Large Scale Matrix Completion and Recommender Systems

by

Lily Robin Amadeo

A thesis
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Master of Science
in
Data Science

July 2015

APPROVED:

Professor Randy C. Paffenroth, Adviser:

Professor Andrew C. Trapp, Reader:
The goal of this thesis is to extend the theory and practice of matrix completion algorithms, and how they can be utilized, improved, and scaled up to handle large data sets. Matrix completion involves predicting missing entries in real-world data matrices using the modeling assumption that the fully observed matrix is low-rank. Low-rank matrices appear across a broad selection of domains, and such a modeling assumption is similar in spirit to Principal Component Analysis. Our focus is on large scale problems, where the matrices have millions of rows and columns. In this thesis we provide new analysis for the convergence rates of matrix completion techniques using convex nuclear norm relaxation. In addition, we validate these results on both synthetic data and data from two real-world domains (recommender systems and Internet tomography). The results we obtain show that with an empirical, data-inspired understanding of various parameters in the algorithm, this matrix completion problem can be solved more efficiently than some previous theory suggests, and therefore can be extended to much larger problems with greater ease.
## Contents

**Abstract**

**Symbols**

### 1 Introduction
1.1 Motivation ................................. 1  
1.2 Problem Background ....................... 4  
1.3 Experiments and Results .................. 8  
1.4 Our Contribution .......................... 9  
1.5 Structure of Thesis ........................ 9

### 2 Background Theory
2.1 Low Rank Matrices .......................... 10  
2.2 Matrix Completion .......................... 11  
2.2.1 Rank Minimizing Matrix Completion and Nuclear Norm .......................... 12  
2.3 RPCA ........................................ 15  
2.3.1 RPCA Algorithm ........................ 17  
2.4 eRPCA ....................................... 17  
2.4.1 eRPCA Algorithm ........................ 19  
2.4.2 Randomized SVD .......................... 19  
2.4.3 Applications .............................. 20

### 3 Experiments
3.1 Data Generation .............................. 22  
3.2 Finding Improved $\mu$ and $\rho$ ............. 22  
3.3 Literature Provided $\mu$ and $\rho$ Values  ........ 23  
3.4 Convergence ................................ 25  
3.5 Varying $\mu$ and $\rho$ ....................... 25  
3.6 Improvement Over Values in the Literature ........ 31  
3.7 Additional Experimentation ................... 34  
3.7.1 Varying Point-Wise Error Bounds ............ 34  
3.7.2 Varying Dimensions and Percent Observed ....... 35  
3.7.3 Allowance of Large Errored Entries to Be Unobserved .................. 37

### 4 Applications
4.1 MovieLens Data Set .......................... 40  
4.2 Internet Tomography ........................ 43
5 Conclusion
5.1 Contributions ........................................ 48
5.2 Future work ........................................ 49

A Additional Details 50
A.1 Matrix Completion Conditions ................. 50
A.2 PCA Conditions .................................. 50
A.3 Computing Resources ............................. 51
A.4 Iteration Time .................................... 51
A.5 Figures ............................................ 52

Bibliography 60
Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mu )</td>
<td>algorithm parameter controlling for solver focus</td>
</tr>
<tr>
<td>( \rho )</td>
<td>algorithm parameter controlling growth of ( \mu )</td>
</tr>
<tr>
<td>( \lambda )</td>
<td>algorithm parameter weight</td>
</tr>
<tr>
<td>( M )</td>
<td>a matrix</td>
</tr>
<tr>
<td>( L )</td>
<td>a low-rank matrix</td>
</tr>
<tr>
<td>( S )</td>
<td>a sparse matrix</td>
</tr>
<tr>
<td>( \epsilon )</td>
<td>a matrix of error bounds</td>
</tr>
<tr>
<td>( \Omega )</td>
<td>set of observed points in a matrix</td>
</tr>
<tr>
<td>( P_{\Omega}(\cdot) )</td>
<td>projection of a matrix onto ( \Omega )</td>
</tr>
<tr>
<td>( | \cdot |_0 )</td>
<td>zero-norm, count of non-zero entries, ( #(k</td>
</tr>
<tr>
<td>( | \cdot |_1 )</td>
<td>one-norm, sum of absolute values of entries, ( \sum_{k=1}^{n}</td>
</tr>
<tr>
<td>( | \cdot |_F )</td>
<td>Frobenius norm, ( \sqrt{\sum_{ij}</td>
</tr>
<tr>
<td>( | \cdot |_F^2 )</td>
<td>in our work, we typically consider the Frobenius norm squared, ( \sum_{ij}</td>
</tr>
<tr>
<td>( | \cdot |_* )</td>
<td>nuclear norm, sum of singular values ( \sum_{k=1}^{n} \sigma_k(X) )</td>
</tr>
<tr>
<td>( S_\epsilon )</td>
<td>shrinkage operator, ( \text{sign}(x)\max(</td>
</tr>
<tr>
<td>( \epsilon_1, \epsilon_2 )</td>
<td>algorithm convergence criteria</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The goal of this project is to extend the theory and practice of algorithms revolving around Principal Component Analysis (PCA), and to understand how they can be utilized, improved, and scaled up to handle very large data sets when solving matrix completion problems. PCA is a statistical procedure that transforms a set of data observations into a group of linearly uncorrelated variables. The typical application of PCA is for dimensionality reduction in data, and to better allow a user to see the internal structure of the data while retaining information about it. In this thesis, the main application is large-scale matrix completion, or, filling in missing entries in real-world data matrices. An incomplete matrix offers extensive opportunities and challenges for analysis, and PCA-like algorithms are an important class of techniques for matrix completion [1], especially for large problems.

In this project we applied the technique to both synthetic, generated data and data from two real-world domains. The results we obtained show that with an understanding of various parameters in the algorithm, this matrix completion problem can be solved more efficiently than previously known.

1.1 Motivation

We are in an age where data is growing at immense speeds, and solving problems has become as much about the technique for analysis as it has handling the data. Making algorithms work for data sets that are very large, (even wanting to consider exabyte scale
data of $O(10^{18})$ bytes) is not only interesting, but absolutely necessary for continued advancement in many fields. Principal Component Analysis is one of the foundational algorithms of data science, and it is an extremely popular statistical technique. Given how applicable it is, PCA has been widely researched and has had numerous extensions created to handle various circumstances that a user encounters within a data set. In particular, in its common application of dimensionality reduction, it has been utilized for image processing, data compression, network mapping, and many more real-world applications [2–4]. PCA is also a foundational element in solving the matrix completion problem [1]. Exploring techniques for making these algorithms efficient on a large-scale is the focus of our work.

PCA is important, because it allows data to be represented in a new, often smaller, set of variables, and, as we focus on here, can to be done in a scalable way. Matrix completion is the situation when a matrix is simply missing some, or many, entries. The user is aiming to fill in those missing entries using some principled approach, which itself is a hard problem, but under some assumptions and conditions on the data, completing the matrix can be accomplished with a high probability of success. In fact, in the case of using PCA for the problem, a surprisingly small number of entries need to be observed in order to recover the full matrix. In our work we emphasize and depend on the fact that too many extraneous entries simply add to computational costs, without improving the recovery process. Luckily, this sparse situation comes up many times in practical business problems. Classic examples of such problems include recommender systems [5], and a famous example of a recommender system is the Netflix Prize[6]. Given a set of user ratings on movies, we wish to predict what individual users will rate movies they have not yet seen. Essentially, the problem becomes completing a matrix of partially observed data, where the missing entries are movies that a user has not yet rated, and the observed are those that the user has rated. In this thesis, we work on both the theory and practice of such methods, and one of our tests on real-world data will be an example of such a recommender system. In addition, among other examples, we will treat mapping of computer networks based upon hop distance. Between these two problems, we expect to demonstrate the applicability of large-scale matrix completion to problems of practical interest.

What is the purpose of applying such algorithms to large data sets? We are quickly moving beyond any smaller measures of data and into the exabyte age. In truth, this
will not last long itself, as data collection of all types continues to grow and businesses, science, government, and others, retain every piece of information they produce and collect. Working with exabyte-level data is a reality, and it is worthy of exploring how current preferred statistical and analysis methods can be expanded into this realm. Many algorithms exist for robust data exploration and analysis, and while they could be applied to increasingly large data sets, their processing time and power required are unreasonable. This is why it is necessary to work towards scaling up these procedures to large data sets in a way that produce time and cost-efficient analysis.

At present, it is estimated that Google holds approximately 15 exabytes of data stored in total [7]. The library of congress only holds approximately 3 petabytes [7]. This begs the question—is exabyte scale work really necessary? The answer of course is yes, it is absolutely worth the time and effort to scale these algorithms up. Estimates predict that this level of data will soon be the norm, in movies [8], cellular data [9], medical and genetic research [10], and social media [11], among others. To best utilize the massive amounts of data we are now generating every day, algorithms for analysis must be able to be applied to that data.

The ability to process big data, then, is clearly necessary. The work in this thesis shows that we can begin to approach these massive exabyte scales with these algorithms. In Chapter 2, we detail how we can take a matrix of data that is large in dimensions, and leverage its second order correlation matrix to perform PCA analysis. This allows us to solve problems efficiently at such scale. Given, say, a matrix that is $10^6$ by $10^6$ in size, or even $10^7$ by $10^7$, we can apply these algorithms directly on their original, first order matrices. However, taking advantage of the tricks detailed in Chapter 2, we can analyze second order matrices arising from data that was originally a $10^6$ by $10^{12}$ matrix. Those original dimensions represent an exabyte of data. It is clear then that this work, and similar, can be applied to very big data. As we proceed in this work, all matrices we work with may be either of the first or second order types.

From the literature[4, 12, 13], we are provided with algorithms that are asymptotically efficient. In fact, the one we work with is linear in the observed number of entries that are given. Indeed, then, when running in $O(n)$, any constant preceding $n$ greatly

\footnote{Of course, the mapping from a $10^6$ by $10^{12}$ first-order matrix to a $10^6$ by $10^6$ second-order matrix is non trivial, but it is easily parallelizable and distributed. We focus on the later step in the process, of doing PCA with that $10^6$ by $10^6$ matrix.}
effects the algorithm’s run time efficiency. A matrix containing twice as many entries as another thus takes twice as much time to run through the algorithm. This is where we see that, if we can manipulate any constants within our algorithm, we may obtain large changes in run time. There are parameters in our algorithm of choice that allow for such manipulation, and it is these that we work with to improve efficiency.

What makes PCA so worthy of improvement? With the ever increasing amount of data in the world, it is only appropriate that current methods be scaled up as large as feasibly possible. A myriad of techniques exist for dealing with small data sets efficiently, and many would be useful for larger data sets as well. PCA and Robust PCA (RPCA, which will be a focus of our exposition) are two examples of these, provided the proper modifications are made for that level of scale. In any data set, linear predictability is a desired trait. This allows one to take simple, linear combinations or transformations of the original data to obtain prediction results, and can be immensely useful in application. In matrix completion, this prediction can be applied to filling in any unknown data points in the matrix. PCA is a key example on the path to the large space of algorithms that do jobs such as this. Applying linear transformations to the data, as PCA does, to obtain new, extracted features, also allows for a broader or altered perspective of the data. In this way, data can be understood in new context. Additionally, PCA offers the user a chance to take advantage of a problem using (unsupervised) linear predictability. This allows for much simpler, straight-forward predictions, and can be very important to problems that can utilize that.

1.2 Problem Background

The matrix completion problem has been explored extensively in the literature[1, 14–17]. It seeks to solve the following: given a matrix $M$ that is $n_1$ by $n_2$, and only a sample $m$ of its entries, is it possible to recover the remaining entries and complete the matrix? One has to wonder if that is even feasible. Intuition says no, this problem can not be solved, as the missing entries could be anything. How can they be recovered? Obviously, one has to make some kind of modeling assumption, and a standard one, in this domain, is that $M$ is low-rank, meaning it has few linearly independent rows and columns. When $M$ is low-rank, it is in fact possible to recover the missing entries given the set $m$ of observed entries. If $M$ is approximately low-rank, and enough entries $m$ are
given, then there is a high probability that $M$ can be recovered. The process of filling in the missing data can be approached in a number of ways. In a paper presented in 2010 by Emmanuel Candes and Terence Tao\[18\], the idea of using convex relaxation made the problem tractable for large-scale data. Utilizing the optimization roots of PCA is one of them, and it can be solved efficiently using convex optimization. Recovering that missing data can be of immense help in applications, and if done properly, can tell the user much more information about the data than they originally had. In our pursuit, we consider matrices of various sizes and, especially, with different amounts of entries observed. We are especially interested in cases where $m \geq n_1 n_2$, although we consider also cases where $m$ is closer to the size of the full matrix.

