Initializing Nonnegative Matrix Factorizations On Distributed Architectures

Alex H. Williams
Neurobiology
Stanford University
Stanford, CA 94305, USA
ahwillia@stanford.edu

1 Introduction

Nonnegative matrix factorization (NMF) [9] approximates a data matrix $A \in \mathbf{R}^{n \times m}$ as a rank-r product of two non-negative factor matrices $W \in \mathbf{R}^{n \times r}$ and $H \in \mathbf{R}^{r \times n}$, typically with $r \ll \min(n, m)$. That is, NMF involves approximately solving the optimization problem:

$$\begin{array}{ll} \underset{W,H}{\text{minimize}} & \|A - WH\|_F^2 \\ \text{subject to} & W \geq 0,\ H \geq 0 \end{array} \tag{1}$$

This problem is nonconvex, and is NP-hard in the worst case [19]. However, approximate solutions to this problem are sought in practice and are successful for a variety of applications (see [9] for a review). A contributing factor to this empirical success is the fact that problem (1) is biconvex, meaning that fixing either W or H and optimizing over the other variable is a convex problem. Specifically, each subproblem amounts to a non-negative least squares solve, which can be solved relatively efficiently using active-set methods (e.g. [15]). This suggests the Alternating Nonnegative Least-Squares (ANLS) procedure, which performs alternating minimizations on W and H until convergence [procedure 1]. This is guaranteed to converge to a stationary point of problem (1), and is the most popular approach for fitting NMF in practice [14].

Recent work has described NMF algorithms in a distributed setting. Implementations for ANLS have been described in Hadoop [16], MPI [13], and Spark [17, 21]. ANLS is not trivially parallelizable in a distributed setting since the non-negative least squares subproblems in (1) decompose over different sets of variables. Minimizing with respect to W (resp. H) can be done in parallel across rows (resp. columns) of the data matrix A, but not across the columns (resp. rows) of A. Thus, each iteration of ANLS involves significant communication costs. Lower bounds for these costs, as well as an algorithm that saturates these bounds, are described in [13]. Other methods for fitting NMF include stochastic gradient-based methods, which are described in a distributed setting by [20, 22].

Despite the popularity of ANLS, this algorithm can get stuck in local minima. Finding a good initialization or "warm start" for NMF is therefore of substantial interest and practical importance. This is particularly true in the distributed setting, since each iteration of ANLS has high communication overheads [13]. In this manuscript, I review several initialization and optimization techniques for NMF, and analyze their complexity in an idealized PRAM and a distributed computing setting. In particular, I focus on the following:

- Successive Projection Algorithm (SPA) [1, 10]
- Nonnegative Double Singular Value Decomposition (NNDSVD) [4]

I conclude that these procedures require relatively little communication and computational overhead, and are relatively easy to extend to distributed architectures.

Procedure 1 Alternating Non-negative Least-Squares (ANLS)

```
Input: $A \in \mathbf{R}^{m \times n}$ holding non-negative data, and desired rank $r$.
Output: Nonnegative factor matrices $W \in \mathbf{R}^{m \times r}$ and $H \in \mathbf{R}^{r \times n}$
1: Initialize $W$ and $H$ randomly
2: while convergence criteria not satisfied do
3:     $W \leftarrow \arg\min_{\hat{W} \geq 0} \|A - \hat{W}H\|_F^2$
4:     $H \leftarrow \arg\min_{\hat{H} \geq 0} \|A - W\hat{H}\|_F^2$
5: end while
```

1.1 Background on Parallel and Distributed Analysis

In most cases, I make the assumption that the dataset can be distributed in 1-dimensional blocks across machines. That is, A is either a "tall-skinny" or a "short-fat" matrix so that either m or n is small enough to be fit on a single machine. These two cases are equivalent from an algorithmic view, since it is trivial to modify a procedure to take A^T as input rather than A.

I analyze algorithms in two theoretical settings. First, in an idealized *parallel random access machine (PRAM)* in which p processors have access to a shared memory block, with no communication overhead. For an introduction and review of this setting see [3]. Briefly, the complexity of an algorithm in the PRAM setting can be characterized by the *work*, the total number of operations performed, and the *depth* (sometimes called the *span*), the longest sequential chain of dependent computations. Work roughly equates to the algorithm runtime when executed sequentially on a single processor, T_1, while depth represents runtime with an unlimited number of processors, T_∞. Work and depth provide upper and lower bounds on the runtime with p processors, T_p, by Brent's theorem [5]:

$$\frac{T_1}{p} \leq T_p \leq \frac{T_1}{p} + T_\infty$$

A shortcoming of Brent's theorem and the PRAM setting is that they do not account for communication costs between processors, which are in practice an important bottleneck on large datasets. Thus, I also study each algorithm in a distributed computation setting involving p machines with private memory banks. In this case, the inputs and outputs of the algorithm, as well as intermediate data structures may be distributed over multiple machines. There are many possible characteristics to report in the distributed setting. I mostly emphasize *(a)* the number of rounds of communication, and *(b)* the largest message size. See [6] for a more thorough review of communication patterns and costs; the algorithms I analyze only use basic communication collectives (*reduce*, *broadcast*, and *all-gather*).

