principal minors are negative (N-matrix) [8], but we will focus from on the lowest rank restriction.

Before discussing what this restriction produces, notice that it makes sense to restrict the rank of the matrix given the context. We can think of the matrix with missing entries as an incomplete survey or distance graph, so when we complete the matrix we are extracting the missing answers of the survey or the distances we could not get a hold off. Notice that in both these contexts, it seems feasible to assume that the matrix representation of the data in reality creates a surplus of real information contained in the two by two coordinate system, therefore producing a low rank matrix. In real world applications, this is a very common assumption, since it makes (at least in recommendation systems) the search for solutions meaningful [2] [9].

Now, the optimization problem stated above is still too hard to solve, in particular it's NP-Hard [10]. Also, the current known algorithms that provide

[1] finding the missing entries of a partially observed matrix

an exact solution have the time complexity of double the exponential of the dimension of the matrix [11], making this approach still very unpractical.

So the approach taken by Emmanuel J. Canès and Benjamin Recht consist on making 2 assumptions and a relaxation on the minimizing criteria:

1. **The given matrix is chosen randomly following a certain model**

   The authors first consider the *random orthogonal model*. This consists simply on assuming that the matrix given has been chosen randomly following a certain model:

   Recall the singular value decomposition (SVD) of a matrix M:

   $$M = \sum_{i=1}^{k} \sigma_i \mathbf{u_i} \mathbf{v_i} *$$

   (* denotes the conjugate transpose)

   Where the u_i's and v_i's are the left and right singular vectors of M^*M and the σ_i's correspond to the singular values of the previous matrix. So the *random orthogonal model* consists of selecting the families: $\{u_i\}$, $\{v_i\}$ and $\{\sigma_i\}$ (where $1 \leq i \leq k$) randomly from the set of all orthonormal vectors and positive real values respectively.

   Recall that SVD always exist [12], therefore we are considering all matrices as candidates for our problem, and this approach gives us a concrete and simple way to consider the origin of the partially observed matrix.

2. **The locations of the visible coordinates of the unknown matrix are chosen uniformly random**

   Even though some real-life contexts may not agree with this assumption (the reconstruction of the sensor layout) it's still a fair assumptions since in general nothing is known about the origin of the visible locations.

3. **Instead of directly minimizing the rank, the *nuclear norm* will be minimized**

   The *nuclear norm* is defined as:

   $$\|X\|_* = \sum_{i=1}^{k} \sigma_i(X)$$

(where $\sigma_i(X)$ is the ith largest singular value of X, the roots of the eigenvalues of X*X).

This was introduced and used as an heuristic by Fazel in [13].

The authors consider an alternative minimization problem **MCP1** stated as follows:

Given a partial observation: $K \subseteq [1, \ldots, n] \times [1, \ldots, m]$ *of the coordinates of a matrix* $A_{n \times m}$ *we wish to:*

$$\textbf{minimize: } \|X\|_* \\ \textbf{subject to: } X_{ij} = A_{ij} \ \forall \ (i,j) \in K$$

Recall that all norms are convex functions[2], so by minimizing this norm instead the rank, we have a convex optimization problem. Furthermore, this problem can be solved efficiently via semidefinite programing as we will see at the end of this chapter.

These changes are made so that the following theorem can be applied:

Theorem 1. *Let* $\boldsymbol{A}$ *be an* $n_1 \times n_2$ *matrix of a rank* r *sampled from a random orthonormal model, and put* $n = max(n_1, n_2)$*. Suppose we observe* m *entries of* $\boldsymbol{A}$ *with locations sampled uniformly at random. Then there are numerical constants* C *and* c *such that if:*

$$m \geq C \, n^{5/4} \, r \, log(n)$$

The minimizer to the problem stated before is unique and equal to $\boldsymbol{A}$ *with probability at least:* $1 - cn^{-3}$

In short, this theorem says that under the assumptions of the authors and the new minimizing objective, the MCP1 can be solved with high probability using semidefinite programing. As a reference, the proof of this theorem is contained in [2]. Notice also that this theorem says that we can recover (with high probability) the entries of a matrix by only getting to see a small part of it.

Now, it only remains to be seen that the MCP1 can be solved via semidefinite programing.

Before the main lemma, we need a small definition:

Definition 1. *For a matrix* $A \in \mathbb{R}^{m \times n}$*, a* ***Moore-Penrose Pseudoinverse*** $A^\dagger \in \mathbb{R}^{m \times n}$ *is a matrix that satisfies:*

- $AA^\dagger A = A$

[2]This fact follows directly from the triangular inequality and the other requirements of being a norm

- $A^\dagger A A^\dagger = A^\dagger$
- $(AA^\dagger)^* = AA^\dagger$

Recall that if A is invertible then $A^\dagger = A^{-1}$

Now for the main lemma [13][14]:

Lemma 1. *Let $X \in \mathbb{R}^{m\times n}$ be a given matrix. Then $rank(X) \leq r$ if and only if there exists matrices $Y \in \mathbb{R}^{m\times m}$ and $Z \in \mathbb{R}^{n\times n}$ such that: $Y = Y^T$, $Z = Z^T$ and*

$$rank(Y) + rank(Z) \leq 2r, \quad \begin{bmatrix} Y & X \\ X^T & Z \end{bmatrix} \geq 0$$

Proof. Let us prove each direction:

- "$\Rightarrow$"
  Suppose that $rank(X) = r_0 \leq r$, the by **Rank Decomposition**, there exists matrices: $L \in \mathbb{R}^{m\times r_0}$ and $R \in \mathbb{R}^{r_0\times m}$ with:

  $$rank(L) = rank(R) = r_0 \quad \text{and} \quad X = LR$$

  Now, if we set:

  $$Y = LL^T, \;\; Z = R^T R \quad \text{and} \quad A = \begin{bmatrix} L & R^T \end{bmatrix}$$

  We get the following:

  – Both Y and Z are symmetric, since:

  $$Y^T = (LL^T)^T = (L^T)^T L^T = LL^T = Y \quad \text{(similar for } Z\text{)}$$

  – The sum of their ranks holds:

  $$rank(Y) + rank(Z) = rank(LL^T) + rank(R^T R) \leq 2r_0 \leq 2r$$

  Finally, our matrix can be expressed as:

  $$\begin{bmatrix} Y & X \\ X^T & Z \end{bmatrix} = \begin{bmatrix} LL^T & LR \\ R^T L^T & R^T R \end{bmatrix} = \begin{bmatrix} L \\ R^T \end{bmatrix} \begin{bmatrix} L^T & R \end{bmatrix} = A^T A$$

  and since for any non zero matrix we have that $A^T A \geq 0$, we conclude that:

  $$\begin{bmatrix} Y & X \\ X^T & Z \end{bmatrix} \geq 0$$

  finishing this direction.