Before proceeding, it is necessary to understand why the assumption of low-rank is a reasonable one. A low-rank matrix can be understood and defined in multiple ways. It is a matrix that has a small number of non-zero singular values in its Singular Value Decomposition (SVD). It is also a matrix with a small number of linearly independent rows (or columns). This means, in a low-rank matrix, one only needs a small number of rows (columns) to produce the remaining rows (columns) as linear combinations of those few. A matrix always has a column-rank equal to its row-rank \[19\]. In more approachable terms, it means that there is a small number of basis rows (columns) that span a unique space in the matrix. In practice, low-rank matrices are encountered very often. Across domains, they occur with such high frequency, that this assumption becomes reasonable for many problems. Using the movie recommender system as an example, the rank of the matrix represents the number of user rows used to predict the rest of the movie preferences of the user base. Being low-rank, this means only a few rows are required—e.g., people’s movie preferences are not very unique. However, in some applications, low-rank matrices are undesirable. A singular matrix, for example, has no inverse. A low-rank matrix by definition can not have its maximum possible rank (i.e., be full rank), which is the value of its larger dimension, and is therefore singular (similarly for rectangular matrices, their being low-rank precludes them from having either left or right inverses). Without an inverse, many techniques that rely on one are not available. However, for our case, and for making predictions, working with low-rank matrices can be extremely advantageous.

PCA happens to have a very neat, closed form solution. Consider a matrix $M$, whose columns have been normalized to a mean of zero. Like most traditional data analysis
problems, here each row represents one record of the collected data, and each column
represents one field. To compute the principal components, one can simply use the
Singular Value Decomposition to find them. This is a common linear algebra technique
used to factorize a matrix into component pieces. Chapter 2 details the relationship
between PCA and SVD. It serves as a linear algebra method to solve the PCA problem,
and can be briefly stated as decomposing a matrix \( M \) into \( M = U \Sigma V^T \), where \( U \) and
\( V \) are unitary (meaning that \( UU^T = U^T U = I \), and \( \Sigma \) is diagonal, with entries along
the diagonal known as singular values. The columns of \( U \) are known as the left singular
vectors of \( M \), and those of \( V \), the right singular vectors.

The covariance matrix for the PCA computation \( MM^T \) can be rewritten as \( WDW^T \),
as it is symmetric and diagonalizable. If we take the SVD matrix \( M = U \Sigma V^T \), and
construct the covariance of that, we obtain \( MM^T = (U \Sigma V^T)(V \Sigma U^T) = U \Sigma^2 U^T \), and as
the square root of the eigenvalues of \( MM^T \) are the singular values of \( M \), the relationship
is then obvious. One important note is that, when using PCA, the required process only
depends on \( U \) and \( \Sigma \) from the SVD computation. This fact, along with the previously
mentioned second-order matrix, means we may again reduce the time and space required
to run our algorithm.

The SVD construction of PCA happens to provide a relatively simple, closed-form so-
lution to the problem. However, what is perhaps less well-known is that the original
definition of PCA as given by Eckart and Young in 1936 was derived from an optimization
viewpoint [20]. In particular, it is this original formulation of the problem that
allows for so much efficiency and improvement. PCA can be thought of as taking a large
matrix with many predictor variables, and finding a smaller number of those that are
sufficient for prediction. Solving this matrix rank minimization problem (while adhering
to the constraint that the given entries of \( M \) match the low-rank component’s entries) is
an alternative to SVD. PCA can be set up as the following optimization problem: Given
a matrix \( M \), one computes a low rank component \( L_0 \) satisfying:

\[
L_0 = \arg \min \| M - L \|_F^2, \quad \text{s.t. rank}(L) \leq k,
\]

where \( \| M - L \|_F^2 \) is the square of the Frobenius norm, or the sum of the squares of the
entries, and \( k \) is some desired maximum rank. Surprisingly, there are many advantages
to using the original optimization version of the problem. Take for example a brief look at speed. The SVD version of solving this problem takes $O(n^3)$ operations. For a matrix with $10^6$ entries, computing the SVD of such a data set would take, leaving room for a constant adjustment, approximately 10.6 years (assuming a processor capable of 3 billion floating point operations per second, using a naive implementation which could be improved upon). A $O(n)$ time version of the problem, which is offered by our current work, would take under one second (with room for a constant time adjustment). Clearly this is a substantial improvement, but, it comes with a cost, and that cost is a detailed understanding of the interaction between the precise optimization problems being solved, and the particular real-world domains of interest. Problem 1.1 is a member of the class of convex optimization problems, for which there are fast solvers available that we leverage, and it is the study of these solvers that is the core of our work.

Modern extensions of PCA [4, 12, 21] typically work on this optimization problem as a starting point, as it allows for these very fast computational approaches. As we will describe, extending this technique to problems where the matrix includes corrupted data gives us Robust Principal Component Analysis (RPCA), also solved using an optimization problem. Lastly, we will discuss the recently proposed eRPCA technique, which aims to solve similar problems for data which has some corruption or uncertainty in all points, and potentially large and varied amounts of it.

The abilities of PCA are impressive, but the algorithm is not without its drawbacks. Its aim is to take possibly correlated explanatory variables and transform them into linearly uncorrelated new variables. It is, however, sensitive to outliers. Onatitice e large outlier can lead to large errors in the computed principal components, which may not be representative of the rest of the data due to that one point. There have been many attempts to robustify the algorithm to those outliers. Weighted PCA [21] attempts to solve this issue by assigning weights to each point and incorporating those into the PCA computation. Robust PCA [4] is another attempt to fix this problem, and is the one we are pursuing. It is a technique that, in essence, takes a data matrix and recovers the previously mentioned low-rank component, and a sparse component. The sparse component $S$ represents the errors, grossly corrupted entries, or outliers that prevent the rest of the matrix from being low-rank. RPCA can be utilized in a number of important domains. Specifically, the literature cited discusses applications for image processing and computer vision [4, 22], and network signal processing [12]. The technique also
allow for extensions to situations when some data is missing and/or corrupted and/or noisy as well.

Finally, the present eRPCA algorithm\cite{12} is an example of an algorithm that solves a similar optimization problem to 1.1, in cost $O(m)$. That is, the cost per iteration is a function of the number of observed entries in the matrix $M$, as opposed to the total number of entries. This fact is vital to understanding the power of this algorithm, as it provides insight into how it may be applied to very large matrices which may not be complete. The goal is to expand this algorithm to one that can handle data on an exabyte scale. This problem is solved using convex optimization, meaning that, given time and memory, it will always converge to the global minimum—but, that convergence is not always fast. The actual rate of convergence is delicate, and there are a few different approaches one can take to substantially speed up the process. This is the main problem that we approach.

\section*{1.3 Experiments and Results}

We look into a number of different pieces of the eRPCA problem that would allow for improved matrix completion results, and accurate results that required less time and algorithm iterations. These included; parameters within the algorithm and how they could be changed based on the size of the data, exploring how the size and sparsity of the data affected the algorithm’s performance at a large scale, and how different error allowances changed the results from the algorithm. By comparing these parameters within the eRPCA algorithm, we are able to substantially improve the time it takes to solve the optimization problem, while maintaining accuracy. On synthetic data, we are able to improve the convergence rate of the algorithm by 2-4 times, when compared to previously used values from the literature\cite{23} for the parameters. Using the values of these parameters that our experiments deemed optimal, we were able to run the algorithm with success on large, real-world data sets and obtain results in a reasonable amount of time.
1.4 Our Contribution

In this work, we make several contributions to the current understanding of the matrix completion problem with noise. At present, there is theory supporting convergence rates and optimal algorithm parameters. In the specific case we worked on, though, the theory is lacking. The work completed here improves significantly on theoretical sufficient conditions to generate empirically necessary conditions. The use of these results is shown in their application to problems of practical importance.

These presented new empirical conditions improve algorithm performance substantially over previously known conditions. In particular, given the linear algorithm, the reduction in run time allows for much larger problems to be solved. This is shown through the our ability to run tests on a very large number of synthetic data examples.

The aforementioned cited works all contributed to our ability to show the effectiveness of these techniques on real-world data. This clearly demonstrates the use of such techniques, and provides motivation for even more experimentation. Promising results on this data show that in application, these algorithms can be used efficiently in large data problems across multiple domains.

1.5 Structure of Thesis

In this work we attempt to provide the reader with enough background information to fully grasp the problem. Chapter 2 describes the background knowledge required to understand the full extent of the work. Chapter 3 describes experiments and results on synthetic data. Chapter 4 contains results on data gathered from two real-world domains. Chapter 5 concludes this work with our advancements.
Chapter 2

Background Theory

Our problem is finding a low-rank matrix whose entries fit a given, incomplete matrix. Taking into account certain assumptions about that matrix, recovery, or generating the complete matrix following chosen modeling assumptions, is possible, and there are many theoretical results that guarantee this [18]. To fully grasp the task at hand, it is necessary to have a thorough understanding of all aspects of the problem, which we provide here. This includes low-rank matrices, the matrix completion problem in general, techniques used to solve the problem, i.e. PCA, RPCA, and eRPCA, and their corresponding algorithms.

2.1 Low Rank Matrices

The root of the problem we are solving in this work is finding low-rank matrices. While there are many equivalent ways to define matrix rank, the most relevant to this pursuit is in terms of singular values. For our discussion, we consider a matrix $M$ to be $n \times n$, although similar notions extend to rectangular matrices as well. Given the SVD of $M$, $M = U\Sigma V^T$, the singular values of $M$ lie on the diagonal of the matrix $\Sigma$. The rank of $M$ is equal to the number of non-zero singular values. Additionally, this rank is equal to the number of linearly independent rows (or columns) the matrix contains[19]. When the matrix of interest is called low rank, it generally comes within a specific context. A matrix that is $5 \times 5$, and is rank 5, i.e. only having 5 nonzero singular values, is a full-rank matrix, and can not in any way be considered low-rank, as it in fact has the
highest rank it possibly can. However, a matrix that is $1000 \times 1000$, with rank 5, is a contender to be low-rank. Even in that situation, though, one has to be cautious. Depending on context, as in a particular domain, that may not be considered especially low-rank. Similarly, when comparing to another $1000 \times 1000$ matrix, it is correct to say that one with rank 10 is of higher rank than one of rank 5, but that rank 10 matrix may actually be low-rank given its context.

Low-rank matrices are encountered quite often in real-world domains, in everything from recommender systems [5], social networks [24], Internet networks [25], and many more. Among data sets coming from all lines of business, low-rank matrices are ubiquitous. That is why the study of low-rank matrices is particularly applicable: any algorithm or technique utilizing them can be extended to real-world data and research. This is also why it is especially useful to attempt to find these low-rank matrices, including those that arise from an incomplete matrix.

### 2.2 Matrix Completion

The matrix completion problem begins with a fairly straightforward goal. Given a matrix $M$, and only a sample $m$ of its entries, the aim is to recover the remaining entries. In Figure 2.1, we show a toy example of this problem. The "?" entries in matrix A represent those that are missing from the matrix. We have no information about their value; only that they are a part of the data set, but are unobserved to us. Matrix B shows a potential completed version of matrix A. All the entries that were seen in A are given in B, with the missing entries filled in.

Although simple in statement, this problem is a hard one to solve, as those missing entries have the potential to be arbitrarily anything. To proceed, some basic but deep
assumptions need to be made about the data. A popular one is that \( M \) is low-rank, given the ubiquitous nature of such matrices. When the low-rank assumption is made, it in fact is possible to recover the missing entries given the set \( m \). In simple terms, that means that a complete matrix is returned such that all the given entries \( m \) remain, while those that were filled in are done so in such a way that the resulting matrix is low-rank. Generally this recovery is a very close approximation of the given entries, and in practice a user can decide how much error is acceptable. There are some additional requirements on the basic matrix completion problem to guarantee recovery, including the number of entries that must be observed (detailed in section A.1), their location within the matrix, and the “incoherence” of the singular vectors [1]. While these conditions appear complicated, they are surprisingly often satisfied in real-world data. These conditions though are not the focus of this thesis, and our work is on how matrices can be completed efficiently using convex optimization.