2 Successive Projections Algorithm (SPA)

2.1 Background

Definition 2.1. *Separability.* We call A separable if it admits a decomposition $A = WH$ in which the columns of H can be permuted to form an $r \times r$ block that is diagonal. That is, for each row in H there exists a column of H that contains a nonzero entry in that row, but contains zeros for all other rows.

Claim. Let $A_\mathcal{S} \in \mathbf{R}^{m \times n}$ denote a separable dataset. Then, separability condition implies that the columns of W appear as columns in A (after scaling).

Proof. Let $\tilde{A}_\mathcal{S}$ and $\tilde{H}$ denote the (separable) data and factor matrix H after (a) scaling each column to be unit length, and (b) permuting the columns such that the first r rows of $\tilde{H}$ contain an identity matrix. Then the first r columns of A contain W:

$$\tilde{A}_\mathcal{S} = W\tilde{H} = W\left[\begin{array}{c|c} I_r & \cdots \end{array}\right] = \left[\begin{array}{c|c} W & \cdots \end{array}\right]$$

□

SPA and similar techniques are motivated by the observation that, by constraining the columns of H to be fixed length, then the vertices of the convex hull of A form W (assuming separability). This leads to the following theorem, which is proved in [2].

Theorem 2.1. (Arora et al., [2]) If a dataset is separable, then there exists a polynomial time algorithm that provably solves the NMF optimization problem.

Although most real-world datasets do not strictly satisfy the separability condition, geometric algorithms based on this assumption are effective in many settings [9]. For example, in document classification [2], NMF can be used to assign r latent topics to m documents for a dataset containing word counts for a dictionary of n words in each document. In this case, certain words (called "anchor words") may only occur in particular topics. For example, "umpire" may only occur in the "sports" section of a corpus of newspaper articles. If there exists at least one anchor word for each topic, then this implies the separability condition.

In practice, many datasets are not perfectly separable. Thus, it can be effective to use the result of SPA as a warm start to ANLS or other descent-based algorithms. Theoretical results on the robustness of SPA to noise are derived in [10], and analysis of similar techniques are given in [8]. SPA was originally introduced by [1].

2.2 Sequential SPA algorithm

As briefly mentioned in the previous section, SPA treats the columns of A as points in $\mathbf{R}^n$ and finds the convex hull of these points; the vertices of the convex hull correspond to columns in W. After W is found, H can be found with a single non-negative least-squares solve.

Procedure 2 Successive Projection Algorithm (SPA)

Input: $A \in \mathbf{R}^{m \times n}$ holding non-negative data, and desired rank r.

Output: Nonnegative factor matrices $W \in \mathbf{R}^{m \times r}$ and $H \in \mathbf{R}^{r \times n}$

```
1: $R = \text{copy}(A), \quad \mathcal{J} = \{\}$
2: for $k = 1 : r$ in parallel do
3:     $q = \arg\max_j \|R_{:j}\|_2^2$
4:     $R = R - \frac{R_{:q} R_{:q}^T R}{\|R_{:q}\|_2^2}$
5:     $\mathcal{J} = \mathcal{J} \cup \{q\}$
6: end parallel for
7: $W = A(\mathcal{J})$
8: $H = \textbf{NNLS}(W, A)$
```

2.3 PRAM model

Claim. SPA takes $O(rmn)$ work and $O(r \log m + r \log n)$ depth to find W (lines 1-7).

Proof. The algorithm takes r sequential iterations, each of which takes $O(mn)$ work and $O(\log m + \log n)$ depth as detailed below. Lines 1 and 7 do not change this complexity (line 1 is completed after $O(mn)$ copy operations, and both are embarrassingly parallel).

Line 3. Each $\|R_{:j}\|_2^2$ for $j \in 1, ..., n$ can be computed in parallel. Each of these norms takes involves summing m numbers, which involves $O(m)$ work and $O(\log m)$ depth (since summation is a binary associative operation). Thus, the work is $O(mn)$ and depth is $O(\log n)$.

Line 4. First compute $R_{:q}^T R$ and then compute $R_{:q}(R_{:q}^T R)$. Again, the work is $O(mn)$ and depth is $O(\log n)$.

