Parallel Stochastic Gradient Algorithms for Large-Scale Matrix Completion

Benjamin Recht and Christopher Ré
Computer Sciences Department, University of Wisconsin-Madison
1210 W Dayton St, Madison, WI 53706

April 2011

Abstract

This paper develops Jellyfish, an algorithm for solving data-processing problems with matrix-valued decision variables regularized to have low rank. Particular examples of problems solvable by Jellyfish include matrix completion problems and least-squares problems regularized by the nuclear norm or γ_2-norm. Jellyfish implements a projected incremental gradient method with a biased, random ordering of the increments. This biased ordering allows for a parallel implementation that admits a speed-up nearly proportional to the number of processors. On large-scale matrix completion tasks, Jellyfish is orders of magnitude more efficient than existing codes. For example, on the Netflix Prize data set, prior art computes rating predictions in approximately 4 hours, while Jellyfish solves the same problem in under 3 minutes.

Keywords. Matrix completion, Incremental gradient methods, Parallel computing, Multicore

1 Introduction

Manipulating and processing large, incomplete data-matrices constitutes a large component of contemporary data analysis. Recent results in *matrix completion* have demonstrated that if the underlying data-matrix is low rank, then the missing entries can be inferred from a vanishingly small fraction of the entries [8, 30]. This body of work demonstrates that the low-rank matrix can be determined as the minimizer of a convex programming problem. These theoretical results have spawned a surge of interest in data-processing algorithms for solving the associated convex programs [7, 15, 21, 24, 37], but none of these methods can currently operate on the scale of data commonly acquired by internet retailers and social networking websites. In response, we present Jellyfish, a parallel incremental gradient algorithm, which can solve problems with millions of rows and columns in minutes on commodity multicore technology.

For moderately sized matrix completion problems (less than 5000 columns), first-order methods based on convex programming are both fast and effective. However, these algorithms require computation of singular values at every iteration and become prohibitively expensive to run when the size of the matrices and number of observations grows beyond a few thousand. To scale to larger data sets, a different paradigm of algorithms is required. Recently, there have been a variety of techniques proposed to factor large scale matrices using incremental, non-convex heuristics [12, 19, 20, 31, 33]. These algorithms obviate the singular value decomposition by explicitly factoring the decision variable and use incremental gradient methods that operate upon very small batches of

data at each iteration. Non-convex methods have been quite successful on real problems, including the Netflix Prize problem, and theoretical analyses suggest that these heuristics succeed in similar sampling regimes as their convex programming counterparts [16, 30].

In this paper, we demonstrate how to implement high-dimensional matrix factorization heuristics in parallel using incremental gradient descent on multicore processors. In particular, using a specially biased ordering of the incremental gradients, we can achieve paralellization with no fine-grained memory locking. This in turn results in speed-ups that are nearly linear in the number of processors assigned to the problem.

We describe the elements of our algorithm, Jellyfish, in Section 2 and describe the algorithm in full detail in Section 3. Here we also describe certain aspects of multicore technology that we can take full advantage of in our framework. For instance, we devote an entire core just to ordering and partitioning the problem data for each pass over the matrix. We demonstrate experimentally that our biased orders do not introduce any reduction in speed or accuracy over standard sampling schemes and show that we get nearly linear speed-up with the number of cores.

Our experiments demonstrate that on small-scale problems, Jellyfish is competitive with existing matrix completion algorithms in accuracy but is 5 to 20 times faster. On large-scale matrix completion tasks, Jellyfish is orders of magnitude more efficient than existing codes. We show that on a multicore workstation, we can solve for matrices with millions of rows and columns in under 18 minutes. On the Netflix Prize data set, Jellyfish solves the associated inference problem in under 3 minutes while prior art required hours.

2 Algorithmic Framework

We briefly review the mathematical foundations of our algorithmic approach. In Section 2.1, we begin by defining the main optimization problems we seek to solve, and describe several convex penalty functions that encourage low-rank matrix solutions. We proceed to show in Section 2.2 that these matrix regularizers can be implemented by explicitly factoring the decision variable into a product of two low-dimensional matrices. This factorization results in a dramatic reduction in the dimension of the search space. In Section 2.3, we then summarize the elements of the projected incremental gradient method which we will employ in the sequel to solve our factored matrix regularization problems.

2.1 Factorization norms and matrix regularizers

Our focus is on a particular class of optimization problems that have a separable cost and constraint structure. Our decision variable $\boldsymbol{X}$ will be a matrix with n_r rows and n_c columns. Without loss of generality, we will assume $n_r \leq n_c$ throughout. We consider problems of the form

$$\text{minimize} \sum_{(i,j)\in\Omega} f_{ij}(X_{ij}) + P(\boldsymbol{X})$$

Here Ω is a subset of $\{1, \dots, n_r\} \times \{1, \dots, n_c\}$ denoting pairs of entries to process. f_{ij} will be a convex function of a scalar. For example, f_{ij} could measure the squared distance from X_{ij} to another scalar M_{ij}, i.e., $f_{ij}(X_{ij}) = (X_{ij} - M_{ij})^2$.

$P : \mathbb{R}^{n_r \times n_c} \to \mathbb{R}$ will be a matrix regularizer which will control some measure of complexity of the matrix $\boldsymbol{X}$. We will be particularly interested in matrix regularizers that promote low-rank

decompositions, motivating our heuristic factorization of $\boldsymbol{X}$. The first regularizer we will study is the *nuclear norm*

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

where σ_k are the singular values of $\boldsymbol{X}$. The nuclear norm (also known as the Schatten 1-norm, the Ky Fan 1-norm, and the trace-norm) forms the foundation for the work on matrix completion, and has been demonstrated to recover low-rank matrices from highly incomplete, noisy data in both theory and practice [8, 9, 13, 29, 30].

A particularly attractive feature of the nuclear norm is that it can be reformulated as a penalty over all possible factorizations of $\boldsymbol{X}$:

$$\|\boldsymbol{X}\|_* = \inf\left\{ \tfrac{1}{2}\|\boldsymbol{L}\|_F^2 + \tfrac{1}{2}\|\boldsymbol{R}\|_F^2 \ \ : \ \boldsymbol{X} = \boldsymbol{L}\boldsymbol{R}^* \right\} \tag{2}$$

where

$$\|\boldsymbol{A}\|_F = \left(\sum_{i=1}^{n_r} \sum_{j=1}^{n_c} A_{ij}^2 \right)^{1/2}$$

is the *Frobenius* or Euclidean norm of $\boldsymbol{A}$ [30]. That is, the nuclear norm is equal to the minimum Frobenius norm factorization of $\boldsymbol{X}$. This minimum is achieved by the singular value decomposition: if $\boldsymbol{X} = \boldsymbol{U}\Sigma\boldsymbol{V}^*$, then $\boldsymbol{L} = \boldsymbol{U}\Sigma^{1/2}$ and $\boldsymbol{R} = \boldsymbol{V}\Sigma^{1/2}$ achieve the minimum value of (2). A quick calculation reveals that the pair $(\boldsymbol{L}, \boldsymbol{R})$ satisfies $\|\boldsymbol{X}\|_* = \frac{1}{2}\|\boldsymbol{L}\|_F^2 + \frac{1}{2}\|\boldsymbol{R}\|_F^2$. To see that no lower value is achievable requires an appeal to semidefinite programming and is proven in Recht *et al* [30].

The other main regularizer of interest is the γ_2-norm [14] (also known as the max-norm [20, 34, 35]). The γ_2-norm also can be computed as a minimization over all possible factorizations of $\boldsymbol{X}$:

$$\|\boldsymbol{X}\|_{\gamma_2} := \inf\left\{ \max\left(\|\boldsymbol{L}\|_{2,\infty}^2, \|\boldsymbol{R}\|_{2,\infty}^2 \right) : \boldsymbol{X} = \boldsymbol{L}\boldsymbol{R}^* \right\} \tag{3}$$

where $\|\cdot\|_{2,\infty}$ denotes the maximum-row-norm of a matrix:

$$\|\boldsymbol{A}\|_{2,\infty} := \max_j \left(\textstyle\sum_k \boldsymbol{A}_{jk}^2 \right)^{1/2}.$$

To compute a factorization $\boldsymbol{L}\boldsymbol{R}^*$ which minimizes (3), one can solve the semidefinite program

$$\|\boldsymbol{X}\|_{\gamma_2} := \inf\ t \ \text{ subject to } \begin{bmatrix} \boldsymbol{A} & \boldsymbol{X} \\ \boldsymbol{X}^* & \boldsymbol{B} \end{bmatrix} \succeq 0\,, \ \operatorname{diag}(\boldsymbol{A}) \le t\,, \ \operatorname{diag}(\boldsymbol{B}) \le t\,,$$

and factor the optimizer using an eigenvalue decomposition. Such γ_2-norm regularized problems appear in applications in collaborative filtering [35], genetic networks [36], and combinatorial optimization [1].

We can thus interpret both the nuclear norm and γ_2-norm as penalties on the rows of factorizations of $\boldsymbol{X}$. The nuclear norm penalizes the average squared Euclidean norm of the rows of $\boldsymbol{L}$ and $\boldsymbol{R}$ while the γ_2-norm penalizes the maximum row-norm. The techniques we detail below can be generalized to any convex function of the row-norms of the factors of $\boldsymbol{X}$.

