On Application of Block Kaczmarz Methods in Low-Rank Matrix Factorization

Edwin Chau[†]

Project advisor: Jamie Haddock[‡]

Abstract. Matrix factorization techniques compute low-rank product approximations of high dimensional data matrices and as a result, are often employed in recommender systems and collaborative filtering applications. However, many algorithms for this task utilize an exact least-squares solver whose computation is time consuming and memory-expensive. In this paper we discuss and test a block Kaczmarz solver that replaces the least-squares subroutine in the common alternating scheme for low-rank matrix factorization. This variant trades a small increase in factorization error for significantly faster algorithmic performance. In doing so we find block sizes that produce a solution comparable to that of the least-squares solver for only a fraction of the runtime and working memory requirement.

1. Introduction. With the recent rise and ubiquity of online services, the volume of data available for and demanding analysis has exploded. This data often resides in extremely high-dimensional space, and for this reason, it is often computationally and intuitively beneficial to reduce its dimension, offering decreased required storage space and insight into the latent trends within the data. Matrix factorizations (e.g., principal component analysis/singular value decomposition (PCA/SVD) [18], CUR factorization [12], pseudo-skeleton approximation [5, 16], and the nonnegative matrix factorization (NMF) [11, 21]) are common approaches to find factor matrices whose product is a low-rank approximation to the original data matrix. These factor matrices, when combined with auxiliary methods, can help identify useful trends and patterns present in the data. As a result, matrix factorization models often find use in tasks such as collaborative filtering [7, 17, 22], which predicts a user's preferences based on historical preferences of other similar/dissimilar users (e.g., for e-commerce product recommendation), and topic modeling [1], which discovers latent themes or topics in a dataset. Specific applications in which these models have been applied are document classification [2], facial recognition [20], and collaborative filtering for movie recommendation (as part of the winning submission to the Netflix Prize competition) [9].

While many dimension-reduction and topic modeling approaches are framed as matrix factorization models, we focus on the simple low-rank factorization model for data $X \in \mathbb{R}^{m\times n}$ that seeks $A \in \mathbb{R}^{m\times k}$, generally called the dictionary matrix, and $S \in \mathbb{R}^{k\times n}$, generally called the representation matrix, such that $X \approx AS$. Here we consider the *factor rank* k to be a user-defined parameter but assume that $k < \min\{m, n\}$ so that the resulting factorization reduces the dimension of the original data and/or reveals latent themes in the data. Generally, the factor rank k can be chosen according to *a priori* information or a heuristic method. Data points are typically stored as columns of X, thus m represents the number of features or attributes of a single observation, and n represents the number of observations. The columns

[‡]Department of Mathematics, Harvey Mudd College, Claremont, CA 91711 (jhaddock@g.hmc.edu, https://jamiehadd.github.io).

[†]Department of Mathematics, University of California, Los Angeles, Los Angeles, CA 90095 (edwinchau@ucla.edu, https://chauedwin.github.io)).

of A are generally referred to as *topics* or *dictionary elements*, which are characterized by features of the dataset. Further, each column of S provides the approximate representation of the respective column in X in the lower-dimensional space spanned by the columns of A. Thus, the data points are well-approximated by a linear combination of the latent topics with coefficients given by the entries of S. Once the approximate factorization is computed, differences between entries of the original data matrix X and its approximation AS can be viewed as potential recommendations. We illustrate this process in the following toy example.

Suppose we have a data matrix consisting of four movie ratings from five different users (here each column represents a single user's ratings for each of the four movies and the presence of value zero indicates that a user has not rated a movie),

$$X = \begin{pmatrix} 5 & 5 & 0 & 5 & 5 \\ 0 & 4 & 0 & 0 & 3 \\ 0 & 0 & 5 & 5 & 0 \\ 1 & 0 & 4 & 0 & 0 \end{pmatrix}$$

and an arbitrary factorization algorithm yields the following low-rank approximation

$$A = \begin{pmatrix} 5 & 0 \\ 3 & 0 \\ 0 & 5 \\ 0 & 3 \end{pmatrix}, \quad S = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}, \quad AS = \begin{pmatrix} 5 & 5 & 0 & 5 & 5 \\ 3 & 3 & 0 & 3 & 3 \\ 0 & 0 & 5 & 5 & 0 \\ 0 & 0 & 3 & 3 & 0 \end{pmatrix}$$

We can interpret the columns of A as latent topics of movie preferences. For example, if the first two movies (rows of the matrices X and A) are in the horror genre and the last two movies are in the romance genre, then the first topic (column of A) corresponds to preferences towards horror and the second topic corresponds to preferences towards romance. The columns of S are weights of these topics for each user. In our example, the first two users and the fifth user (columns of X and S) appear to prefer horror films, the third user appears to prefer romance films, and the fourth user appears to enjoy both. Recommendations can then be made based on differences between the approximation AS and the original data matrix X, with larger differences representing more suitable recommendations (for entries in X equal to zero, which indicate movies a user has not yet tried). For example, we could recommend the second movie to the first and fourth users or the fourth movie to the fourth user.