- Computing $R_{:q}^T R$ is a simply matrix-vector multiply which can be done in $O(mn)$ work and $O(\log m)$ depth. Briefly, n dot products are computed in parallel, each with work $O(m)$ and depth $O(\log m)$.
- Computing $R_{:q}(R_{:q}^T R)$ is simply an outer product of two vectors, which can be done in $O(mn)$ work and $O(1)$ depth. For two vectors $\mathbf{u} \in \mathbf{R}^m$ and $\mathbf{v} \in \mathbf{R}^n$, each entry in the result can be computed in parallel $(\mathbf{u}\mathbf{v}^T)_{ij} = u_i v_j$.

Lines 5. This is a $O(1)$ work and depth operation.

□

Remark. On a sequential machine the non-negative least squares problem on line 8 can be solved up to a tolerance parameter ϵ with $O(\log \frac{1}{\epsilon})$ iterations of a first-order (i.e. gradient-based) convex solver. Calculating the gradient is linear in the number of parameters, $O(rn)$, leading to a total of $O(rn \log \frac{1}{\epsilon})$ work. A variety of active-set methods are more typically used in practice, and a more thorough analysis is outside of the scope of this paper. Note that line 8 of [procedure 2] will have the same cost as half an iteration of ANLS.

2.4 Distributed Setting

Since the SPA iterations involve repeated operations on the columns of A, it is natural to distribute the columns of A across machines so that the majority of computation can be done locally. We restrict ourselves to the case where each column can be fit on a single machine (i.e. A is short and fat).

$$A = \left[\begin{array}{c|c|c|c} A^{(1)} & A^{(2)} & \cdots & A^{(b)} \end{array} \right] \tag{2}$$

In this regime $W \in \mathbf{R}^{m \times r}$ is a relatively small matrix that can be stored on each machine (typically $r \ll m$). We distribute H similar to A, so that machine ℓ holds the relevant partitions of H to approximate $A^{(\ell)}$, given W:

$$\left[\begin{array}{c|c|c} A^{(1)} & \cdots & A^{(b)} \end{array} \right] \approx W \left[\begin{array}{c|c|c} H^{(1)} & \cdots & H^{(b)} \end{array} \right] \quad \Rightarrow \quad A^{(\ell)} \approx W H^{(\ell)}$$

This suggests [procedure 3] for SPA in a distributed setting, which I analyze in terms of storage space, communication and computation cost.

Memory Allocation. Distributed SPA allocates $O(mn)$ new memory, due to the initialization of R in step 2. (Allocations for W, and H, are $O(mr)$ and $O(nr)$ and temporary allocations for $R_{:q}$ in each iteration are $O(m)$). This is perhaps the biggest shortcoming of translating SPA into a distributed setting. For very large datasets creating a copy may be prohibitive.

Communication Costs. Distributed SPA requires $O(r \log p)$ rounds of communication with a maximum message size $O(m)$. Each iteration involves a fixed number of map (all-to-one) and broadcast (one-to-all) communications. The largest message size is length m (broadcast $R_{:q}$ in step 9). There are exactly r sequential iterations, so the total number of communication rounds is $O(r \log p)$. Step 13 requires an all-to-all communication, but this does not alter the order of the result. Consider each column of W as a length m message. Then there are r messages of length m; a naïve algorithm that sequentially broadcasts each message takes $O(r \log p)$ rounds of communication.

Reduce-Key Complexity. The only reduce step occurs on line 6, which takes the maximum of p scalars. This is $O(p)$ if done sequentially.

Other Computational Costs. The PRAM analysis outlined in the previous section is directly applicable. The computations done on each machine require $O(\frac{rmn}{p})$ work and $O(r \log m + r \log \frac{n}{p})$ depth.

3 NNDSVD

3.1 Background

It is well-known that the best unconstrained rank-r approximation of a matrix in terms of the Frobenius norm is given by the truncated singular value decomposition (SVD). This represents a rare case