- "$\Leftarrow$"

  By assumption we have that Z is symmetric, and therefore admits a diagonalization by orthogonal matrices. Then, let U be such a matrix that:

$$U^T Z U = \begin{bmatrix} \Sigma & 0 \\ 0 & 0 \end{bmatrix}$$

  Where Σ is diagonal with all its entries strictly positive, so in particular $\Sigma > 0$. Now consider the following matrix:

$$\begin{bmatrix} I & 0 \\ 0 & U \end{bmatrix}$$

  which by definition of U is orthogonal and hence not singular. So we know that[3]:

$$\begin{bmatrix} Y & X \\ X^T & Z \end{bmatrix} \geq 0 \Leftrightarrow \begin{bmatrix} I & 0 \\ 0 & U^T \end{bmatrix} \begin{bmatrix} Y & X \\ X^T & Z \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & U \end{bmatrix} \geq 0 \tag{3.1}$$

  and since:

$$\begin{aligned} \begin{bmatrix} I & 0 \\ 0 & U^T \end{bmatrix} \begin{bmatrix} Y & X \\ X^T & Z \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & U \end{bmatrix} &= \begin{bmatrix} I & 0 \\ 0 & U^T \end{bmatrix} \begin{bmatrix} Y & XU \\ X^T & ZU \end{bmatrix} \\ &= \begin{bmatrix} Y & XU \\ U^T X^T & U^T Z U \end{bmatrix} \\ &= \begin{bmatrix} Y & X_1 & X_2 \\ X_1^T & \Sigma & 0 \\ X_2^T & 0 & 0 \end{bmatrix} \end{aligned}$$

  Where $[X_1 X_2] = XU$ with an adequate partition. So by 3.1, we have that:

$$\begin{bmatrix} Y & X_1 & X_2 \\ X_1^T & \Sigma & 0 \\ X_2^T & 0 & 0 \end{bmatrix} \geq 0$$

  Then we must have that $X_2 = 0$, which holds if and only if:[4]

$$X(I - ZZ^\dagger) = 0$$

  But by definition of $Z^\dagger$, we know that $I - ZZ^\dagger$ is a projection operator onto the kernel of Z, denoted by: $ker(Z)$. [15]

[3] `http://www.math.ucsd.edu/~njw/Teaching/Math271C/Lecture_03.pdf`
[4] This is a consequence of **Schur Complement** [13]

So:

$$(I - ZZ^{\dagger})v \in ker(Z) \quad \text{for all } v \in \mathbb{R}^m$$

but:

$$X(I - ZZ^{\dagger})v = 0$$

and hence we have that:

$$ker(Z) \subseteq ker(X) \quad \text{and} \quad dim\big(ker(Z)\big) \leq dim\big(ker(X)\big)$$

Consequently, we can say that:

$$rank(X) = n - dim\big(ker(X)\big) \leq n - dim\big(ker(Z)\big) = rank(Z).$$

So we have showed that:

$$rank(X) \leq rank(Z) \tag{3.2}$$

Now, recall that changing rows and columns only alters the sign of the determinant and so:

$$det\left(\begin{bmatrix} Y - \lambda I & X \\ X^T & Z - \lambda I \end{bmatrix}\right) = \pm\, det\left(\begin{bmatrix} Z - \lambda I & X^T \\ X & Y - \lambda I \end{bmatrix}\right)$$

since we did $n + m$ to get from one matrix to the other. Notice that if we equal any of the two previous determinants to zero and solve for λ we get the same sets. This only shows that both matrix share the same eigenvalues and by the characterization of non-negative eigenvalues of a semipositive definite matrix, we can say that:

$$\begin{bmatrix} Y & X \\ X^T & Z \end{bmatrix} \geq 0 \quad \Leftrightarrow \quad \begin{bmatrix} Z & X^T \\ X & Y \end{bmatrix} \geq 0$$

So we can proceed in analogous fashion from 3.1 and get that:

$$dim\big(ker(Y)\big) \leq dim\big(ker(X^T)\big)$$

and consequently:

$$rank(X) = rank(X^T) = m - dim\big(ker(X^T)\big) \leq m - dim\big(ker(Y)\big) = rank(Y).$$

showing that:

$$rank(X) \leq rank(Y) \tag{3.3}$$

Finally by 3.2, 3.3 and the hypothesis that $rank(Y) + rank(Z) \leq 2r$, we conclude that:

$$rank(X) \leq r$$

as we wanted.

This concludes our proof of the lemma.

Notice that with this lemma, we see that the problem

$$\begin{gathered} \textbf{minimize: } rank(X) \\ \textbf{subject to: } X_{ij} = A_{ij} \ \forall \ (i,j) \in K \end{gathered}$$

is equivalent to the problem

$$\begin{gathered} \textbf{minimize: } rank(Z) + rank(Y) \\ \textbf{subject to: } X_{ij} = A_{ij} \ \forall \ (i,j) \in K \\ \textbf{and} \quad \begin{bmatrix} Y & X \\ X^T & Z \end{bmatrix} \geq 0 \end{gathered}$$

with Z and Y as in Lemma 1. Recall that if $W \in \mathbb{R}^{n \times m}$ is a symmetric matrix then: $W^*W = W^2$, all its eigenvalues are positive and hence:

$$\sigma_i(W) = \lambda_i^W \quad \text{(the ith largest eigenvalue of } W)$$

so in conclusion, if W is symmetric we have that:

$$\|W\|_* = \sum_{i=1}^{k} \sigma_i(W) = \sum_{i=1}^{k} \lambda_i^W = trace(W)$$

So with this in mind, if we apply the nuclear norm as an heuristic for the rank of the given matrices of the last mentioned problem, it becomes equivalent to:

$$\begin{gathered} \textbf{minimize: } trace(Z) + trace(Y) \\ \textbf{subject to: } X_{ij} = A_{ij} \ \forall \ (i,j) \in K \\ \textbf{and} \quad \begin{bmatrix} Y & X \\ X^T & Z \end{bmatrix} \geq 0 \end{gathered}$$

Which can be solved using semidefinite programming [2].