Many methods for matrix factorization utilize an alternating scheme due to the non-convexity of some common problem formulations [23]. In the model $X \approx AS$, this non-convexity means that there may exist many locally optimal factor matrices A and S that approximate X. Take the one-dimensional case as an example, and suppose we are trying to approximate $X = \begin{pmatrix} 1 \end{pmatrix}$. There exist two optimal factorizations for X, namely $A = \begin{pmatrix} 1 \end{pmatrix}, S = \begin{pmatrix} 1 \end{pmatrix}$ and $A = \begin{pmatrix} -1 \end{pmatrix}, S = \begin{pmatrix} -1 \end{pmatrix}$, and the resulting solution produced by an iterative method may depend on the initializations of A and S. Thus, matrix factorization methods often iteratively alternate through the factor matrices, holding the others constant and updating one at a time. For example, the common Frobenius norm formulation of the matrix factorization problem asks, given a data matrix X, to minimize the objective function

$$\|X - AS\|_F^2 \tag{1.1}$$

with respect to $A \in \mathbb{R}^{m\times k}$ and $S \in \mathbb{R}^{k\times n}$. Alternating methods that approximately minimize the norm do so by first fixing A and iteratively solving for S such that $X \approx AS$ (usually using the same formulation (1.1)), and then fixing S and iteratively solving for A such that $X \approx AS$. After fixing all but one of the factors, the resulting problem is often convex and the formulation requires approximate solution of a simple linear system; in the case of (1.1), the resulting problems are simple least-squares problems. While the details of this process will be discussed in Section 1.3, we note now that the quality and computational time of the solution is dictated by the chosen linear system or least-squares solver.

One commonly utilized solver is the method of least-squares, where the exact least-squares solution ($\text{argmin}_x \|Ax - b\|^2$) is computed using the pseudo-inverse of the matrix, $A^\dagger$. Alternating methods that utilize least-squares solvers are commonly referred to as alternating least squares (ALS). Provided the data matrix and its factor matrices can fit into running memory, the factorization can simply be computed by setting one of the factor matrices equal to the matrix product of the data matrix and the pseudo-inverses of the other factors. However, large-scale data will rarely completely fit into running memory and must be processed using *batch* methods; see e.g., [10]. Such techniques lessen the running memory requirement by sampling portions of the factor matrices to compute updates that correspond to the samples, and are the focus of our work. To avoid confusion, we will refer to alternating least squares methods that utilize batch sampling as batch alternating least squares (BALS).

While quick to converge, the least-squares solver has a two-fold problem. First, calculating an exact solution for commonly encountered large linear systems is computationally expensive, requiring significantly more time to run as data matrices increase in dimension. Second, the least-squares solver requires the entire data matrix to be loaded into memory to solve for a solution, which significantly limits the size of systems for which this technique can be employed. Because of these drawbacks, we focus instead on utilizing an iterative method of solving linear systems within the alternating scheme of matrix factorization, namely *randomized Kaczmarz (RK)* and its generalization, *block randomized Kaczmarz (BRK)*.

Kaczmarz methods are a large family of methods used to approximate the solution of a linear system by iteratively sampling and solving a subset of the system. The Kaczmarz method was first introduced by S. Kaczmarz [8]. The randomized Kaczmarz (RK) method, a variant of the Kaczmarz method that randomly samples rows proportional to their ℓ_2 norm, was later introduced by T. Strohmer and R. Vershynin and proven to have exponential convergence [19]. While Kaczmarz methods approximate the exact least-squares solution when the linear system is consistent, they do not necessarily approximate the least-squares solution when the system is inconsistent. This is because the method projects iterates into the solution space of individual equations which likely do not contain the least-squares solutions in the case that the system is inconsistent. The *randomized extended Kaczmarz (REK)* is a variant of RK suitable for noisy or inconsistent systems and which converges to the exact least-squares solution [24]. Furthermore, the generalization of RK known as block randomized Kaczmarz (BRK), where a submatrix is sampled rather than a single row, was introduced and shown to outpace RK on both well paved and randomly generated matrices [14]. Employment of nonstandard stepsizes has further sped the convergence of the BRK method [13].

Due to the often sparse and redundant nature of large scale data, a Kaczmarz method is a natural alternative to least-squares in a matrix factorization approach. We therefore consider

an approach which replaces the traditional least-squares solver with BRK. This family of methods aims to approximate a solution to each least-squares problem rather than computing an exact one, trading accuracy for a quicker runtime.

We note that the methods studied in this paper are not novel; similar methods have been proposed in the context of sensor network localization [4]. The main similarity between the methods we focus on and those addressed in the aforementioned work is the utilization of Kaczmarz methods to efficiently compute approximate matrix factorizations. In both cases, the algorithm partitions a factor matrix (i.e., A or S in (1.1)) before computing the block Kaczmarz update for the uniformly chosen partition (See Section 2.2 for details).

However, the methods we consider differ in the way we partition the factor matrices. While the aforementioned work fixes the partitions of rows and columns of the factor matrices throughout the duration of the algorithm and iteratively samples from them, our method randomly samples individual rows or columns of the factor matrix at each iteration until the specified partition size is fulfilled. By relaxing the requirement of fixed partitions, we demonstrate that this BRK based approach performs well when the sampling procedure is more flexible. Our work also differs in its application to recommender systems. Furthermore, we illustrate the potential computational speedup offered by these methods in the large-scale data regime.

Our contributions include empirical results on sparse large-scale synthetic and real-world datasets as well as a publicly available Python package[1] implementing the methods discussed.

1.1. Organization. The rest of the paper is organized as follows. Section 2 provides details of the aforementioned ALS and BRK methods. Section 3 applies these methods on small-scale synthetic, large-scale synthetic, and large-scale real-world datasets and compares their approximation error and runtimes. Finally, in Section 4, we provide some final conclusions and discussion of future work.