2.2 Low rank approximations

The first approximation that we make is to assume the decision variable $\boldsymbol{X}$ has rank at most r. If this is the case then $\boldsymbol{X}$ can be written as $\boldsymbol{L}\boldsymbol{R}^*$ where $\boldsymbol{L}$ is $n_r \times r$, $\boldsymbol{R}$ is $n_c \times r$. This reformulation dramatically reduces the number of primal decision variables from $n_r n_c$ to $(n_r + n_c)r$. With this reduction, we can store the entire decision variable in memory even if n_r and n_c are in the millions.

Using this explicit factorization, we can approximate the nuclear norm problem

$$\text{minimize} \sum_{(i,j)\in\Omega} f_{ij}(X_{ij}) + \mu \|\boldsymbol{X}\|_* \tag{4}$$

with the optimization problem

$$\text{minimize} \sum_{(i,j)\in\Omega} f_{ij}(\boldsymbol{L}_i \boldsymbol{R}_j^*) + \tfrac{\mu}{2} \|\boldsymbol{L}\|_F^2 + \tfrac{\mu}{2} \|\boldsymbol{R}\|_F^2 \tag{5}$$

and $\boldsymbol{L}_i$ (resp. $\boldsymbol{R}_j$) denotes the ith (resp. jth) row of $\boldsymbol{L}$ (resp. $\boldsymbol{R}$).

Similarly, the γ_2-norm constrained problem

$$\text{minimize} \sum_{(i,j)\in\Omega} f_{ij}(X_{ij}) \quad \text{subject to } \|\boldsymbol{X}\|_{\gamma_2} \le B\,, \tag{6}$$

is approximated by the factored problem

$$\text{minimize} \sum_{(i,j)\in\Omega} f_{ij}(\boldsymbol{L}_i \boldsymbol{R}_j^*) \quad \text{subject to } \|\boldsymbol{L}\|_{2,\infty}^2 \le B,\ \|\boldsymbol{R}\|_{2,\infty}^2 \le B\,. \tag{7}$$

These factored optimization problems have fewer explicit decision variables than their counterparts in Section 2.1. However, the new formulations are nonconvex and potentially subject to stationary points that are not globally optimal. Fortunately, under relatively mild assumptions, one can show that the optimal values of (5) and (7) coincide with the optimal values of (4) and (6) respectively. For instance, using the analysis of Burer and Monteiro, all of the local minima of (5) and (7) are global solutions provided that r is chosen to be larger than the rank of the optimal solution of (4) and (6) respectively [6].

2.3 Projected incremental gradient methods

Shifting gears for a moment, consider a general optimization problem of the form

$$\min_{\boldsymbol{x}\in\mathbb{R}^d} \sum_{i=1}^{N} f_i(\boldsymbol{x}) + P(\boldsymbol{x}) \tag{8}$$

where the f_i are differentiable and P is a convex extended-real valued function.

Projected incremental gradient methods are a class of optimization algorithms ideally suited to solve (8). These methods are summarized by the following iterative rule that describes how one produces the $(k+1)$-th iterate, $\boldsymbol{x}^{(k+1)}$, given $\boldsymbol{x}^{(k)}$.

$$\boldsymbol{x}^{(k+1)} = \Pi_{\alpha P}\left(\boldsymbol{x}^{(k)} - \alpha_k \nabla f_{\eta(k)}(\boldsymbol{x}^{(k)})\right) \tag{9}$$

In this expression, $\eta(k)$ is a subset of $\{1, \ldots, N\}$, chosen at each iteration. $f_{\eta(k)}$ is the approximation $f_{\eta(k)}(\boldsymbol{x}) = \sum_{j \in \eta(k)} f_j(\boldsymbol{x})$. α_k is a positive step-size parameter. The function $\Pi_{\alpha P}$ is given by the expression

$$\Pi_{\alpha P}(\boldsymbol{x}) = \arg\min_{\boldsymbol{w}} \tfrac{1}{2}\|\boldsymbol{x} - \boldsymbol{w}\|_2^2 + \alpha P(\boldsymbol{w})$$

$\Pi_{\alpha P}$ is called *a proximal-point operator*, and in the case where P is the indicator function of a set C,

$$P(x) = \begin{cases} 0 & x \in C \\ \infty & \text{otherwise} \end{cases},$$

$\Pi_{\alpha P}$ is simply the Euclidean projection onto C [32]. An example proximal-point operator ensures that the model has unit Euclidean norm by projecting the model on to the the unit ball. $P(\boldsymbol{x})$ might also be a regularization penalty such as total-variation or negative entropy. These methods are called incremental because they minimize an objective function by examining an item or a small subset of items, modifying the model, and then fixing (projecting) the model to satisfy priors or constraints (e.g., norm constraints).

In incremental methods, one tries to improve the cost function by following an approximate gradient constructed from a very small subset of the summands. When the functions $\sum_{i=1}^{n} f_i(\boldsymbol{x})$ and $P(\boldsymbol{x})$ are both convex, the incremental projected gradient method is guaranteed to converge to a globally optimal solution [27]. Commonly, the set η is chosen by sampling a subset of a given size, but the method converges even if η_k is chosen deterministically, although at a possibly slower rate [3, 22, 23, 25, 38].

Many authors have proposed using projected subgradient methods on problems (5) and (7) [12, 19, 20, 33]. Indeed, one of the first publicly announced methods for attacking the Netflix Prize minimized (5) with $f_{ij}(\boldsymbol{L}_i \boldsymbol{R}_j) = (\boldsymbol{L}_i \boldsymbol{R}_j - M_{ij})^2$ [12]. In what follows, we describe a novel sampling scheme for problems (5) and (7) that allow for dramatic speed-ups over this prior art.

3 Algorithm Description and Implementation Details

The basic operation of our algorithm is that of incremental projected gradient sampling *one pair* (i,j) from Ω at each iteration. We sample a term (i, j) according to some distribution and then apply step (9). Our main contribution lies in exactly *how* we sample the entries. As we now describe, we choose a biased permutation of the set Ω which allows for massive parallelization of the gradient computations. In Section 4, we demonstrate that on large data sets, this bias introduces minimal loss in the convergence rate while enabling eleven-fold speed-ups on a twelve-core machine.

3.1 Elements of individual iterations

For (5), we write the problem as

$$\text{minimize} \sum_{(i,j) \in \Omega} \left\{ f_{ij}(\boldsymbol{L}_i \boldsymbol{R}_j^*) + \tfrac{\mu}{2|\Omega_{i-}|}\|\boldsymbol{L}_i\|_F^2 + \tfrac{\mu}{2|\Omega_{-j}|}\|\boldsymbol{R}_j\|_F^2 \right\} \tag{10}$$

where $\Omega_{i-} = \{j \ : \ (i,j) \in \Omega\}$ and $\Omega_{-j} = \{i \ : \ (i,j) \in \Omega\}|$. That is, we weight penalty of $\boldsymbol{L}_i$ by the number of terms involving the ith row of the decision variable and the penalty of $\boldsymbol{R}_j$ by the

number of terms involving the jth column of the decision variable. Under this reformulation, (10) is a of the form (8) with $P = 0$. In this case, step (9) with $|\eta(k)| = 1$ becomes

$$\begin{aligned} \boldsymbol{L}_{i_k}^{(k+1)} &= \left(1 - \frac{\mu\alpha_k}{|\Omega_{i_k-}|}\right) \boldsymbol{L}_{i_k}^{(k)} - \alpha_k f'_{i_k j_k}\left(\boldsymbol{L}_{i_k}^{(k)} \boldsymbol{R}_{j_k}^{(k)*}\right) \boldsymbol{R}_{j_k}^{(k)} \\ \boldsymbol{R}_{j_k}^{(k+1)} &= \left(1 - \frac{\mu\alpha_k}{|\Omega_{-j_k}|}\right) \boldsymbol{R}_{j_k}^{(k)} - \alpha_k f'_{i_k j_k}\left(\boldsymbol{L}_{i_k}^{(k)} \boldsymbol{R}_{j_k}^{(k)*}\right) \boldsymbol{L}_{i_k}^{(k)} \end{aligned} \tag{11}$$

and all the other entries in $\boldsymbol{L}$ and $\boldsymbol{R}$ are unchanged.

For (7), step (9) becomes

$$\begin{aligned} \boldsymbol{L}_{i_k}^{(k+1)} &= \Pi_B\left(\boldsymbol{L}_{i_k}^{(k)} - \alpha_k f'_{i_k j_k}\left(\boldsymbol{L}_{i_k}^{(k)} \boldsymbol{R}_{j_k}^{(k)*}\right) \boldsymbol{R}_{j_k}^{(k)}\right) \\ \boldsymbol{R}_{j_k}^{(k+1)} &= \Pi_B\left(\boldsymbol{R}_{j_k}^{(k)} - \alpha_k f'_{i_k j_k}\left(\boldsymbol{L}_{i_k}^{(k)} \boldsymbol{R}_{j_k}^{(k)*}\right) \boldsymbol{L}_{i_k}^{(k)}\right) \end{aligned} \tag{12}$$

where

$$\Pi_B(\boldsymbol{v}) = \begin{cases} \frac{\sqrt{B}\cdot v}{\|v\|} & \|v\|^2 \geq B \\ v & \text{otherwise} \end{cases}.$$

Again, all of the other rows of $\boldsymbol{L}$ and $\boldsymbol{R}$ are unchanged. Each of these iterations can be computed in time linear in r and the time it takes to compute the gradient of the univariate function $f_{i_k j_k}$.