Chapter 4

The Experiments

For the experimental part of this project, a simple script was written in *Python* that used the library: *fancyimpute 0.0.19*[3] to guess the entries of a given incomplete matrix. First, the code generates random sets of orthonormal vectors, using the *random* library from python and a simple implementation of the Gram-Schmdit process. With this capability, the code is able to generate random matrices following the *random orthonormal model* as introduced in the previous chapter. The code also selects random entries, so that the observed coordinates follows the schema proposed by the authors of the main paper.

Specifically from the *fancyimpute* libarry, we use the classes: *NuclearNormMinimization* and *Solver*. The authors of this code use *cvxpy*[16] to minimize the Nuclear norm under the mentioned assumptions.

Since the matrix have decimal numbers in their entries, it very improbable that a recovered entry is exactly the same as the original cooordinate. Therefore, the matrices are compared with a tolerance $\varepsilon > 0$, where two coordinates x_{ij} and y_{ij} are considered equal if:

$$|x_{ij} - y_{ij}| < \varepsilon$$

With the previous code, we where able to conduct several simple experiments around the version of the matrix completion problem mentioned in the previous chapters:

- **Experiment 1**

  While the size of a matrix remains constant, we wanted to see the behaviour of the recovered percentage of the matrix as the number of observed entries grew. For this experiment, we worked with matrices of size: 20×20 and a tolerance of $\varepsilon = 0.05$. For each number of observed entries, twenty random matrices where constructed with their entries chosen at

random, so that the mean of their recovered percentage could be calculated.

Results: The following image shows the results of the experiment:

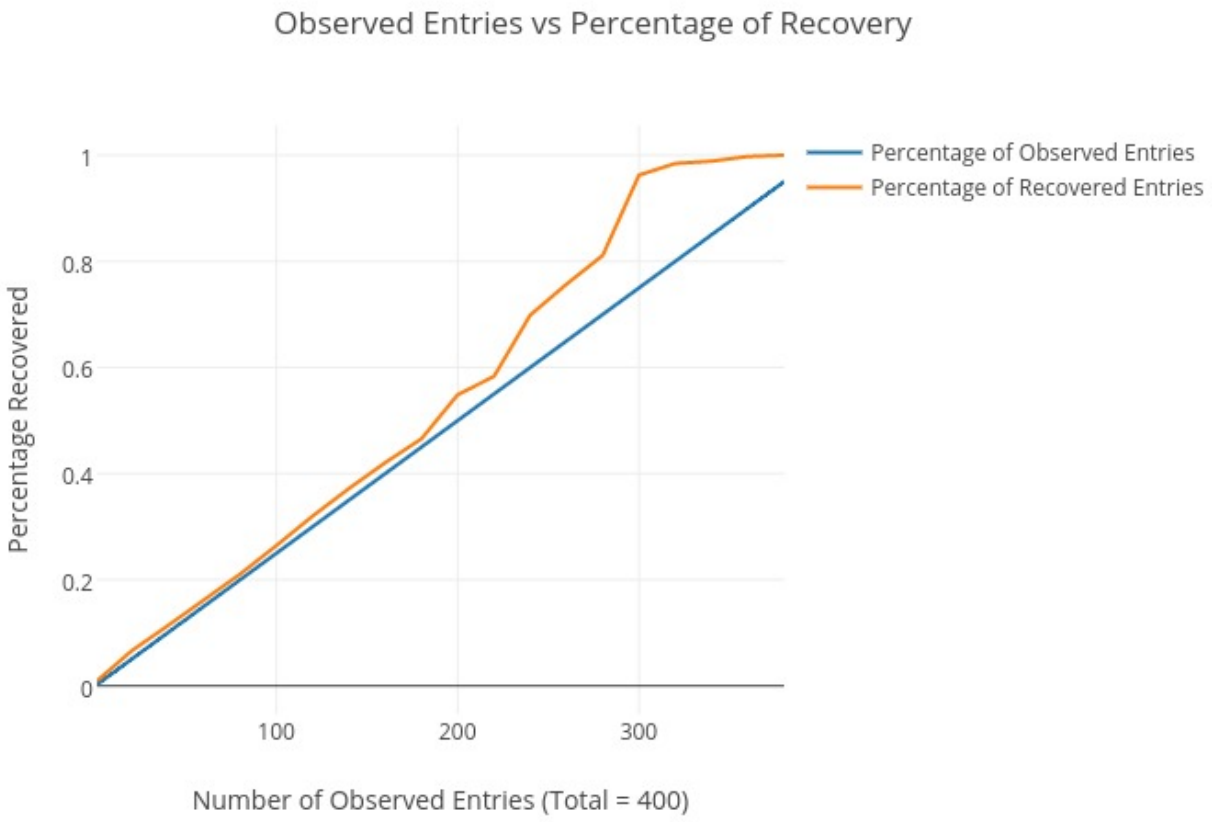

Figure 4.1: Results Experiment 1

Notice that before the number of observed entries is more than half of the total coordinates, this method is useless. After the middle benchmark is reached, this implementation slowly improves, recovering more than 96.5% after 70% of the original matrix is observed.

- **Experiment 2**

After the first experiment, our interest is to see if the percentage of recovered entries changes as the size of the matrix increases while the same percentage of entries is observed. For each size in the range: $[5, 50]$, twenty random square matrices where created with only 70% of their entries visible, and the mean of the percentage of recovered matrix was recorded. Also, each of the matrices has rank: $r = \sqrt[3]{n}$, where n is the number of columns/rows of the matrix.

