Matrix Completion with Queries

Natali Ruchansky
Boston University
natalir@cs.bu.edu

Mark Crovella
Boston University
crovella@cs.bu.edu

Evimaria Terzi
Boston University
evimaria@cs.bu.edu

ABSTRACT

In many applications, e.g., recommender systems and traffic monitoring, the data comes in the form of a matrix that is only partially observed and low rank. A fundamental data-analysis task for these datasets is *matrix completion*, where the goal is to accurately infer the entries missing from the matrix. Even when the data satisfies the low-rank assumption, classical matrix-completion methods may output completions with significant error – in that the reconstructed matrix differs significantly from the true underlying matrix. Often, this is due to the fact that the information contained in the observed entries is insufficient. In this work, we address this problem by proposing an active version of matrix completion, where queries can be made to the true underlying matrix. Subsequently, we design `Order&Extend`, which is the first algorithm to unify a matrix-completion approach and a querying strategy into a single algorithm. `Order&Extend` is able identify and alleviate insufficient information by judiciously querying a small number of additional entries. In an extensive experimental evaluation on real-world datasets, we demonstrate that our algorithm is efficient and is able to accurately reconstruct the true matrix while asking only a small number of queries.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*data mining*

Keywords

matrix completion; recommender systems; active querying

1. INTRODUCTION

In many applications the data comes in the form of a low-rank matrix. Examples of such applications include recommender systems (where entries of the matrix indicated user preferences over items), network traffic analysis (where the matrix contains the volumes of traffic among source-destination pairs), and computer vision (image matrices). In many cases, only a small percentage of the matrix entries are observed. For example, the data used in the Netflix prize competition was a matrix of 480K users $\times$ 18K movies, but only 1% of the entries were known.

A common approach for recovering such missing data is called *matrix completion*. The goal of matrix-completion methods is to accurately infer the values of missing entries, subject to certain assumptions about the complete matrix [2, 3, 5, 7, 8, 9, 19]. For a true matrix T with observed values only in a set of positions Ω, matrix-completion methods exploit the information in the observed entries in T (denoted T_Ω) in order to produce a "good" estimate $\widehat{T}$ of T. In practice, the estimate may differ significantly from the true matrix. In particular, this can happen when the observed entries T_Ω are not adequate to provide sufficient information to produce a good estimate.

In many cases, it is possible to address the insufficiency of T_Ω by actively obtaining additional observations. For example, in recommender systems, users may be asked to rate certain items; in traffic analysis, additional monitoring points may be installed. These additional observations can lead to an augmented Ω' such that $T_{\Omega'}$ carries more information about T and can lead to more accurate estimates $\widehat{T}$. In this *active* setting, the data analyst can become an *active* participant in data collection by posing *queries* to T. Of course such active involvement will only be acceptable if the number of queries is small.

In this paper, we present a method for generating a small number of queries so as to ensure that the combination of observed and queried values provides sufficient information for an accurate completion; i.e., the $\widehat{T}$ estimated using the entries $T_{\Omega'}$ is significantly better than the one estimated using T_Ω. We call the problem of generating a small number of queries that guarantee small reconstruction error the ActiveCompletion problem.

The difference between the classical matrix-completion problem and our problem is that in the former, the set of observed entries is *fixed* and the algorithm needs to find the best completion given these entries. In ActiveCompletion, we are asked to design both a *completion* and a *querying* strategy in order to minimize the reconstruction error. On the one hand, this task is more complex than standard matrix completion – since we have the additional job of designing a good querying strategy. On the other hand, having the flexibility to ask some additional entries of T to be revealed should yield lower reconstruction error.

KDD'15, August 10-13, 2015, Sydney, NSW, Australia.

DOI: http://dx.doi.org/10.1145/2783258.2783259.

At a high level, ActiveCompletion is related to other recently proposed methods for active matrix completion, e.g., [4]. However, existing approaches identify entries to be queried *independently* of the method of completion. In contrast, a strength of our algorithm is that it addresses completion and querying in an integrated fashion.

The main contribution of our work is `Order&Extend`, an algorithm that simultaneously minimizes the number of queries to ask and produces an estimate matrix $\widehat{T}$ that is very close to the true matrix T. The design of `Order&Extend` is inspired from recent matrix-completion methods that view the completion process as solving a sequence of (not necessarily linear) systems [10, 11, 13, 16]. We adopt this general view, focusing on a formulation that involves only linear systems. Although existing work uses this insight for simple matrix completion, we go one step further and observe that there is a relationship between the ordering in which systems are solved, and the number of additional queries that need to be posed. Therefore, the first step of `Order&Extend` focuses on finding a good ordering of the set of linear systems. Interestingly, this ordering step relies on the combinatorial properties of the mask graph, a graph that is associated with the positions (but not the values) of observed entries Ω.

In the second step, `Order&Extend` considers the linear systems in the chosen order, and asks queries every time it encounters a *problematic* linear system $Ax = b$. A linear system can problematic in two ways: (a) when there are not enough equations for the number of unknowns, so that the system does not have a unique solution; (b) when solving the system $Ax = b$ is numerically unstable *given the specific* b *involved.* Note that, as we explain in the paper, this is not the same as simply saying that A is ill-conditioned; part of our contribution is the design of fast methods for detecting and ameliorating such systems.

Our extensive experiments with datasets from a variety of application domains demonstrate that `Order&Extend` requires significantly fewer queries than any other baseline querying strategy, and compares very favorably to approaches based on well-known matrix completion algorithms. In fact, our experiments indicate that `Order&Extend` is "almost optimal" as it gives solutions where the number of entries it queries is generally very close to the information-theoretic lower bound for completion.

2. RELATED WORK

To the best of our knowledge we are the first to pose the problem of constructing an algorithm equipped simultaneously with a completion and a querying strategy. However, matrix completion is a long studied problem, and in this section we describe the existing work in this area.

Statistical matrix completion: The first methods for matrix completion to be developed were statistical in nature [1, 2, 3, 5, 7, 8, 9, 14, 19]. Statistical approaches are typically interested in finding a low-rank completion of the partially observed matrix. These methods assume a random model for the positions of known entries, and formulate the task as an optimization problem. A key characteristic of statistical methods is that they estimate a completion regardless of whether the information contained in the visible entries is sufficient for completion. In other words, on any input they output their best estimate, which can have high error. Moreover, statistical methods are not equipped with a querying strategy, nor a mechanism to signal when the information is insufficient.

Random sampling: Candés and Recht introduced a threshold on the number of entries needed for accurate matrix completion [2]. Under the assumption of randomly sampled locations of known entries, they prove that an $n_1 \times n_2$ matrix of rank r should have at least $m > Cn^{\frac{6}{5}} r \log(n)$ for their algorithm to succeed with high probability, where $n = \max(n_1, n_2)$. Different authors in the matrix-completion literature develop slightly different thresholds, but all are essentially $O(nr \log(n))$ [9, 15]. We point out that achieving this bound in the real world often requires a significantly large number of samples. For example, adopting the rank $r \approx 40$ of top solutions to the Netflix Challenge, over 151 million entries would need to be queried.