Procedure 3 Distributed SPA

```
Input: $A \in \mathbf{R}^{m\times n}$ holding non-negative data, distributed across $p$ machines with each machine
    holding $n/p$ columns.
Output: Each machine holds the full matrix $W \in \mathbf{R}^{m\times r}$. The second factor matrix $H \in \mathbf{R}^{r\times n}$ is
    distributed as $A$ across machines.
 1: $\mathcal{J}^{(\ell)} = \{\}$        ▷ Each machine, indexed by $\ell$, stores the set of selected indices.
 2: $R^{(\ell)} = \text{copy}(A^{(\ell)})$        ▷ Each machine copies the local part of $A$ in parallel.
 3: for $k = 1 : r$ do
 4:     $q^{(\ell)} = \arg\max_j \|R^{(\ell)}_{:j}\|$
 5:     map: Each machine emits $\langle 1 \to (q^{(\ell)}, \|R^{(\ell)}_{:q}\|)\rangle$
 6:     reduce: Maximum on $\|R^{(\ell)}_{:j}\|$, finding $q$ the column of max norm
 7:     broadcast: $q$ to all machines
 8:     broadcast: Machine holding column index $q$ broadcasts $R_{:q}$.
 9:     $R^{(\ell)} = R^{(\ell)} - \frac{R_{:q}R_{:q}^T R^{(\ell)}}{\|R_{:q}\|_2^2}$, $\mathcal{J}^{(\ell)} = \mathcal{J}^{(\ell)} \cup \{q\}$
10: end for
11: $W^{(\ell)} = A^{(\ell)}(\mathcal{J}^{(\ell)})$
12: all-gather: $W \leftarrow W^{(\ell)}$        ▷ $W$ needs to be consolidated on each machine for NNLS.
13: $H^{(\ell)} = \textbf{NNLS}(W, A^{(\ell)})$
```

in which there is an efficient method to find a provably optimal solution to a non-convex problem. One might hope that spectral methods for computing the SVD could be leveraged to find fast and effective algorithms for NMF.

There are unfortunately many limitations to spectral algorithms for NMF [14]. However, they can nevertheless serve as useful initializations ("warm starts") for ANLS and other descent-based methods. A useful theorem in this respect is that the leading pair of singular vectors of a non-negative matrix are also non-negative. This is a simple extension of the Perron-Frobenius theorem to non-square matrices.

Theorem 3.1. Denote the singular value of a decomposition of a non-negative matrix A of rank-r as $A = U\Sigma V^T = \sum_{k=1}^r s_r \mathbf{u}_r \mathbf{v}_r^T$. Then $\mathbf{u}_1$ and $\mathbf{v}_1$ are non-negative.

Corollary. When $r = 1$, a solution to the NMF problem (1) is given by $W = \mathbf{u}_1$ and $H = \mathbf{v}_1^T$.

It is well-known that the full SVD can be exactly computed by successive matrix deflation: that is, repeatedly computing and then subtracting off the leading singular vectors. Unfortunately, this approach does not generalize to NMF when $r > 2$ since $A - \mathbf{u}_1\mathbf{v}_1^T$ may contain negative entries. Nevertheless, approximate solutions to NMF based on the SVD can be computed and can serve as effective initializations in practice. Boutsidis [4] proposed [procedure 4], which they call nonnegative double singular value decomposition (NNDSVD).[1]

3.2 PRAM

Claim. In a PRAM setting, the work and depth required to compute a rank-r NNDSVD matches that of computing a truncated SVD. Methods for parallel truncated SVD are an active research area, but are currently believed to require $O(mnr)$ work and $O(\log^3 n)$ depth [18].

[1]Note that [4] use a sequential for loop; since this for loop is trivially parallelizable I indicate this in [procedure 4] with a parallel for loop.

Procedure 4 Parallel Nonnegative Double Singular Value Decomposition (NNDSVD)

Input: $A \in \mathbf{R}^{m \times n}$ holding non-negative data, and the desired rank r.

Output: Approximate non-negative factor matrices $W \in \mathbf{R}^{m \times r}$ and $H \in \mathbf{R}^{r \times n}$.

```
 1: Compute the leading $r$ singular vectors, $U, \mathbf{s}, V = \texttt{psvd}(A, r)$
        ▷ $U \in \mathbf{R}^{m \times r}$, $\mathbf{s} \in \mathbf{R}^r$, and $V \in \mathbf{R}^{n \times r}$. By theorem 3.1, $U_{:1}$ and $V_{:1}$ are nonnegative.
 2: $W_{:1} \leftarrow \sqrt{s_1} \cdot U_{:1}$
 3: $H_{:1} \leftarrow \sqrt{s_1} \cdot V_{:1}$
 4: for $k = 2 : r$ in parallel do
 5:     $\mathbf{u}_+ = \texttt{pos}(U_{:k}),\ \ \mathbf{u}_- = \texttt{neg}(U_{:k})$
 6:     $\mathbf{v}_+ = \texttt{pos}(V_{:k}),\ \ \mathbf{v}_- = \texttt{neg}(V_{:k})$
 7:     if $\|\mathbf{u}_+\|\|\mathbf{v}_+\| > \|\mathbf{u}_-\|\|\mathbf{v}_-\|$ then
 8:         $\mathbf{u} = \mathbf{u}_+/\|\mathbf{u}_+\|,\ \ \mathbf{v} = \mathbf{v}_+/\|\mathbf{v}_+\|,\ \ \sigma = \|\mathbf{u}_+\|\|\mathbf{v}_+\|$
 9:     else
10:         $\mathbf{u} = \mathbf{u}_-/\|\mathbf{u}_-\|,\ \ \mathbf{v} = \mathbf{v}_-/\|\mathbf{v}_-\|,\ \ \sigma = \|\mathbf{u}_-\|\|\mathbf{v}_-\|$
11:     end if
12:     $W_{:k} \leftarrow \sigma\sqrt{s_k} \cdot \mathbf{u}$
13:     $H_{k:} \leftarrow \sigma\sqrt{s_k} \cdot \mathbf{v}$
14: end parallel for
```