3.2 Epochs

Computing a solution to desired tolerance typically requires several passes over the data set. We call each of these passes an *epoch*. Classically, incremental gradient schemes are described as a two-part process: (1) given a data set, perform a with-replacement sample from that data set to obtain a single data item, and (2) take an (incremental) gradient step with respect to that data item. As we discussed in Section 2.3, part (1) of this procedure is unnecessary to guarantee convergence as any ordering of the elements of Ω will converge.

On the other hand, with-replacement sampling currently admits the best provable convergence rates. In fact, the provable rates for without-replacement sampling are significantly slower: for example, Nedic and Bertsekas show that one cannot guarantee closer proximity to the optimal solution than after a full gradient step [26]. However, there are reasons to suspect that without-replacement sampling is actually better on large data sets [5]. One reason is related to the *coupon collector phenomenon*. Informally, the coupon collector phenomenon says that if there are a large number of data values, say N, then it will require on average $N \log N$ samples to simply see each data item once (collect each coupon). Obviously, in without replacement sampling, all of the items will have been touched after exactly N samples. On small data sets, this $\log N$ is discrepancy not an issue. However, on the Netflix Prize data set analyzed in our experiments $\log N \approx 18$. One can construct instances, where with-replacement sampling would need 18 times more passes over a similarly sized data set to reach the same convergence as a without-replacement sample. Estimating the discrepancy between with and with-out replacement sampling is a long-standing problem in numerical optimization and machine learning [4, pg. 628] and is out of the scope of this paper. We defer such theoretical investigations to future work.

3.3 Parallelization via cyclic partitioning

We now describe our biased, without-replacement, sample ordering that allows us to parallelize stochastic gradient descent while obviating fine-grained locking. We describe our scheme, called *cyclic partitioning*, that achieves these goals.

$$\begin{bmatrix} L_1 \\ L_2 \\ L_3 \end{bmatrix} \begin{bmatrix} R_1 & R_2 & R_3 \end{bmatrix} = \begin{bmatrix} L_1R_1 & L_1R_2 & L_1R_3 \\ L_2R_1 & L_2R_2 & L_2R_3 \\ L_3R_1 & L_3R_2 & L_3R_3 \end{bmatrix}$$

Figure 1: Cyclic Partitioning. There are 9 Chunks in 3 rounds. Chunks in the same round are highlighted in the same way. Here, π_{row} and π_{col} are identity permutations.

Note that both of the basic iterations (11) and (12) act on very local portions of the matrix $\boldsymbol{L}$ and $\boldsymbol{R}$. Indeed, if we knew in advance that we were going to first take a gradient of $f_{i_1j_1}$ and then of $f_{i_2j_2}$, $i_1 \neq i_2$, $j_1 \neq j_2$, then we could calculate these two steps in parallel. Neither has any bearing on each other. More aggressively, if we were going to run a series of gradient steps from a set $S_1 = \{(i,j) \ : \ i \in I_1, \ j \in J_1\}$ and $S_2\{(i,j) \ : \ i \in I_2, \ j \in J_2\}$ with $I_1 \cap I_2 = \emptyset$ and $J_1 \cap J_2 = \emptyset$, we could run these operations completely in parallel. This observation underlies our main contribution.

Suppose we have access to p parallel processors. First, Jellyfish generates a permutation the rows and columns of the matrix, i.e., generates two (random) permutations of integers: π_{row}, a permutation on the row indexes, and π_{col}, a permutation on the column indexes. We then partition Ω into l^2 sets $C_{a,b}$ for $a, b = 1, \ldots, p$. Each $C_{a,b}$ is called a *chunk*. We define the contents of a chunk by the following rule. Given $(i,j) \in \Omega$ it is put in chunk $C_{a,b}$ where

$$a = \left\lfloor \frac{l}{n_r}(\pi_{row}(i) - 1) \right\rfloor + 1 \text{ and } b = \left\lfloor \frac{l}{n_c}(\pi_{col}(j) - 1) \right\rfloor + 1$$

This partitioning can be done using our without-replacement sampler: first select an element (i,j) from the remaining pool, then determine which chunk to assign it to.

We say that two chunks C_{ab} and $C_{a'b'}$ *overlap* if $a = a'$ or $b = b'$. For example, C_{11} overlaps C_{12} but C_{11} does not overlap C_{22}. Our scheme groups non-overlapping chunks into *rounds*. Figure 1 shows an example: chunks in the same round are shaded in the same way.

For each round, Jellyfish processes each chunk in that round in a separate process. Fine-grained locks are not needed on either $\boldsymbol{L}$ or $\boldsymbol{R}$. Figure 1 illustrates the case $l = 3$. Suppose that two different processes both need to update the same row of $\boldsymbol{L}$ or $\boldsymbol{R}$. From the update rule, this implies that there is a data item (i,j) in one chunk and there is a data item (i',j') in a different chunk such that either $i = i'$ or $j = j'$. This is contradiction: the chunks in a given round do not overlap. Thus, we are able to process all the chunks in a given round without fine-grained locking. Algorithm 1 summarizes the preceding discussion giving the execution of one epoch of the Jellyfish algorithm.

3.4 Permutations

As one can see, much of the effort of the computation is devoted to permuting and arranging the data. This gives rise to the efficiency and lock-free parallelism that is fundamental to our approach. It is imperative that the data be permuted and arranged efficiently to not degrade from our code.

We keep two copies of the data necessary to compute the gradients of the f_{ij} in memory. Our permutation of this data proceeds in two phases. First, the data set, row order, and column order is shuffled in place (the subroutine generate_perms() in Algorithm 1). Next, the data is moved in its

Algorithm 1 Given $(\boldsymbol{L}, \boldsymbol{R})$ and *data* a training set, update $(\boldsymbol{L}, \boldsymbol{R})$ using p partitions

```
1: (π_row, π_col) ← generate_perms()
2: [C_{1,1}, C_{1,2}, ..., C_{p,p}] ← permute_data(π_row, π_col, data)
3: Round[u] = [C_{a,b} s.t. b = a + u  mod l for a = 1, ..., p]
4: for ℓ = 1, ..., p do
5:    for C_{a,b} in Round[ℓ] in parallel do
6:       for (i, j) in C_{a,b} do
7:          apply (11) for the nuclear norm or (12) for the max norm to the (L, R) pair.
8:       end for
9:    end for
10: end for
```

shuffled order into collocated parts of memory corresponding to the block partition defined by the row and column permutations. That is, the data is collected into the appropriate chunks, and the ordering inside these chunks is that of the global data set order assigned by the generate_perms() subroutine. The data is only moved when it is put in the appropriate bucket for the next iteration. This is executed as permute_data() in Algorithm 1.

Our permutations are implemented using the *Knuth shuffle* [17, pg. 145]. An entry is selected at random from $\{1, \ldots, m\}$ and is swapped with the first element in the list. Then an entry is selected at random from $\{2, \ldots, m\}$ and swapped with the second element in the list. Proceeding in this fashion, a uniform permutation is generated after a single pass over the indices. Importantly, this operator does not require a sort of any kind. This allows us to avoid use of excess random bits or superlinear time algorithms.

3.5 Pipeline data parallelism

A straightforward optimization is to overlap the gradient computations of the current epoch with the determinations of the row, column, and data permutations for the subsequent epoch. Suppose the data shuffling takes time s, and an epoch of gradient steps gradient takes time G. In this case, one can see that executing m epochs sequentially (assuming linear speedup) on p processors takes $m(G/p + s)$ since the shuffling and permutations cannot be partitioned per epoch.

In contrast, suppose we run an interleaved plan where we devote k processors to permuting the data and $p-k$ to computing gradients. Each permutation thread operates on an independent copy of the data, and thus this will require a k-fold increase in memory. However, it can also result in a dramatic increase in speed.

Suppose, for instance, that the time to compute one epoch of gradient calculations, $G/(p-k)$ is greater than s/k. In this case, the interleaved plan will require a total time of $mG/(p-k) + s$: we wait a total time s for the initial permutation, but then never wait for the permutation during the gradient computations. Hence, the interleaved plan has lower end-to-end time whenever

$$\left(\tfrac{1}{p} \cdot \tfrac{m}{m-1} \cdot \tfrac{k}{p-k}\right) G \leq s \leq \tfrac{k}{p-k} \cdot G\,.$$

The optimal value for k can be computed automatically by running a few sample epochs, computing G and s empirically, and then solving a discrete optimization problem. In our experiments, we set $k = 2$ on a 12 core machine.

3.6 Choosing the rank

Our algorithm requires an estimate for the rank of the decision variable. This rank can be used as an algorithm parameter and tuned by the user. However, it is often times advantageous to have the rank selected automatically. For such an automatic rank selection, we use the following heuristic algorithm.