Results: The following graph shows the results for the experiment

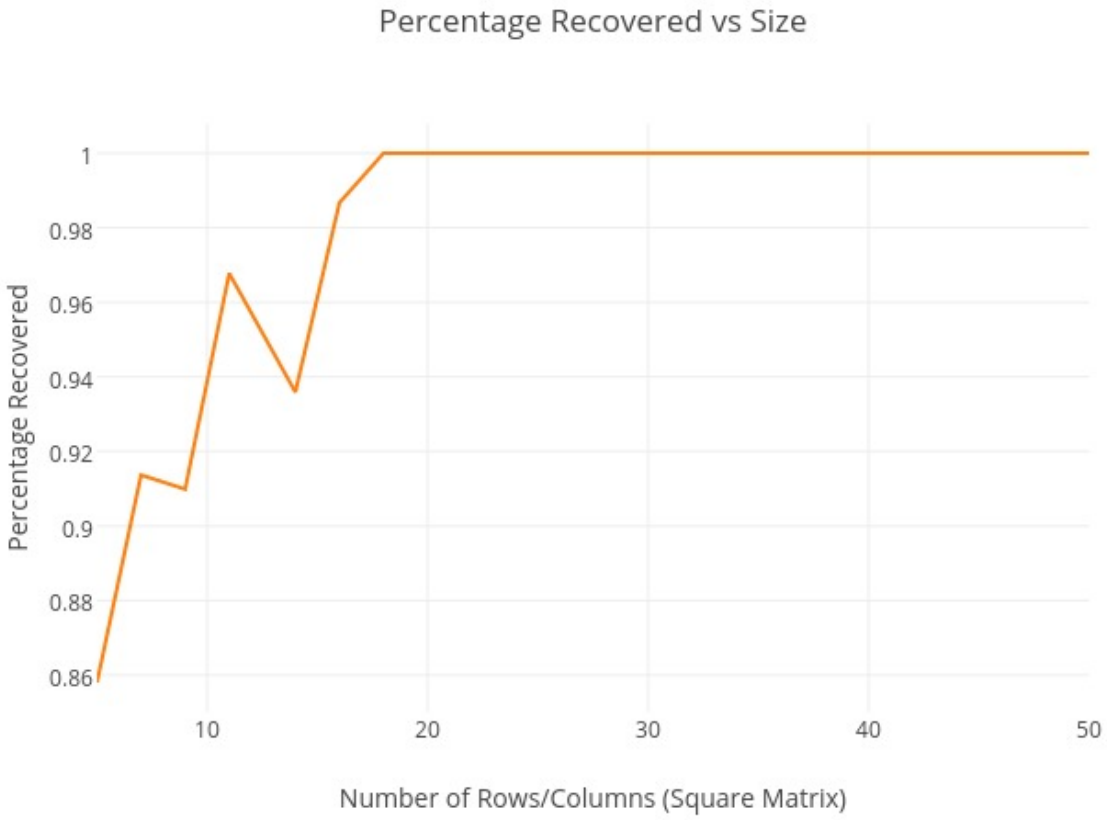

Figure 4.2: Results Experiment 2

Notice that the percentage of recovered entries changes dramatically between the sizes 12 and 20, increasing in almost 22% and staying above

97% after this size. Recall that Theorem 1 says that if m is the amount of recovered entries, we can recover (with high probability) most of the entries if:

$$m \geq C\ n^{5/4}\ r\ log(n)$$

where the matrix's size is $n \times n$ and its rank is r. Now, since $r = \sqrt[3]{n}$, we know that for any constant C, there is an $N \in \mathbb{N}$ such that if: $n > N$, we have that:

$$0.7n^2 > C\ n^{5/4}\ \sqrt[3]{n}\ log(n)\ \Rightarrow\ 0.7n^2 > C\ n^{19/12}\ log(n)$$

Which seems very aligned with our graphic, since $0.7n^2$ is exactly the amount of observed entries of the incomplete matrix (following the authors schema).

Following the previous remark, we re-executed the experiment but this time observing only $n^{16/12}\, log(n)$ coordinates of each matrix of size $n \times n$, instead of $0.7n^2$ (70% of the matrix) as was the previous configuration. This small tweak resulted in the following graph:

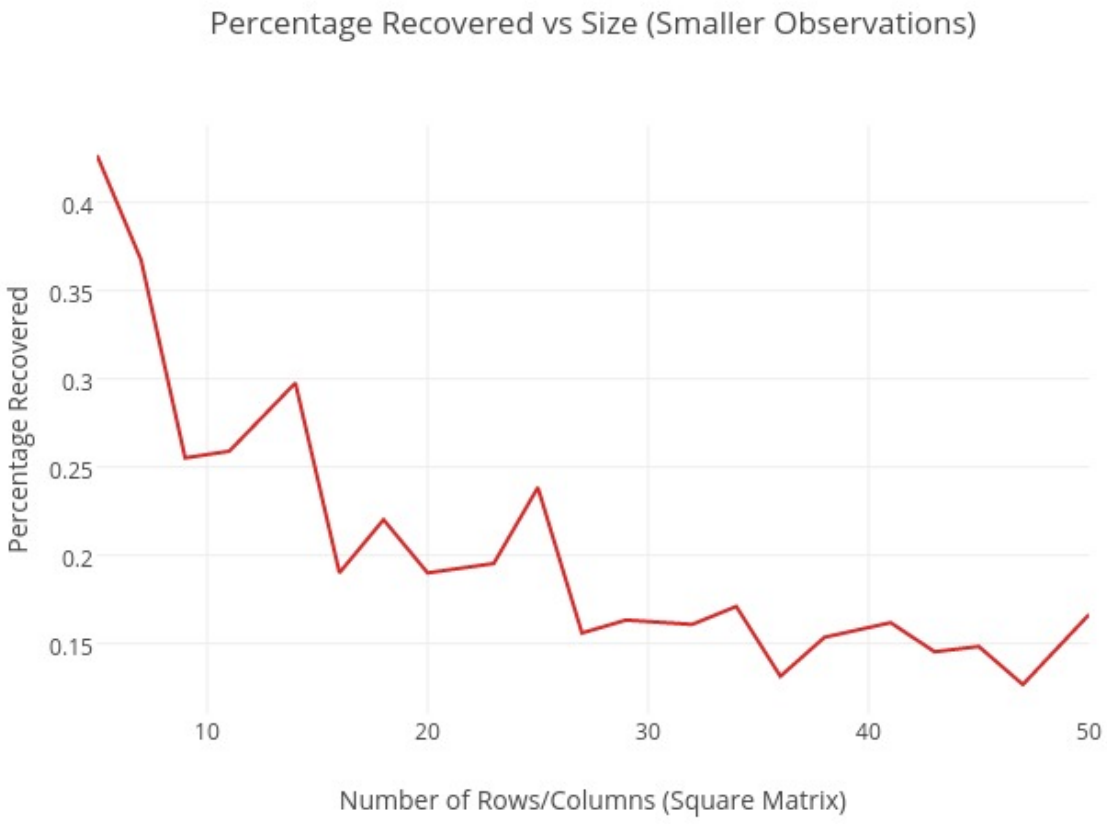

Figure 4.3: Results Experiment 3 (re-executed with smaller observations)

Notice the significant drop in the recovered percentage of each matrix. This again is aligned with the formula of Theorem 1, since now (for some n) we eventually get the other direction of the inequality :

$$n^{16/12}\, log(n) > C\, n^{19/12}\, log(n)$$

- **Experiment 3**

Our next experiment focuses on the efficiently and real-life applicability of the studied implementation. Our procedure simply increases the size of a 70% visible matrix and measures the mean time required to recover the

rest of the matrix. The given matrix and its visible values are generated as in the previous examples.

Results: The following image consolidates the results of the experiments

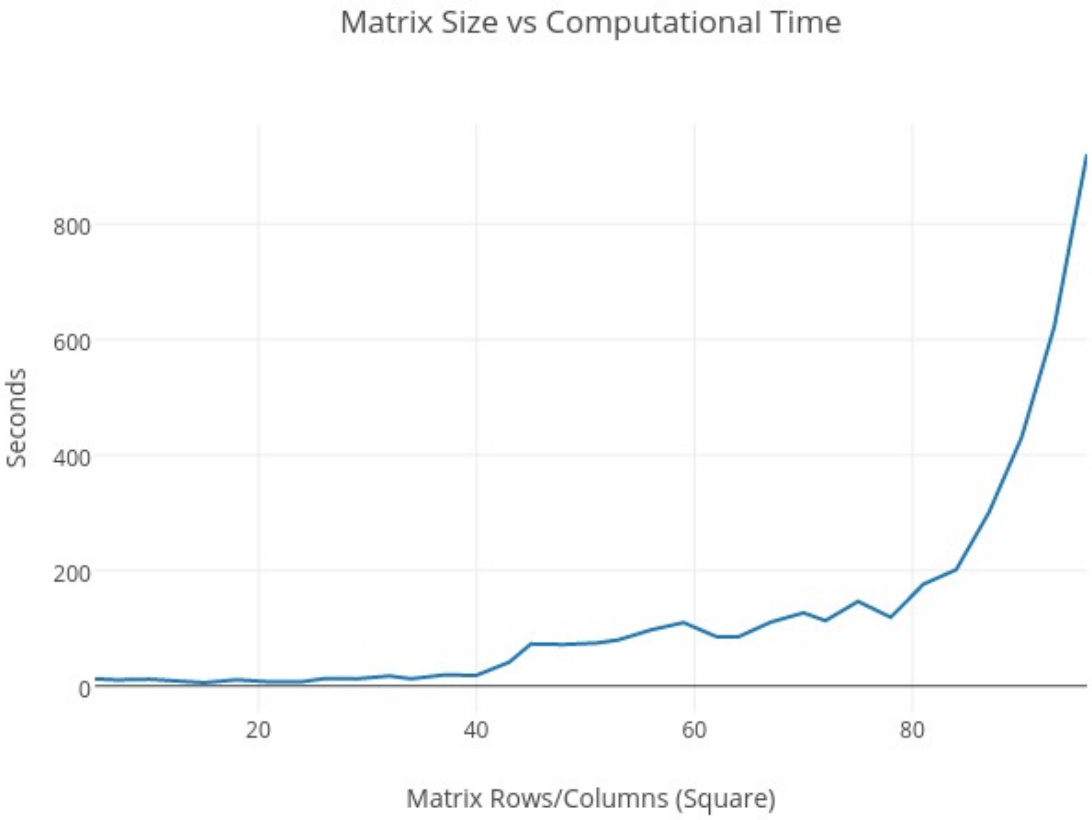

Figure 4.4: Results Experiment 3

Notice how beyond matrices of dimensions 80×80, the library takes considerably more time and becomes somewhat inefficient. This comes as no surprise since the creators of this code explicitly alert the user of the slowness of the library for large matrices [3].

- **Experiment 4**

  This final experiment was conducted as a "Toy Example" to make a section of the presentation more easy going, it serves simply as an amusing

motivation to the MCP. The idea of this experiment is to consider the following scenario:

Suppose you work in a small branch office, where everyone except your boss, can be considered as a peer. One day, an important official peer, superior and subordinate evaluation survey comes from headquarters. The instructions tell you to rate the performance of your peers and superiors on a numeric scale.

Now, of course it's of great value and interest to know exactly what rating did "The Boss" give you on the survey. But he is evil, so sucking up will get you no where. It is also very probable that your are not the only person interested in the boss's review and since your very likeable and charismatic, you can assume that everyone considers you their friend and therefore you can ask them for their survey answers without any drawbacks.

How would you go about this?[1] Notice that there is a incomplete matrix that needs to be filled, where a single row is completely missing. This is not exactly the model of random observation of entries mentioned by the authors, and we need at least one value of this row for the MCP to make sense[2]. So the question becomes a matter of bravery: How many people need to find out their assigned score (by any means) for the whole office to know what the boss thinks of them?

