Distributed Nonnegative Matrix Factorization for Web-Scale Dyadic Data Analysis on MapReduce

Chao Liu
Microsoft Research
One Microsoft Way
Redmond, WA 98052
chaoliu@microsoft.com

Hung-chih Yang
Microsoft Research
One Microsoft Way
Redmond, WA 98052
hunyang@microsoft.com

Jinliang Fan
Microsoft Research
One Microsoft Way
Redmond, WA 98052
jifan@microsoft.com

Li-Wei He
Microsoft Research
One Microsoft Way
Redmond, WA 98052
lhe@microsoft.com

Yi-Min Wang
Microsoft Research
One Microsoft Way
Redmond, WA 98052
ymwang@microsoft.com

ABSTRACT

The Web abounds with dyadic data that keeps increasing by every single second. Previous work has repeatedly shown the usefulness of extracting the interaction structure inside dyadic data [21, 9, 8]. A commonly used tool in extracting the underlying structure is the matrix factorization, whose fame was further boosted in the Netflix challenge [26]. When we were trying to replicate the same success on real-world Web dyadic data, we were seriously challenged by the scalability of available tools. We therefore in this paper report our efforts on scaling up the nonnegative matrix factorization (NMF) technique. We show that by carefully partitioning the data and arranging the computations to maximize data locality and parallelism, factorizing a tens of millions by hundreds of millions matrix with billions of nonzero cells can be accomplished within tens of hours. This result effectively assures practitioners of the scalability of NMF on Web-scale dyadic data.

Categories and Subject Descriptors

G.4 [**Mathematics of Computing**]: Mathematical Software – Parallel and vector implementations

General Terms

Algorithms, Experimentation, Performance

Keywords

distributed computing, nonnegative matrix factorization, MapReduce, dyadic data

1. INTRODUCTION

The Web abounds with dyadic data that keeps increasing by every single second. In general, dyadic data are the measurements on *dyads*, which are pairs of two elements coming from two sets [13, 21]. For instance, the most well-known dyadic data on the Web is the term-by-document representation of the Web corpus, where the measurement on a dyad (term, document) can be the count of how many times the term appears in the document, or some transformed value such as the TF-IDF score.

The list of dyadic data on the Web keeps growing, especially with the booming of social media. Because dyadic data contains rich information about the interactions between the two participating sets, its usefulness for practical applications has been repeatedly reported in previous work. For instance, in Web search, the (query, clicked URL) data is probably the most exploited source for query clustering [12], query suggestions [3] and improving search relevance [2]. In Internet monetization, the (bid keyword, ad) dyads with measurements on impressions and clicks constitute a valuable source for estimating click-through rate (CTR) and optimizing ad placement [9]. Finally, the booming social media generate a lot of useful dyadic data, *e.g.*, tags on flickr images, users and their joined communities, *etc.*. Previous work has shown that these data can be effectively leveraged for improved image retrieval [31] and community recommendation [8].

In general, Web dyadic data shares the following characteristics.

- **High-dimensional**: The two involved sets are usually very huge, *e.g.*, the set of distinct terms and all Web documents for the (term, document) dyadic data.
- **Sparse**: Measurements are sparse relative to the all possible dyads. For example, a term does not appear in all the documents and not all URLs are clicked for a given query.
- **Nonnegative**: Most measurements on Web dyadic data are based on event observations (*e.g.*, impressions and clicks), which are positive if observed and 0 otherwise.
- **Dynamic**: Web dyadic data keeps growing every single second, in terms of both the observed dyads and the dimensionality, *e.g.*, new users join the social network and tag things they are interested all the time.

While exploiting the rich information of Web dyadic data, we must face a grand challenge it poses at the same time:

WWW 2010, April 26–30, 2010, Raleigh, North Carolina, USA.
ACM 978-1-60558-799-8/10/04.

Will our analysis tools scale to the Web-scale, and keep pace with the explosion of Web data?