Ideally, we would like to guarantee that the optimal solution is a local minimum of our cost function. In particular, we would like to guarantee that the Hessian at the optimal solution is strictly positive definite. For an individual f_{ij}, we have

$$\nabla^2 f_{ij}(\boldsymbol{L}_i, \boldsymbol{R}_j) = f''_{ij}(\boldsymbol{L}_i \boldsymbol{R}_j^*) \begin{bmatrix} \boldsymbol{R}_j & \boldsymbol{L}_i \end{bmatrix}^* \begin{bmatrix} \boldsymbol{R}_j & \boldsymbol{L}_i \end{bmatrix} + f'_{ij}(\boldsymbol{L}_i \boldsymbol{R}_j^*) \begin{bmatrix} \boldsymbol{0} & \boldsymbol{I} \\ \boldsymbol{I} & \boldsymbol{0} \end{bmatrix}$$

The global Hessian is thus the sum of m rank one matrices plus a rank indefinite matrix. In order for the Hessian to have any chance of being full rank about the optimal solution, the sum of rank one matrices must be positive definite, and the number of summands thus must exceed $r(n_1 + n_2)$. Using this intuition, Jellyfish begins with an initial guess of the rank to be $r = \lfloor \frac{m}{3(n_1+n_2)} \rfloor$. The additional factor of 3 is used to guaranteed a sufficient amount of redundancy in the data terms to encourage local strong convexity.

Jellyfish prunes this initial guess of the rank by running for a few epochs, and then estimating the singular values of the individual factors. If a significant gap in the singular values is detected, the rank is reduced and the algorithm is rerun from scratch with this smaller rank. We note that in our implementation, rather than computing an exact SVD of $\boldsymbol{L}$ and $\boldsymbol{R}$, we sample $2r \log r$ rows of $\boldsymbol{L}$ and compute the SVD of this considerably smaller matrix. This sampling scheme has been shown by many authors to provide high quality estimates of the true spectrum of the larger matrix (see, for example, [11]). The main advantage of this approximate SVD is that it can be computed in time $O(r^3 \log(r))$, which is nearly instantaneous when compared to the other subroutines of Jellyfish.

4 Experiments

We first verify our most important conceptual claim: that our parallel incremental gradient algorithm can achieve state-of-the-art performance on large scale matrix factorization problems orders of magnitudes faster than existing techniques. We then validate our technical contributions using scalability, scale-up, and lesion studies.

Experimental setup: All experiments are run on an identical configuration: a dual Xeon X650 CPUs (6 cores each x 2 hyperthreading) machine with 24GB of RAM and a software RAID-0 over 4 2TB Seagate Constellation 7200RPM disks. The kernel is Linux 2.6.18-128.

4.1 Random low-rank matrices

In our first batch of experiments, we generated hundreds of different problem instances. We experimented with different sizes of matrices, different ranks, and different amounts of noise. For a fixed choice of the number of rows n_r, number of columns n_c, rank r, and noise level σ^2, we generated a low-rank matrix $\boldsymbol{M} = \boldsymbol{Y}_L \boldsymbol{Y}_R^*$ where $\boldsymbol{Y}_L$ and $\boldsymbol{Y}_R$ have i.i.d. Gaussian entries and are normalized so the average squared magnitude of $\boldsymbol{M}$ is 1.

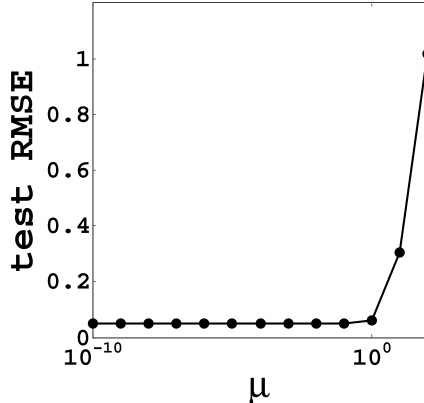

Figure 2: Sensitivity to μ in random matrix completion experiments. This is for a fixed sampling of a fixed 1000×5000, rank 10 matrix ($\beta = 5$, $\sigma^2 = 10^{-2}$). The same experiment was run 5 times for each value of μ. We plot the average test error here.

A train set was generated by sampling $\beta r(n_r + n_c - r)$ entries from $\boldsymbol{M}$ and adding noise with variance σ^2. β is a parameter and chosen to be either 3, 5, or 10. These entries were provided to Jellyfish and used to fit the unknown low-rank matrix. The number of samples is chosen to be a multiple of the number of parameters in the hidden low-rank model, $r(n_r + n_c - r)$. A test set was generated with 1% of the number of examples as in the training set and created without any additive noise. This set is used to validate that the algorithm does not overfit the train set and provides evidence that we correctly reconstruct $\boldsymbol{M}$.

The goal of these experiments was to minimize the nuclear norm regularized problem

$$\text{minimize}_{\boldsymbol{X}} \sum_{(i,j)\in\Omega} (X_{ij} - M_{ij})^2 + \mu \|\boldsymbol{X}\|_*$$

where Ω denotes the training set. On each instance ran Jellyfish for 40 epochs using the rank detection heuristic described in Section 3.6. We always dedicated 10 cores to gradient computations and 2 cores to permuting and organizing the data. We chose the gradient step-size to start at 0.1 and then reduced this step-size by a factor of 0.9 each epoch. We found empirically that when we used our rank identification heuristic, Jellyfish was very insensitive to the choice of the regularization parameter μ. That is, we were able to achieve the same error on the validation set for all sufficiently small μ. See Figure 2. We conjecture that this is because our explicit matrix factorization serves to regularize the problem by reducing the size of the parameter space. We emphasize that this insensitivity to the rank appears special to these random gaussian problems, and is not a general feature. Here, we report results with $\mu =$ 1e-5.

The total CPU time and the root-mean-square error (RMSE) achieved on both the train and test sets were recorded at each epoch. The results are tabulated in the appendix, scanning 243 different instances.

We compared our Jellyfish implementation to the fastest freely distributed code that we could find NNLS [37]. While there are many codes for matrix completion, NNLS consistently returns the fastest results most accurately (see [2] for a comparison of several different matrix completion codes). NNLS runs an implementation of Nesterov's accelerated method for composite gradient descent [28]. The code employs several heuristics to accelerate the individual iterations

which use iterative singular value decompositions and implements a continuation scheme for the regularization parameter μ. NNLS was written in Matlab, and some of the functions are available as mex files to speed up processing time. Unfortunately, the mex version is somewhat unstable and frequently crashes. Though we have been in contact with the authors, we have not been able to get a consistent build of NNLS with mex optimizations at the time of this submission. Hence, we compare JELLYFISH against the non-mex version of NNLS, compiled using Matlab's mcc compiler.[1]

We plot performance profiles comparing NNLS and JELLYFISH in Figure 3 [10]. Recall that a performance profile is a empirical cumulative distribution comparing some statistic $\alpha(s,i)$ indexed by a set of solvers (s) and a set of instances (i). For a fixed solver, s, the performance profile f_s is given by

$$f_s(\tau) = \frac{|\{i \ : \ \alpha(s,i) < \tau \min_s \alpha(s,i)\}|}{\text{number of instances}}$$

That is, how frequently is α less than τ times the minimum achievable value of the test statistic. We plot the profiles of train RMSE, test RMSE, and total computation time. JELLYFISH is always faster than NNLS, and is up to 25 times faster even on these small problems. On over 86% of the instances, the JELLYFISH returns either lower test RMSE than NNLS or at most a factor of 2 worse.

The discrepancy in the performance profiles for RMSE can be explained by four instances where JELLYFISH yields test RMSE more than 10 times worse than NNLS. However, when we rerun all of these instances, the minimum training error is always *better* than NNLS. That is, JELLYFISH converges to a local optimum in some instances, but the quality of this solution can always be improved by running with a different random seed. We demonstrate this phenomenon in Figure 4. Since JELLYFISH is always at least 5 times faster than NNLS, it makes sense to restart the computation if a suspiciously high RMSE is acquired. Moreover, such a restart is trivially parallelizable, as we can run from different starting points on completely different machines or on disjoint sets of cores.

Note that the performance profiles are for only for very small problems (1000 rows). On larger problems, the speedups are far more dramatic. For instance, on a problem with 100K rows and 500K columns, NNLS takes 8 hours to complete, while JELLYFISH takes a mere 3 minutes. Even with the mex optimization, NNLS will always be at least an order of magnitude slower than JELLYFISH. We were even able to run JELLYFISH on an instance with 1 million rows and columns, with rank 10 and $\beta = 5$. This instance required less than 18 minutes to run.

4.2 Collaborative filtering experiments

Recommendation Tasks. We also tested our formulation at solving movie recommendation tasks on the Movielens (1M and 10M rating data sets) and Netflix Prize data sets. In these sets, tables indexed by movie id and user id are provided, and the goal is to infer the values in an unseen test set. The ratings are on an integer scale from 1 to 5. While it is not clear that matrix factorization is the only solution to this problem, several researchers have shown that this approach can yield high quality predictions [12, 19]. In our experiments, we test a γ_2-norm formulation

$$\text{minimize} \sum_{(i,j)\in\Omega} (X_{ij} - M_{ij} - c)^2 \quad \text{subject to } \|\boldsymbol{X}\|_{\gamma_2} \leq B\,.$$

[1] We were also unable to find any implementations of similar algorithms written in a low-level language such as C. While coding directly in C or Fortran would likely yield a version of NNLS which is considerably faster than the Matlab version, we doubt that it would yield the hundred-fold speedups necessary to be competitive with JELLYFISH.