Structural matrix completion: Recently, a class of matrix completion has been proposed, which we call *structural.* Rather than taking an optimization approach, the methods of structural completion explicitly analyze the information content of the visible entries and are capable of stating definitively that the observed entries are information-theoretically sufficient for reconstruction [10, 11, 13, 16].

Structural methods are implicitly concerned with the number of possible completions that are consistent with the partially observed matrix; this could be infinite, a finite, one, or none. A key observation shared by all structural approaches is that the number of possible completions does not depend on the values of the observed entries, but rather only on their positions. This statement, proved by Kiraly et al. [10], means that in our search for good ordering of linear systems we can work solely with the locations of known entries.

The common characteristic between our method and structural methods is that they also view matrix completion as a task of solving a sequence of (not necessarily linear) systems of equations where the result of one is used as input to another. In fact, Meka et al. [13] adopt the same view as ours. However, the key difference between these works and ours is that we are concerned particularly with the *active* version of the problem and we need to effectively design both a reconstruction and a querying strategy simultaneously.

The active problem: Although active learning has been studied for some time, work in the active matrix completion area has only appeared recently [4, 17]. In both these works, the authors are interested in determining which entries need to be revealed in order to reduce error in matrix reconstruction. Their methods choose to reveal entries with the largest predicted uncertainty based on various measures. Algorithmically, the difference with our work is that the previous approaches construct a querying strategy *independently* of the completion algorithm. In fact, they use off-the-shelf matrix completion algorithms for the reconstruction phase, while the strength in our algorithm is precisely its integrated nature of querying and completing. These methods appear to have other drawbacks. In the experiments, Chakraborty et al. start with partial matrices where 50-60% of entries are already known – far greater than that required by our method. Further, their proposed query strategy does not lead to a significant improvement over pure random querying. While Sutherland et al. report low reconstruction error, the main experiments are run over 10×10 matrices, providing no evidence that the methods scale.

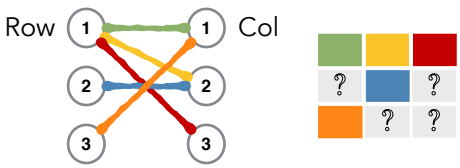


Figure 1: The mask graph G_Ω of mask $\Omega = \{(1,1), (1,2), (1,3), (2,2), (3,1)\}$.

3. PROBLEM DEFINITION

In this section, we describe our setting and provide the problem definition.

3.1 Notation and setting

Throughout the paper, we assume the existence of a true matrix T of size $n_1 \times n_2$; T may represent the preferences of n_1 users over n_2 objects, or the measurements obtained in n_1 sites over n_2 attributes. We assume that the entries of T are real numbers ($T_{ij} \in \mathbb{R}$) and that only a subset of these entries $\Omega \subset \{(i,j)\,|, 1 \le i \le n_1, 1 \le j \le n_2\}$ are observed. We refer to the set of positions of known entries Ω as the *mask* of T. When we are referring to the values of the visible entries in T we will denote that set as T_Ω.

The mask Ω has an associated *mask graph,* denoted G_Ω. The mask graph is a bipartite graph $G_\Omega = (V_1, V_2, E)$, where V_1 and V_2 correspond to the set of nodes in the left and right parts of the graph, with every node $i \in V_1$ representing a row of T and every $j \in V_2$ representing a column of T. The edges in G_Ω correspond to the positions Ω, meaning $(i,j) \in E \iff (i,j) \in \Omega$. An example is shown in Figure 1.

Throughout the paper, we will use $\widehat{T}_\Omega$ to denote the estimate of T that was computed using T_Ω as input. Of course, this estimate depends not only on Ω, but also on the algorithm $\mathcal{A}$ used for completion. Therefore, we denote the estimate obtained by a particular algorithm $\mathcal{A}$ on input T_Ω as $\mathcal{A}(T_\Omega) = \widehat{T}_{\mathcal{A},\Omega}$. When the algorithm is unspecified or clear from the context, we will omit it from the subscript.

Finally, we define the *reconstruction error* of the estimate $\widehat{T}_\Omega$ of T as:

$$\textsc{RelError}(\widehat{T}_\Omega) = \frac{\|T - \widehat{T}_\Omega\|_F}{\|T\|_F}. \tag{1}$$

Recall that for an $n_1 \times n_2$ matrix X, $\|X\|_F$ (or simply $\|X\|$) is the Frobenius norm defined as $\|X\|_F = \sqrt{\sum_{i=1}^{n_1}\sum_{j=1}^{n_2} X_{ij}^2}$.

Low effective rank: For the matrix-completion problem to be well-defined, some restrictions need to be placed on T. Here, the *only* assumption we make for T is that it has *low effective rank* r ($\ll \min(n_1, n_2)$). Were T exactly rank r, it would have r non-zero singular values; when T is *effectively* rank r, it has full rank but its top r singular values are significantly larger in magnitude than the rest. In practice, many matrices obtained through empirical measurements are found to have low effective rank.

Rather than stipulating low rank, it is often simpler to postulate that the effective rank r is known. This assumption is used in obtaining many theoretical results in the matrix-completion literature [2, 10, 11, 14]. Yet in practice, the important assumption is that of low effective rank. Even if r is unknown but required as input to an algorithm, one could try a several values of r and choose the best performing. For the rest of the paper, we consider r to be known and omit reference to it when it is understood from context.

Querying entries: For simplicity of exposition we discuss our problem and algorithms in the context of unlimited access to all unobserved entries of T. However, our results still apply and our algorithm can still work in the presence of constraints on which entries of T may be queried.

3.2 The ActiveCompletion problem

Given the above, we define our problem as follows:

Problem 1 (ActiveCompletion). *Given an integer $r > 0$ that corresponds to the effective rank of T and the values of T in certain positions Ω, find a set of additional entries Q to query from T such that for $\Omega' = \Omega \cup Q$, $\textsc{RelError}(\widehat{T}_{\Omega'})$ as well as $|Q|$ are minimized.*

Note that the above problem definition has two minimization objectives: (1) the number of queried entries and (2) the reconstruction error. In practice we can only solve for one and impose a constraint on the other. For example, we can impose a *query budget* b on the number of queries to ask and optimize for the error. Alternatively, one can use *error budget* ϵ to control the error of the output, and then minimize for the number of queries to ask. In principle our algorithm can be adjusted to solve any of the two cases. However, since setting a desired b is more intuitive for our active setting, in our experiments we do this and optimize for the error. We will focus on this version of the problem (with the budget on the queries) for the majority of the discussion.

The exact case: A special case of ActiveCompletion is when T is exactly rank r, and the maximum allowed error ϵ is zero. In this case, the problem asks for the minimum number of queries required to reconstruct $\widehat{T}$ that is exactly equal to the true matrix T. This can only be guaranteed by ensuring that the information observed in $T_{\Omega'}$ is adequate to restrict the solution space to a single unique completion, which will then necessarily be identical to T.

Intuitively, one expects that the larger the set of observed entries Ω, the fewer the number of possible completions of T_Ω. In fact, Kiraly et al. [10] make the fundamental observation that under certain assumptions, the number of possible completions of a partially observed matrix does not depend on the values of the visible entries, but *only* on the positions of these entries. This result implies that the uniqueness of matrices that agree with T_Ω is a property of the mask Ω and not of the actual values T_Ω.