Results: The following image shows the mean number of heroes you must have among your peers to complete the challenge:

[1] Surprise! Matrix Completion Problem

[2] An incomplete matrix of dimension $n \times m$ with this characteristics, is simply a complete $(n-1) \times m$ and the minimum rank solution is filling the missing row with zeros

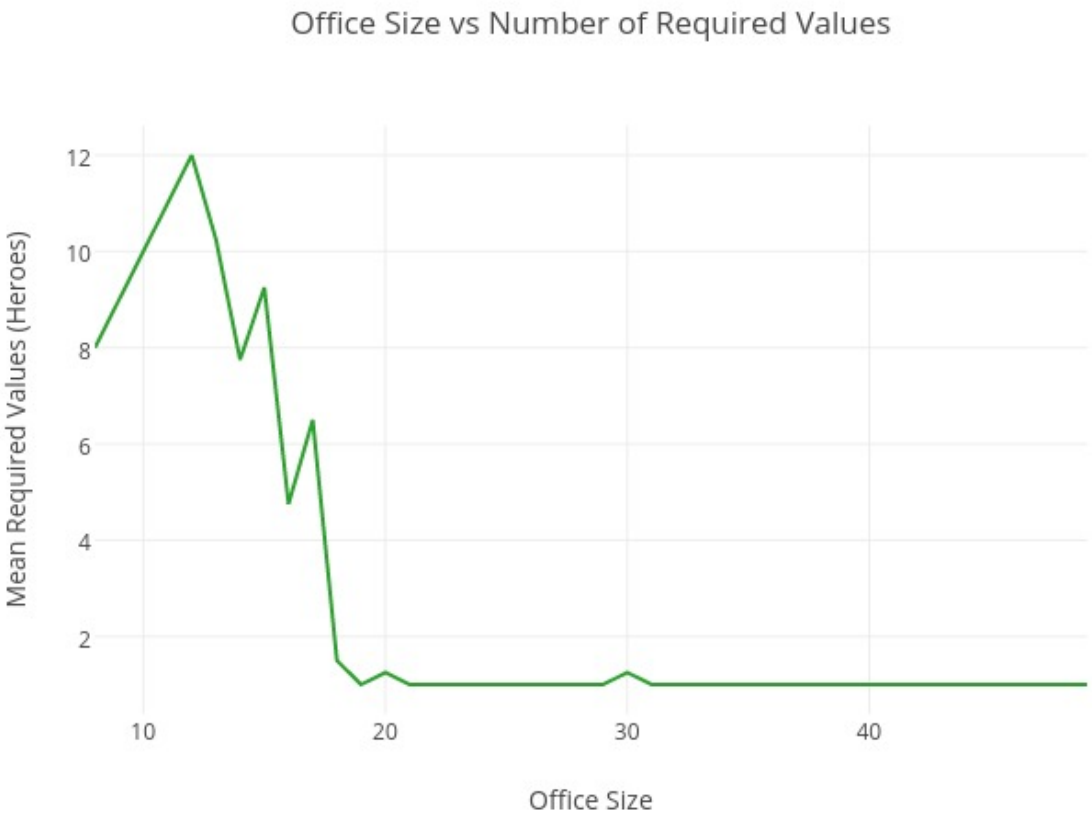

Figure 4.5: Results Experiment 4

Notice that if the office has more than 18 people, basically a single answer from the boss's survey will suffice!

Now, the matrices where created using the random orthogonal model like the rest of the examples, but notice that they did not follow the uniform random selection of observations, since all the missing values where in the same row. The fact that for matrices of dimension bigger than 18×18 only one entry is required, comes mainly from the fact that the matrices in this experiment have small rank ($r = 8$) and recalling again the formula from Theorem 1, we have that:

$$n^2 - n > C\, n^{5/4}\, 8\, log(n)$$

for some n, so eventually we are recovering the whole matrix. Notice that, even though we relaxed the condition of the selection of the entries, we still get a behaviour aligned we the previous formula.

Chapter 5

The Conclusion

Even though the matrix completion problem is very popular, we would be lying if we said it's a problem that's been solved. Focusing only in the papers and publications mentioned in this work, we notice that there are several assumptions that need to be made for an algorithm to tackle this problem and beyond our bibliography, there are still a lot of compromises and context dependant relaxations that are assumed to get better results.

As for the library *fancyimpute* (the part of Norm Minimization), we can say that is an efficient and usable code for small matrices. Recall that for large matrices (above 80×80), even though the code recovers almost the complete matrix, it becomes very slow. In real world applications, the matrices that need to be completed are massive! For instance: *Netflix*, only in the USA[1], has over 47 million subscribers [17]. So even if they offered only about 100 movies (which is obviously not the case) the matrix that Netflix needs to complete is bit larger for any generic solver such as *cvxpy* or its analogue in other languages. For many real life problems, specific theory and code is developed for the MCP to be solved.[18] [8] [19]

At least for small matrices, open source libraries provide good support and efficient ways to calculate approximations of the matrix completion problem.

[1] and recall that it's available in over 190 countries

The Code

The following contains the printed main code written for this project. The complete code is available at: `https://github.com/minigonche/matrix_completion`