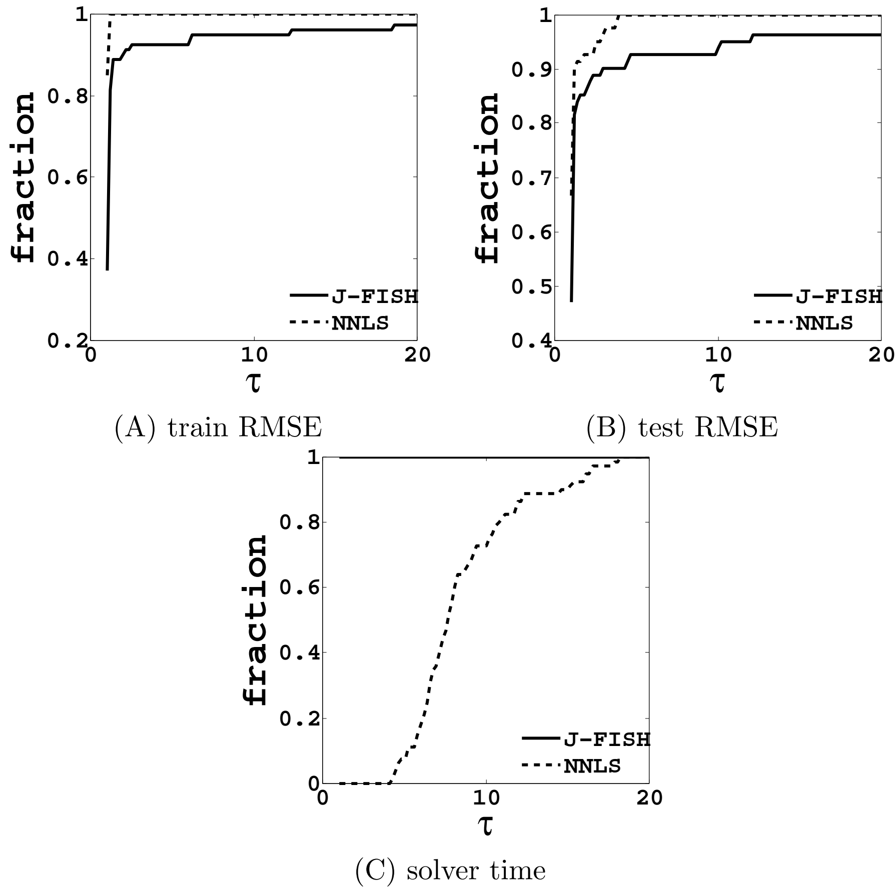

Figure 3: Perfomance profiles for (A) train RMSE (B) test RMSE (C) and solver time. In all graphs, JELLYFISH is plotted with a solid line and NNLS is plotted with a dashed line.

n_c	rank	β	σ^2	NNLS		initial		final	
				train RMSE	test RMSE	train RMSE	test RMSE	train RMSE	test RMSE
1000	20	3	1e-3	2.59e-2	2.39e-2	1.58e-1	2.41e-1	2.56e-2	2.39e-2
1000	20	3	1e-4	8.18e-3	7.33e-2	1.52e-1	2.42e-1	8.18e-2	7.35e-2
5000	10	3	1e-4	8.18e-3	7.47e-3	2.45e-1	3.59e-1	8.73e-3	1.11e-2
10000	10	3	1e-4	8.21e-3	9.32e-3	2.35e-1	3.44e-1	8.55e-3	1.72e-2

Figure 4: Demonstrating that bad results can be fixed by running again. Here we show 4 instances where JELLYFISH returned an solution with error more than 10x larger than NNLS. When we rerun from a different starting position, the difference between our old and new results evaporate.

	Dataset	Dim	#Example	Size
Fact.	Movielens1M	6k × 4k	1M	24M
	Movielens10M	72k × 65k	10M	225M
	Netflix	480k × 18k	100M	1.4G
Fact.	Movielens1M	6k × 4k	1M	24M
	Movielens10M	72k × 65k	10M	225M
	Netflix	480k × 18k	100M	1.4G

Figure 5: Recommendation data set statistics.

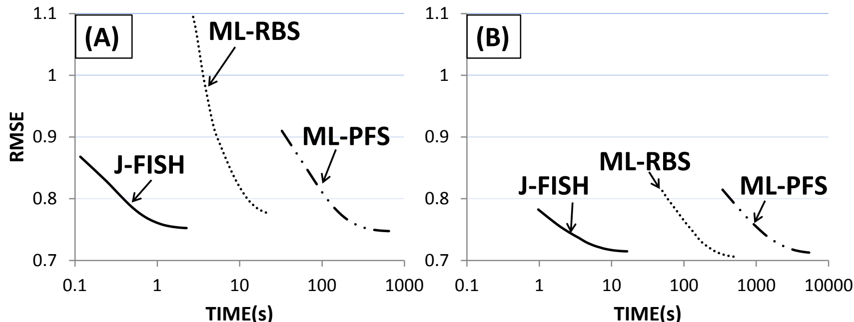

Figure 6: Low-Rank Factorization Graph on (A) Movielens1M and (B) Movielens10M.

with $B = 1.5$ and c set to be the mean of M_{ij} on Ω. In Figure 6, we plot on the x-axis the time during execution (in log scale) and the y-axis the current quality of the solution. We run all algorithms for a total of 20 epochs using a rank of 30. The gradient stepsize is initialized at 5e-2 and is reduced by a factor of 0.8 after each epoch..

We compare against implementations in Matlab that we hand tuned to be as fast as possible and then compiled using mcc. These Matlab codes were used to perform the experiments in Lee *et al* [20]. Rather than using one example per incremental gradient step, this code uses randomly sampled batches of size $100,000$ to estimate the gradient of the least-squares costs and additionally implements a "momentum" parameter for over-relaxation. We also compare against a straightforward implementation with only modest tuning called ML-PFS (Projected Fully Stochastic). Jellyfish is much faster than all competitor implementations. Figure 7(A) shows that the trend continues as we scale up to the larger Netflix data set. This suggests that the gap between our algorithms and the competitors will increase with the data set size.

4.3 Sampling and scheduling

We validate the technical details of contributions.

Thread scaling . We validate our scheduling algorithms with a thread scale-up experiment. In Figure 7(B), we force the scheduler to use the number of worker threads (i.e., threads running gradients) indicated on the x axis and measure the average time per single epoch for Jellyfish on MovieLens1M (reported on the y axis). We also verify that the worker threads do not wait on shuffling. Moving from left-to-right in the graph, we see that as we add threads, the running

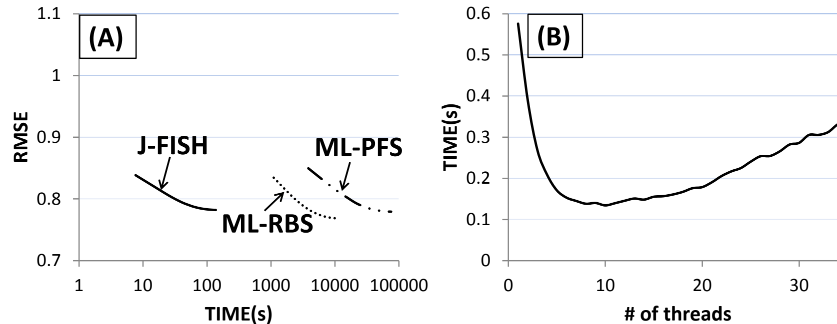

Figure 7: (A) Low-rank Netflix. (B) Time-per-epoch for VMM for varying number of threads.

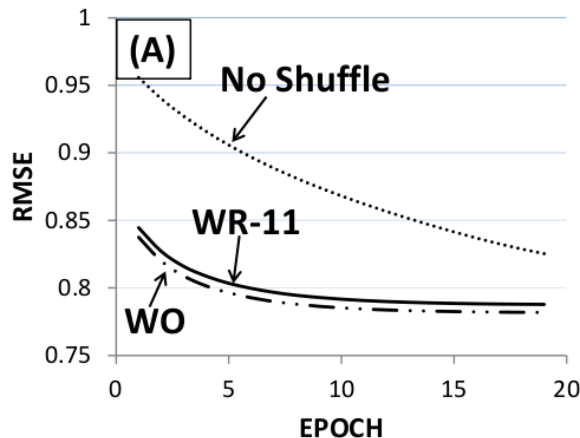

Figure 8: Effect of Order on Epochs v. RMSE on Netflix.

time decreases to a sweet spot around 11 threads. This downward trend verifies our claim that our approach to factored models provides a large improvement over the sequential approach (which is 1-thread on this graph). Recall that our experiment machine has 12 physical cores. Since we use 1 core for shuffling, this gives evidence that our scheduling algorithm of saturating the available cores can provide substantial improvement. Continuing after 11, the graph starts to trend upward indicating longer epochs (although the operating system reports 24 processors). If our scheduling algorithm were to use these processors, our performance would substantially degrade. This suggests that our performance is essentially data access bound (hyperthreading splits the caches between the two threads that operate on the same core).

This experiment also confirms that it is worthwhile, if available, to dedicate a core to shuffling. If we do not use that core then each epoch is lengthened here on average by almost 0.1 seconds which is approximately 90% of an epoch. That is, a sequential shuffle would almost double the time of a single epoch (we separately confirmed this assertion).