Proof. Lines 2-14 require $O(rm + rn)$ work and depth of $O(\log m + \log n)$ as detailed below. Note that the iterations of the for loop can be executed in parallel.

Lines 2-3, 8, 10, 12-13. These are simple scalar multiplications of vectors. They require either $O(m)$ or $O(n)$ work (for columns of U or V, respectively) and $O(1)$ depth.

Lines 5-6. These are also embarrassingly parallel operations, resulting in $O(m)$ or $O(n)$ work and $O(1)$ depth. Each element of U_{ik} and V_{ik} is found to be positive or negative in parallel, and is input to index i of one of $\mathbf{u}_+$, $\mathbf{u}_+$, $\mathbf{v}_+$, or $\mathbf{v}_+$.

Line 7. Computing the Euclidean norms of $\mathbf{u}_+$ and $\mathbf{u}_+$ requires $O(m)$ work and $O(\log m)$ depth. Likewise, these computations for $\mathbf{v}_+$ and $\mathbf{v}_+$ require $O(n)$ work and $O(\log n)$ depth.

□

3.3 Distributed Setting

It is easy to see that the SVD is the again the largest cost in terms of computation and communication costs. For completeness, a full algorithm for NNDSVD using distributed framework is outlined in appendix A. The only additional communication costs for NNDSVD are *allreduce* commands to compute $\mathbf{u}_+$, $\mathbf{u}_-$, $\mathbf{v}_+$, and $\mathbf{v}_-$. These require $O(\log p)$ rounds of communication.

I analyze two methods for distributed SVD in appendices B and C. The first is based on the calculation of the Gramian matrix, $A^T A$, and is one of the current methods implemented in Spark's Mllib [21]. The second method is based on a randomized algorithm [12]; although this algorithm is well-known I am unaware of any implementation or analysis of this algorithm in a distributed setting. Both methods have similar communication costs, but the number of operations required by the randomized method is linear in m and n, while the Gramian method is quadratic in one of these dimensions.

4 Conclusions and Future Work

In a distributed setting, every iteration of ANLS requires at least $O(\frac{mnr}{p})$ operations and $O(\log p)$ rounds of communication [13]. Because NMF is an NP-hard problem with few theoretical guarantees, it may take many iterations to converge to a stationary point. This report shows how to extend standard NMF initialization procedures to a distributed setting. These algorithms require similar resources to just a single ANLS iteration, and have been empirically shown to drastically reduce time to convergence. Indeed, under certain conditions, SPA and similar techniques provably recover the optimal solution without the need to resort to alternating minimization [2, 10].

In practice, SPA and similar techniques can be sensitive to noise. Gillis and Vavasis [10] showed that this sensitivity is scaled by the condition number of the data matrix $\kappa(A) = \frac{\sigma_{\max}}{\sigma_{\min}}$, such that ill-conditioned data is much more sensitive to noise. Recently, the same authors presented a preconditioning algorithm, which addresses this shortcoming and renders the algorithm much more robust to noise [11]. They suggest an active set method for solving a semi-definite program as an optimal solution, but also suggest a heuristic procedure based the SVD to precondition the data. It would thus be feasible and likely useful to combine SPA with some form of pre-conditioning in the distributed setting.

For datasets that are not nearly separable, SPA and similar techniques may not provide useful initializations. Another downside is that SPA requires a copy of the data to be made, which can be quite expensive for very large datasets. Thus, I also studied a more generic procedure, NNDSVD [4]. I showed that the communication and computational costs of this algorithm are dominated by computing the SVD. Using a common current algorithm for distributed SVD, this would require $O(m^2 n)$ or $O(mn^2)$ operations aggregated across machines, unlike SPA which requires $O(mnr)$ operations. However, I outline a randomized algorithm in appendix C which makes the total cost of NNDSVD similar to SPA.

Overall, these results suggest that popular initialization procedures for NMF on a single machine are scalable to large architectures.

Acknowledgements

This report was prepared as a final project for a class on distributed algorithms and optimization at Stanford University during the spring quarter of 2016. I am grateful to Reza Zadeh and Andreas Santucci for providing both formal instruction and informal conversations and feedback during this course.