However, as an example for the interested reader, a problem that cannot be solved using these techniques is as follows. Given a matrix with rank 1, but that contains all zero’s except for one entry, it would be impossible to recover the correct matrix in any principled way, when only a portion of its entries were observed. Additionally, a matrix may be low-rank and well-conditioned for recovery, but the observed sampled entries given are not sufficient—for example, if they are all in one column.

To further investigate this problem, consider the singular value decomposition (SVD) representation of \( M \), \( M = U \Sigma V^T \). A low-rank matrix, then, can be thought of as one where the left singular vectors, the entries of \( U \) and the right singular vectors, those in \( V \), are selected uniformly at random from all possible orthonormal vectors [1]. There is a condition that is imposed on this matrix for its recovery. The singular values of the matrix \( M \) must be sufficiently spread out in order to solve the problem with the minimum number of entries observed [1]. This is the coherence condition, as defined in [1], and matrices must be of a low enough coherence, or incoherent, for recovery to happen.

### 2.2.1 Rank Minimizing Matrix Completion and Nuclear Norm

We denote by \( \Omega \) the locations of the observed entries \( m \), i.e. \( (i,j) \in \Omega \) if \( M_{(i,j)} \) is observed. We also use the set projection operator \( P_\Omega \), where \( P_\Omega(M) \) is the projection of
$M$ onto the set $\Omega$. This process maps the given elements $m$ of matrix $M$ to the set $\Omega$.

Given that sufficient conditions have been met, to recover that completed low-rank matrix $L$, given $M$, the following optimization problem can be solved:

$$\min \text{rank}(L), \text{s.t. } L_{(ij)} = M_{(ij)}, (i,j) \in \Omega,$$

(2.1)

If there is only one solution to the recovery problem, $M$ will be recovered. Unfortunately, this problem is NP-hard. In relevance to our pursuit, this means that the algorithm required to solve would be too expensive to scale to any substantial size. In particular, all known algorithms with exact solutions for this problem require time at least doubly exponential in $n$, and are therefore not practical for use [18]. Consider, instead, the alternative nuclear norm minimization problem

$$\min \|L\|_*, \text{s.t. } L_{(ij)} = M_{(ij)}, (i,j) \in \Omega,$$

(2.2)

This new problem presents a convex function to optimize, which can be done much more efficiently. First, it is vital to understand why this change in problems is allowed.

The nuclear norm of a matrix $X$ is defined as $\|X\|_* = \sum_{j=1}^{b}\sigma_{j}(X)$, or, the sum of the $kth$ largest singular values of $X$ [1]. Alternatively, the rank of the matrix $X$ is the number of linearly independent rows or columns of the matrix. This number is equal to the number of nonzero singular values of the matrix, or the number of nonvanishing singular values. The nuclear norm is the sum of those singular values, as the singular values of $0$ will not contribute to the sum.

One way to describe the relationship between nuclear norm and rank is that nuclear norm is a convex relaxation of rank [18]. Such convex relaxation operators are not our focus, but as a motivating example, consider a vector $x$ and a problem statement, minimize the zero-norm of $x$ such that $Ax = b$. We could try a version of $x$ that solves the constraint and has a zero-norm of 1 (lowest non-trivial possibility). If it does not obey the constraint, we could try a version of $x$ that has a zero-norm of 2. But this means checking all possible zero-norm 2 versions of $x$, or all possible vectors with non-zero values in two places. Then we would move up to 3, and so on. This problem is in fact NP-hard, so it is not something that can be pursued for vectors of any significant size.
Now consider instead minimizing the 1-norm of $x$. This is the sum of the absolute values of the entries in the vector $x$, or, $\|x\|_1 = \sum_{k=1}^{n} |x_k|$. A graph of the results of the 1-norm for $x$ would result in concentric diamonds around an origin, seen in Figure 2.2. This represents the level set of the 1-norm of each possible $x$. Fortunately, as explained by [18, 26], the constraint line $Ax = b$ will generally intersect these diamonds at vertexes, which aligns with the versions of $x$ that would provide a minimal zero-norm. Such ideas provide the foundation of the field of compressed sensing [27], and similar ideas also appear in lasso regression [28].

![Figure 2.2: The concentric diamonds represent the level set of the one-norm. The red line is the constraint, $Ax = b$. The constraint line generally intersects the level-set at a vertex, implying a minimized zero-norm.](image)

When applied to the PCA problem, it follows the exact same intuition. Originally, the problem is attempting to minimize the rank of the low-rank component $L_0$. The rank is the same as the number of non-zero singular values, so we can instead say minimize the zero-norm of the $\sigma$, the singular values. Given that we know this is equivalent (under the proper assumption) to minimizing the sum of the absolute value of singular values, we do that instead. And, that is exactly what the nuclear norm is: the sum of these always positive singular values. It is for these reasons that the matrix completion problem can be solved by minimizing the nuclear norm, as opposed to the rank itself, when searching for a low-rank solution. The convex relaxation is applicable to the eRPCA algorithm discussed shortly, as it is how that problem is solved as well.
2.3 RPCA

Expanding upon traditional PCA, Robust PCA was recently popularized by Candes, Li, Ma, and Wright in 2009 [4] to deal with corrupted data matrices. PCA matrix completion provides a solid basis for problems where this lower-dimensional space is desired, but they do not extend well to highly corrupted data. That is, the noise in the data must be relatively small, even if it is sparse. Small errors in the data will cause the PCA solution to have small changes in the singular values, as they are continuous functions of the entries of the matrix. In particular, the number of non-zero singular values gives the rank of the matrix, so if errors are small, and make only small changes to those singular values, then the true zero singular values will still be close to zero. However, large errors may make large changes to the singular values, and drastically alter the number of non-zero singular values. This makes the rank of the underlying matrix hard to approximate. RPCA provides the ability to solve the same problem as PCA, under relatively simple and broad conditions, for potentially largely corrupted data. The focus of RPCA is on decomposing the given matrix $M$ into $M = L_0 + S_0$, where $L$ is low-rank and $S$ is sparse. This sparse $S$ is analogous to the dense, but small-valued $N$ of the original PCA problem, but may contain large values, whereas the errors captured in $N$ are only small. If this decomposition were exactly possible, one could gain substantial knowledge about $M$ just by looking at $L$ or $S$, and also that the principal components are of the underlying low-rank matrix are discoverable even with corrupted data. Under some very basic assumptions about the rank and sparsity of $L_0$ and $S_0$ [4] that are deep in meaning but in practice often occur naturally in real-world data (see A.2), it turns out the matrix $M$ is recoverable [4].

The RPCA problem is solved by the following tractable optimization:

$$L, S = \arg\min_{L_0, S_0} \|L_0\|_* + \lambda\|S_0\|_1,$$

$$\text{s.t. } M - L_0 - S_0 = 0$$

where $\lambda$ is a constant, determined by the user. This is an extension of the PCA optimization problem that is solved using the nuclear norm, which is the convex relaxation of rank, and the one-norm, which is the convex relaxation of sparsity. In real-world
applications, it is very possible that the corruptions in M are of arbitrarily large magni-
tude. Because of this, RPCA is an extremely useful extension of PCA to corrupted or
noisy data that is commonly found in practice.

Figure 2.3: A visual example of Robust Principal Component Analysis. The original
video frames are decomposed into a low-rank component, showing the background
image, and a sparse component, showing the moving pieces of the image that change
with each frame.

(a) Original frames  (b) Low-rank $L$  (c) Sparse $\hat{S}$

As an example of an application of RPCA, we provide a visual demonstration, borrowed
from [18]. Figure 2.3 shows three frames from a security camera video. The first three
vertical pictures represents the given frames. To actually solve this problem, the values
of each pixel in the frame can be mapped into a column, and each frame’s column lined
up to form a matrix. The second set of frames represents the low-rank component output
by running RPCA. This translates to the background of the image. These components
are not changing substantially from frame to frame, and thus form a set of low-rank
columns. The last set of images represent the sparse components of the frames. Those
pixels that have substantial or large changes from frame to frame occur when someone
walks through the video, or otherwise is present only in a few instances. These show up
as sparse anomalies in the measure of pixel value.
2.3.1 RPCA Algorithm

A number of methods can be used to solve the RPCA problem. Theoretically, any algorithm for solving convex optimization problems could potentially be applied under the proper implementation circumstances. Methods discussed in the literature [4] include interior point methods and many other techniques. Our method of choice for this algorithm though is based off of using augmented Lagrange multipliers (ALM), presented in [23], and proximal approximation method, as detailed in [23] as well.

Lagrange multipliers are a very common technique used to solve optimization problems with equality constraints [29]. The Lagrange multiplier is a suitable alternative because it is a replacement for the original problem, using Lagrangian relaxation [29]. This method can be used to solve a hard optimization problem by relaxing the strict problem constraint, which, here, is that the entries of the recovered matrix match those of the given one. Instead, a Lagrange multiplier is put into the optimization problem, and any penalty necessary is shouldered by that term.

In addition, for accelerating convergence for RPCA, we use the Augmented Lagrange Multiplier method, and pursue the Lagrangian [23]:

\[ \mathcal{L}(L, S, Y) = \|L\|_* + \lambda \|S\|_1 + \langle Y, M - L - S \rangle + \frac{\mu}{2} \|M - L - S\|_F^2 \]  

(2.4)

To solve, the Lagrangian \( \mathcal{L} \) is first minimized with respect to \( L \), with \( S \) fixed, after which \( \mathcal{L} \) is minimized with respect to \( S \), with \( L \) fixed. Following, the Lagrange multiplier \( Y \) is updated based on upon errors in the constraint. The exact algorithm for this can be seen in [4]. This method is called the Alternating Directions Method of Multipliers, or ADMM [23]. As noted in [4], the dominating cost of each iteration of this algorithm is in computing the SVD when updating \( L \). In implementation, the choice of the parameter \( \mu \) and the stopping criterion (for convergence) greatly affect the number of iterations, which we explore extensively in Chapter 3.

2.4 eRPCA

RPCA provides a very robust method for dealing with the discussed noisy data, but as stated in [4] does not extend to the matrix completion problem. However, eRPCA
is an extension of RPCA that deals with globally noisy data, and can be used for the incomplete case. In particular, while RPCA allows for large error, eRPCA allows for pointwise variable error, where each entry of the matrix may have its own personal error allowance or noise estimate, rather than a global noise constraint [12]. In practice, this can mimic real-world situations more precisely than a global error constraint. Take for example the primary domain discussed in [12] and [13]. In sensor networks, it may be known that some sensors have larger error than others, based on domain knowledge about these systems. Encoding this prior knowledge about sensor accuracy can improve the depiction of the problem at hand.

Additionally, this eRPCA approach is applicable for problems with only a fraction of the data observed, which allows for significant amounts of missing data, another common occurrence in applications. Say that $\epsilon$ is a matrix of entrywise error bounds. That is, each entry in $\epsilon$ represents the absolute error allowance in each corresponding entry of the target matrix $M$. Then the matrix that the user is attempting to be found can be represented, using the projection operator, as

$$|P_{\Omega}(M) - P_{\Omega}(L_0 + S_0)| \leq \epsilon,$$  \hspace{1cm} (2.5)

The minimization problem solved by eRPCA is similar to that of RPCA, but modified as such [13]:

$$L_1, S_1 = \arg \min_{L_0, S_0} \|L_0\|_* + \lambda \|S_1(P_{\Omega}(S_0))\|_1,$$  \hspace{1cm} (2.6)

$$\text{s.t. } P_{\Omega}(M) - P_{\Omega}(L_0 + S_0) = 0,$$

$$L, S = L_1, S_1(P_{\Omega}(S_1)).$$

In this partially observed case, two common choices of $\lambda$ are $\lambda = \sqrt{\frac{n}{|\Omega|}}$, which accounts for the proportion of entries that are seen, or $\lambda = \frac{1}{\sqrt{n}}$, the traditional choice in [4]. The method of choice for solving this problem, as discussed above briefly, is ADMM, the same that is used in traditional RPCA.