Without-replacement sampling . We validate that without-replacement sampling allows faster rates of convergence (we describe this in theoretical detail in the next section). To verify this assertion, we run the Netflix data set in Jellyfish and force it to shuffle the data in three

ways: (1) without-replacement at each epoch (WO), (2) with-replacement shuffling (WR), and (3) a deterministic order (No Shuffle). In the No Shuffle approach, since there is no need to shuffle data, we can and do use an additional core for computation. The other two approaches use 11 cores to compute the gradient and 1 core for shuffling using the appropriate shuffling algorithm. Figure 8(B) shows the running time (x-axis) versus the RMSE while (B) plots each epoch versus RMSE. We can see is that No Shuffle converges substantially slower than either random order in terms of epochs. Moreover, without-replacement shuffle converges notably faster per epoch. The reported runs above are averaged, but on every run, without-replacement sampling had a lower RMSE than with-replacement sampling on every epoch. We also observed that the shuffle phase for with-replacement shuffling took significantly more time than without-replacement shuffling (almost 20s to shuffle Netflix). This further justifies our decision to pursue without-replacement sampling.

5 Discussion and Open Questions

Our stochastic block partitioning scheme should have many applications to different model geometries that arise in data analysis. A straightforward extension would be to data mining of low-rank tensors [18] where the block partitioning would be over high dimensional data cubes. We are also interested in adapting our code to enable near-real time updating of factor models under changing conditions. Such an adaptation could be useful in recommender systems where users are constantly updating their preferences and could also be extended to finding anomalies in network traffic behavior [2].

The success of our stochastic parallelization heuristic also raises several interesting theoretical questions. A key insight that underlies our algorithm is that we can constrain the ordering of the steps in the incremental gradient method to yield parallelism. But, this begs a simpler (and far deeper) question: *What effect does the ordering have on the convergence rate of the incremental gradient method?* To answer this question, we first need to understand the discrepancy in convergence rate between with- and without-replacement sampling for incremental gradients (which, as mentioned above, is long-standing open problem in optimization [4, pg. 628]). Certainly, with-replacement sampling may suffer from a coupon collector's penalty, i.e., with-replacement sampling may require a logarithmic factor more samples just to touch all of the summands in the objective function. When the number of terms is in the billions, the log factor is substantial and can lead to significant slow-downs. However without-replacement sampling is an empirically superior sampling regime even for small numbers of summands. Is there a straightforward analysis that can explain such a discrepancy? We believe an answer to such a question would provide a first step toward understanding how to trade quality and runtime performance in sampling for gradient methods.

Acknowledgements

This work was supported in part by ONR Contract N00014-11-M-0478. Christopher Ré is additionally supported by the National Science Foundation under IIS-1054009, and gifts from Google, Microsoft, and Johnson Controls, Inc. Any opinions, ndings, conclusions, or recommendations expressed in this work do not necessarily reect the views of the US government.

References

[1] N. Alon and A. Naor. Approximating the cut-norm via Grothendieck's inequality. *SIAM Journal on Computing*, 35(4):787–803, 2006.

[2] L. Balzano, R. Nowak, and B. Recht. Online identification and tracking of subspaces from highly incomplete information. In *Proceedings of the 48th Annual Allerton Conference*, 2010.

[3] D. P. Bertsekas. A hybrid incremental gradient method for least squares. *SIAM Journal on Optimization*, 7:913–925, 1997.

[4] D. P. Bertsekas. *Nonlinear Programming.* Athena Scientific, Belmont, MA, 2nd edition, 1999.

[5] L. Bottou and O. Bousquet. The tradeoffs of large scale learning. In *Neural Information Processing Systems*, 2008.

[6] S. Burer and R. D. C. Monteiro. Local minima and convergence in low-rank semidefinite programming. *Mathematical Programming*, 103(3):427–444, 2005.

[7] J.-F. Cai, E. J. Candès, and Z. Shen. A singular value thresholding algorithm for matrix completion. *SIAM Journal on Optimization*, 20(4):1956–1982, 2008.

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

[9] E. J. Candès and T. Tao. The power of convex relaxation: Near-optimal matrix completion. *IEEE Transactions on Information Theory*, 56(5):2053–2080, 2009.

[10] E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming, Series A*, 91:201–213, 2002.

[11] P. Drineas, A. Frieze, R. Kannan, S. Vempala, and V. Vinay. Clustering in large graphs and matrices. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1999.

[12] S. Funk. Netflix update: Try this at home, December 2006. `http://sifter.org/~simon/journal/20061211.html`.

[13] D. Gross. Recovering low-rank matrices from few coefficients in any basis. *IEEE Trans. on Information Theory*, 57:1548–1566, 2011.

[14] G. J. O. Jameson. *Summing and Nuclear Norms in Banach Space Theory.* Number 8 in London Mathematical Society Student Texts. Cambridge University Press, Cambridge, UK, 1987.

[15] S. Ji and J. Ye. An accelerated gradient method for trace norm minimization. In *Proceedings of the ICML*, 2009.

[16] R. H. Keshavan, A. Montanari, and S. Oh. Matrix completion from a few entries. *IEEE Transactions on Information Theory*, 56(6):2980–2998, 2009.

[17] D. E. Knuth. *The Art of Computer Programming.* Addison-Wesley Professional, 2nd edition, 1998.

[18] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009.

[19] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[20] J. Lee, , B. Recht, N. Srebro, R. R. Salakhutdinov, and J. A. Tropp. Practical large-scale optimization for max-norm regularization. In *Advances in Neural Information Processing Systems*, 2010.

[21] Z. Liu and L. Vandenberghe. Interior-point method for nuclear norm approximation with application to system identification. *SIAM Journal on Matrix Analysis and Applications*, 31(3):1235–1256, 2009.

[22] Z. Luo. On the convergence of the lms algorithm with adaptive learning rate for linear feedforward networks. *Neural Computation*, 3(2):226–245, 1991.

[23] Z. Q. Luo and P. Tseng. Analysis of an approximate gradient projection method with applications to the backpropagation algorithm. *Optimization Methods and Software*, 4:85–101, 1994.

[24] S. Ma, D. Goldfarb, and L. Chen. Fixed point and Bregman iterative methods for matrix rank minimization. *MAthematical Programming*, pages 1–33, 2009. Published online first at `http://dx.doi.org/10.1007/s10107-009-0306-5`.

[25] O. L. Mangasarian and M. V. Solodov. Serial and parallel backpropagation convergence via nonmonotone perturbed minimization. *Optimization Methods and Software*, 4:103–116, 1994.

[26] A. Nedic and D. P. Bertsekas. Convergence rate of incremental subgradient algorithms. In S. Uryasev and P. M. Pardalos, editors, *Stochastic Optimization: Algorithms and Applications*, pages 263–304. Kluwer Academic Publishers, 2000.

[27] A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro. Robust stochastic approximation approach to stochastic programming. *SIAM Journal on Optimization*, 19(4):1574–1609, 2009.

[28] Y. Nesterov. Gradient methods for minimizing composite functions. Technical report, CORE Discussion Paper, 2007. Preprint available at `http://www.optimization-online.org/DB_HTML/2007/09/1784.html`.

[29] B. Recht. A simpler approach to matrix completion. *Journal of Machine Learning Research*, 2009. To Appear. Preprint available at `http://arxiv.org/abs/0910.0651`.

[30] B. Recht, M. Fazel, and P. Parrilo. Guaranteed minimum rank solutions of matrix equations via nuclear norm minimization. *SIAM Review*, 52(3):471–501, 2010.

[31] J. D. M. Rennie and N. Srebro. Fast maximum margin matrix factorization for collaborative prediction. In *Proceedings of the International Conference of Machine Learning*, 2005.

[32] R. T. Rockafellar. Monotone operators and the proximal point algorithm. *SIAM Journal on Control and Optimization*, 14(5):877–898, 1976.

[33] R. Salakhutdinov and A. Mnih. Probabilistic matrix factorization. In *Advances in Neural Information Processing Systems*, 2008.

[34] N. Srebro, J. Rennie, and T. Jaakkola. Maximum margin matrix factorization. In *Advances in Neural Information Processing Systems*, 2004.

[35] N. Srebro and A. Shraibman. Rank, trace-norm and max-norm. In *18th Annual Conference on Learning Theory (COLT)*, 2005.

[36] A. Tanay, R. Sharan, and R. Shamir. Discovering statistically significant biclusters in gene expression data. *Bioinformatics*, 18(1):136–144, 2002.

[37] K.-C. Toh and S. Yun. An accelerated proximal gradient algorithm for nuclear norm regularized least squares problems. *Pacific Journal of Mathematics*, 6:615–640, 2010.