References

[1] Mário César Ugulino Araújo, Teresa Cristina Bezerra Saldanha, Roberto Kawakami Harrop Galvão, Takashi Yoneyama, Henrique Caldas Chame, and Valeria Visani. The successive projections algorithm for variable selection in spectroscopic multicomponent analysis. *Chemometrics and Intelligent Laboratory Systems*, 57(2):65–73, 2001.

[2] Sanjeev Arora, Rong Ge, Ravindran Kannan, and Ankur Moitra. Computing a nonnegative matrix factorization–provably. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 145–162. ACM, 2012.

[3] Guy E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, March 1996.

[4] C. Boutsidis and E. Gallopoulos. {SVD} based initialization: A head start for nonnegative matrix factorization. *Pattern Recognition*, 41(4):1350 – 1362, 2008.

[5] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, April 1974.

[6] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[7] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication-optimal parallel and sequential qr and lu factorizations. *SIAM Journal on Scientific Computing*, 34(1):A206–A239, 2012.

[8] Nicolas Gillis. Successive nonnegative projection algorithm for robust nonnegative blind source separation. *SIAM Journal on Imaging Sciences*, 7(2):1420–1450, 2014.

[9] Nicolas Gillis. The why and how of nonnegative matrix factorization. *Regularization, Optimization, Kernels, and Support Vector Machines*, 12:257, 2014.

[10] Nicolas Gillis and Stephen A Vavasis. Fast and robust recursive algorithmsfor separable nonnegative matrix factorization. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36(4):698–714, 2014.

[11] Nicolas Gillis and Stephen A. Vavasis. Semidefinite programming based preconditioning for more robust near-separable nonnegative matrix factorization. *SIAM Journal on Optimization*, 25(1):677–698, 2015.

[12] Nathan Halko, Per-Gunnar Martinsson, Yoel Shkolnisky, and Mark Tygert. An algorithm for the principal component analysis of large data sets. *SIAM Journal on Scientific computing*, 33(5):2580–2594, 2011.

[13] Ramakrishnan Kannan, Grey Ballard, and Haesun Park. A high-performance parallel algorithm for nonnegative matrix factorization. *arXiv preprint arXiv:1509.09313*, 2015.

[14] Jingu Kim, Yunlong He, and Haesun Park. Algorithms for nonnegative matrix and tensor factorizations: a unified view based on block coordinate descent framework. *Journal of Global Optimization*, 58(2):285–319, 2014.

[15] Jingu Kim and Haesun Park. Fast nonnegative matrix factorization: An active-set-like method and comparisons. *SIAM Journal on Scientific Computing*, 33(6):3261–3281, 2011.

[16] Ruiqi Liao, Yifan Zhang, Jihong Guan, and Shuigeng Zhou. Cloudnmf: A mapreduce implementation of nonnegative matrix factorization for large-scale biological datasets. *Genomics, Proteomics & Bioinformatics*, 12(1):48 – 51, 2014.

[17] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *arXiv preprint arXiv:1505.06807*, 2015.

[18] Sanguthevar Rajasekaran and Mingjun Song. *High Performance Computing and Communications: Second International Conference, HPCC 2006, Munich, Germany, September 13-15, 2006. Proceedings*, chapter A Novel Scheme for the Parallel Computation of SVDs, pages 129–137. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[19] Stephen A Vavasis. On the complexity of nonnegative matrix factorization. *SIAM Journal on Optimization*, 20(3):1364–1377, 2009.

[20] Hyokun Yun, Hsiang-Fu Yu, Cho-Jui Hsieh, S. V. N. Vishwanathan, and Inderjit Dhillon. Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion, 2013.

[21] Reza Bosagh Zadeh, Xiangrui Meng, Aaron Staple, Burak Yavuz, Li Pu, Shivaram Venkataraman, Evan Sparks, Alexander Ulanov, and Matei Zaharia. Matrix computations and optimization in apache spark. *arXiv preprint arXiv:1509.02256*, 2015.

[22] Yong Zhuang, Wei-Sheng Chin, Yu-Chin Juan, and Chih-Jen Lin. A fast parallel sgd for matrix factorization in shared memory systems. In *Proceedings of the 7th ACM Conference on Recommender Systems*, RecSys '13, pages 249–256, New York, NY, USA, 2013. ACM.