Main Source Code

```
# Felipe Gonzalez Casabianca
# Source code for the experiments of the first project of the course:
# Introduccion to Convex Optimization at Universidad de los Andes
# This script uses exclusevly the library fancyimpute, focusing on the
# NuclearNormMinimization, since it implements the approach explained in
# the project

#-------- Script Imports ----------
#For simple syntax divition
from __future__ import division
#The main library
from fancyimpute import NuclearNormMinimization
#For the definition of matrices:
import numpy as np
#For pseudorandom number generation
import random as rand
#For math simple operations
import math
#For Graphing. The methods export the grapgics on plotly, the user only needs
# to enter his/her username and api-key
import plotly.plotly as py
import plotly.graph_objs as go
py.sign_in('minigonche', '8cjqqmkb4o') #This api-key has been changed already
#For time calculations
import time



#-------- Random Orthogonal Vector Sets ----------
# This code constructs pseudorandom sets of orthogonal vectors,
# using the gram schmit process to orthogonalize the random vectors


#The gram schmit coeficient
def gs_cofficient(v1, v2):
```

```
    return np.dot(v2, v1) / np.dot(v1, v1)

#Multiply a vector by a coefficient
def multiply(cofficient, v):
    return map((lambda x : x * cofficient), v)

#Calculates the projection of v2 onto v1
def proj(v1, v2):
    return multiply(gs_cofficient(v1, v2) , v1)

#Calclates the norm of the given vector
def norm(v1):
    return math.sqrt(np.dot(v1, v1))

#Generates a random vector
def random_vector(dim):
    vec = []
    for i in range(dim):
        vec.append(rand.uniform(-1, 1))

    if(norm(vec) == 0):
        return random_vector(dim)

    return vec

#Generates a random positive vector
def random_postive_vector(dim,max_value):
    vec = []
    for i in range(dim):
        vec.append(rand.uniform(0, max_value))

    if(norm(vec) == 0):
        return random_vector(dim)

    return vec

#Genrates a random orthonormal set of vectors
def random_ortho_set(n, dim):
    """
        Parameters
        ----------
        n : int
            The number of vectors in the set
        dim : int
            The dimention of the vectors
    """
    if(n > dim):
        raise ValueError('''The size of the linearly independent set cannot be
        grater than the dimention of the vectors''')

    Y = []
    for i in range(n):
        temp_vec = random_vector(dim)
        for inY in Y :
            # Finds the projection
            proj_vec = proj(inY, temp_vec)
            # Substracts the vector
```

```
            temp_vec = map(lambda x, y : x - y, temp_vec, proj_vec)
        #Normalizes the vector
        temp_vec = multiply(1/norm(temp_vec), temp_vec)
        Y.append(temp_vec)
    return Y

#-------- Random Orthogonal Model Matrix ----------
# This code construct a random matrix using the random
# orthgonal model

#Genrates a random matrix using the random orhonormal model
def random_matrix(dim, r):
    """
        Parameters
        ----------
        dim : int
            The dimention of the matrix (square)
        r : int
            The number of singular vectors
    """
    #First set of orthonormal vectors (single vectors)
    set1 = random_ortho_set(r,dim)
    #Second set of orthonormal vectors (single vectors)
    set2 = random_ortho_set(r,dim)
    #Set of single values
    sing_values = random_postive_vector(r,50)

    #generates the matrix
    M = sing_values[0]*np.matrix(set1[0]).transpose()*np.matrix(set2[0])
    for i in range(1,r):
        temp = np.matrix(set1[i]).transpose()*np.matrix(set2[i])
        M = M + sing_values[i]*temp

    return M



#Gets a matrix wth a certain number of observation of the given matrix.
# this values are chosen uniformly at random and the rest of the values are
# marked as NaN
def get_observed_matrix(M, num_observations):

    n = M.shape[0]
    total = n*n
    empty_matrix = np.empty((n,n,))
    empty_matrix[:] = np.NAN
    empty_matrix = np.matrix(empty_matrix)

    m_origin = np.copy(M)
    m_dest = np.copy(empty_matrix)
    flag = False

    # If the total amount of observations is to large, is better to simply
    # remove values instead of adding
    if(total/2 < num_observations):
        m_origin = np.copy(empty_matrix)
        m_dest = np.copy(M)
```

```
        num_observations = total - num_observations
        flag = True

    for i in range(num_observations):
        coor = [rand.randint(0,n-1),rand.randint(0,n-1)]
        # Selects randomly untils the coordinate selected is idle
        while(np.isnan(m_dest[coor[0],coor[1]]) == flag):
            coor = [rand.randint(0,n-1),rand.randint(0,n-1)]

        #Updates the coordinate
        m_dest[coor[0],coor[1]] = m_origin[coor[0],coor[1]]

    return m_dest


# ------------------------------------------------------------------------
# ------------------------- Experiments ----------------------------------
# ------------------------------------------------------------------------

# Graphs the mean percentage of the recovered matrix vs the amount of
# observed entries

def graph_percentage_vs_observed(dim,
                                 ite,
                                 tries,
                                 r,
                                 tol,
                                 fast_but_approximate):

    """
        Parameters
        ----------
        dim : int
            The dimention of the matrix (square)
        ite : int
            The number of iterations per observations
        tries : int
            The number of values (evenly distributed) that the number of
            observations will take
        r : int
            The number of singular vectors
        tol : float
            The tolerance for two values to be considered equal
        fast_but_approximate : bool
            Use the faster but less accurate Splitting Cone Solver (part of
            FancyImpute)
    """

    if(dim <= 1):
        raise ValueError('''The dimention must be larger than 1''')

    total = dim*dim
    skip = (dim*dim - 1)/tries
    m_values = np.arange(1,total, skip).astype(int)

    #The percentages
    per = []
```

```
    for m in m_values:
        ite_values = []
        for i in range(ite):
            #Complete Random Matrix
            initial_matrix = random_matrix(dim, r)
            #Incomplete Matrix
            incomplete_matrix = get_observed_matrix(initial_matrix, m)
            #Guessed Matrix
            guessed_matrix = NuclearNormMinimization(fast_but_approximate=
            fast_but_approximate).complete(incomplete_matrix)

            #The boolean matrix indicating if two coordenates are equal
            # (with the given tolerance)
            bool_matrix = np.isclose(np.matrix(initial_matrix),
                                     guessed_matrix,
                                     atol = tol)

            #Calculates percentage of correcly guessed entries
            percentage = np.sum(bool_matrix)/total

            #Saves percentage
            ite_values.append(percentage)
        #Saves mean of percentages
        per.append(np.mean(ite_values))

    #Draws the initial trace
    trace_m = go.Scatter(x = m_values.tolist(),
                         y =  multiply(1/total, m_values.tolist()))
    #Draws the observed trace
    trace_r = go.Scatter(x = m_values.tolist(),
                         y =  per)
    #Export graph
    plot_url = py.iplot([trace_m,trace_r])


# Graphs the mean percentage of the recovered matrix vs the size of the matrix
# at a constant percentage of observed

def graph_percentage_vs_size(per_obs,
                             ite,
                             tries,
                             min_size,
                             max_size,
                             r,
                             tol,
                             fast_but_approximate):

    """
        Parameters
        ----------
        per_obs : float
            The percentage corresponding to the observed entries of the matrix
        ite : int
            The number of iterations per size
        tries : int
            The number of values (evenly distributed) that the number of
```

```
        observations will take
    min_size : int
        The minimum size of the matrix (row or column dimention)
    max_size : int
        The maximum size of the matrix (row or column dimention)
    r : int
        The number of singular vectors
    tol : float
        The tolerance for two values to be considered equal
    fast_but_approximate : bool
        Use the faster but less accurate Splitting Cone Solver (part of
        FancyImpute)
"""
if(max_size <= min_size):
    raise ValueError('''The maximum size must be striclty  larger than
                        the minimum size''')

skip = (max_size - min_size)/tries
s_values = np.arange(min_size, max_size + skip, skip).astype(int)

#The percentages
per = []

for s in s_values:
    ite_values = []
    for i in range(ite):
        #Complete Random Matrix
        initial_matrix = random_matrix(s, r)
        #Incomplete Matrix
        incomplete_matrix = get_observed_matrix(initial_matrix,
                                                int(s*s*per_obs))
        #Guessed Matrix
        guessed_matrix = NuclearNormMinimization(fast_but_approximate=
        fast_but_approximate).complete(incomplete_matrix)

        #The boolean matrix indicating if two coordenates are equal
        # (with the given tolerance)
        bool_matrix = np.isclose(np.matrix(initial_matrix),
                                 guessed_matrix,
                                 atol = tol)

        #Calculates percentage of correcly guessed entries
        percentage = np.sum(bool_matrix)/(s*s)

        #Saves percentage
        ite_values.append(percentage)
    #Saves mean of percentages
    per.append(np.mean(ite_values))
    print('Size: ' + str(s) + ' Finished')

#Draws the observed trace
trace = go.Scatter(x = s_values.tolist(),
                   y =  per)
#Export graph
plot_url = py.iplot([trace])
```

```
#The average time necessary to guess a given matrix or certain size
def average_time(dim,
                 per_obs,
                 ite,
                 r,
                 fast_but_approximate):

    """
        Parameters
        ----------
        dim : int
            The dimention of the matrix (square)
        per_obs : float
            The percentage corresponding to the observed entries of the matrix
        ite : int
            The number of iterations
        r : int
            The number of singular vectors
        fast_but_approximate : bool
            Use the faster but less accurate Splitting Cone Solver (part of
            FancyImpute)
    """

    clock = []
    for i in range(ite):
         #Complete Random Matrix
        initial_matrix = random_matrix(dim, r)
        #Incomplete Matrix
        incomplete_matrix = get_observed_matrix(initial_matrix,
                                                int(dim*dim*per_obs))
        #Starts clock
        #-----
        start_time = time.time()
        #Guessed Matrix
        guessed_matrix = NuclearNormMinimization(fast_but_approximate=
        fast_but_approximate).complete(incomplete_matrix)
        #Stops clock
        total_time = (time.time() - start_time)

        #Saves time
        clock.append(total_time)

    return(np.mean(clock))
```