[38] P. Tseng. An incremental gradient(-projection) method with momentum term and adaptive stepsize rule. *SIAM Joural on Optimization*, 8(2):506–531, 1998.

rank	σ^2	β	Jellyfish time (s)	Jellyfish train RMSE	Jellyfish test RMSE	NNLS time (s)	NNLS train RMSE	NNLS test RMSE
5	0.01	3	0.39	1.115e-01	1.723e-01	5.89	5.857e-02	2.280e-01
		5	0.54	8.820e-02	5.666e-02	4.03	8.942e-02	5.099e-02
		10	0.73	9.489e-02	3.397e-02	4.45	9.489e-02	3.417e-02
	0.001	3	0.39	4.770e-02	8.482e-02	5.63	2.416e-02	1.361e-01
		5	0.55	2.805e-02	7.075e-02	3.62	2.819e-02	1.572e-02
		10	0.71	3.025e-02	1.169e-02	4.56	3.017e-02	1.076e-02
	0.0001	3	0.38	3.745e-02	1.063e-01	5.78	1.524e-02	3.188e-01
		5	0.52	1.225e-02	6.431e-02	4.15	8.966e-03	5.347e-03
		10	0.71	9.507e-03	3.496e-03	4.44	9.499e-03	3.415e-03
10	0.01	3	0.62	8.154e-02	7.264e-02	4.53	8.128e-02	7.165e-02
		5	0.86	8.954e-02	5.286e-02	5.35	8.952e-02	5.319e-02
		10	1.32	9.482e-02	3.431e-02	6.25	9.482e-02	3.424e-02
	0.001	3	0.63	2.664e-02	2.709e-02	4.37	2.588e-02	2.332e-02
		5	0.88	2.831e-02	1.620e-02	4.99	2.829e-02	1.615e-02
		10	1.28	2.999e-02	1.057e-02	6.53	2.999e-02	1.058e-02
	0.0001	3	0.63	1.087e-02	2.058e-02	4.21	8.156e-03	7.410e-03
		5	0.88	8.917e-03	4.909e-03	5.29	8.918e-03	4.937e-03
		10	1.28	9.674e-03	3.723e-03	6.44	9.505e-03	3.304e-03
20	0.01	3	1.17	8.156e-02	6.917e-02	7.57	8.155e-02	6.940e-02
		5	1.61	8.926e-02	5.166e-02	7.46	8.926e-02	5.164e-02
		10	2.58	9.486e-02	3.308e-02	11.27	9.485e-02	3.306e-02
	0.001	3	1.17	1.581e-01	2.406e-01	7.82	2.585e-02	2.390e-02
		5	1.59	2.832e-02	1.594e-02	7.21	2.830e-02	1.593e-02
		10	2.55	3.004e-02	1.096e-02	10.67	3.002e-02	1.083e-02
	0.0001	3	1.16	1.519e-01	2.418e-01	8.11	8.183e-03	7.328e-03
		5	1.63	9.077e-03	5.069e-03	8.27	8.949e-03	4.963e-03
		10	2.52	9.496e-03	3.320e-03	11.09	9.497e-03	3.331e-03

Figure 9: 1000×1000.

rank	σ^2	β	Jellyfish time (s)	Jellyfish train RMSE	Jellyfish test RMSE	NNLS time (s)	NNLS train RMSE	NNLS test RMSE
5	0.01	3	0.79	8.461e-02	9.522e-02	14.37	6.438e-02	2.702e-01
		5	1.07	8.941e-02	5.521e-02	10.86	8.934e-02	5.267e-02
		10	1.42	9.470e-02	3.403e-02	11.44	9.470e-02	3.410e-02
	0.001	3	0.81	3.501e-02	8.706e-02	13.00	2.827e-02	2.087e-01
		5	1.04	2.846e-02	3.656e-02	11.01	2.820e-02	1.665e-02
		10	1.53	3.001e-02	1.060e-02	11.53	3.001e-02	1.062e-02
	0.0001	3	0.80	2.057e-02	6.312e-02	12.90	1.642e-02	2.396e-01
		5	1.05	9.643e-03	1.052e-02	10.88	8.973e-03	5.429e-03
		10	1.41	9.582e-03	4.225e-03	11.47	9.501e-03	3.361e-03
10	0.01	3	1.31	8.186e-02	7.205e-02	12.22	8.171e-02	7.141e-02
		5	1.69	8.946e-02	4.991e-02	13.38	8.946e-02	4.990e-02
		10	2.68	9.486e-02	3.387e-02	18.70	9.486e-02	3.387e-02
	0.001	3	1.35	2.622e-02	2.578e-02	13.65	2.577e-02	2.284e-02
		5	1.78	2.832e-02	1.615e-02	14.03	2.832e-02	1.615e-02
		10	2.72	2.998e-02	1.043e-02	19.55	2.998e-02	1.043e-02
	0.0001	3	1.33	2.448e-01	3.593e-01	14.44	8.181e-03	7.465e-03
		5	1.63	8.950e-03	5.321e-03	12.88	8.958e-03	5.270e-03
		10	2.71	9.472e-03	3.350e-03	19.15	9.467e-03	3.350e-03
20	0.01	3	2.50	8.178e-02	7.376e-02	19.77	8.178e-02	7.374e-02
		5	3.30	8.936e-02	4.988e-02	21.63	8.936e-02	4.986e-02
		10	6.05	9.483e-02	3.327e-02	34.66	9.483e-02	3.327e-02
	0.001	3	2.34	1.579e-01	2.261e-01	21.65	2.582e-02	2.279e-02
		5	3.37	2.830e-02	1.587e-02	24.20	2.829e-02	1.587e-02
		10	6.12	3.003e-02	1.037e-02	34.87	3.003e-02	1.036e-02
	0.0001	3	2.53	8.173e-03	7.307e-03	20.86	8.194e-03	7.537e-03
		5	3.41	8.972e-03	5.217e-03	24.56	8.943e-03	5.083e-03
		10	6.06	9.498e-03	3.363e-03	39.07	9.494e-03	3.361e-03

Figure 10: 1000×5000.

rank	σ^2	β	Jellyfish time (s)	Jellyfish train RMSE	Jellyfish test RMSE	NNLS time (s)	NNLS train RMSE	NNLS test RMSE
5	0.01	3	1.35	8.363e-02	1.128e-01	24.12	6.860e-02	2.758e-01
		5	1.51	8.937e-02	5.662e-02	16.82	8.934e-02	5.517e-02
		10	3.23	8.570e-02	4.904e-02	20.71	9.481e-02	3.437e-02
	0.001	3	1.33	3.557e-01	4.700e-01	21.88	2.900e-02	2.270e-01
		5	1.62	2.837e-02	2.202e-02	19.16	2.831e-02	1.718e-02
		10	3.43	2.936e-02	1.194e-02	21.61	2.998e-02	1.096e-02
	0.0001	3	1.36	1.515e-02	6.241e-02	22.36	1.383e-02	2.365e-01
		5	1.61	9.638e-03	2.357e-02	18.96	8.969e-03	5.482e-03
		10	2.39	9.500e-03	3.461e-03	22.03	9.500e-03	3.461e-03
10	0.01	3	2.16	8.180e-02	7.908e-02	26.60	8.174e-02	7.728e-02
		5	2.69	8.929e-02	5.131e-02	25.03	8.928e-02	5.132e-02
		10	4.80	9.497e-02	3.342e-02	36.15	9.496e-02	3.343e-02
	0.001	3	2.11	2.592e-02	2.423e-02	22.08	2.583e-02	2.360e-02
		5	2.72	2.830e-02	1.613e-02	24.57	2.830e-02	1.620e-02
		10	4.81	3.004e-02	1.076e-02	37.01	3.003e-02	1.076e-02
	0.0001	3	1.92	2.346e-01	3.444e-01	23.61	8.208e-03	9.318e-03
		5	2.75	8.954e-03	5.148e-03	24.35	8.953e-03	5.213e-03
		10	4.74	9.498e-03	3.431e-03	35.88	9.480e-03	3.431e-03
20	0.01	3	4.17	8.162e-02	7.156e-02	34.25	8.161e-02	7.175e-02
		5	5.58	8.944e-02	5.100e-02	41.86	8.944e-02	5.101e-02
		10	11.25	9.475e-02	3.359e-02	65.64	9.474e-02	3.359e-02
	0.001	3	3.98	2.580e-02	2.311e-02	46.80	2.579e-02	2.310e-02
		5	5.59	2.827e-02	1.609e-02	44.71	2.829e-02	1.616e-02
		10	11.11	3.000e-02	1.064e-02	67.52	3.000e-02	1.064e-02
	0.0001	3	4.12	8.172e-03	7.430e-03	43.90	8.183e-03	7.463e-03
		5	5.58	8.956e-03	5.128e-03	47.44	8.957e-03	5.177e-03
		10	11.18	9.490e-03	3.379e-03	65.22	9.491e-03	3.389e-03

Figure 11: 1000×10000.

rank	σ^2	β	time (s)	train RMSE	test RMSE
5	0.01	3	2.38	3.515e-01	6.062e-01
		5	2.79	9.148e-02	6.325e-02
		10	4.56	9.502e-02	3.393e-02
	0.001	3	2.03	1.724e-01	3.326e-01
		5	2.83	2.816e-02	1.966e-02
		10	4.50	2.999e-02	1.072e-02
	0.0001	3	2.15	6.353e-02	1.361e-01
		5	2.87	9.715e-03	1.261e-02
		10	4.44	9.495e-03	3.344e-03
10	0.01	3	3.52	8.251e-02	7.626e-02
		5	4.93	8.954e-02	4.977e-02
		10	9.44	9.479e-02	3.354e-02
	0.001	3	3.46	2.680e-02	2.896e-02
		5	4.72	2.828e-02	1.608e-02
		10	9.38	3.002e-02	1.059e-02
	0.0001	3	3.45	1.913e-02	4.333e-02
		5	4.80	9.047e-03	5.113e-03
		10	9.53	9.480e-03	3.354e-03
20	0.01	3	7.01	8.157e-02	7.104e-02
		5	10.36	8.943e-02	5.086e-02
		10	20.02	9.491e-02	3.339e-02
	0.001	3	7.05	2.585e-02	2.280e-02
		5	10.43	2.826e-02	1.588e-02
		10	20.47	3.000e-02	1.050e-02
	0.0001	3	7.52	8.177e-03	7.235e-03
		5	10.26	8.952e-03	5.037e-03
		10	20.67	9.484e-03	3.339e-03