Critical mask size: The number of degrees of freedom of an $n_1 \times n_2$ matrix of rank exactly r is $r(n_1 + n_2 - r)$, which we denote $\phi(T, r)$. Hence, regardless of the nature of Ω, any solution with $\epsilon = 0$ must have $|\Omega'| \ge \phi(T, r)$. We therefore call $\phi(T, r)$ the *critical mask size* as it can be considered as a (rather strict) *lower bound* on the number of entries that need to be in Ω' to achieve small reconstruction error.

Empty masks: For the special case of exact rank and $\epsilon = 0$, if the input mask Ω is empty, i.e., $\Omega = \emptyset$, then ActiveCompletion can be solved optimally as follows: simply query the entries of r rows and r columns of T. This will require $\phi(T, r)$ queries, which will construct a mask Ω' that determines a unique reconstruction of T. Therefore, when the initial mask $\Omega = \emptyset$, the ActiveCompletion problem can be solved in polynomial time.

4. ALGORITHMS

In this section we present our algorithm, `Order&Extend`, for addressing the ACTIVECOMPLETION problem.

The starting point for the design of `Order&Extend` is the low (effective) rank assumption of T. As it will become clear, this means that the unobserved entries are related to the observed entries through a set of linear systems. Thus one approach to matrix completion is to solve a sequence of linear systems. Each system in this sequence uses observed entries in T, or entries of T reconstructed by previously solved linear systems to infer more missing entries.

The reconstruction error of such an algorithm depends on the quality of the solutions to these linear systems. As we will show below, each query of T can yield a new equation that can be added to a linear system. Hence, if a linear system has fewer equations than unknowns, a query must be made to add an additional equation to the system. Likewise, if solving a system is numerically unstable then a query must be made to add an equation that will stabilize it. Crucially, the need for such queries depends on the nature of the solutions obtained to linear systems *earlier in the order*. Thus the order in which systems are solved, and the nature of these systems are inter-related. A good ordering will minimize the number of "problematic" systems being encountered. However, problematic systems can appear even in the best-possible order, meaning that good ordering alone is insufficient for accurate reconstruction.

At a high level, `Order&Extend` operates as follows: first, it finds a promising ordering of the linear systems. Then, it proceeds by solving the linear systems in this order. If a linear system that requires additional information is encountered, the algorithm either strategically queries T or moves the system to the end of the ordering. When all systems have been solved, $\hat{T}$ is computed and returned. The next subsections describe these steps in detail.

4.1 Completion as a sequence of linear systems

In this section we explain the particular linear systems that the completion algorithm solves, the sequence in which it solves them, and how the ordering in which systems are solved affects the quality of the completion.

For the purposes of this discussion, we assume that T is of rank exactly r. In this case T can be expressed as the product of two matrices X and Y of sizes $n_1 \times r$ and $r \times n_2$; that is, $T = XY$. Furthermore, we assume that any subset of r rows of X, or r columns of Y, is linearly independent. (Later we will describe how `Order&Extend` addresses the case when these assumptions do not hold – i.e., when T is only *effectively* rank-r, or when an r-subset is linearly dependent). To complete T, it suffices to find such factors X and Y.[1]

The `Sequential` completion algorithm: We start by describing an algorithm we call `Sequential`, which estimates the rows of X and columns of Y. `Sequential` takes two inputs: (1) an ordering π over the set of all rows of X and columns of Y, which we call the *reconstruction order*, and (2) the partially observed matrix T_Ω.

To explain how `Sequential` works, consider the example in Figure 2, where $r = 2$, T is on the left and G_Ω is on the right. The factors X and Y are shown on the side of and above T to convey how their product results in T. The nodes V_1 of G_Ω correspond to the rows of X, and nodes V_2 to the columns of Y. In this figure, we illustrate an intermediate step of `Sequential`, in which the values of the i-th and i'-th rows of X have already been computed. Each entry of T is the inner product of a row of X and a column of Y. Hence we can represent the depicted entries in T by the following linear system:

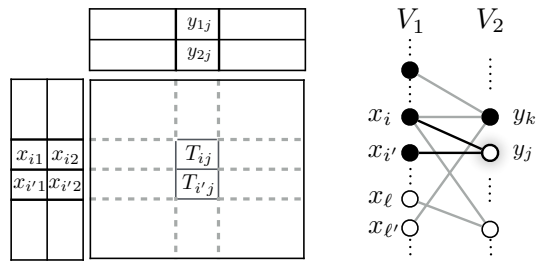


Figure 2: An intermediate step of `Sequential` algorithm.

$$T_{ij} = x_{i1}y_{1j} + x_{i2}y_{2j} \quad (2)$$

$$T_{i'j} = x_{i'1}y_{1j} + x_{i'2}y_{2j} \quad (3)$$

Observe that x_i and $x_{i'}$ are known, and that the edges (x_i, y_j) and $(x_{i'}, y_j)$ corresponding to T_{ij} and $T_{i'j}$ exist in G_Ω. The only unknowns in (2) and (3) are y_{1j} and y_{2j}, which leaves us with two equations in two unknowns. As stated above and by assumption, any r-subset of X or Y is linearly independent; hence one can solve uniquely for y_{1j} and y_{2j} and fill in column j of Y.

To generalize the example above, the steps of `Sequential` can be partitioned in x- and y-steps; at every y-step the algorithm solves a system of the form

$$A_x y = t. \quad (4)$$

In this system, y is a vector of r unknowns corresponding to the values of the column of Y we are going to compute; A_x is an $r \times r$ fully-known submatrix of X and t is a vector of r known entries of T which are located on the same column as the column index of y. If A_x and t are known, and A_x is full rank, then y can be computed exactly and the algorithm can proceed to the next step.

In the x-steps `Sequential` evaluates a row of X using an $r \times r$ already-computed subset of columns of Y, and a set of r entries of T from the row of T corresponding to the current row of X being solved for. Following the same notational conventions as above, the corresponding system becomes $A_y x = t'$. For simplicity we will focus our discussion on y-steps; the discussion on x-steps is symmetric.

The completion on the mask graph: The execution of `Sequential` is also captured in the mask graph shown in the right part of Figure 2. In the beginning, no rows or columns have been recovered and all nodes of G_Ω are white (unknown). As the algorithm proceeds they become black (known), and this transformation occurs in the order suggested by the input reconstruction order π. Thus a black node denotes a row of X or column of Y that has been computed. In our example, the fact that we can solve for the j-th column of Y (using Equations (2) and (3)) is captured

[1]Note that X and Y are not uniquely determined; any invertible $r \times r$ matrix W yields new factors XW^{-1} and WY which also multiply to yield T.

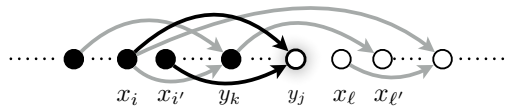


Figure 3: The direction on the edges imposed by the ordering of nodes shown in Figure 2.

by y_j's *two* connections to black/known nodes (recall $r = 2$). For general rank r, the j-th column of Y can be estimated by a linear system, if in the mask graph y_j is connected to at least r already computed (black) nodes. This is symmetric for the i-th row of X and node x_i. Intuitively, this transformation of nodes from black to white is reminiscent of an information-propagation process. This analogy was first drawn by Meka et al. [13].