1.2. Notation. Let $A \in \mathbb{R}^{m\times n}$ be a matrix with m rows and n columns. For a matrix A, we denote the i^{th} row of A as $A_{i,:}$ and the i^{th} column as $A_{:,i}$. We let $\|A\|$ denote the spectral norm of A, $\|A\|_F$ denote the Frobenius norm of A, $A^\dagger$ denote the Moore-Penrose pseudoinverse of A, and $\sigma_{\max}(A)$ denote the maximum singular value of A. Finally, the methods considered in this work iteratively produce approximations to factor matrices A and S in the matrix factorization formulation (1.1); we denote the approximation to matrix A produced in the j^{th} iteration of a method as $A^{(j)}$. We let $[n]$ denote the integers in the interval from 1 to n, $[n] = \{1, 2, \ldots, n\}$.

1.3. Preliminaries. As mentioned previously, the alternating approach to computing the matrix factorization,

$$AS \approx X, \tag{1.2}$$

is to repeatedly fix one factor matrix and update the other. Here, ALS will simply perform a pseudo-inverse calculation and matrix multiplication calculation(s) to achieve a solution. However, a typical batch alternating scheme attempts to minimize the objective function

[1] https://github.com/chauedwin/mf-algorithms

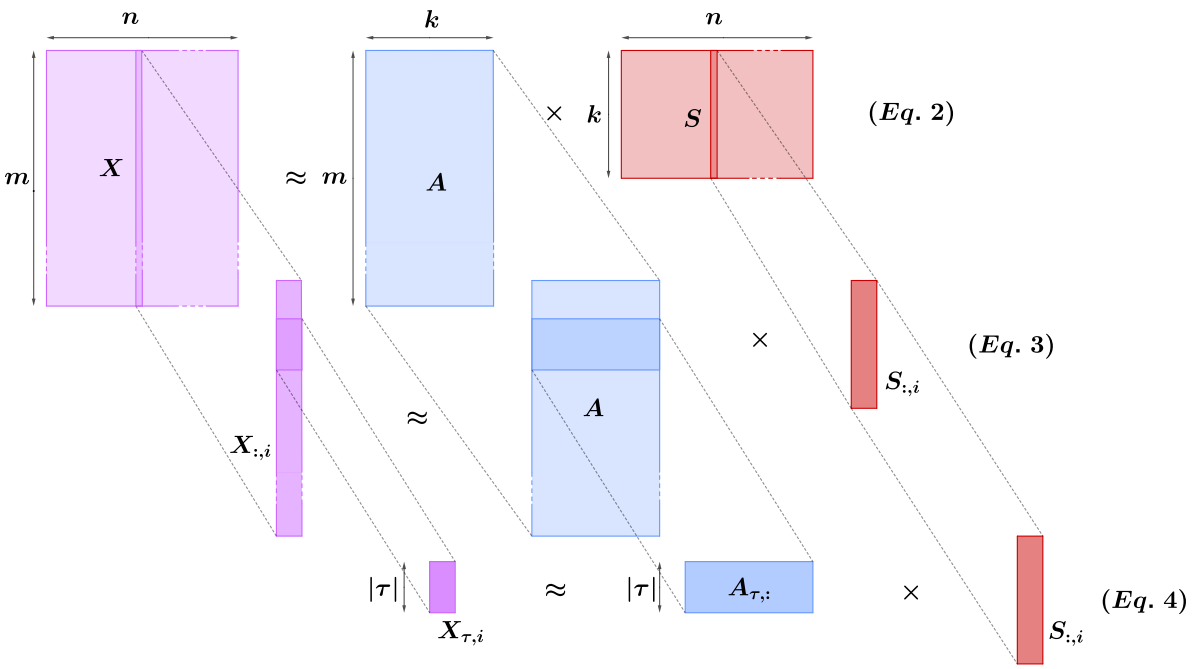


Figure 1: The matrix equation (2) is reduced to a linear system (3) and solved using least-squares or further reduced to (4) and solved with an iterative method.

(1.1) by randomly sampling a column $i \in [n]$ (or row $j \in [m]$) and reducing equation (1.2) column-wise (or row-wise) to linear systems

$$AS_{:,i} \approx X_{:,i} \text{ or } A_{j,:}S \approx X_{j,:} \tag{1.3}$$

if solving for S (or A). We will refer to this type of sampling as *matrix-vector sampling*. The values of indices i and j will depend on the chosen sampling strategies discussed in the next section. The batch alternating least-squares method would then utilize the method of least-squares to update $S_{:,i}$ or $A_{j,:}$ before continuing to the next iteration. On the other hand, a Kaczmarz method takes a sample of the equations τ_1 or τ_2 and further reduces approximation (1.3) to

$$A_{\tau_1,:}S_{:,i} \approx X_{\tau_1,i} \text{ or } A_{j,:}S_{:,\tau_2} \approx X_{j,\tau_2} \tag{1.4}$$

before updating approximate solutions for $S_{:,i}$ and $A_{j,:}$ as outlined in Section 2.2. We will refer to this second type of sampling as *vector-sample sampling*. See Figure 1 for a visualization of these algorithmic reductions.

2. Algorithms. The methods we consider follow the alternating scheme, which iteratively matrix-vector samples a row from the left factor matrix (or column from the right factor matrix) and updates it while holding the other factor constant. They reduce the computation required in each iteration of alternating least-squares via the sequence of reductions visualized

in Figure 1. We refer to the first reduction from (1.2) to (1.3) as the *matrix-vector reduction*, and the second reduction from (1.3) to (1.4) as the *vector-sample reduction*.

Matrix-vector sampling and their corresponding reductions can be performed cyclically or stochastically, both of which have differing effects on the quality of the resulting approximation. Cyclic samples iterate through and update every row/column of the factor matrix. In this context, one iteration of the algorithm refers to one epoch, where all rows and columns of A and S are updated once.