Appendix A Distributed NNDSVD

Procedure 5 Distributed NNDSVD

Input: $U \in \mathbf{R}^{m\times r}$, $\mathbf{s} \in \mathbf{R}^r$, $V \in \mathbf{R}^{n\times r}$ holding the SVD of A. U and V are distributed as block row matrices across p_u and p_v machines, respectively, i.e. $U^{(\ell)} \in \mathbf{R}^{\frac{m}{p_u}\times r}$ and $V^{(\ell)} \in \mathbf{R}^{\frac{n}{p_v}\times r}$ are held on machine ℓ. The singular values s are broadcast/held on all machines.

Output: Approximate non-negative factor matrices $W \in \mathbf{R}^{m\times r}$ and $H \in \mathbf{R}^{r\times n}$, distributed across machines as U and V.

```
1: $W^{(\ell)}_{:1} \leftarrow \sqrt{s_1}\cdot U^{(\ell)}_{:1},\ \ H^{(\ell)}_{:1} \leftarrow \sqrt{s_1}\cdot V^{(\ell)}_{:1}$
2: for $k = 2 : r$ on each machine $\ell$ do
3:     $\mathbf{u}^{(\ell)}_{+k} = \texttt{pos}(U^{(\ell)}_{:k}),\ \ \mathbf{u}^{(\ell)}_{-k} = \texttt{neg}(U^{(\ell)}_{:k})$
4:     $\mathbf{v}^{(\ell)}_{+k} = \texttt{pos}(V^{(\ell)}_{:k}),\ \ \mathbf{v}^{(\ell)}_{-k} = \texttt{neg}(V^{(\ell)}_{:k})$
5:     Compute $\|\mathbf{u}^{(\ell)}_{+k}\|^2$, $\|\mathbf{v}^{(\ell)}_{+k}\|^2$, $\|\mathbf{u}^{(\ell)}_{-k}\|^2$, $\|\mathbf{v}^{(\ell)}_{-k}\|^2$
6: end for
7: allreduce: $\|\mathbf{u}_{+k}\| = \sqrt{\sum_\ell \|\mathbf{u}^{(\ell)}_{+k}\|^2},\ \ \|\mathbf{v}_{+k}\| = \sqrt{\sum_\ell \|\mathbf{v}^{(\ell)}_{+k}\|^2}$
8: allreduce: $\|\mathbf{u}_{-k}\| = \sqrt{\sum_\ell \|\mathbf{u}^{(\ell)}_{-k}\|^2},\ \ \|\mathbf{v}_{-k}\| = \sqrt{\sum_\ell \|\mathbf{v}^{(\ell)}_{-k}\|^2}$
9: for $k = 2 : r$ on each machine $\ell$ do
10:    if $\|\mathbf{u}_{+k}\|\|\mathbf{v}_{+k}\| > \|\mathbf{u}_{-k}\|\|\mathbf{v}_{-k}\|$ then
11:        $\mathbf{u}^{(\ell)}_k = \mathbf{u}^{(\ell)}_{+k}/\|\mathbf{u}_{+k}\|,\ \ \mathbf{v}^{(\ell)}_k = \mathbf{v}^{(\ell)}_{+k}/\|\mathbf{v}_{+k}\|,\ \ \sigma_k = \|\mathbf{u}_{+k}\|\|\mathbf{v}_{+k}\|$
12:    else
13:        $\mathbf{u}^{(\ell)}_k = \mathbf{u}^{(\ell)}_{-k}/\|\mathbf{u}_{-k}\|,\ \ \mathbf{v}^{(\ell)}_k = \mathbf{v}^{(\ell)}_{-k}/\|\mathbf{v}_{-k}\|,\ \ \sigma_k = \|\mathbf{u}_{-k}\|\|\mathbf{v}_{-k}\|$
14:    end if
15:    $W^{(\ell)}_{:k} \leftarrow \sigma_k\sqrt{s_k}\cdot \mathbf{u}^{(\ell)}_k$
16:    $H^{(\ell)}_{:k} \leftarrow \sigma_k\sqrt{s_k}\cdot \mathbf{v}^{(\ell)}_k$
17: end for
```

Appendix B Distributed Singular Value Decomposition by calculating Gramian Matrix

Let $A \in \mathbf{R}^{m\times n}$ with $n \gg m$ distributed as column-wise as in (2). One simple way to compute the SVD (assuming n is not too large) is to first compute compute the SVD of AA^T, since this provides the left singular vectors of A:

$$A = U\Sigma V^T \Rightarrow AA^T = (U\Sigma V^T)(U\Sigma V^T)^T = U\Sigma^2 U^T$$

To solve for the right singular vectors:

$$V^T = \Sigma^{-1}U^T A$$

This suggests [procedure 6], which I outline and briefly review below.

Memory Allocation. Distributed SVD requires $O\left(m^2 + \frac{nr}{p}\right)$ new memory on any single machine. The m^2 term comes from the computation of $A^{(\ell)}A^{(\ell)T}$ on line 1; the $\frac{nr}{p}$ term comes from the (distributed) allocation of V^T on line 6. Note that the allocation of U does not contribute since $mr \le m^2$.