Incomplete and unstable linear systems: As it has already been discussed in the literature [13], the performance of an algorithm like `Sequential` is heavily dependent on the input reconstruction order. Meka et al. [13] have discussed methods for finding a good reconstruction order in the special case where the mask graph has a power-law degree distribution. However, even with the best possible reconstruction order `Sequential` may still encounter linear systems which are either *incomplete* or *unstable*. Incomplete linear systems are those for which the vector t has some missing values and therefore the system $A_x y = t$ cannot be solved. Unstable linear systems are those in which all the entries in t are known, but the resulting expression $A_x^{-1} t$ may be very sensitive to small changes in t. These systems raise a numerous problems in the case where the input T is a noisy version of a rank r matrix, i.e., it is a matrix of effective rank r.

In the next two sections we describe how `Order&Extend` deals with such systems.

4.2 Ordering and fixing incomplete systems

First, `Order&Extend` devises an order that minimizes the number of incomplete systems encountered in the completion process.

Let us consider again the execution of `Sequential` on the mask graph, and the sequential transformation of the nodes in $G_\Omega = (V_1, V_2, E)$ from white to black. Recall that in this setting, an incomplete system occurs when the node in G_Ω that corresponds to y is connected to less than r black nodes.

Consider an order π of the nodes $V_1 \cup V_2$. This order conceptually imposes a direction on the edges of E; if x is before y in that order, then $\pi(x) < \pi(y)$, and edge (x, y) becomes directed edge $(x \to y)$. Figure 3 shows this transformation for the mask graph in Figure 2 and the order implied there. For fixed π, a node becomes black if it has at least r incoming edges, i.e., indegree at least r. In this view, an incomplete system manifests itself by the existence of a node that has indegree less than r. Clearly, if an order π guarantees that all nodes have r incoming edges, then there are no incomplete systems, and π is a *perfect reconstruction order*.

In practice such perfect orders are very hard to find; in most of the cases they do not exist. The goal of the first step of `Order&Extend` is to find an order π that is as close as possible to a perfect reconstruction order. It does so by constructing an order that minimizes the number of edges that need to be added so that the indegree of any node is r.

To achieve this, the algorithm starts by choosing the node from $G_\Omega = (V_1, V_2, E)$ with the lowest degree. This node is placed last in π, and removed from $G_\Omega = (V_1, V_2, E)$ along with its incident edges. Of the remaining nodes, the one with minimum degree is placed in the next-to-last position in π, and again removed from $G_\Omega = (V_1, V_2, E)$. This process repeats until all nodes have been assigned a position in π.

Next, the algorithm makes an important set of adjustments to π by examining each node u in the order it occurs in π. For a particular u the adjustments can take two forms:

1. *if u has degree $\leq r$*: it is repositioned to appear immediately after the neighbor v with the largest $\pi(v)$.

2. *if u has degree $> r$*: it is repositioned to appear immediately after the neighbor its neighbor v with the the r-th smallest $\pi(v)$.

These adjustments aim to construct a π such that when the implied directionality is added to edges, each node has indegree as close to r as possible. While it is possible to iterate this adjustment process to further improve the ordering, in our experiments this showed little benefit.

Once the order π is formed as described above, then the incomplete systems can be quickly identified: as `Order&Extend` traverses the nodes of G_Ω in the order implied by π, every time it encounters a node u with in-degree less than r, it adds edges so that u's indegree becomes r; by definition, the addition of a new edge (x, u) corresponds to querying a missing entry T_{xu} of T.

4.3 Finding and alleviating unstable systems

The incomplete systems are easy to identify – they correspond to nodes in G_Ω with degree less than r. However, there are other "problematic" systems which do not appear to be incomplete, yet they are *unstable*. Such systems arise due to noise in the data matrix or to an accumulation of error that happens through the sequential system-solving process. These systems are harder to detect and alleviate. We discuss our methodology for this below.

Understanding unstable linear systems: Recall that a system $A_x y = t$ is unstable if its solution is very sensitive to the noise in t. To be more specific, consider the system $A_x y = t$, where A_x has full rank and t is fully known. Recall that the solution of this system, $y = A_x^{-1} t$, will be used as part of a subsequent system: $A_y x = t'$, where y will become a row of matrix A_y. Let $A_x = U \Sigma V^T$ be the singular value decomposition of A_x with singular values $\sigma_1 \geq \ldots \geq \sigma_r$. Now if there is a σ_j such that σ_j is very small, then the solution to the linear system will be very unstable when the singular vector v_j corresponding to σ_j has a large projection on t. This is because in A_x^{-1}, the small σ_j will be inverted to a very large $1/\sigma_j$. Thus the inverse operation will cause any component of t that is in the direction of v_j to be *disproportionally-strongly expressed*, and any small amount of noise in t to be amplified in y. Thus, unstable systems may be catastrophic for the reconstruction error of `Sequential` as a single such system may initiate a sequence of unstable systems, which can amplify the overall error.

Unstable vs ill-conditioned systems: It is important to contrast the notion of an unstable system with that of an ill-conditioned system, which is widely used in the literature. Recall, that system $A_x y = t$ is ill-conditioned if *there exists* a vector s and a small perturbation s' of s, such that the

results of systems $A_x y = s$ and $A_x y' = s'$ are significantly different. Thus, whether or not a system is ill-conditioned depends only on A_x, and not on its relationship with any target vector t in particular. An ill-conditioned system is also characterized by a large condition number $\kappa(A_x) = \frac{\sigma_1}{\sigma_n}$. This way of stating ill-conditioning emphasizes that $\kappa(A_x)$ measures a property of A_x and *does not depend* on t. Consequently (as we will document in Section 5.3) the condition number $\kappa(A_x)$ generates too many false positives to be used for identifying unstable systems.

Identifying unstable systems: To provide a more precise measure of whether a system $A_x y = t$ is unstable, we compute the following quantity:

$$\ell(A_x, t) = \|A_x^{-1}\| \frac{\|t\|}{\|y\|}. \tag{5}$$

We call this quantity the *local condition number*, which was also discussed by Trefethen and Bau [18]. The local condition number is more tailored to our goal as we want to quantify the proneness of a system to error with respect to a particular target vector t. In our experiments, we characterize a system $A_x y = t$ as unstable if $\ell(A_x, t) \geq \theta$. We call the threshold θ the *stability threshold* and in our experiments we use $\theta = 1$. Loosely, one can think of this threshold as a way to control for the error allowed in the entries of reconstructed matrix. Although it is related, the value of this parameter does not directly translate into a bound on the RelError of the overall reconstruction.

Selecting queries to alleviate unstable systems: One could think of dealing with an unstable system via regularization, such as ridge regression (Tikhonov Regularization) which was also suggested by Meka et al. [13]. However, for systems $A_x y = t$, such regularization techniques aim to dampen the contribution of the singular vector that corresponds to the smallest singular value, as opposed to boosting the contribution of the singular vectors that are in the direction of t. Further, the procedure can be expressed in terms of only A_x without taking t into account; as we have discussed this is not a good measure for our approach.