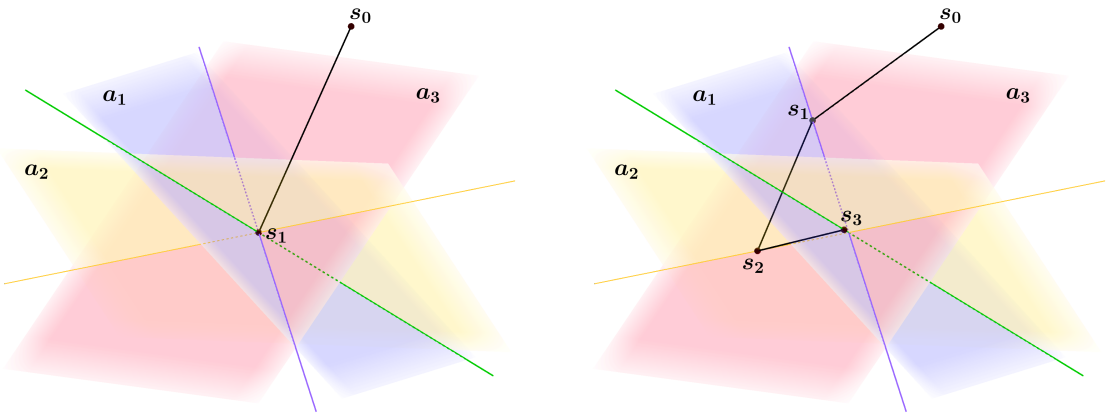
On the other hand, stochastic sampling randomly selects a row/column of the factor matrix at each iteration. These are sampled without replacement to prevent solving the subsampled linear system so precisely that it reduces the quality of the overall matrix factorization approximation, thus stochastic sampling may not necessarily update every row/column of the factor matrix. In the case that $X \in \mathbb{R}^{m\times n}$ is not square, the number of row updates to A and column updates to S can be made proportional; that is, for every one column update to S, we perform $\lceil m/n \rceil$ row updates to A (assuming $m > n$). Here, one iteration refers to a single matrix-vector sample and one epoch corresponds to $\min(m, n)$ iterations.

The algorithms we will discuss and compare follow the alternating update scheme, the iterations of which we will refer to as *matrix-vector iterations*. We will also refer to iterations of the BALS/BRK methods as vector-sample iterations, which perform either exact least-squares or iterations of a Kaczmarz method while matrix-vector iterations alternate between the factor matrices. Our focus is on the BRK-based alternating method, where the vector-sample reduction is performed according to a stochastic sample. While algorithms could employ a combination of cyclic and stochastic schemes (for example, performing cyclic matrix-vector reductions and stochastic vector-sample reductions), they are beyond the scope of this paper. We focus instead only on a method which applies stochastic sampling in both the matrix-vector and vector-sample reductions.

Finally, the manner of initialization of the factor matrices A and S also affects the final approximation. Because the objective function (1.1) is non-convex in both A and S as mentioned previously, there are many possible local optima within reach of matrix factorization algorithms and which local optimum the method finally reaches depends upon the initialization of the factor matrices. For this reason, our methods will simply randomly initialize two factor matrices before iteratively computing an improved approximation. While more sophisticated initialization techniques have been developed to promote faster convergence in specific applications, most do not guarantee this property in a general setting [3].

2.1. Batch Alternating Least-Squares. The BALS algorithm is as follows. Given a data matrix $X \in \mathbb{R}^{m\times n}$ and a target factor rank k, two factor matrices $A \in \mathbb{R}^{m\times k}$ and $S \in \mathbb{R}^{k\times n}$ are randomly initialized. It first samples an i^{th} column of S and performs a matrix-vector sample and reduction specified in (1.3).

It then solves for $S_{:,i}$ a user-specified number of times using the least-squares method, updating $S_{:,i}$ with the explicit least-squares solution of the now reduced equation . The algorithm then repeats for a row of A. This iterative process continues until an exit condition is met. The pseudocode for this algorithm is provided in Method 2.1; note that this algorithm exploits only the matrix-vector reductions. Furthermore, BALS would specifically use the explicit least-squares solution as the least-squares solver for the matrix-vector reduced equation



(a) BALS update (b) BRK update

Figure 2: Visualizations of updates. Linear equations are represented by hyperplanes and the solution space of two equations is represented by their intersection. **(Left)** A BALS update solves the linear system in one step. **(Right)** Three BRK updates defined by orthogonally projecting the previous iterate onto the solution space of the two sampled equations.

of each matrix-vector iteration.

While it is common to use a relative tolerance as a stopping condition, we opted to instead specify the exact number of matrix-vector iterations for both the BALS method and the following Block Randomized Kaczmarz method. This choice was due to the fact that the relative tolerance may not be reached in a reasonable amount of time when factorizing a full rank matrix (see Section 3.3 for an example). However, a relative tolerance can be utilized in conjunction with a maximum number of matrix-vector iterations in a practical setting.

2.2. Block Randomized Kaczmarz. Similar to BALS, the BRK algorithm first randomly initializes A and S and performs a matrix-vector sample and reduction for a column of S. The BRK algorithm then departs from BALS by performing a subsequent vector-sample reduction for $|\tau_1| \leq m$ rows of A and X.