The method of choice for our matrix completion pursuit is eRPCA because of how versatile it is. All of the problems presented so far may be solved with eRPCA. The matrix completion problem is solved by using a large value of $\lambda$ to encourage entries in
the low-rank component. The RPCA problem is covered by setting $\epsilon$, the point-wise error, to a constant.

2.4.1 eRPCA Algorithm

Like RPCA, eRPCA uses ADMM to actually implement the algorithmic solution. Here, the modified Lagrangian looks like [13]:

$$\mathcal{L}(L, S, Y, \mu) = \|L\|_* + \lambda \|S_{\epsilon}(P_{\Omega}(S))\|_1 + \frac{\mu}{2}\|P_{\Omega}(M) - P_{\Omega}(L + S) + \frac{1}{\mu}Y\|_F^2.$$  \hspace{1cm} (2.7)

At each iteration of the algorithm, the Lagrangian is optimized first with respect to $L$ with $S$ fixed, and then vice versa. In the step to optimize $L$, an SVD must be computed. See [12] for details on the algorithm’s implementation. Given the stated algorithm for use, each iteration of the optimization problem costs on the order of the number of observed and partially observed entries of $M$, both in time and memory [13]. Equation (2.7) runs in $O(n^3)$ time, so it must be modified to obtain that $O(m)$ efficiency. If rank is fixed, and a randomized SVD [30] is used in the computation of $L$, that can be reduced to $O(n^2)$. Storing the recovered matrices $L$ and $S$ would alone require $O(n^2)$ time, so that must be improved as well. Two tricks are used to solve this particular issue. First, we note that $S$ is a sparse matrix. Therefore it is straightforward to store it as such, instead of storing it as a regular, full matrix. Second, a slightly more complicated trick to exploit is based on the fact that instead of even storing $L$, instead, we may just store just its SVD. Given these modifications, the algorithm now runs in $O(m)$, as desired. This linear run time is what allows us to make significant modifications to the efficiency, as now even a constant in the algorithm can substantially change the resulting efficiency capabilities. The fact that you can use this modified version of the original problem to solve 2.7 is proven in [12]. The minimizer for one of these problems provides a minimum for the other, so solving the second problem’s modified Lagrangian is appropriate for solving the original.

2.4.2 Randomized SVD

For the reader unfamiliar with randomized SVD, we provide some basic intuition about what that is. When applying the ADMM algorithm, the most costly step is in updating
the approximation of $L$ at each iteration. At step $k$, there is a present approximated SVD for $L$, say $L_k = U \Sigma V^T$. Some sparse change $A_0$ is added to $L_k$, and the next step is to find the SVD of $L_{(k+1)} = L_k + A_0$, which is to say the SVD of $(U \Sigma V^T + A_0)$. SVD computations are very costly, and thus this is the slowest part of the algorithm.

Randomized SVD provides a matrix free version of this computation that greatly improves the time needed to solve. Presented in [30], the general scheme for computing the randomized SVD is as follows:

Given a target matrix $M$, which is $m$ by $n$, and a target number of singular vectors $k$, and an exponent $q$, usually $q = 1$ or 2, generate an $n \times 2k$ test matrix $\Omega$. Compute $Y = (MM^T)^qM\Omega$, and construct a matrix $Q$ whose columns are an orthonormal basis for the range of $Y$. Compute a matrix $B = QM$, and $SVD(B) = \tilde{U} \Sigma V^T$. Setting $U = Q\tilde{U}$ then gives $SVD(M) = UEV^T$. All details can be found in [30]. This entire process requires $2(q+1)$ passes over the matrix, making it very efficient. The details of applying the randomized SVD to the eRPCA problem can be seen in section 6 of [13].

2.4.3 Applications

The literature [12], [13] has so far focused on applying eRPCA for sensor network problems. A network’s data may present patterns of sparse correlations among nodes that do not follow the expected, low-rank background correlation. These anomalies may be evidence of network disruption, attacks, or other oddities. They can be short and high intensity, or longer and lower intensity. While using eRPCA to detect them does not guarantee that these anomalies are points of concern, it provides a simple approach to their detection, and allows them to be further investigated once found. Similar approaches are suggested across all types of networks, from surveillance to social. Our applications in this thesis are on a recommender system, and Internet tomography mapping.

The recommender problem provides an incomplete matrix of user ratings. The aim is to fit a low-rank complete matrix to those entries that are given. The low-rank assumption made in many matrix completion problems is appropriate here, because, in general, movie watchers tend to not be unique. One person who enjoys genres such as sci-fi and action tends to have ratings data similar to another user with those preferred genres.
This results in only a few rows needed in a linear combination to generate the others, and therefore, comes from a low-rank matrix.

Previous work on Internet tomography provides theory to show that a matrix of Euclidean distances (when measured on a plane) has rank at most 4 [25]. Therefore it is reasonable to assume that other distance measures will result in a similarly low-rank matrix. Hop distance, the number of nodes data passes through in transmission, is one area that can be studied to see if this indeed holds true, and is one that we pursue in this work.
Chapter 3

Experiments

3.1 Data Generation

For the purpose of the following experiments in this thesis, synthetic data was generated. Unless otherwise noted, this was done as follows:

Matrix dimensions $n_1 \times n_2$, true rank, and number of points observed $|\Omega|$ were chosen by the user. True rank of the generated data was selected as 1, and the maximum rank for the algorithmically-produced recovered matrix was 4. A random number generator generated coordinates $ij$ for $|\Omega|$ points within the dimensions. Two matrices $U$ and $V$ were generated to have dimensions of $n_1$ or $n_2$ by the chosen rank, and were filled with random numbers between 0 and 1.0 from a random distribution. For each randomly generated coordinate, the value at $U_i$ and $V_j^T$ were multiplied, and that was taken as the observed entry. A random 10% of the entries were given an addition of 1, to simulate sparse corruptions. The $\epsilon$ values were constant at $10^{-3}$. Unless otherwise noted, $\lambda$ was set as the given value from [4].

3.2 Finding Improved $\mu$ and $\rho$

In order to use the eRPCA algorithm, a value for each of the parameters involved is needed. In the Lagrangian function 2.7, we have the parameter $\mu$. In addition, we have $\rho$, which is not directly in the Lagrange, but is part of the algorithm. Both of these influence the number of iterations the algorithm will take to complete. Given that our
algorithm runs in $O(n)$, if we can halve the number of iterations the algorithm requires, we can solve problems twice as large in the same amount of time. For this reason, we spend much time on finding the best values of the parameters that allow us to lessen the iteration count. This study of convergence leads to an empirical starting point on which more theory can be developed and extended to larger problems. Ensuring the most efficient use of the algorithm is what allows us to proceed with advances on large data.

3.3 Literature Provided $\mu$ and $\rho$ Values

As a starting point, previous work [12, 13] used the values given in [23]. Our focus is an extensive empirical analysis of these values. Differing values for both parameters could produce a version of the algorithm that is significantly improved over one using those literature-provided values. Most notably, if different values drastically reduce the number of iterations the algorithm needs to run, there is potential for significant speed-up and the ability to solve much larger problems. It is relevant to note why the literature values are not necessarily optimal for our problem. To begin, they are geared towards a similar problem with substantial differences. They do not encompass all that we are working with, ie matrix completion, variable error, etc. In addition, we are solving the problem using an inexact method, described shortly. The literature is not. Therefore any improvement we make in convergence does not nullify the results given in the literature, nor does it mean they are not sufficient for their particular problem. Instead, we merely utilize them as starting points, as we must begin somewhere, and learn how we can improve on them for our problem at hand.

To fully understand what impact $\mu$ and $\rho$ have on the algorithm, we refer to the Lagrangian 2.7. $\mu$ represents the coefficient of the term that is added to consider the constraint of the problem. If $\mu$ is small, the multiplier on the constraint term is small, and thus the focus of the minimization is on the objective terms. Similarly, if $\mu$ is large, the constraint term is large, and the focus of the minimization becomes the constraint. To accurately solve the problem quickly, there must be a balance between the constraint term and the objective term. If one is too large, it will dominate the minimization problem, rendering the other term essentially irrelevant. If $\mu$ is too large, the constraint term may be minimized quickly, but the objective may take a long time to converge. If too
small, the objective may achieve a minimum value quickly, at the expense of ignoring the constraint, which then may not be solved.

As is known in the literature, there is no fixed value of $\mu$ that is optimal for the convergence rate. Therefore, at each iteration of the problem, there is the potential to update $\mu$. Because the algorithm may be satisfied with the intermediate value of the objective when starting off with a small value for $\mu$, $\mu$ may be increased to turn more focus to the constraint. This potential increase is the job of $\rho$. [23] provides rules for updating the value of $\mu$ at each iteration. Based off of their version, the rule implemented in eRPCA is:

$$
\mu_{k+1} = \begin{cases} 
\rho \mu_k & \text{if } \mu \|L_0 - L_\Omega - S\|_F / \|M\|_F < \epsilon_2 \\
\mu_k & \text{otherwise}
\end{cases}
$$

(3.1)

where $\epsilon_2$ is a selected stopping criterion. Here, $\rho$ will be greater than 1 always, as a value less than 1 would prevent the algorithm’s solving emphasis to move on to the constraint. Additionally, $\mu$ must be greater than 0 always, as a negative value would only increase the focus of the algorithm towards the objective.

The provided values for $\mu$ and $\rho$, which we call $\mu_0$ and $\rho_0$, in [23] are the values used in [12]. It is our aim to improve upon the work done in [12] and [13], and focus on manipulating these variables in an attempt to find values that would make the algorithm converge faster while maintaining acceptable accuracy. The given values are $\mu_0 = 1/\text{largest } \sigma_M$, and $\rho_0 = 1.2172 + 1.8588 \rho_s$, where $\rho_s = |\Omega|/|n_1 n_2|$, or the sampling density of the problem. (The relationship between $\rho_0$ and $\rho_s$ was found through regression.) These values, provided that one is using the exact ALM method, guarantee convergence of the algorithm [23]. However, we are using an inexact version, where we do not fully update our current estimates of $L$ and $S$ at each step. In our inexact implementation, convergence is not guaranteed, and the theory fails. This is why we must proceed using empirical studies. Instead, if the choices of $\mu$ meets specified conditions (mainly $\mu$ nondecreasing and $\sum_{k=1}^{+\infty} \mu_k^{-1} = +\infty$) stated in [23], the iterations will converge, but we have no details on the rate of convergence. This provides a starting point for studying that convergence rate. Given initial values taken from [23], page 12, we are able to manipulate them to understand how their size will change the convergence rate.
3.4 Convergence

Before proceeding, it is necessary to understand what it means to say that the algorithm has converged. Embedded within the eRPCA algorithm are two convergence parameters, $\epsilon_1$ and $\epsilon_2$. (In experimental practice in this paper, these values were based on those given in [23].) At some point, there needs to be criteria for the algorithm to stop. These $\epsilon$ values provide evaluation points for the algorithm to do that. In general terms, they indicate that, since the last iteration, the changes in the constraint and objective are not significant enough to continue making more updates, and that the present estimates are close enough. They are defined as follows: criterion 1 represents the constraint, and is calculated as

$$\frac{\|M - L_\Omega - S\|_F}{\|M\|_F} < \epsilon_1 \quad (3.2)$$

Criterion 2 represents the objective, and is calculated as

$$\min(\mu_k, \sqrt{\mu_k}) \frac{\|L_k - L_{k-1} - S\|_F}{\|M\|_F} < \epsilon_2 \quad (3.3)$$

If both of the above thresholds are satisfied, the algorithm will stop, and we say convergence has been achieved.

Thus, if the values of either $\epsilon$ are changed, the number of iterations will change. A larger $\epsilon_2$ will mean that less progress in minimizing the objective between iterations is required for the algorithm to complete, and convergence will therefore be faster. A larger $\epsilon_1$ allows the constraint to be less well-satisfied. Smaller $\epsilon$ values will ensure higher number of iterations are required, as even minute progress between iterations will push the algorithm to continue.