Procedure 6 Distributed Truncated Singular Value Decomposition

```
Input: Matrix $A$ and desired rank $r$. $A$ is distributed as in (2), so that machine $\ell$ holds sub-matrix
    $A^{(\ell)} \in \mathbf{R}^{m \times \frac{n}{p}}$, where $p$ is the number of machines.
Output: $U \in \mathbf{R}^{m\times r}$, $\Sigma \in \mathbf{R}^{r\times r}$, $V^T \in \mathbf{R}^{r\times n}$. $V^T$ is distributed like $A$ across machines.
1: Each machine computes $A^{(\ell)}A^{(\ell)T}$
2: reduce: $AA^T = \sum_{\ell=1}^{p} A^{(\ell)}A^{(\ell)T}$
3: Compute $U, \Sigma^2, U^T = \texttt{svd}(AA^T, r)$ locally
4: $\Sigma = \texttt{sqrt}(\Sigma^2)$
5: broadcast: $\Sigma^{-1}U^T$
6: Each machine computes $V^{(\ell)T} = \Sigma^{-1}U^T A^{(\ell)}$
```

Communication Costs. The only communication costs are the all-to-one communication on line 2 with message size n^2, and the one-to-all communication on line 5 with message size nr. These require $O(\log p)$ rounds of communication when a recursive doubling communication pattern is used.

Computational Costs. Computing AA^T on line 1 requires $O(m^2 n)$ flops and dominates the order of the computational complexity. All other computations involve r and from $r \le \min(m,n)$. Computing the SVD of AA^T requires $O(m^2 r)$ flops. Computing Σ and Σ^{-1} requires $O(r)$ operations since these are diagonal matrices. Likewise, $\Sigma^{-1}U^T$ is a simple scaling of each column of U^T so takes $O(mr)$ flops. Finally, line 6 requires $O(rmn)$ flops in aggregate across machines.

Appendix C Randomized algorithm for distributed SVD

Procedure 7 Distributed Randomized SVD (based on [12])

```
Input: Desired rank $r$ and data $A \in \mathbf{R}^{m\times n}$ distributed as a block row matrix across $p$ machines
    so that machine $\ell$ holds $A^{(\ell)} \in \mathbf{R}^{\frac{m}{p}\times n}$. All machines initialize their random number generator
    with a common seed.
Output: Approximate factor matrices $W \in \mathbf{R}^{m\times r}$ and $H \in \mathbf{R}^{r\times n}$.
1: Each machine draws $\Omega \in \mathbf{R}^{n\times r}$ as a Gaussian random matrix
                                                  ▷ $\Omega$ is equal on all machines.
2: Compute $Y^{(\ell)} = A^{(\ell)}\Omega$
3: Compute distributed QR decomposition, $Y^{(\ell)} = Q^{(\ell)}R$.
                                  ▷ $Q$ is orthonormal and distributed like $Y$ [7].
4: reduce: Compute $B = \sum_{\ell=1}^{p} Q^{(\ell)T}A^{(\ell)}$
                     ▷ $B \in \mathbf{R}^{r\times n}$ is small and should fit on a single machine.
5: Locally compute SVD, $B = \hat{U}\Sigma V^T$
6: broadcast: $\hat{U}$
7: $U = Q^{(\ell)}\hat{U}$
```

Communication costs. Communication occurs on lines 3, 4, and 6. A communication-optimal algorithm for QR decomposition is given by [7] and involves $O(\log p)$ rounds of communication.

The reduce and broadcast operation on lines 4 and 6 also require $O(\log p)$ communication rounds, with $O(rn)$ and $O(r^2)$ message sizes, respectively.

Computation costs. The computation on any machine is upper bounded by $O(\frac{mnr}{p}+\frac{mr^2}{p}+r^3\log p+nr^2+nrp)$ operations. Critically, this algorithm has linear complexity in m and n, unlike the method covered in appendix B.

Line 2. $O\left(\frac{mnr}{p}\right)$ operations on each machine.

Line 3. The algorithm for distributed QR decomposition provided by [7] requires $O(\frac{mr^2}{p}+r^3\log p)$ computations.

Line 4. Computing $Q^{(\ell)T}A^{(\ell)}$ on each machine requires $O\left(\frac{mnr}{p}\right)$ operations. The reduce complexity involves summing $r\times n$ matrices over machines, resulting in $O\left(nrp\right)$ operations.

Line 5. Calling a standard SVD algorithm here requires $O(nr^2)$ operations.

Line 7. $O(\frac{mr^2}{p})$ operations on each machine.