It then performs a BRK update,

$$S_{:,i}^{(j+1)} = S_{:,i}^{(j)} + (A_{\tau_1,:})^{\dagger}(X_{\tau_1,i} - A_{\tau_1,:}S_{:,i}^{(j)}). \tag{2.1}$$

We can similarly update A by selecting a subset $|\tau_2| \leq n$ of the columns of S,

$$A_{i,:}^{(j+1)} = A_{i,:}^{(j)} + (X_{i,\tau_2} - A_{i,:}S_{:,\tau_2})(S_{:,\tau_2})^{\dagger}. \tag{2.2}$$

This vector-sample reduction and subsequent BRK update is also performed a user-specified number of times. The pseudocode for this algorithm is given in Method 2.2. Note that Method 2.1 details the alternating matrix factorization algorithm whereas Method 2.2 details a BRK subroutine which is a specific option for the LS solver in Method 2.1. Furthermore, the subsets τ_1, τ_2 do not need to be contiguous. Note that BRK is *equivalent to a Randomized Kaczmarz (RK) update* when $|\tau_1| = |\tau_2| = 1$ and *equivalent to a BALS update* when $|\tau_1| = m$ and $|\tau_2| = n$.

In this paper we will focus on uniform sampling methods. For matrix-vector sampling, this means the rows of A are picked with probability $\frac{1}{m}$ and columns of S are picked similarly. For vector-sample sampling, this means that τ_1, τ_2 are picked uniformly without replacement. We will refer to this method as uniform block randomized Kaczmarz (UBRK).

Others have previously found that a weighted sampling method offers the benefit of quicker convergence [19], however we did not find success with it in this application. When testing its performance against UBRK, calculations required for a weighted sample resulted in a significantly longer runtime for little relative error improvement.

Method 2.1 Alternating Scheme

procedure AlternatingScheme($X \in \mathbb{R}^{m\times n}, k, M, L$)
  initialize $A \in \mathbb{R}^{m\times k}, S \in \mathbb{R}^{k\times n}$
  for $l = 1, \ldots, M$ **do**
    sample column index $i \in [n]$
    use L iterations of LS solver to approximately solve for $\mathbf{s} \in \mathbb{R}^k$ in

$$X_{:,i} \approx A\mathbf{s}$$

    replace ith column of S with $\mathbf{s}$,

$$S_{:,i} = \mathbf{s}$$

    sample row index $j \in [m]$
    use L iterations of LS solver to approximately solve for $\mathbf{a} \in \mathbb{R}^k$ in

$$X_{j,:}^\top \approx S^\top \mathbf{a}$$

    replace jth row of A with $\mathbf{a}$,

$$A_{j,:} = \mathbf{a}^\top$$

  end for return A, S
end procedure

Remark 2.1. The choice of L, or number of iterations of the least-squares solver, depends on the specific least-squares solver being utilized. While certain methods may have a natural heuristic or indication of sufficient solution, block Kaczmarz does not (on an inconsistent system). Thus, we leave L as a user-specified parameter. Also, while alternative stopping cri-

Method 2.2 Block Randomized Kaczmarz (BRK)

procedure BRK($A \in \mathbb{R}^{m\times k}, \mathbf{b} \in \mathbb{R}^m, r, L$)
  initialize $\mathbf{y}_0 \in \mathbb{R}^k$
  for $l = 1, \ldots, L$ **do**
    sample $\tau \subset [n]$ such that $|\tau| = r$
    $\mathbf{y}_l = (I - A^{\dagger}_{\tau,:}A_{\tau,:})\mathbf{y}_{l-1} + A^{\dagger}_{\tau,:}\mathbf{b}_\tau$
  end for return $\mathbf{y}_L$
end procedure

teria may be chosen (rather than specifying a number of iterations), they are highly dependent on the structure of the problem or rate of convergence and may not be reached for particularly difficult inconsistent systems. For the same reason, we decided to leave the number of matrix-vector sampling iterations as a user-specified parameter as well.

3. Experimental Results. In this section, we apply the discussed algorithms to both synthetic and real data. We first compare their performances and results in detail on a small-scale synthetic data matrix before conducting a more significant comparison on synthetic and real large-scale data. Our implementations of these methods utilize Python's Numpy library [6], and pseudo-inverse calculations were avoided using Numpy's lstsq() function. The test matrices in all tests were randomly generated using Numpy as well. We note that ALS found a solution in a single iteration as expected while BALS and UBRK required many iterations to converge. This large difference is the reason we excluded ALS from experimental results and plots.

We used an Intel(R) Core(TM) i7-4770 CPU(3.40 GHz) with 16.0 GB of RAM running on Windows OS for relative error computations. Runtime calculations were performed using an Intel(R) Core(TM) i3-3225 CPU(3.30 GHz) with 6.0 GB of RAM running on GNU/Linux.

3.1. Small-scale Synthetic Data. We applied our matrix factorization method on a random matrix $X \in \mathbb{R}^{1000\times 1000}$. We generate X as the product of factor matrices $A \in \mathbb{R}^{1000\times 50}$ and $S \in \mathbb{R}^{50\times 1000}$, which were generated by sampling each entry from $\{0, 1, 2, 3\}$ with probabilities $\{0.97, 0.1, 0.1, 0.1\}$ and $\{0, 1\}$ with probabilities $\{0.99, 0.01\}$, respectively. The resulting matrix X had a sparsity of 98.58% and true rank of 50.

Our proposed iterative methods have four main parameters: factor rank, number of matrix-vector iterations, number of vector-sample iterations, and block size. Because the ideal factor rank is known to be 50, it will be fixed at that value for all small-scale tests. Note that BALS will be used as a baseline to gauge performance and will be fixed to employ one vector-sample iteration and the maximum possible block size (m rows of A and n columns of S) for all tests. Each data point in our plots is the average of 10 trials.