3.5 Varying $\mu$ and $\rho$

Given the eRPCA problem is geared towards partially observed matrices and those with varying error constraints, it was worthwhile to investigate which values of $\mu$ and $\rho$ would potentially produce results that required less algorithm iterations. The ADMM method in [23] provides two versions of solving: exact, and inexact. Here, we are forced to use
the inexact method to achieve the desired performance and functionality, because our particular objective of interest does not have a closed form solution for the minimum of $L$ with $S$ fixed. The exact method provides a proof for convergence, and theoretical values for $\mu$ and $\rho$ that provide convergence, but they do not provide any insight into the exact convergence rate. We instead use the inexact version, and do an empirical study to determine advantageous choices of the parameters under that case.

To begin, the values of $\mu$ and $\rho$ were varied, based off of the original ones provided in [23]. For $\mu$, the original value was multiplied by powers of 2 ranging from .0625 to 512. For $\rho$, the value was tested for the range $\frac{\rho - 1}{8} + 1...64(\rho - 1) + 1$. Matrix size varied, as did the percent of observed entries. Figure 3.1 shows initial results, starting with a matrix of 1,000 by 1,000 entries.

To initially find values of $\rho$ and $\mu$ that were an improvement over the ones in the literature, we looked at the number of iterations the algorithm required to satisfy the $\epsilon_1$ and $\epsilon_2$ thresholds. The 'best' value under this criterion, for any fixed $\epsilon_1$ and $\epsilon_2$, would be to choose arbitrarily large values of $\mu$ and $\rho$ for any matrix, and indeed we see that in our work. This result is fairly intuitive, because as $\mu$ grows, the Lagrangian will have a larger dependence on the constraint term. It is then straightforward for the algorithm to satisfy solely the constraint, and with that term dominating, the algorithm can converge very quickly. We see in these results, an increased $\mu$ value will produce a solution that requires fewer iterations than a smaller $\mu$ value. Unfortunately, satisfying fixed constraints with $\epsilon_1$ and $\epsilon_2$, does not guarantee that the solution is useful, since the value of the Lagrangian depends on $\mu$. In particular, for the same fixed $\epsilon$ values, the objective may not be well minimized, so one has to be judicious when interpreting what "converged" means for large $\mu$ values. This is evinced in Figure 3.4b. Note that the theory in [23] is true, and that you are guaranteed convergence, eventually. However, it may happen very slowly!

The largest values of $\mu$ benefit the time and iteration performance of the algorithm, up until a plateauing point witnessed at approximately $128\mu_0$. Additionally, as seen in Figure 3.3, the iterations also decrease as $\rho$ decreases. However, the change is not as pronounced, as the iterations are much more dependent on the value of $\mu$.

Together, these results indicate that the largest value of $\mu$ and $\rho$ paired together is the most efficient choice. However, for values of $\mu$ that large, one would expect the
Figure 3.1: Results for a 1,000 by 1,000 matrix with 10,000 entries. The color scale shows that the algorithm runs with the least amount of iterations (dark blue) at the largest values of $\mu$ and $\rho$. However, the contour lines show that the objective value is no longer minimized when that $\mu$ value is too large. The red dot represents the original $\mu \rho$ pair. The green dot represents the $\mu$ and $\rho$ pair that run the algorithm with the least amount of iterations, while minimizing the objective value.

Figure 3.2: Plot of $\mu$ vs. iterations for the same matrix. This plot clearly shows the decline in number of iterations as the $\mu$ value grows, up until a plateau at approximately $128\mu_0$. There is some variation within particular values of $\mu$, as each is paired with multiple $\rho$ values.

algorithm to fail in its objective pursuit, as it would be focusing solely on the constraint for such a large $\mu$. When $\mu$ is that large, the constraint term of the Lagrangian becomes correspondingly large. Thus, in attempting to minimize, the algorithm needs to assert much more care in minimizing the constraint term. The objective term is comparatively small. This failure does indeed occur. The contour lines in plot 3.1 represent the objective value. It is clear that the objective value begins to increase when the value of $\mu$ has passed a certain threshold. This occurs when the algorithm has met the thresholds
Figure 3.3: Plot of $\rho$ vs. iterations for the same matrix. This plot shows a decline in number of iterations as the $\rho$ value grows. There is large variation among iteration counts at each value of $\rho$, as each was paired with multiple $\mu$ values.

$\epsilon_1$ and $\epsilon_2$ for convergence, but has not actually solved the minimization problem. With every iteration, the estimates of $L$ and $S$ are slightly changing. $\mu$ affects the scale of the criterion that is compared with $\epsilon_2$. Thus, $\mu$ changing will affect this portion of the convergence.

Given the need to both solve the objective correctly, and do so efficiently, the problem then becomes finding the best values of $\mu$ and $\rho$ that not only lessen solving time and iterations, but still minimize the objective. This is a minimization problem, in which we require the minimum objective, so not satisfying that is not an acceptable solution. Again we see much more variation in the measures dependent on $\rho$, as it has less direct impact on the algorithm when compared with $\mu$.

Digging deeper, we can see that the value of the optimal objective becomes significantly larger with the value of $\mu$ (for fixed $\epsilon_1$ and $\epsilon_2$), and somewhat less so with the value of $\rho$, seen in Figures 3.4b and 3.5. The iteration count is included so we can see the relationship between the two measurements.

In the plot where $\mu$ is tested, we can see the heart of our solution in Figure 3.4. This Figure shows both the decreasing iteration count and the increasing objective value. Where the objective is at a minimum, we have an available range of $\mu$ values we can utilize to solve our problem. This Figure also marks the value of $\mu_0$ with a vertical black line. There is a range of $\mu$ values that provide both the lowest objective value
Figure 3.4: Log plot of $\mu$ vs. Iterations and Objective Value. The red dots represent the number of iterations the algorithm takes to solve the problem. The blue dots represent the value of the objective. The original value of $\mu$ is marked on both plots. In plot 3.4a, we point out the improvement in the number of iterations in our best pair of $\mu$ and $\rho$, compared with the original values used. The original $\mu$ is marked with a line in plot 3.4b.

Figure 3.5: $\rho$ vs. Iterations and Objective Value. The red dots represent the number of iterations the algorithm takes to solve the problem. The blue dots represent the value of the objective. The original value of $\rho$ is marked with a line.

obtained, and a low number of iterations. This range encompasses the values of $16\mu_0$ and below. Within that range, there is one run of the algorithm that uses the least number of iterations, depending on the selected value for $\rho$ that $\mu$ is paired with.

Listing the $\mu$ and $\rho$ pairs that give the minimum objective, and finding the pair of those that solves within the least iterations, gives us the values of $8\mu_0$ and $64(\rho_0 - 1) + 1$. The pairing of these values is vital, as they work in tandem to affect the algorithm. Simply choosing $8\mu_0$ without regard for $\rho$ can result in a large range of iteration counts; here,
between 18 and 43, although all pairings do result in a minimized objective. Choosing $64(\rho_0 - 1) + 1$ with no care for choice in $\mu$ can result in solutions ranging from 3 iterations to 63 iterations, but the objective is not minimized in many of those cases.

In the preceding discussion, we have been working with an example matrix of size 1,000 by 1,000. For our purposes, this is a very small example, and serves only to demonstrate our results. We proceed with much larger examples to truly demonstrate the power of the algorithm on large data sets. Additionally, the parameter values we deem best, in terms of convergence rate and objective minimization, are robust to matrices of different sizes. There is some minimal variation around the iteration counts within a small range of $\mu$ values.

In this particular example, we see the value of these experiments. Simply choosing the original $\mu_0$ and $\rho_0$ values will solve the problem. However, it can be done much more efficiently! In this case, using $\mu_0$ and $\rho_0$ requires 58 iterations of the algorithm, while the best pair found experimentally only requires 18 iterations. This observation is the crux of the $\mu$ and $\rho$ experiments. Certain choices of the parameter values can result in solvings that take impressively small numbers of iterations, and require very little time to compute. However, these do not actually provide solutions to the minimization problem. Instead, we must choose within values that do provide a minimized solution. Unfortunately those that do not are of little use for the sake of our problem, when the goal is the recovery of the low-rank and sparse components for matrix completion.

Instead, the situations where the problem is not actually solved can help to provide benchmarks for sanity tests. We can run these versions and check briefly that the algorithm is indeed not solving the problem by checking the results of perturbing the solution. At present, [23] provides $\epsilon_1$ and $\epsilon_2$ to define convergence. In fact, these values are not necessarily correct. As mentioned, they measure if the change in the objective or constraint is large between iterations. So, for example, if the change in the objective is minimal, the algorithm may complete, as it has converged. But, along the way, there is the possibility that the objective is not actually minimized, even if inter-iteration change is minimal. For this reason, we check a posteriori the value of the objective, and compare to a slightly different value. While this does not guarantee a minimum, it does show when the objective is not at a minimum.
If the objective has been successfully minimized, a minor perturbation in the provided recovered solution would result in a higher objective value. In the case of our problem, we do this by slightly changing the singular values of $L$ that are returned. To ensure we preserve the constraint, we also then must perturb $S$ accordingly. If we call the singular values of $L$ returned $\Sigma_{L}$, we make small changes to $\Sigma_{L}$, and recalculate $L$ as $\hat{L} = U\Sigma_{L}V^{T} = L - \gamma$. This requires a change to the original $S$, as in $\hat{S} = S + \gamma$. We used these new $\hat{L}$ and $\hat{S}$ to calculate the new objective value. When the objective is minimized, after perturbation, we will only see values of the new objective (using $\hat{L}$ and $\hat{S}$) that are somewhat higher. For those where the objective is not minimized, the objective value after perturbation could be higher or lower. This provides a simple way to see that the algorithm is indeed doing its job. It is relevant to note that there is a chance that this method has only found a local minimum, and not a global one.

The proceeding results are all obtained from a test case where the matrix is of size $1,000 \times 1,000$, with $10,000$ points observed, or $1\%$. For matrices of larger dimensions, and with different percentages of observed entries, we see slightly different results in the best values of the parameters, but they are all within a similar range (i.e., $2\mu_{0}$ to $8\mu_{0}$). There is a similar range of values for $\rho$ that would produce the minimum objective. When we increase the size of the matrix to $10,000 \times 10,000$, or $100,000 \times 100,000$, we see patterns very similar to the $1,000 \times 1,000$ size, as show in Figures 3.6, 3.7, and 3.9. In a few cases, such as that shown in Figure 3.8, the literature-provided pair of values for $\mu$ and $\rho$ did not fully minimize the objective. Indeed, while convergence is met based upon the $\epsilon$ values given in [23], their rules do not guarantee that the objective is at a minimum. Instead, it merely renders convergence to respond to change in objective value. Furthering the theoretically or empirically sufficient values, or rules, for $\epsilon$ and convergence are another remaining area to pursue.

3.6 Improvement Over Values in the Literature

The experiments run show that, in general, the literature-provided values of $\mu$ and $\rho$ as used in [23] are not the best choices for these cases. There is always room for improvement, and often by a factor of at least two, which we see in the table below. This allows the user to solve problems that are twice as big, given the same resources and
Figure 3.6: Results for a 100,000 by 100,000 matrix with .1% entries observed. The blue is where iteration counts are low, and lighter colors, where iteration counts are higher. The contours show the value of the objective, which needs to be minimized. The red dot represents the previously used $\mu$ and $\rho$ values, while the green dots represent multiple improved experimental values that minimize the objective.

Figure 3.7: Results for a 10,000 by 10,000 matrix with 1,000,000 (1%) entries. The color scale shows that the algorithm runs with the least amount of iterations (dark blue) at the largest values of $\mu$ and $\rho$. However, the contour lines show that the objective value is no longer minimized when that $\mu$ value is too large. The red dot represents the previously used $\mu$ $\rho$ pair, while the green represents the best experimental value pair found.

time. This substantially impacts the size of the problem that the user is able to handle, given that the algorithm runs in linear time, and can have significant implications when applied to real-world data.
Figure 3.8: $\mu$ and $\rho$, colored by the objective value. Contours represent the number of iterations the algorithm took to complete. The blue values represent a minimized objective, while those tending towards red are no longer minimized. The red dot represents the previously used $\mu \rho$ pair, while the green represents the best pair that minimizes the objective. In this case, the literature-provided pair did not minimize the objective value. The patterns seen here are an anomaly in our experiments, and offer chance for further explanation.