The advantage of our setting is that we can actively query entries from T. Therefore, our way of dealing with this problem is by adding a direction to A_x (or as many as are needed until there are r strong ones). We do that by extending our system from $A_x y = t$ to $\begin{bmatrix} A_x \\ \alpha \end{bmatrix} \tilde{y} = \begin{bmatrix} t \\ \tau \end{bmatrix}$. Of course, in doing so we implicitly shift from looking for an exact solution to the system, to looking for a least-squares solution.

Clearly α cannot be an arbitrary vector. It must be an already computed row of X, it should be independent of A_x, and it must boost a direction in A_x which is poorly expressed and also in the direction of t. Given the intuition we developed above, we iterate over all previously computed rows of X that are not in A_x, and set each row as a candidate α. Among all such α's we pick α^* as the one with the smallest $\ell(A_x, t)$, and use it to extend A_x to $\begin{bmatrix} A_x \\ \alpha^* \end{bmatrix}$.

Querying T judiciously: Although the above procedure is conceptually clear, it raises a number of practical issues. If the system $A_x y = t$ solves for the j-th column of matrix Y, then every time we try a different α, which suppose is the already-computed $X(i,:)$, then the corresponding τ must be the entry T_{ij}. Since T_{ij} is not necessarily an observed entry, this would require a query even for rows $\alpha \neq \alpha^*$, which is clearly a waste of queries since we will only pick one α^*. Therefore, instead of querying the unobserved values of τ, `Order&Extend` simply uses random values following the distribution of the values observed in the i-th row and j-th column of T. Once α^* is identified, we only query the value of τ corresponding to row α^* and column j.

If there is no α^* that leads to a system with local condition number below our threshold, we postpone solving this system by moving the corresponding node of the mask graph to the end of the order π.

Algorithm 1 The `local_condition` routine

Input: C, A_x, α, t
$D = C - \frac{C\alpha^T \alpha C}{1+\alpha C \alpha^T}$
$\tilde{A}_x = \begin{bmatrix} A_x \\ \alpha \end{bmatrix}$
$\tau =$ `Random` $(T(i,:), T(:,j))$
$\tilde{t} = \begin{bmatrix} t \\ \tau \end{bmatrix}$
$\tilde{y} = D\tilde{A}_x \tilde{t}$
return $\|D\tilde{A}_x\| \frac{\|\tilde{t}\|}{\|\tilde{y}\|}$

Algorithm 2 The `Stabilize` routine

Input: A_x, t, θ
j : the column of Y being computed
$C = (A_x^T A_x)^{-1}$
for $i \in$ {Computed rows of X} **do**
  $\alpha_i = X(i,:)$
  if $X(i,:)$ not in A_x **then**
    $c(i) =$ `local_condition`$(C, A_x, \alpha_i, t, \tau)$
$i^* = \arg\min_i c(i)$, $\alpha^* = X(i^*,:)$
if $c(i^*) < \theta$ **then**
  return $(i^*, j), \alpha^*$
return null

Computational speedups: From the computational point of view, the above approach requires computing a matrix inversion per α. With a cubic algorithm for matrix inversion, this could induce significant computational cost. However, we observe that this can be done efficiently as all the matrix inversions we need to perform are for matrices that differ only in their last row – the one occupied by α.

Recall that the least-squares solution of the system $A_x y = t$ is $y = (A_x^T A_x)^{-1} A_x^T t$. Now in the extended system $\begin{bmatrix} A_x \\ \alpha \end{bmatrix} \tilde{y} = \begin{bmatrix} t \\ \tau \end{bmatrix}$ or $\tilde{A}_x \tilde{y} = \tilde{\tau}$, the corresponding solution is $\tilde{y} = (\tilde{A}_x{}^T \tilde{A}_x)^{-1} \tilde{A}_x{}^T \tilde{t}$. Observe that we can write:

$$\tilde{A}_x{}^T \tilde{A}_x = \begin{bmatrix} A_x^T & \alpha^T \end{bmatrix} \begin{bmatrix} A \\ \alpha \end{bmatrix} = A_x^T A_x + \alpha^T \alpha.$$

Thus, $\tilde{A}_x{}^T \tilde{A}_x$ can be seen as a rank-one update to $A_x^T A_x$. In such a setting the Sherman-Morrison Formula [6] provides a way to efficiently calculate $D = (\tilde{A}_x{}^T \tilde{A}_x)^{-1}$ given $C = (A_x^T A_x)^{-1}$. The details are shown in Algorithm 1. Using the Sherman-Morrison Formula we can find $\tilde{y}$ via matrix multiplication, which requires $O(r^2)$ for at most $n = \max\{n_1, n_2\}$ candidate queries. Since the values of r we encounter in real datasets are small constants (in the range of 5-40), this running time is small.

The pseudocode of this process is shown in Algorithm 2. The process of selecting the right entry to query is summarized in the `Stabilize` routine. Observe that `Stabilize` either returns the entry to be queried, or if there is no entry that can lead to a stable systems it returns null. In the latter case the system is moved to the end of the order.

4.4 Putting everything together

Given all the steps we described above we are now ready to summarize `Order&Extend` in Algorithm 3.

Algorithm 3 The `Order&Extend` algorithm

Input: T_Ω, r, θ
Compute G_Ω
Find ordering π (as per Section 4.2)
for $A_x y = t$ (corresponding to the j-th column of Y) encountered in π **do**
  solve_system $= true$
  if $A_x y = t$ is incomplete **then**
    Query T and complete $A_x y = t$
  while `local_condition`$(A_x, t) > \theta$ **do**
    if $\{(i^*, j), \alpha^*\} =$ `Stabilize`$(A_x, t, \theta) \neq$ null **then**
      $A_x = \begin{bmatrix} A_x \\ \alpha^* \end{bmatrix}$
      $t = \begin{bmatrix} t \\ T(i^*, j) \end{bmatrix}$
    else
      move $A_x y = t$ to the end of π
      solve_system $= false$
      break
  if solve_system **then**
    $Y(:, j) = y = A_x^\dagger t$ (using least squares)
return $\widehat{T} =$ XY

`Order&Extend` constructs the rows of X and columns of Y in the order prescribed by π – the pseudocode shows the construction of columns of Y, but it is symmetric for the rows of X. For every linear system the algorithm encounters, it completes the system if it is incomplete and tries to make it stable if it is unstable. When a complete and stable version of the system is found, the system is solved using least squares. Otherwise, it is moved to the end of π.

Running time: The running time of `Order&Extend` consists of the time to obtain the initial ordering, which using the algorithm of Matula and Beck [12] is $O(n_1 + n_2)$, plus the time to detect and alleviate incomplete and unstable systems. Recall that for each unstable system we compute an inverse $O(r^3)$ and check n candidates $O(r^3 + r^2 n)$. Thus the overall running time of our algorithm is $O\big((n_1 + n_2) + N \times (r^3 + r^2 n)\big)$, where N is the number of unstable system the algorithm encounters. In practice, the closer a matrix is to being of rank exactly r, the smaller the number of error prone systems it encounters and therefore the faster its execution time.[2]

Partial completions: If the budget b of allowed queries is not adequate to resolve the incomplete or the unstable systems, then `Order&Extend` will output $\widehat{T}$ with only a portion of the entries completed. The entries that remain unrecovered are those for which the algorithm claims inability to produce a good estimate. From the practical viewpoint this is extremely useful information as the algorithm is able to inform the data analyst which entries it was not able to reconstruct from the observations in T_Ω.