The first question we address is how much of the data matrix needs to be sampled at each iteration in order to converge to a solution. To answer this, we compared the UBRK method at block sizes of 20% increments and BALS. The results are shown in Figure 3. Although a wide range of block sizes can be used, we excluded block sizes that were not large enough to converge to a local solution.

We can see that block sizes of about 40% and above are sufficient to converge to a factor-

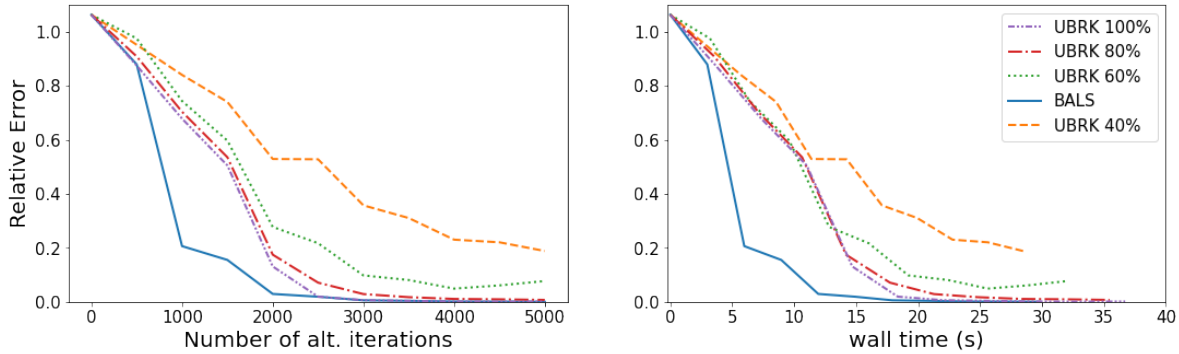


Figure 3: Results of small-scale synthetic data experiments. The data matrix $X \in \mathbb{R}^{1000\times1000}$ is generated by multiplying two synthetically generated factor matrices. The percentage following "UBRK" denotes the percentage of rows or columns sampled per iteration, i.e., UBRK 40% means 40% of the rows of A and 40% of the columns of S were sampled by UBRK. **(Left)** Relative error of methods with various numbers of matrix-vector iterations. **(Right)** Wall time of methods.

ization that captures most of the original matrix data (Figure 3 left). Furthermore, as block size increases, the rate of convergence increases and final relative error decreases. However, while larger block sizes lead to faster convergence, they are more expensive in terms of runtime (Figure 3 right). In particular, while a block size of 100% for UBRK achieves the same relative error as BALS, it requires more time to run. However, BALS is optimal for small datasets due to their manageable size allowing a better tradeoff of runtime for accuracy, as well as lower memory limitations. We will see in the large-scale experiments that UBRK has a computational advantage over BALS in larger dimensions.

We also varied the number of vector-sample iterations amongst 1, 10, 20, and 30 while fixing the number of matrix-vector iterations at 1000. An exit condition was included when updating a single row or column; the method exited vector-sample iterations and continued to the next matrix-vector iteration when $\|AS_{:,i} - X_{:,i}\| < \epsilon$. We found that for block sizes greater than 1, increasing the number of vector-sample iterations did not affect the relative error, meaning that the block Kaczmarz updates either converged after the first update or would have trouble converging at all.

3.2. Large-scale Synthetic Data. We now turn to a large-scale data matrix more reflective of real world data, and increase the data matrix size to $X \in \mathbb{R}^{10^5\times1000}$. This matrix was generated similar to our previous experiment. The factor matrices $A \in \mathbb{R}^{10^5\times50}$ and $S \in \mathbb{R}^{50\times1000}$ were generated by sampling each entry from $\{0, 1, 2, 3\}$ with probabilities $\{0.97, 0.1, 0.1, 0.1\}$ and $\{0, 1\}$ with probabilities $\{0.999, 0.001\}$, respectively. This resulted in a slightly higher sparsity of 99.86% and a true matrix rank of 29, which did not significantly affect the relative error to which the algorithms converged. In light of the previous set of experiments, we will fix the number of vector-sample iterations at 1 since increasing the vector-sample iteration count had no discernable effect on the resulting relative error.

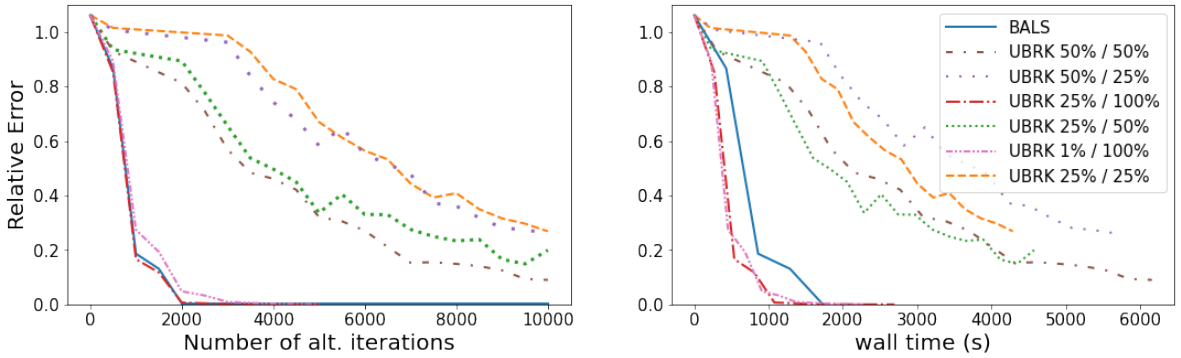


Figure 4: Results of large-scale synthetic data experiments. The data matrix $X \in \mathbb{R}^{10^5 \times 1000}$ is generated by multiplying two synthetically generated factor matrices. The percentages following "UBRK" denotes the percentage of rows/columns sampled per iteration, i.e. UBRK 25%/50% means 20% of the rows of A and 50% of the columns of S were sampled by UBRK. **(Left)** Relative error of methods with various numbers of matrix-vector iterations. **(Right)** Wall time of methods.