Figure 3.9: $\mu$ and $\rho$, colored by the objective value. Contours represent the number of iterations the algorithm took to complete. The blue values represent a minimized objective, while those tending towards red are no longer minimized. The red dot represents the previously used $\mu \rho$ pair, while the green represents the best pair that minimizes the objective. While all runs of the algorithm were relatively low in iterations, iteration time itself was high, so it is still important to aim for a low number of iterations.
*In some cases, $\mu_0$ and $\rho_0$ did not minimize the objective. Based on one run of the data with each pair of parameter values.

3.7 Additional Experimentation

3.7.1 Varying Point-Wise Error Bounds

The large advantage eRPCA has over traditional RPCA is that it can accommodate for varying error bounds. This means that some points may be observed, and their values are set in stone as correct and unchangeable, while others may be "observed", or partially observed, yet there is some allowance for their error. The fully observed points have an error allowance of zero. The partially observed points have some positive value given for their error bound. Testing various levels of error can also provide more information to the user about what performance they may expect from the algorithm. In Figure 3.10,

![Figure 3.10: (Normalized) iteration count vs. a constant error allowance. Each line represents a different matrix dimension or different percent of entries observed. Regardless of size and entries given, we can see that as the constant error allowance grows, the number of iterations required to solve decreases.](image)

we can see the number of iterations required as a function of the amount of error each point has. This is for a constant error allowance. As the amount of error each point is allowed increases, the number of iterations required to solve the algorithm decreases. This does confirm what a user may expect, intuitively. When a point has a larger error bound, there is more flexibility allowed in the algorithm’s need to fit its solution to that...
point. So, overall, it will be an easier, quicker process to fit the recovered $L$ to the given points, if larger error is permitted. When the error matrix $\epsilon$ is set constant and reported correctly to the algorithm, RPCA and eRPCA perform similarly [12]. However, when varying the error, as the matrix $\epsilon$ does for the eRPCA algorithm, there is significant improvement in recovery over traditional RPCA [12].

3.7.2 Varying Dimensions and Percent Observed

When solving problems on such potentially large scales, it is vital to understand the most efficient way to approach them before even beginning. This means going into the problem with some expectations of how long and hard it may be to potentially solve. It has been shown that eRPCA can handle problems on larger scale, and with different amount of the data actually available for use, and can be made more and less efficient when changing the few variables within the problem. What matters deeply is also understanding where the data’s size influences the problem. In both the size of the entire possible matrix the observed data is coming from, and the size of the observed set of points, there will be differences in run time and in number of iterations. In appendix A.4, we generate examples to confirm results proven in [13]. These help to solidify our understanding of the algorithm in practice. Additionally, they are an example of data-driven results that support the theory. These results show that iteration time is approximately constant, when a constant number of entries are observed.

The attempt to develop theory around the number of iterations any problem will require can result in more robust expectations of how long it will take to solve. This is especially useful if there is a limited amount of time or resources available. This can include knowing an expected iteration count. Here, we ask that if there is a limited amount of time, but a constant number of entries, does the size of the matrix we obtain those entries from affect iteration count?

The constant seen in iteration time for the same number of observed data points held constant does not translate to the actual number of iterations. Instead, a matrix with larger dimensions takes less iterations to solve, given the same number of observed entries, than a smaller matrix. This is because there are more columns in the larger matrix that the data is originated from. With more dimensions, when trying to fit a low-rank component, there are more places to put the additional entries, rather than
fitting them tightly into the smaller matrix, and maintain the desired low rank. This is evinced in the tests with results in Figure 3.11.

Put in other words, there is not a constant number of iterations when the percent of observed entries is held constant, but the matrix size changes. This knowledge can help a user judge their requirements for a problem. While their data may be limited, given time constraints, it could be beneficial to use a smaller portion of what is available, to guarantee a smaller number of iterations. This is dependent on user and domain needs, but does provide a chance for greater efficiency if the domain allows for it.

Figure 3.11 originates from data with true rank 1, and maximum allowed rank 4, meaning the algorithm may produce an $L$ with at most rank 4. On data that has a true rank of 4, and a maximum allowed rank of 16, we see similar patterns in the run time per iteration, as shown in Figure 3.12. However, in Figure 3.13, we see a slightly different pattern for data with true rank 16. In fact, this is the same situation as seen in the rank 1 and rank 4 data, but, the pattern becomes more pronounced with this higher rank. As the matrices begin to have larger dimensions, but the same number of entries, we see the required number of iterations go down. In cases with very large matrices that are extremely sparse, we do expect rank to have an effect on these results.
Figure 3.12: Number of iterations, compared with the number of entries observed, and what percent of the data they represent. This data originates from matrices with true rank 4, and maximum allowed rank (in the algorithm) of 16. As the number of observed entries is held constant, but the percentage that they represent grows, the algorithm requires more iterations to solve.

Figure 3.13: Number of iterations, compared with the number of entries observed, and what percent of the data they represent. This data originates from matrices with true rank 16, and maximum rank allowed 64. As the number of observed entries is held constant, but the percentage that they represent grows, the algorithm requires more iterations to solve.

3.7.3 Allowance of Large Errored Entries to Be Unobserved

Left for further work is analysis on manipulating the given error bounds. As noted, in the eRPCA problem, the target matrix $M$ is not always fully observed. The algorithm itself is shown in [13] to run in $O(m)$, where $m$ is the number of observed entries, in both
time and memory. This includes entries that are fully observed with no error allowance, and those that have some error allowance captured in the matrix $\epsilon$. The only place where points are not contributing to run space/time is points that are not observed at all. That is, in the eyes of the algorithm, a fully observed point with no error is just as expensive as a partially observed point.

In practical applications, this fact is important. In a specific domain, some error allowances could be given to certain points based off of prior knowledge. Certain values for the error are small, but some are very, very large. That is, the point is considered observed, but the error bound on it means it potentially falls within a very large range of possible values it can take. The algorithm must fit its solution through it, as we require that $P_\Omega(M - L_0 - S_0) = 0$. It is possible that, given the domain, the user knows that a point with a very large error allowance, is for all intents and purposes, unobserved. If this occurred, the user could remove that point fully, and run the algorithm on a smaller data set, lessening both the time and memory cost. This would, naturally, be advantageous. But, it would require the user to know what threshold of those error bounds makes them large enough to render the point essentially unobserved.

One possible way to deal with this problem would be the following: choose an arbitrary (although smart) choice of a threshold for $\epsilon$, called $g$ (or as an alternative, pick the largest $j\%$ of $\epsilon$ values, for some $j$). For all points $m$ in $\Omega$ where $\epsilon_m > g$, remove them from $\Omega$. Run the algorithm on this new, smaller set of points, called $\Omega_g$. Then, check that the values given in the recovered matrix $L + S$ at those points is within their original value, plus or minus $\epsilon_m$. One would expect that initially, some points in the recovered matrix may not fit correctly. The next step would then be adding those points back into $\Omega_g$, and run the algorithm again. This process could be repeated until all values are correct, or close enough, to the user’s satisfaction. In the end, the user would have a value for $g$ that produces sufficient results for them.

This process appears to lose the advantage of time savings, as the algorithm must be run over and over. Given the data and the original choice of $g$, this is certainly possible, and may in fact cause for much more time spent solving the problem. If, however, the user knew this data, or similar data from the same domain, needed to be dealt with multiple times and the algorithm applied to similar data again, this process may have its advantages. It would be used to set a threshold $g$ for $\epsilon$ that renders a point essentially
unobserved in the particular domain, and would potentially speed up future uses of the algorithm within that domain, if those points would be discarded prior to solving.
Chapter 4

Applications

In these applications, we attempt to show that the work completed on synthetic data can be extended to real-world domains. These represent two very different areas of research that could benefit from using algorithms such as the ones we have been discussing. Our results indicate that similar patterns are seen across synthetic data and real-world data, and that real-world data presents somewhat more varied results.

4.1 MovieLens Data Set

The first domain we applied this work to was recommender systems. A recommender system is, generically, any algorithm or filtering system that is attempting to predict how a user will rate a service, product, movie, or other offered element [5]. To an external user, this looks something like "you might enjoy this movie", or, "other products you might enjoy are". From the system’s view point, this is a prediction problem. Based on the user’s past history and ratings, what will they rate a product they haven’t experienced yet? If it is predicted they will rate it high, then it will be recommended.

The MovieLens data set has been collected by the GroupLens Research lab at the University of Minnesota, Twin Cities, since 1998 [31]. The data consists of basic user demographic information (in some cases), and their ratings of various movies. They provide data sets in a range of sizes, from approximately 100,000 ratings to 20,000,000 ratings. In particular, we focus on applying the eRPCA algorithm to subsets taken from their data set containing approximately 1 million entries. As we are not working with
anything beyond ratings, the only pertinent information was the ratings themselves, and not the user demographic information.

The MovieLens 1M data set contains 1,000,209 movie ratings from 6,040 users, and 3,952 different movies. In this work, we sampled that data into two other matrices. Matrix $A$ is 1,000 by 6,000, with 242,726 entries observed, for approximately 4%. Note that matrix $A$ is a departure from the usually square matrices we have been working with thus far. Matrix $B$ is 600 by 400, with 10,377 entries observed, for 4.3% observed. These dimensions maintain the approximate ratio of the original data set.

For matrix $B$, Figure 4.1 represents the same parameter evaluation we performed in the experimental data. In this case, the literature-provided $\mu$ and $\rho$ values require 95 iterations of the algorithm. Our best choice within our typical range requires 54 iterations. The values used to produce that were $4\mu$ and $64(\rho - 1) + 1$. This is well within the usual pattern of best values we saw in the synthetic data. To ensure we have entries in the low-rank $L$ component of the recovered matrix, we used twice the $\lambda$ value from the literature (otherwise, all entries in the recovered matrices $L$ and $S$ are put into the sparse component $S$). Additionally, we evaluated the number of nonzero singular values of the recovered $L$ component to help determine rank. Consistent results on multiple allowed maximum ranks showed the data to have a true rank of 13, which we do consider as low-rank in this context.

![Figure 4.1](image-url)

**Figure 4.1**: Number of iterations, represented by the colors, as objective value, represented by the contour, for varied $\mu$ and $\rho$. The patterns seen here are very similar to those we saw in the synthetic generated data. The best pair of $\mu$ and $\rho$ is marked with a green dot. The pair provided in the literature are marked with a red dot.
Matrix $A$ allows us to test our chosen parameter pairs and compare with those provided in the literature. This represents approximately one quarter of the original data. Using our chosen values of $8\mu$ and $64(\rho - 1) + 1$, the algorithm converges in 41 iterations. Using the literature-provided values, it requires 77 iterations, so we again see that there are more efficient values to be found using our experiments.

What this work demonstrates is that based on experimentation on synthetic data, we can choose appropriate values for our parameters on real-world data. Additionally, we see patterns in real data similar to those we see on the synthetic. One issue that is present on real data, though, and not synthetic, is that there may be further restrictions placed on the data. Our current data set provides an example of this. The entries in the matrix represent movie ratings, and therefore only belong to a small bounded range of values that users may select as a rating. When entries are generated in the returned, recovered $L$ and $S$, they may be outside that range. In Figure 4.2, we can see that while the provided observed values were within the range 0 to 5, some of the predicted values lie slightly outside that range.

![Figure 4.2: The recovered matrix $L + S$, for matrix $B$. The original observed values lie within the range of 0 to 5. The recovered values lie slightly beyond those bounds.](image)
There are a few possible ways to handle this problem. First, given the $\epsilon$ error bounds, the user may manipulate them so as to force all entries within a certain range. Second, there are possible extensions to PCA that attempt to model logistic regression [32], which forces entries to be percentages within 0.0 and 1.0. This can be extended to generating probabilities for belonging to a number of classes (such as ratings of 0, 1, 2, 3, 4, or 5, in our case). Like our work does, this work returns to the optimization roots of PCA in order to meet its goals. This provides yet another example of the versatility of that original PCA inception, and ideas from it could potentially be combined with our method to work on bounded data.