[2]Code and information are available at `http://cs-people.bu.edu/natalir/matrixComp`

5. EXPERIMENTS

In this section we experimentally evaluate the performance of `Order&Extend` both in terms of reconstruction error as well as the number of queries it makes. Our experiments show that across all datasets `Order&Extend` requires very few queries to achieve a very low reconstruction error. All other baselines we compare against require many more queries for the same level of error, or can ever achieve the same level of reconstruction error.

Datasets: We experiment on the following nine real-world datasets, taken from a variety of applications.

MovieLens: This dataset contains ratings of users for movies as appearing in the MovieLens website.[3] The original dataset has size $6\,040 \times 3\,952$ and only 5% of its entries are observed. For our experiments we obtain a denser matrix of size $4\,832 \times 3\,162$.

Netflix: This dataset also contains user movie-ratings, but from the Netflix website. The dataset's original size is $480\,189 \times 17\,770$ with 1% of observed entries. Again we focus on a submatrix with higher percentage of observe entries and size $48\,019 \times 8\,885$.

Jester: This dataset corresponds to a collection of user joke ratings obtained for joke recommendation on the Jester website.[4] For our experiments we use the whole dataset with size $23\,500 \times 100$ with 72% of its entries being observed.

Boat: This dataset corresponds to a fully-observed black and white image of size 512×512.

Traffic: This is a set of four datasets; each is part of a traffic matrix from a large Internet Service Provider where rows and the columns are source and destination prefixes (i.e., groups of IP addresses), and each entry is the volume of traffic between the corresponding source-destination pair. The largest dataset size $7\,371 \times 7\,430$ and 0.1% of its entries are observed; we call this *TrafficSparse*. The other two are fully-observed of sizes $2\,016 \times 107$, and $2\,016 \times 121$; we call these *Traffic1* and *Traffic2*.[5]

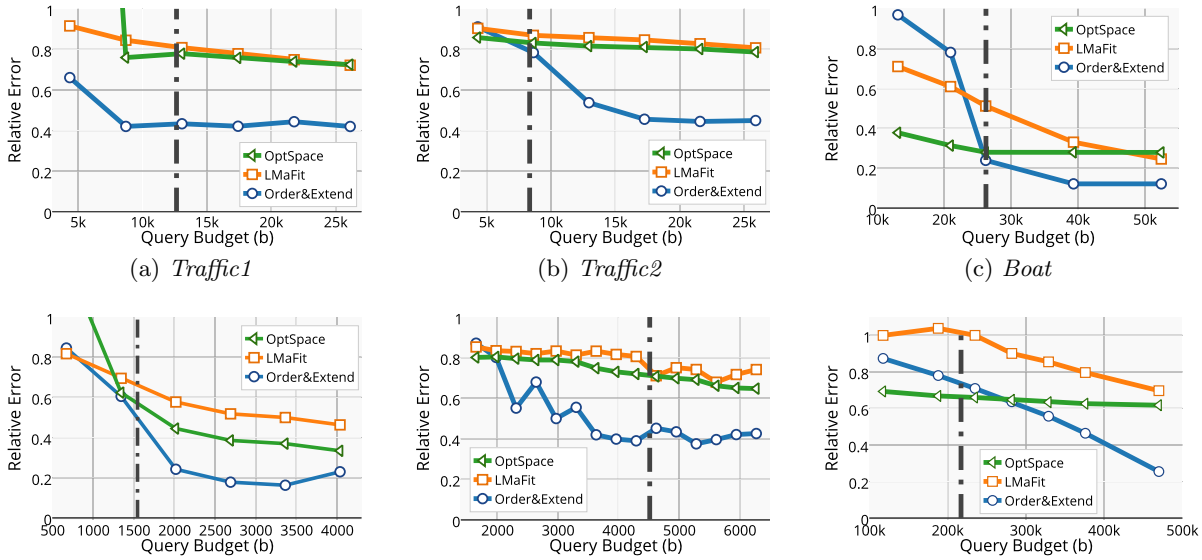
Latency: Here we use two datasets consisting of Internet network delay measurements. Rows and columns are hosts, and each entry indicates the minimum ping delay among a particular time window. The datasets are fully-observed and of sizes 116×116, and 869×19; we call these *Latency1* and *Latency2*.[5]

Baseline algorithms: We compare the performance of our algorithm to two state-of-the-art matrix-completion algorithms, `OptSpace` and `LmaFit` .

`OptSpace`: An SVD-based algorithm introduced by Keshavan et al.[8]. The algorithm centers around a convex-optimization step that aims to minimize the disagreement of the estimate $\widehat{T}$ on the initially observed entries T_Ω. We use the original implementation of `OptSpace`.[6]

`LmaFit`: A popular alternating least-squares method for matrix completion [19]. In our experiments we use the orig-

[3]Source `http://www.grouplens.org/node/73`.
[4]Source `http://goldberg.berkeley.edu/jester-data/`
[5]Source `https://www.cs.bu.edu/~crovella/links.html`
[6]`http://web.engr.illinois.edu/~swoh/software/optspace/code.html`



(a) *Traffic1* (b) *Traffic2* (c) *Boat*

(d) *Latency1* (e) *Latency2* (f) *Jester*

Figure 4: RelError of completion achieved by `Order&Extend`, `LmaFit` and `OptSpace` on datasets with *approximate* rank; x-axis: query budget b; y-axis: RelError of the completion.

inal implementation of this algorithm provided by Wen et. al. [7], and in particular the version where the rank r is provided, as we observed it to perform best.

As neither `OptSpace` nor `LmaFit` are algorithms for active completion, we set up our experiment as follows: first, we run `Order&Extend` on T_{Ω_0}, which asks a budget of b queries. Before feeding T_{Ω_0} to `LmaFit` and `OptSpace` we extend it with b *randomly* chosen queries. In this way both algorithms query the same number of additional entries. A random distribution of observed entries has been proved to be (asymptotically) optimal for statistical methods like `OptSpace` and `LmaFit` [2, 8, 19]. Therefore, picking randomly distributed b additional entries is the best querying strategy for these algorithms, and we have also verified that experimentally.

5.1 Methodology

For all our experiments, the ground-truth matrix T is known but not fully revealed to the algorithms. The input to the algorithms consists of an *initial mask* Ω_0, the observed matrix T_{Ω_0}, and a budget b on the number of queries they can ask. Each algorithm $\mathcal{A}$ outputs an estimate $\widehat{T}_{\mathcal{A},\Omega_0}$ of T.