Figure 12: 10000×10000.

rank	σ^2	β	time (s)	train RMSE	test RMSE
5	0.01	3	5.55	8.511e-02	1.054e-01
		5	8.35	8.949e-02	5.410e-02
		10	13.94	9.488e-02	3.394e-02
	0.001	3	5.67	3.751e-02	9.244e-02
		5	7.89	2.845e-02	2.004e-02
		10	14.03	3.000e-02	1.071e-02
	0.0001	3	5.45	2.679e-02	8.276e-02
		5	8.06	9.610e-03	1.362e-02
		10	14.40	9.482e-03	3.415e-03
10	0.01	3	9.59	8.179e-02	7.559e-02
		5	14.92	8.952e-02	5.156e-02
		10	27.07	9.492e-02	3.363e-02
	0.001	3	9.88	2.621e-02	2.688e-02
		5	14.59	2.829e-02	1.607e-02
		10	26.58	3.000e-02	1.059e-02
	0.0001	3	9.57	8.978e-03	1.412e-02
		5	14.81	8.964e-03	5.211e-03
		10	28.06	9.485e-03	3.358e-03
20	0.01	3	20.09	8.164e-02	7.262e-02
		5	29.55	8.944e-02	5.083e-02
		10	61.80	9.488e-02	3.358e-02
	0.001	3	19.89	2.582e-02	2.291e-02
		5	32.48	2.828e-02	1.606e-02
		10	60.96	2.999e-02	1.059e-02
	0.0001	3	20.16	8.176e-03	7.345e-03
		5	28.40	8.945e-03	5.125e-03
		10	61.74	9.491e-03	3.361e-03

Figure 13: 10000×50000.

rank	σ^2	β	time (s)	train RMSE	test RMSE
5	0.01	3	10.24	8.279e-02	9.575e-02
		5	14.35	8.948e-02	5.309e-02
		10	25.38	9.485e-02	3.402e-02
	0.001	3	9.86	2.871e-02	6.618e-02
		5	14.19	2.837e-02	1.857e-02
		10	24.69	3.001e-02	1.083e-02
	0.0001	3	9.85	1.762e-02	6.379e-02
		5	14.56	9.097e-03	7.485e-03
		10	23.92	9.490e-03	3.409e-03
10	0.01	3	17.30	8.175e-02	7.566e-02
		5	26.73	8.947e-02	5.120e-02
		10	52.41	9.488e-02	3.359e-02
	0.001	3	17.41	2.593e-02	2.543e-02
		5	26.26	2.828e-02	1.612e-02
		10	52.56	3.000e-02	1.067e-02
	0.0001	3	17.07	2.577e-01	3.922e-01
		5	27.03	8.947e-03	5.150e-03
		10	52.52	9.484e-03	3.366e-03
20	0.01	3	36.16	8.167e-02	7.267e-02
		5	56.20	8.947e-02	5.053e-02
		10	117.45	9.487e-02	3.350e-02
	0.001	3	36.78	2.583e-02	2.304e-02
		5	56.51	2.830e-02	1.602e-02
		10	121.01	3.000e-02	1.060e-02
	0.0001	3	35.01	8.172e-03	7.403e-03
		5	56.06	8.949e-03	5.131e-03
		10	119.00	9.486e-03	3.351e-03

Figure 14: 10000×100000. We ran NNLS on a problem of this size with rank 10, $\beta = 5$ and noise variance equalling 0.01. This took 600s to run, and resulted in a train RMSE of 8.948e-02 and a test RMSE of 5.119e-02.

rank	σ^2	β	time (s)	train RMSE	test RMSE
5	0.01	3	19.21	3.794e-01	6.862e-01
		5	28.25	8.644e-02	5.900e-02
		10	49.32	9.486e-02	3.362e-02
	0.001	3	18.55	3.343e-01	6.264e-01
		5	28.57	2.890e-02	3.004e-02
		10	48.65	3.000e-02	1.067e-02
	0.0001	3	18.80	3.450e-01	6.397e-01
		5	29.76	1.369e-02	2.762e-02
		10	49.43	9.491e-03	3.409e-03
10	0.01	3	35.44	8.299e-02	8.007e-02
		5	54.22	8.944e-02	5.089e-02
		10	108.55	9.487e-02	3.347e-02
	0.001	3	34.66	4.761e-02	8.908e-02
		5	53.94	2.829e-02	1.602e-02
		10	110.67	2.999e-02	1.063e-02
	0.0001	3	34.16	2.212e-02	4.912e-02
		5	54.44	8.949e-03	5.141e-03
		10	107.48	9.489e-03	3.345e-03
20	0.01	3	78.44	8.162e-02	7.173e-02
		5	123.82	8.943e-02	5.024e-02
		10	265.34	9.487e-02	3.342e-02
	0.001	3	78.54	2.585e-02	2.275e-02
		5	122.92	2.829e-02	1.593e-02
		10	264.24	3.000e-02	1.057e-02
	0.0001	3	76.53	8.195e-03	7.348e-03
		5	123.94	8.947e-03	5.044e-03
		10	263.33	9.486e-03	3.344e-03

Figure 15: 100000×100000.

rank	σ^2	β	time (s)	train RMSE	test RMSE
5	0.01	3	63.71	9.903e-02	1.617e-01
		5	96.31	8.950e-02	5.290e-02
		10	180.82	9.485e-02	3.388e-02
	0.001	3	60.46	1.132e-01	2.535e-01
		5	101.16	2.834e-02	2.611e-02
		10	176.20	3.000e-02	1.072e-02
	0.0001	3	62.91	7.936e-02	1.963e-01
		5	93.44	9.716e-03	1.278e-02
		10	175.51	9.486e-03	3.388e-03
10	0.01	3	117.82	8.184e-02	7.545e-02
		5	193.13	8.944e-02	5.122e-02
		10	388.60	9.488e-02	3.365e-02
	0.001	3	126.78	2.634e-02	2.870e-02
		5	195.01	2.829e-02	1.616e-02
		10	383.14	3.000e-02	1.064e-02
	0.0001	3	122.82	9.265e-03	1.723e-02
		5	192.44	8.950e-03	5.160e-03
		10	385.14	9.488e-03	3.362e-03
20	0.01	3	252.25	8.162e-02	7.229e-02
		5	434.56	8.945e-02	5.053e-02
		10	924.28	9.488e-02	3.352e-02
	0.001	3	253.23	2.582e-02	2.287e-02
		5	447.77	2.828e-02	1.596e-02
		10	945.28	3.000e-02	1.059e-02
	0.0001	3	246.46	8.174e-03	7.318e-03
		5	432.81	8.945e-03	5.050e-03
		10	931.65	9.487e-03	3.348e-03

Figure 16: 100000×500000. We ran NNLS on a problem of this size with rank 10, $\beta = 5$ and noise variance equalling 0.01. This took 28500s to run, and resulted in a train RMSE of 8.944e-02 and a test RMSE of 5.121e-02.

rank	σ^2	β	time (s)	train RMSE	test RMSE
5	0.01	3	125.21	8.330e-02	9.998e-02
		5	193.23	8.945e-02	5.323e-02
		10	364.40	9.487e-02	3.405e-02
	0.001	3	119.62	3.087e-02	7.229e-02
		5	190.04	2.836e-02	1.888e-02
		10	367.14	3.000e-02	1.075e-02
	0.0001	3	122.64	1.642e-02	6.248e-02
		5	191.49	9.160e-03	9.191e-03
		10	363.12	9.486e-03	3.403e-03
10	0.01	3	246.36	8.167e-02	7.563e-02
		5	390.54	8.944e-02	5.116e-02
		10	784.25	9.487e-02	3.368e-02
	0.001	3	239.93	2.611e-02	2.849e-02
		5	394.65	2.829e-02	1.618e-02
		10	799.48	3.000e-02	1.064e-02
	0.0001	3	236.44	9.054e-03	1.746e-02
		5	392.01	8.948e-03	5.159e-03
		10	772.97	9.488e-03	3.370e-03
20	0.01	3	508.30	8.167e-02	7.272e-02
		5	890.34	8.944e-02	5.060e-02
		10	1910.87	9.488e-02	3.352e-02
	0.001	3	502.64	2.583e-02	2.303e-02
		5	872.89	2.828e-02	1.599e-02
		10	1922.64	3.000e-02	1.060e-02
	0.0001	3	505.03	8.175e-03	7.408e-03
		5	858.09	8.945e-03	5.060e-03
		10	2187.10	9.488e-03	3.350e-03

Figure 17: 100000×1000000