4.2 Internet Tomography

The second domain we applied our work to was a group of data sets demonstrating Internet tomography. This is the study that deals with mapping the Internet. The Internet can be thought of as a series of interconnected nodes that send and receive information from each other. There are various distance measures that can be used to describe this node graph. Euclidean distance is one. Hop distance, or, the number of nodes data must pass through to reach a target node, is another. Transmission or latency time is another measure that can be used as a proxy for distance. Currently, theorems have been developed stating that the Euclidean distances between nodes form a matrix with at most rank 4 (assuming the nodes lie on a plane) [25]. This is a low-rank matrix, which is exactly the type our work has focused on. However, there are not currently similar theorems about hop distance or transmission time, though preliminary results indicate these problems are low-rank as well. When distance is measured across a large network, it is often not likely that a full matrix of distances will be available. This is in part due to the fact that it would simply take a long time to measure all the possible distances. So, we begin with an incomplete matrix, for which we want to fill in the remaining distances between nodes.

When working with synthetic data, we generated our entries such that they were small (Section A.3 addresses this process). In real-world domains such as this, it is not always the case the the value of each entry may be similar to ours, nor that it be within any prescribed range. In the case of our generated data, the mean of the observed entries was approximately .36. In the case of this data, the mean of the observed entries is
An absolute change in the objective value is then less appropriate to evaluate as a relative change. This is the reason that $\epsilon_1$ and $\epsilon_2$ are normalized by $\|M\|_F$. In an attempt to account for the actual value of the data points, the convergence criteria must depend somewhat on the actual values in the objective. As an example, an objective value composed of data that falls within the range 0.0 to 1.0 will be smaller than one with data between 100 and 1000, given everything else is held constant. To fully address this, and to continue to strengthen the theory behind choosing $\epsilon_1$ and $\epsilon_2$, we would need to proceed with an in depth analysis of the appropriate criteria, which is left for future work.

The data we used for this application was generated to represent potential hop distances between nodes. We used matrices of four different sizes for this particular application. Matrix $A$ was 100 x 100, with 270 observed entries, for 2.7%. Matrix $B$ was 10,000 x 10,000 with 10,478 entries, for .01%. Matrix $C$ was 10,000 x 10,000 with 1,105,612 entries, for 1.1% observed. Matrix $D$ was 1,000 by 1,000, with 10,000 entries observed, for 1% observed. Note that matrix $A$ and matrix $B$ are not so similar in dimensions to the matrices we dealt with in test data. This is to allow us to explore the applicability of our work on a wider variety of possible matrices similar to what we might encounter in real-world domains.

All of these matrices produced interesting results, so we discuss them each in turn. Matrix $A$ is substantially smaller than any of the data we dealt with in experimentation. Indeed, it is smaller than much of the data we would expect our results to extend to, and serves only as a simple toy example. In Figure 4.3, we see that patterns do appear similar to what we encountered in the synthetic data. In this case, though, our range of possible values for $\mu$ and $\rho$ that have a low iteration count and minimize the objective is fairly large. There are a number of pairs that would accomplish the task at hand in a reasonable number of iterations, the lowest of which are marked. This includes both our experimental values of choice, $8\mu$ and $64(\rho - 1) + 1$, as well as the values provided in [23]. We attribute this to the size of the data. Such a small matrix does follow the results we propose, but allows for more departure from the consistent results.

Matrix $D$ imitates patterns we found in the synthetic data fairly closely. Using $\epsilon_1$ and $\epsilon_2$ values ten times smaller than those provided in the literature, we see the results in Figure 4.4. This graph also shows us the worst values of $\mu$ we have yet to witness.
Figure 4.3: Number of iterations, represented by the colors, and objective value, represented by the contour, for varied $\mu$ and $\rho$. The patterns seen here are very similar to those we saw in the synthetic generated data, although there is a wider range of acceptable parameter values. Those marked in green all have the same minimized objective value, and the same least iteration count.

When $\mu$ is much too big for the problem, our iteration count and objective value grow substantially. We also see the results of our experimental work fall in line with these. For this matrix, the literature-provided pair of parameters resulted in 42 iterations, while our choice of $4\mu$ and $32(\rho - 1) + 1$ resulted in only 17, an 2.5 times improvement. This shows that our empirical results can and do extend to real-world data.

Figure 4.4: Number of iterations, represented by the colors, as objective value, represented by the contour, for varied $\mu$ and $\rho$. The patterns seen here are very similar to those we saw in the synthetic generated data. The literature-provided pair of parameters is marked in red, while our experimentally derived pair is marked in green.
Matrix $C$ represents a chance to compare our selected parameter values with those provided in the literature. With again smaller values of $\epsilon_1$ and $\epsilon_2$, now by a factor of 100, and using our selected values of $8\mu$ and $64(\rho - 1) + 1$, the algorithm converged within 76 iterations. In the case of the literature-provided values, the algorithm never reached convergence. We required the algorithm to stop after 1,000 iterations, so while it may converge eventually, it does not do so in the allowed iterations. Again, this demonstrates the use of our experimentally derived parameter values, in aiding in algorithm convergence.

Matrix $B$ is one that is much more sparse than we experimented with on synthetic data. With only 10,000 entries total, there is approximately one entry per row or column. In this extremely sparse case, we see similar patterns to what we previously noted in the experimental work, but with one caveat made clear. In Figure 4.5, it appears that the objective value is minimized for a much smaller range of possible $\mu$ and $\rho$ pairs. This is a consequence of working with real data. In this situation, there are many small details of the problem that we must consider.

First, we address the objective value. As noted, there is a small range for which it appears to be minimized. Both our experimental choice of $\mu$ and $\rho$ values, and those provided in the literature [23], do not lie in this range. This is a consequence of how convergence is determined, and essentially serves as a symptom of the problem at hand. We initiated the a posteriori objective test noted in Chapter 3, due to the possibility that the algorithm may complete, and deem a run converged, but not having yet solved the objective. This results from what the convergence criteria $\epsilon_1$ and $\epsilon_2$ have been set to. What those appropriate convergence values are, for such a sparse data set, is still left to be determined. We used the values provided in [23] for our tests, but it prompts the question of how meaningful $\epsilon_1$ and $\epsilon_2$ are in the case of such sparse data. In particular, we determine that this data is not reasonable to consider with the methods at hand. This is an important result, as it helps to both demonstrate the limitations of the method, and prompt further experimentation into the problem.
Figure 4.5: Tests run on an extremely sparse data set, showing number of iterations, represented by the colors, and objective value, represented by the contour, for varied $\mu$ and $\rho$. The patterns seen here are very similar to those we saw in the synthetic generated data. There is a smaller range of minimized objective values based on parameter pairs, and both our experimental parameter values (green) and those provided in the literature (red) do not fall in that range.
Chapter 5

Conclusion

5.1 Contributions

The matrix completion problem is seen many times in real-world domains, and thus is a pursuit worthy of more research and experimentation. When there is an opportunity to improve upon solving the problem under various situations, and doing so more efficiently, it presents a chance to see great advancement. In this work, we showed that with judicious choice of parameter values, the eRPCA algorithm can be efficiently applied to matrix completion problems of larger scale than before. Particularly, we saw 2, 3, and sometimes 4 times speed up in test runs of the algorithm on synthetic data with those preferred choices of $\mu$ and $\rho$, over the previously used default values. Given that the algorithm runs in linear time, this means it would handle a matrix potentially 4 times larger in the same amount of time with the right choice of parameters, vs. one with the default choices. These results provide experimental data that can lead to theoretical justification. The consistent patterns we saw indicate that there is potential for robust theory around the most useful parameter values.

Additionally, we applied these techniques to real-world data from multiple domains, and demonstrated their use. This indicates that the work indeed was beneficial, and is a worthwhile pursuit. Exploring more real-world problems can lead to both stronger results empirically and more indications of possible theoretical results. By using both recommender systems data and network data, we showed that a variety of domains can benefit from this work.
Lastly, we affirmed previous results regarding these algorithms, and demonstrated again how useful the eRPCA algorithm is. While this work was mainly focused on using it for the matrix completion problem, we see that the algorithm can be manipulated in various ways as to apply it to multiple types of related problems. eRPCA is a relatively recent derivative of RPCA. Demonstrating its use will aid in confirming how helpful a technique it could be in certain problems.

5.2 Future work

At present, this work provides an empirical direction to follow, and potentially find theoretical justification. In the experimental results, we saw clear patterns of preferred parameter values. Providing the theory behind why these choices are more efficient, and proving that theory, is left for future work. The experimental patterns achieved show that there is likely a direction of theoretical work possible, in understanding which parameter values are best.

Particularly detailed pieces of the problem are left for further exploration. This includes finding sufficient error bounds to discard entries, as mentioned in Chapter 3, and developing more theory around the meaning of $\epsilon_1$ and $\epsilon_2$ and their appropriate values. eRPCA has shown to be an extremely versatile algorithm. With various versions, it can solve the matrix completion problem, traditional PCA or RPCA, and the case where errors are varied. Having one comprehensive algorithm that can do so many things will inspire much future work. Particularly, algorithm tuning and modification to suit the particular problem at hand. There are many different avenues left to be pursued.
Appendix A

Additional Details

A.1 Matrix Completion Conditions

In many matrix completion problems, there are various requirements for the number of points observed, and their location within the incomplete matrix. In order to complete the matrix as given in [1], the number of entries of $M$ given must obey

$$m \geq Cn^{(1.2)r \log(n)}$$

for some positive constant $C$, where $M$ is $n \times n$ and rank $r$. Additional conditions, for example the previously mentioned incoherence, are also required.

Incoherence is another requirement often imposed on the matrix. The coherence of a matrix is the largest (absolute) cross-correlation between columns of the matrix [33]. An incoherent matrix has a small value for this measure, which corresponds to sufficiently spread out singular values.

A.2 PCA Conditions

The required conditions for solving the RPCA problem are straightforward, as given in [4]. First, $L_0$ must not be sparse. That is, $L_0$ must be incoherent, as defined in [1]. Given the SVD of the matrix $L_0$, the singular values must be reasonably spread out, or not sparse. See [4] for details. Similarly, matrix $S_0$ must not be low-rank. This
condition requires the assumption that the pattern of entries in $S_0$ is random and selected uniformly. In terms more understandable to an application: the corrupted values in the matrix $M$ may be arbitrarily large, but must be randomly distributed throughout $M$, as opposed to concentrated to a particular region in the matrix. Under these conditions, the minimization problem recovers the components $M = L_0 + S_0$.

### A.3 Computing Resources

The computer used for all parts of this thesis provided the following resources. A Linux machine, encompassing eight cores, each 2800.032 Mhz. Each has 512 KB cache, and a total of 18498236 KB available. Python was used to write all algorithms, generate data and testing scripts, and produce all final graphs and charts. R was used for some intermediate statistic calculations.

### A.4 Iteration Time

Given the case where there are, say, 10,000 entries observed, the cost in time and memory, and therefore time per iteration, should be the same, regardless of the size of the matrix that these data came from. This comes from the fact that the eRPCA algorithm runs in $O(m)$ in both time and memory[13]. In Figure A.1 below, we can see this confirmed for matrices of various sizes, all with constant number of entries observed. Iteration time is approximately constant, allowing for some variation. And, in Figure A.1, we see that iteration time increases for larger number of observed entries, regardless of matrix size.

This result seems intuitive and simple, and in many ways it is. It is necessary to rely on it, though, for practical applications of eRPCA. Given a set number of entries, regardless of where they came from, we would expect time per iteration to be approximately constant. But, that does not mean the number of iterations is constant! Knowing ahead of time an estimated value for the time per iteration can allow the user to make smart and informed decisions about how to use the algorithm. Armed with knowledge of iteration time, coupled with an approximate guess at number of iterations (gleaned perhaps from results testing $\mu$ and $\rho$), the user can allocate that time to using the algorithm.
Figure A.1: Time per iteration compared with the number of entries observed, and what percent of the data they represent. As the number of entries increases, so does the time per iteration required, linearly. But, as the number of entries is held constant across varying matrix size, as demonstrated by the percent of entries they represent, time per iteration is approximately constant. True rank is 1, maximum rank is 4.

Figure A.1 was evaluated on data with a true rank of 1, and a maximum allowed rank in the algorithm of 4. On data simulated to have a true rank of 4, and a maximum allowed rank of 16, we see similar patterns, as well as that with true rank 16 and maximum allowed rank 64. Figure A.2 and A.3 show that, regardless of true rank, we still see time per iteration is approximately constant according to the number of entries.