Selecting the input mask Ω_0: The initial mask Ω_0, with cardinality m_0 is selected by picking m_0 entries uniformly at random from the ground-truth matrix T.[8] The cardinality m_0 is selected so that $m_0 > 0$ and $m_0 < \phi(T, r)$; usually we chose m_0 to be $\approx 30-50\%$ of $\phi(T, r)$. The former constraint guarantees that the input is not trivial, while the latter guarantees that additional queries are definitely needed.

Range for the query budget b: We vary the number of queries,b, an algorithm can issue among a wide range of values. Starting with $b < \phi(T, r) - m_0$, we gradually increase it until we see that the performance of our algorithms stabilize (i.e., further queries do not decrease the reconstruction error). Clearly, the smaller the value of b the larger the reconstruction error of the algorithms.

Reconstruction error: Given a ground-truth matrix T and input T_{Ω_0}, we evaluate the performance of a reconstruction algorithm $\mathcal{A}$, by computing the relative error of $\widehat{T}_{\mathcal{A},\Omega_0}$ with respect to T, using the RelError function defined in Equation (1). This measure takes into consideration *all* entries of T, both the observed and the unobserved. The closer $\widehat{T}_{\mathcal{A},\Omega}$ is to T the smaller the value of $\textsc{RelError}(\widehat{T}_{\mathcal{A},\Omega})$. In general, $\textsc{RelError}(\widehat{T}_{\mathcal{A},\Omega}) \in [0, \infty)$ and at perfect reconstruction $\textsc{RelError}(\widehat{T}_{\mathcal{A},\Omega}) = 0$.

Although our baseline algorithms always produce a full estimate (i.e., they estimate all missing entries), `Order&Extend` may produce only partial completions (see Section 4.4 for a discussion in this). In these cases, we assign value 0 to the entries it does not estimate.

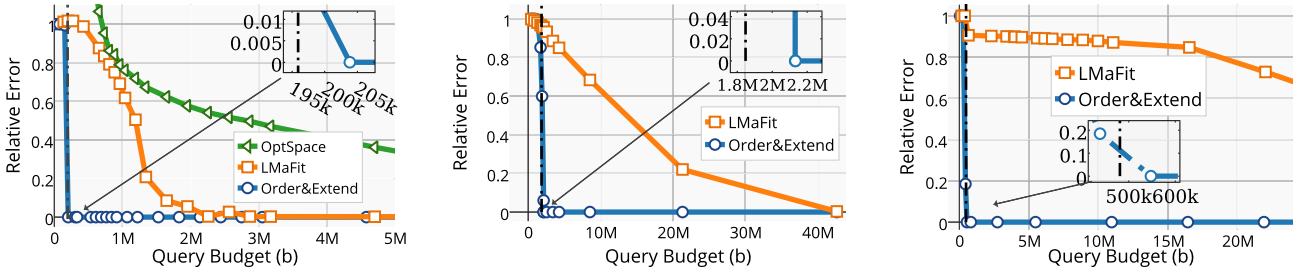
5.2 Evaluating `Order&Extend`

Experiments with real noisy data: For our first experiment, we use datasets for which we know all off the entries. This is true for six out of our nine datasets: *Traffic1* ,*Traffic2*, *Latency1*, *Latency2*, *Jester*, *Boat*. Note that *Jester* is missing 30% of the entries, but we treat them as true zero-values ratings; the remaining datasets are fully known and able to be queried as needed. As these are real datasets they are not exactly low rank, but plotting their singular values reveals that they have low effective rank. By inspecting their singular values, we chose: $r = 7$ for the *Traffic* and *Latency* datasets, $r = 10$ for *Jester* and $r = 40$ for *Boat*.

Figure 4 shows the results for each dataset. The x-axis is the query budget b; note that while `LmaFit` and `OptSpace` always exhaust this budget, for `Order&Extend` it is only an upper bound on the number of queries made. The y-axis is

[7] `http://lmafit.blogs.rice.edu/`

[8] We also test other sampling distributions, but the results are the same as the ones we report here and thus omitted.



(a) *MovieLens* (b) *Netflix* (c) *TrafficSparse*

Figure 5: RelError of completion achieved by `Order&Extend`, `LmaFit` and `OptSpace` on datasets with *exact* rank; x-axis: query budget b; y-axis: RelError of the completion.

the $\text{RelError}(T, \widehat{T}_{\mathcal{A},\Omega})$. The black vertical line marks the number of queries needed to reach the critical mask size; i.e., it corresponds to budget of $(\phi(T,r) - m_0)$. One should interpret this line as a very conservative lower bound on the number of queries that an optimal algorithm would need to achieve errorless reconstruction in the absence of noise.

From the figure, we observe that `Order&Extend` exhibits the lowest reconstruction error across all datasets. Moreover, it does so with a very small number of queries, compared to `LmaFit` and `OptSpace`; the latter algorithms achieve errors of approximately the same magnitude in all datasets. On some datasets `LmaFit` and `OptSpace` come close to the relative error of `Order&Extend` though with significantly more queries. For example for the *Latency1* dataset, `Order&Extend` achieves error of 0.24 with $b = 2K$ queries; `LmaFit` needs $b = 4K$ to exhibit an error of 0.33, which is still more than that of `Order&Extend`. In most datasets, the differences are even more pronounced; e.g., for *Traffic2*, `Order&Extend` achieves a relative error of 0.50 with about $b = 13K$ queries; `OptSpace` and `LmaFit` achieve error of more than 0.8 even after $b = 26K$ queries. Such large differences between `Order&Extend` and the baselines appear in all datasets, but *Boat*. For that dataset, `Order&Extend` is still better, but not as significantly as in other cases – likely an indication that the dataset is more noisy. We also point out that the value of b for which the relative error of `Order&Extend` exhibits a significant drop is much closer to the indicated lower bound by the black vertical line. Again this phenomenon is not so evident for *Boat* probably because this datasets is further away from being low rank.

Extremely sparse real-world data: For the purpose of experimentation our algorithm needs to have access to all the entries of the ground truth matrix T – in order to be able to reveal the values of the queried entries. Unfortunately, the *Movielens*, *Netflix*, and *TrafficSparse* datasets consist mostly of missing entries, therefore we cannot query the majority of them. To be able to experiment with these datasets, we overcome this issue by approximating each dataset with its closest rank r matrix T_r. The approximation is obtained by first assigning 0 to all missing entries of the observed T, and then taking the singular value decomposition and setting all but the largest r singular values to zero. This trick grants us the ability to study the special case discussed in Section 3 where the matrix is of exact rank r.

Using $r = 40$, the results for these datasets are depicted in Figure 5 with the same axes and vertical line as in Figure 4. Again, we observe a clear dominance of `Order&Extend`. In this case the differences in the relative error it achieves are much more striking. Moreover, `Order&Extend` achieves almost 0 relative error for extremely small number of queries b; in fact the error of `Order&Extend` consistently drops to an extremely small value for b very close to the lower bound of the optimal algorithm (as marked by the black vertical line shown in the plot). On the other hand `LmaFit` and `OptSpace` are far from exhibiting such a behavior. This signals that `Order&Extend` devises a querying strategy that is almost optimal. Interestingly the performance of `OptSpace` changes dramatically in these cases as compared to the approximate rank datasets. In fact on *TrafficSparse* and *Netflix* the error is so high it does not appear on the plot.