Bibliography

[1] David Goldberg, David Nichols, Brian M Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, 1992.

[2] Emmanuel J Candès and Benjamin Recht. Exact matrix completion via convex optimization. *Foundations of Computational mathematics*, 9(6):717–772, 2009.

[3] Alex Rubinsteyn and Sergey Feldman. fancyimpute 0.0.19. https://pypi.python.org/pypi/fancyimpute, 2016.

[4] James Bennett and Stan Lanning. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35, 2007.

[5] Nathan Linial, Eran London, and Yuri Rabinovich. The geometry of graphs and some of its algorithmic applications. *Combinatorica*, 15(2):215–245, 1995.

[6] Sewoong Oh. *Matrix completion: Fundamental limits and efficient algorithms*. PhD thesis, Stanford University, 2010.

[7] Akshay Soni, Troy Chevalier, and Swayambhoo Jain. Noisy inductive matrix completion under sparse factor models. *arXiv preprint arXiv:1609.03958*, 2016.

[8] C Mendes Araújo, Juan R Torregrosa, and Ana M Urbano. The n-matrix completion problem under digraphs assumptions. *Linear algebra and its applications*, 380:213–225, 2004.

[9] Emmanuel J Candès and Terence Tao. The power of convex relaxation: Near-optimal matrix completion. *IEEE Transactions on Information Theory*, 56(5):2053–2080, 2010.

[10] Moritz Hardt, Raghu Meka, Prasad Raghavendra, and Benjamin Weitz. Computational limits for matrix completion. In *COLT*, pages 703–725, 2014.

[11] Alexander L Chistov and D Yu Grigor'ev. Complexity of quantifier elimination in the theory of algebraically closed fields. In *International Symposium on Mathematical Foundations of Computer Science*, pages 17–31. Springer, 1984.

[12] Lloyd N Trefethen and David Bau III. *Numerical linear algebra*, volume 50. Siam, 1997.

[13] Maryam Fazel. *Matrix rank minimization with applications*. PhD thesis, PhD thesis, Stanford University, 2002.

[14] Stephen P Boyd, Laurent El Ghaoui, Eric Feron, and Venkataramanan Balakrishnan. *Linear matrix inequalities in system and control theory*, volume 15. SIAM, 1994.

[15] João Carlos Alves Barata and Mahir Saleh Hussein. The moore–penrose pseudoinverse: A tutorial review of the theory. *Brazilian Journal of Physics*, 42(1-2):146–165, 2012.

[16] Steven Diamond and Stephen Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.

[17] Number of netflix streaming subscribers worldwide from 3rd quarter 2011 to 2nd quarter 2016 (in millions). `https://www.statista.com/statistics/250934/quarterly-number-of-netflix-streaming-subscribers-worldwide/`. Accessed: 2016-10-01.

[18] Robert M Freund, Paul Grigas, and Rahul Mazumder. An extended frank-wolfe method with" in-face" directions, and its application to low-rank matrix completion. *arXiv preprint arXiv:1511.02204*, 2015.

[19] Emmanuel J Candes and Yaniv Plan. Matrix completion with noise. *Proceedings of the IEEE*, 98(6):925–936, 2010.