With a tall matrix, it may be useful to explore asymmetric block sizes - block sizes that are different for each factor matrix - to "balance" the optimality of both factor matrices. The results of these experiments are depicted in Figure 4.

Note that BALS was terminated at 2000 matrix-vector iterations as its relative error effectively reached zero, and UBRK was terminated at 5000 iterations for the same reason (Figure 4 left). While symmetric block sizes of 25% and 50% were slower to converge to a solution than BALS despite their faster runtimes, they can still be advantageous when facing a memory limitation, as they use 25% and 50% of the required memory for BALS, respectively.

However, an interesting result is that an asymmetric block size of 1% of rows of A and the full S matrix was comparable in performance to BALS (Figure 4 left). This choice of block sizes not only cuts the runtime of the matrix factorization task in half, but also requires a bit less than 2% of the working memory required by BALS.

It is also important, as evident in Figure 4 (left), to note that the runtime of block size pairs 25%/25% (orange) and 1%/100% (pink) are very similar despite the decrease from sampling 25% to 1% of the left factor matrix (or from $25,000$ rows to 1000). This is due to the left factor requiring more updates than the right factor. Because our data matrix has 100 times more rows than columns, the methods update the left factor matrix 100 times for every 1 update of the right factor. As a result the increase from sampling 25% to 100% of the right factor (or from 250 columns to 1000) overshadows the decrease in sampled rows.

The algorithms have been able to achieve a low final relative error due to choosing a factor rank close to the true matrix rank of the synthetically generated data. When testing on real data, where the true rank is often unknown, our choice of factor rank may significantly affect the relative error of resulting approximations. This will be the case in the following experiment.

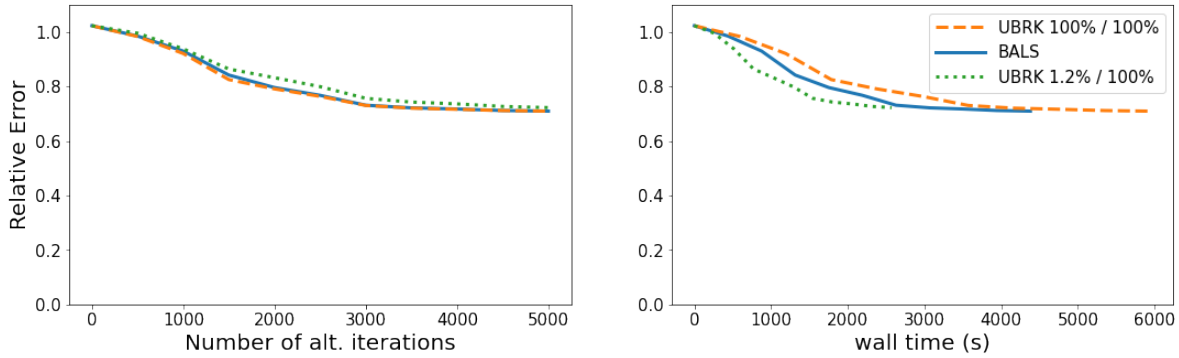


Figure 5: Results of Amazon dataset experiments. The data matrix $X \in \mathbb{R}^{128877 \times 1548}$ was full rank with a factor rank of 50. **(Left)** Relative error of methods with various numbers of matrix-vector iterations. **(Right)** Wall time of methods.

3.3. Real World Data. Finally, we compare algorithm performance on a real-world dataset from Amazon, which is publicly available as part of Dr. J. McAuley's Recommender Systems Datasets [7, 15]. The data was originally a dataframe of reviewer IDs, product IDs, time and ratings, which we reformatted by generating a data matrix with the reviewer IDs as the row indices, product IDs as the column indices, and ratings as the matrix entries. The resulting matrix was 128877×1548 and had a sparsity of 0.99926. Zero entries represented the absence of a rating and nonzero integer entries ranged from 1 to 5.

We once again chose a factor rank of 50. However, the true matrix rank was 1548, and this difference will impact the quality of resulting approximations, as shown in Figure 5. Both BALS and UBRK converged to the solution at a slower rate and converged to a higher relative error at about 0.7. This means that the solution was only able to capture about 30% of the information provided by the test matrix. This is due in part to the mismatch between the factor rank and the true data rank.

Similar to the large-scale synthetic data, sampling the entire S matrix and a small percentage of A enabled UBRK to reach a solution with final relative error similar to that of ALS, and to converge far quicker. In this scenario, UBRK with less than maximal S block size was not able to converge to final relative error similar to that of BALS.

4. Conclusions. This work rigorously compares the quality and speed of BALS and BRK subroutines in a common low-rank matrix factorization setting. We explore how different combinations of block sizes for BRK affect its solution quality and runtime when compared to an BALS benchmark. During this process we also observe patterns in particularly successful BRK block sizes; sampling most, or all, of the smaller factor matrix and only a small portion of the larger factor matrix is sufficient for converging to a solution similar in quality to BALS in less time.