Note that the striking superiority of `Order&Extend` in the case of exact-rank matrices is consistent across all datasets we considered, including others not shown here.

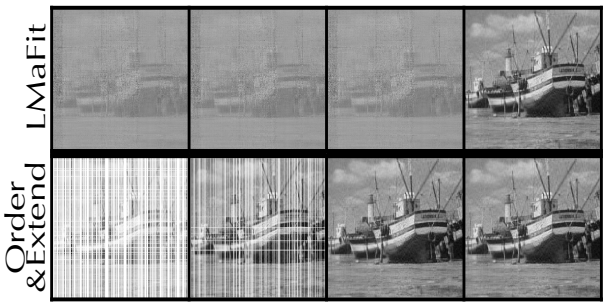


Figure 6: Recovery process using `LmaFit`, and `Order&Extend`. Each column is a particular b, increasing from left to right.

Running times: Though the algorithmic composition is quite different, we give some indicative running times for our algorithm as well as `LmaFit` and `OptSpace`. For example, in the *Netflix* dataset the running times were in the order of 11 000 seconds for `LmaFit`, 80 000 seconds for `Order&Extend`, and 200 000 seconds for `OptSpace`. These numbers indicate that `Order&Extend` is efficient despite the fact that in addition to matrix completion it also identifies the right queries; the running times of `LmaFit` and `OptSpace` simply correspond to running a single completion on the extended mask that is randomly formed. Note that these running times are

computed using an unoptimized and serial implementation of our algorithm; improvements can be achieved easily e.g., by parallelizing the local condition number computations.

Partial completion of `Order&Extend`: As a final experiment, we provide anecdotal evidence that demonstrates the difference in the philosophy behind `Order&Extend` and other completion algorithms. Figure 6 provides a visual comparison of the recovery process of `Order&Extend` and `LmaFit` for different values of query budget b. For small values of b, `Order&Extend` does not have the sufficient information to resolve all incomplete and unstable systems. Therefore the algorithm does not estimate the entries of T corresponding to these systems, which renders the white areas in the two left-most images of `Order&Extend` . In contrast `LmaFit` outputs full estimates, though with significant error. This can be seen by incremental sharpening of the image, compared to the piece-by-piece reconstruction of `Order&Extend`.

5.3 Discussion

Here we discuss some alternatives we have experimented with, but omitted due to significantly poorer performance.

Alternative querying strategies: `Order&Extend` uses a rather intricate strategy for choosing its queries to T. A natural question is whether a simpler strategy would be sufficient. To address this we experimented with versions of `Sequential` that considered the same order as `Order&Extend` but when stuck with a problematic system they queried either randomly, or with probability proportional (or inversely proportional) to the number of observed entries in a cell's row or column. All these variants were significantly and consistently worse than the results we reported above.

Condition number: Instead of detecting unstable systems using the local condition number we also experimented with a modified version of `Order&Extend`, which characterized a system $A_x y = t$ as unstable if its condition number $\kappa(A_x)$ was above a threshold. For values of threshold between 5 and 100 the results were consistently and significantly worse than the results of `Order&Extend` that we report here, both in terms of queries and in terms of error. Further, there was no threshold of the condition number that would perform comparably to `Order&Extend` for any dataset.

6. CONCLUSIONS

In this paper we posed the ActiveCompletion problem, an active version of matrix completion, and designed an efficient algorithm for solving it. Our algorithm, which we call `Order&Extend`, approaches this problem by viewing querying and completion as two interrelated tasks and optimizing for both simultaneously. In designing `Order&Extend` we relied on a view of matrix completion as the solution of a sequence of linear systems, in which the solutions of earlier systems become the inputs for later systems. In this process, reconstruction error depends both on the order in which systems are solved and on the stability of each solved system. Therefore, a key idea of `Order&Extend` is to find an ordering for the systems in which as many as possible give good estimates of the unobserved entries. However, even in the perfect order problematic systems arise; `Order&Extend` employs a set of techniques for detecting these systems and alleviating them by querying a small number of additional entries from the true matrix. In a wide set of experiments with real data we demonstrated the efficiency of our algorithm and its superiority both in terms of the number of queries it makes, and the error of the reconstructed matrices it outputs.

Acknowledgments: This research was supported in part by NSF grants CNS-1018266, CNS-1012910, IIS-1421759, IIS-1218437, CAREER-1253393, IIS-1320542, and IIP-1430145. We also thank the anonymous reviewers for their valuable comments and suggestions.

7. REFERENCES

[1] S. Bhojanapalli and P. Jain. Universal Matrix Completion. *ArXiv e-prints*, 2014.

[2] E. J. Candès and B. Recht. Exact matrix completion via convex optimization. *Commun. ACM*, 2012.

[3] E. J. Candès and T. Tao. The power of convex relaxation: near-optimal matrix completion. *IEEE Transactions on Information Theory*, 2010.

[4] S. Chakraborty, J. Zhou, V. N. Balasubramanian, S. Panchanathan, I. Davidson, and J. Ye. Active matrix completion. In *ICDM*, 2013.

[5] Y. Chen, S. Bhojanapalli, S. Sanghavi, and R. Ward. Coherent matrix completion. In *ICML*, 2014.

[6] G. H. Golub and C. F. V. Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences, 2012.

[7] P. Jain, P. Netrapalli, and S. Sanghavi. Low-rank matrix completion using alternating minimization. In *STOC*, pages 665–674, 2013.

[8] R. H. Keshavan, A. Montanari, and S. Oh. Matrix completion from a few entries. *IEEE Transactions on Information Theory*, 2010.

[9] R. H. Keshavan, A. Montanari, and S. Oh. Matrix completion from noisy entries. *JMLR*, 2010.

[10] F. J. Király, L. Theran, R. Tomioka, and T. Uno. The algebraic combinatorial approach for low-rank matrix completion. *CoRR*, 2013.

[11] F. J. Király and R. Tomioka. A combinatorial algebraic approach for the identifiability of low-rank matrix completion. In *ICML*, 2012.

[12] D. W. Matula and L. L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *Journal of the ACM (JACM)*, 30(3):417–427, 1983.

[13] R. Meka, P. Jain, and I. S. Dhillon. Matrix completion from power-law distributed samples. In *NIPS*, 2009.

[14] S. Negahban and M. J. Wainwright. Restricted strong convexity and weighted matrix completion: Optimal bounds with noise. *JMLR*, 2012.

[15] B. Recht. A simpler approach to matrix completion. *The Journal of Machine Learning Research*, 2011.

[16] A. Singer and M. Cucuringu. Uniqueness of low-rank matrix completion by rigidity theory. *SIAM J. Matrix Analysis Applications*, 2010.

[17] D. J. Sutherland, B. Póczos, and J. Schneider. Active learning and search on low-rank matrices. In *ACM SIGKDD*, 2013.

[18] L. N. Trefethen and D. Bau III. *Numerical linear algebra*. SIAM Press, 1997.

[19] Z. Wen, W. Yin, and Y. Zhang. Solving a low-rank factorization model for matrix completion by a nonlinear successive over-relaxation algorithm. *Math. Program. Comput.*, 2012.