A.5 Figures

Provided are additional Figures demonstrating results on more synthetic data tests. These Figures are similar to those found in Chapter 3. We provide them to show similarity in patterns across multiple data sets. The Table before each set of Figures, with a row highlighted, indicates which data set they were produced by.
Figure A.2: Time per iteration compared with the number of entries observed, and what percent of the data they represent. Here the true rank is 4, and the maximum allowed rank is 16. As the number of entries increases, so does the time per iteration required, linearly. But, as the number of entries is held constant across varying matrix size, as demonstrated by the percent of entries they represent, time per iteration is approximately constant.

![Figure A.2](image)

Figure A.3: Time per iteration compared with the number of entries observed, and what percent of the data they represent. Here the true rank is 4, and the maximum allowed rank is 16. As the number of entries increases, so does the time per iteration required, linearly. But, as the number of entries is held constant across varying matrix size, as demonstrated by the percent of entries they represent, time per iteration is approximately constant.

| n         | |O|    | % Observed | Best | |r| | pair      | Iterations with literature-provided values | Iterations with best pair |
|-----------|----------------|-------|------------|---------|----------------|--------------------------------|-----------------------------|-----------------------------|
| 1,000     | 10,000         | 1%    | 4µ₀, 64(ρ₀ − 1) + 1 | 58      | 18                           |
| 1,000     | 100,000        | 10%   | 4µ₀, 64(ρ₀ − 1) + 1 | 211     | 312*                         |
| 10,000    | 100,000        | 1%    | 4µ₀, 64(ρ₀ − 1) + 1 | 27      | 10                           |
| 10,000    | 1,000,000      | 1%    | 4µ₀, 64(ρ₀ − 1) + 1 | 43      | 17                           |
| 10,000    | 10,000,000     | 10%   | 6µ₀, 64(ρ₀ − 1) + 1 | 130     | 242*                         |
| 100,000   | 1,000,000      | 0.1%  | 4µ₀ or 2µ₀, 64(ρ₀ − 1) + 1 or 32(ρ₀ − 1) + 1 | 16      | 6                            |
| 1,000,000 | 10,000,000     | 0.01% | 2µ₀, 32(ρ₀ − 1) + 1 | 9       | 4                            |
| 1,000,000 | 20,000,000     | 0.02% | 4µ₀, 64(ρ₀ − 1) + 1 | 10      | 4                            |
| 1,000,000 | 50,000,000     | 0.005%| 4µ₀, 64(ρ₀ − 1) + 1 | 12      | 5                            |
Figure A.4: $\mu$ and $\rho$, colored by the objective value. Contours represent the number of iterations the algorithm took to complete. The blue values represent a minimized objective, while those tending towards red are no longer minimized. The red dot represents the previously used $\mu \rho$ pair, while the green represents the best pair that minimizes the objective. In this case, the literature-provided pair did not minimize the objective value. The patterns seen here are a slight deviation in our experiments, resembling the other 10% observed case seen in Figure 3.8.

Figure A.5: $\mu$ vs. Iterations and Objective Value. The red dots represent the number of iterations the algorithm takes to solve the problem. The blue dots represent the value of the objective. The default value of $\mu$ is marked with a line.
Figure A.6: $\rho$ vs. Iterations and Objective Value. The red dots represent the number of iterations the algorithm takes to solve the problem. The blue dots represent the value of the objective. The default value of $\rho$ is marked with a line.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$m$</th>
<th>% Observed</th>
<th>Best $\mu \rho$ pair</th>
<th>Iterations with literature-provided values</th>
<th>Iterations with best pair</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>10,000</td>
<td>1%</td>
<td>$4 \mu_0, 64(\rho_0 - 1) + 1$</td>
<td>58</td>
<td>14</td>
</tr>
<tr>
<td>1,000</td>
<td>100,000</td>
<td>10%</td>
<td>$6 \mu_0, 64(\rho_0 - 1) + 1$</td>
<td>211</td>
<td>312*</td>
</tr>
<tr>
<td>10,000</td>
<td>100,000</td>
<td>1%</td>
<td>$4 \mu_0, 64(\rho_0 - 1) + 1$</td>
<td>27</td>
<td>10</td>
</tr>
<tr>
<td>10,000</td>
<td>1,000,000</td>
<td>1%</td>
<td>$4 \mu_0, 64(\rho_0 - 1) + 1$</td>
<td>43</td>
<td>17</td>
</tr>
<tr>
<td>10,000</td>
<td>10,000,000</td>
<td>10%</td>
<td>$6 \mu_0, 64(\rho_0 - 1) + 1$</td>
<td>130</td>
<td>242*</td>
</tr>
<tr>
<td>100,000</td>
<td>1,000,000</td>
<td>0.1%</td>
<td>$4 \mu_0, 32(\rho_0 - 1) + 1$ or $2(\rho_0 - 1) + 1$</td>
<td>16</td>
<td>6</td>
</tr>
<tr>
<td>1,000,000</td>
<td>10,000,000</td>
<td>0.01%</td>
<td>$2 \mu_0, 32(\rho_0 - 1) + 1$</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>1,000,000</td>
<td>20,000,000</td>
<td>0.002%</td>
<td>$4 \mu_0, 64(\rho_0 - 1) + 1$</td>
<td>19</td>
<td>4</td>
</tr>
<tr>
<td>1,000,000</td>
<td>50,000,000</td>
<td>0.005%</td>
<td>$4 \mu_0, 64(\rho_0 - 1) + 1$</td>
<td>42</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure A.7: $\mu$ and $\rho$, colored by the objective value. Contours represent the number of iterations the algorithm took to complete. The blue values represent a minimized objective, while those tending towards red are no longer minimized. The red dot represents the previously used $\mu \rho$ pair, while the green represents the best pair that minimizes the objective.
Figure A.8: $\mu$ vs. Iterations and Objective Value. The red dots represent the number of iterations the algorithm takes to solve the problem. The blue dots represent the value of the objective. The default value of $\mu$ is marked with a line.

Figure A.9: $\rho$ vs. Iterations and Objective Value. The red dots represent the number of iterations the algorithm takes to solve the problem. The blue dots represent the value of the objective. The default value of $\rho$ is marked with a line.

| $n$     | $|\Omega|$ | % Observed | Best $\rho\mu$ pair | Iterations with literature-provided values | Iterations with best pair |
|---------|-----------|------------|----------------------|-------------------------------------------|--------------------------|
| 1,000   | 10,000    | 1%         | $4\mu_0, 64(\rho_0 - 1) + 1$ | 58                                        | 18                       |
| 1,000   | 100,000   | 10%        | $6\mu_0, 64(\rho_0 - 1) + 1$ | 211                                       | 312*                     |
| 10,000  | 100,000   | 4%         | $4\mu_0, 64(\rho_0 - 1) + 1$ | 27                                        | 10                       |
| 10,000  | 1,000,000 | 1%         | $4\mu_0, 64(\rho_0 - 1) + 1$ | 43                                        | 17                       |
| 10,000  | 10,000,000| 10%        | $4\mu_0, 64(\rho_0 - 1) + 1$ | 130                                       | 242*                     |
| 100,000 | 1,000,000 | 0.1%       | $4\mu_0$ or $2\mu_0, 64(\rho_0 - 1) + 1$ or $32(\rho_0 - 1) + 1$ | 16 | 6 |
| 1,000,000| 10,000,000| 0.01%      | $2\mu_0, 32(\rho_0 - 1) + 1$ | 9                                         | 4                        |
| 1,000,000| 20,000,000| 0.002%     | $4\mu_0, 64(\rho_0 - 1) + 1$ | 10                                        | 4                        |
| 1,000,000| 50,000,000| 0.0005%    | $4\mu_0, 64(\rho_0 - 1) + 1$ | 12                                        | 5                        |
Figure A.10: $\mu$ vs. Iterations and Objective Value. The red dots represent the number of iterations the algorithm takes to solve the problem. The blue dots represent the value of the objective. The default value of $\mu$ is marked with a line.

Figure A.11: $\rho$ vs. Iterations and Objective Value. The red dots represent the number of iterations the algorithm takes to solve the problem. The blue dots represent the value of the objective. The default value of $\rho$ is marked with a line.

| $\sigma$ | $|F|$ | % Observed | Best $\mu\rho$ pair | Iterations with literature-provided value | Iterations with best pair |
|----------|------|------------|----------------------|-----------------------------------------|--------------------------|
| 1,000    | 10,000 | 1%         | $\mu_0, 64(\rho_0 - 1) + 1$ | 58                                      | 18                       |
| 1,000    | 100,000 | 10%        | $\mu_0, 64(\rho_0 - 1) + 1$ | 211                                     | 312*                     |
| 10,000   | 100,000 | 1%         | $\mu_0, 64(\rho_0 - 1) + 1$ | 27                                      | 18                       |
| 10,000   | 1,000,000 | 5%        | $\mu_0, 64(\rho_0 - 1) + 1$ | 43                                      | 17                       |
| 10,000   | 10,000,000 | 10%     | $\mu_0, 64(\rho_0 - 1) + 1$ | 130                                     | 242*                     |
| 100,000  | 1,000,000 | 0.1%      | $\mu_0, 32(\rho_0 - 1) + 1$ or $32(\rho_0 - 1) + 1$ | 16                                       | 6                        |
| 1,000,000 | 10,000,000 | 0.001% | $\mu_0, 32(\rho_0 - 1) + 1$ | 33                                      | 8                        |
| 1,000,000 | 20,000,000 | 0.002% | $\mu_0, 64(\rho_0 - 1) + 1$ | 10                                      | 3                        |
| 1,000,000 | 50,000,000 | 0.005% | $\mu_0, 64(\rho_0 - 1) + 1$ | 12                                      | 5                        |
Figure A.12: $\mu$ vs. Iterations and Objective Value. The red dots represent the number of iterations the algorithm takes to solve the problem. The blue dots represent the value of the objective. The default value of $\mu$ is marked with a line.

Figure A.13: $\rho$ vs. Iterations and Objective Value. The red dots represent the number of iterations the algorithm takes to solve the problem. The blue dots represent the value of the objective. The default value of $\rho$ is marked with a line.

| n     | |D|  | % Observed | Best $\mu$ pair | Iterations with literature-provided values | Iterations with best pair |
|-------|---|---|-------------|----------------|---------------------------------------------|--------------------------|
| 1,000 | 10,000 | 1% | $2\mu_0, 64(\rho_0 - 1) + 1$ | 58 | 18 |
| 1,000 | 100,000 | 10% | $2\mu_0, 64(\rho_0 - 1) + 1$ | 211 | 312* |
| 10,000 | 100,000 | 1% | $2\mu_0, 64(\rho_0 - 1) + 1$ | 27 | 10 |
| 10,000 | 10,000,000 | 10% | $2\mu_0, 64(\rho_0 - 1) + 1$ | 40 | 17 |
| 10,000 | 100,000,000 | 0% | $64\mu_0, 64(\rho_0 - 1) + 1$ | 110 | 242* |
| 100,000 | 1,000,000 | .01% | $2\mu_0$ or $2\mu_0, 64(\rho_0 - 1) + 1$ or $32(\rho_0 - 1) + 1$ | 16 | 6 |
| 1,000,000 | 10,000,000 | 0.01% | $2\mu_0, 64(\rho_0 - 1) + 1$ | 9 | 4 |
| 1,000,000 | 20,000,000 | 0.002% | $2\mu_0, 64(\rho_0 - 1) + 1$ | 10 | 4 |
| 1,000,000 | 50,000,000 | 0.005% | $2\mu_0, 64(\rho_0 - 1) + 1$ | 12 | 5 |
Figure A.14: $\rho$ vs. Iterations and Objective Value, for a matrix with 10,000,000 entries that is 1,000,000 by 1,000,000. The red dots represent the number of iterations the algorithm takes to solve the problem. The blue dots represent the value of the objective. The default value of $\rho$ is marked with a line.

Figure A.15: $\mu$ vs. Iterations and Objective Value, for a matrix with 10,000,000 entries that is 1,000,000 by 1,000,000. The red dots represent the number of iterations the algorithm takes to solve the problem. The blue dots represent the value of the objective. The default value of $\mu$ is marked with a line.