Directions for future work regarding Kaczmarz subroutines include finding optimal step-sizes, further theoretical proofs detailing relative error behavior as the solution approaches a local minimum, and a qualitative comparison between BALS and BRK solutions. It may be

that the BRK update strategy might improve factor matrix sparsity; BALS often produces highly dense factor matrices due to the application of exact least-squares as a subroutine. This direction needs further investigation. Future work on matrix factorization in general can also include finding an optimal factor matrix initialization. We also plan to generalize these techniques to tensor decompositions, where the size of the data can be extremely large and subsampling can offer drastic speedup.

Acknowledgements. The author would like to thank the manuscript referees for their thoughtful and detailed comments which significantly improved earlier versions of this work and has recommended interesting future directions for consideration. We would like to thank Jacob Moorman for his guidance and resources for preparing and publishing Python packages. Sponsor JH was partially supported by NSF CAREER DMS #1348721, NSF DMS #2011140 and NSF BIGDATA DMS #1740325 which are led by Professor Deanna Needell. JH is also partially supported by NSF DMS #2111440.

REFERENCES

[1] M. Berry and M. Browne. Email Surveillance Using Non-negative Matrix Factorization. *Comput. Math. Organ. Th.*, 11:249–264, 10 2005.

[2] E. Gaussier and C. Goutte. Relation between PLSA and NMF and implications. In *Proc. ACM SIGIR Conf. on Research and Development in Inform. Retrieval*, pages 601–602, 2005.

[3] N. Gillis. The Why and How of Nonnegative Matrix Factorization. *Regularization, Optimization, Kernels, and Support Vector Machines*, 12:257–291, 01 2014.

[4] A. Gogna and A. Majumdar. Matrix factorization model using Kacmarz algorithm: Application in sensor localization. In *IEEE International Conf. on Digital Signal Processing (DSP)*, pages 219–223, 2015.

[5] S. Goreinov, E. Tyrtyshnikov, and N. Zamarashkin. A theory of pseudoskeleton approximations. *Linear Algebra Appl.*, 261(1):1–21, 1997.

[6] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del R'ıo, M. Wiebe, P. Peterson, P. G'erard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020.

[7] R. He and J. McAuley. Ups and Downs: Modeling the Visual Evolution of Fashion Trends with One-Class Collaborative Filtering. *Proc. International Conference on World Wide Web - WWW '16*, 2016.

[8] S. Kaczmarz. Angenäherte auflösung von systemen linearer gleichungen. *Bull. Internat. Acad. Polon. Sci. Lettres A*, pages 335–357, 1937.

[9] Y. Koren. Collaborative filtering with temporal dynamics. In *Proc. ACM SIGKDD International Conf. on Knowledge Discovery and Data Mining*, pages 447–456, 2009.

[10] B. W. Larsen and T. G. Kolda. Practical leverage-based sampling for low-rank tensor decomposition. *arXiv preprint arXiv:2006.16438*, 2020.

[11] D. D. Lee and H. S. Seung. Algorithms for non-negative matrix factorization. In T. K. Leen, T. G. Dietterich, and V. Tresp, editors, *Adv. Neur. In.*, pages 556–562. MIT Press, 2001.

[12] M. W. Mahoney and P. Drineas. CUR matrix decompositions for improved data analysis. *Proc. National Academy of Sciences*, 106(3):697–702, 2009.

[13] I. Necoara. Faster randomized block Kaczmarz algorithms. *SIAM J. Matrix Anal. A.*, 40(4):1425–1452, 2019.

[14] D. Needell and J. Tropp. Paved with Good Intentions: Analysis of a Randomized Block Kaczmarz Method. *Linear Algebra Appl.*, 441, 08 2012.

[15] J. Ni, J. Li, and J. McAuley. Justifying recommendations using distantly-labeled reviews and fine-grained aspects. In *Proc. Conf. on Empirical Methods in Natural Language Processing and the 9th*

International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), pages 188–197, Hong Kong, China, 11 2019. Association for Computational Linguistics.

[16] A. Osinsky and N. Zamarashkin. Pseudo-skeleton approximations with better accuracy estimates. *Linear Algebra Appl.*, 537, 10 2017.

[17] R. Pan, Y. Zhou, B. Cao, N. N. Liu, R. Lukose, M. Scholz, and Q. Yang. One-class Collaborative Filtering. In *Proc. IEEE International Conf. on Data Mining*, pages 502–511, 2008.

[18] K. Pearson. LIII. On lines and planes of closest fit to systems of points in space. *Philos. Mag.*, 2(11):559–572, 1901.

[19] T. Strohmer and R. Vershynin. A randomized Kaczmarz algorithm with exponential convergence. *J. Fourier Anal. Appl.*, 15, 03 2007.

[20] M. A. Turk and A. P. Pentland. Face recognition using eigenfaces. In *IEEE Comput. Soc. Conf.*, pages 586–587. IEEE Computer Society, 1991.

[21] E. Tyrtyshnikov and E. Shcherbakova. Methods for Nonnegative Matrix Factorization Based on Low-Rank Cross Approximations. *Comp Math Math Phys+*, 59:1251–1266, 08 2019.

[22] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In *Proc. International Conf. on Algorithmic Aspects in Inform. and Management*, pages 337–348, 06 2008.

[23] Z. Zhu, Q. Li, G. Tang, and M. B. Wakin. Global optimality in low-rank matrix optimization. *IEEE T. Signal Proces.*, 66(13):3614–3628, Jul 2018.

[24] A. Zouzias and N. M. Freris. Randomized Extended Kaczmarz for Solving Least-Squares. *SIAM J. Matrix. Anal. A.*, 34(2):773–793, 2013.