This paper gives an affirmative answer to this question regarding the nonnegative matrix factorization (NMF), which is a handy tool in analyzing dyadic data. For the convenience of rigorous analysis within a formal framework, dyadic data is usually modeled as matrices such that techniques proposed for one dyadic dataset will appeal to other forms of dyadic data. For instance, Latent Semantic Indexing (LSI) [15] and co-clustering [16] that are originally proposed for (term, document) dyadic data have found their ways into bioinformatics [20, 18].

Matrix factorization is a commonly used approach to understanding the latent structure of the observed matrix for various applications (*e.g.*, see [4, 40, 26]). There are many forms of matrix factorization, and [39] offers a unified view of several important factorizations including Singular Value Decomposition (SVD) and NMF. We in this paper choose to scale up NMF because it respects the nonnegativity that is inherent in most Web dyadic data. Previous work has shown that by respecting the nonnegativity, the factorization results will be easier to interpret while being comparable to, or better than, other techniques like SVD on effectiveness (*e.g.*, [43, 38, 28]).

We are not the first to scale up matrix factorization on noticing the data explosion. Many researchers have tried to scale up different factorizations including NMF (*e.g.*, [1, 44, 24, 25, 37]) through delicate algorithm designs. They generally assume that the data can be held in memory (or efficiently read from disk), and have reported on successes on factorizing tens of thousands by tens of thousands matrices with millions of nonzero values. While these studies aim at "large scale," the target of this study is the "Web scale," which can be informally interpreted as at least millions-by-millions matrices with billions of observations. To analyze Web-scale data, we can no longer assume the data can be held on a single powerful desktop. We therefore propose to scale up NMF through parallelism on distributed computer clusters with thousands of machines.

Because of the wide utility of NMF, we are not the first one trying to parallelize NMF either. Previous work has successfully parallelized NMF for multi-core machines through multi-threading [23, 36]. They propose to partition matrices in a way that takes advantage of the lightweight data sharing on multi-core machines. Unfortunately, these algorithms do not transfer to distributed clusters because data sharing is no longer lightweight in clusters (details in Section 3.1). In order to maximize the parallelism and data locality, we choose to partition matrices in the opposite direction to the previous work, and successfully parallelize NMF on thousands of machines in a distributed cluster. In contrast to previous work on parallel NMF, our algorithm is termed *distributed NMF*.

In summary, this paper makes the following contributions

- By observing a variety of Web dyadic data that conforms to different probabilistic distributions, we put forward a probabilistic NMF framework, which not only encompasses the two classic NMFs [28, 29] but also presents the Exponential NMF (ENMF) for modeling Web lifetime dyadic data (*e.g.*, the dwell time of users on browsed pages).
- A bigger contribution, as we deem, is the success of scaling up NMF to (potentially) arbitrarily large matrices on MapReduce [14] clusters. We show that by carefully partitioning the data and arranging the computations to maximize data locality and parallelism, factorizing tens of millions by hundreds of million matrices with billions of nonzero values can be accomplished within tens of hours. This is several-orders-of-magnitude larger than the largest factorization reported in literature, which essentially assures real-world applications of NMFs on scalability.
- Finally, a set of systematic experiments on both simulated and real-world data are performed to demonstrate the desired scalability and the usefulness of NMF.

While this paper focuses on scaling up NMF on MapReduce cluster, the same scaling-up scheme can be easily ported to MPI [34] clusters. Our preference of MapReduce clusters to MPI clusters merely comes from the fact that the MapReduce programming paradigm is often, if not always, implemented on the same cluster where Web data is collected. It is generally more convenient to run an algorithm where data is stored.

The rest of this paper is organized as follows. Section 2 lays out the probabilistic NMF framework and elucidates the Exponential NMF with details. Section 3 is devoted to describing how to scale up NMF on MapReduce clusters. We report on the experimental evaluation in Section 4, and discuss related work in Section 5. Finally, Section 6 concludes this study with brief discussion on future work.

2. PROBABILISTIC NMF

We use regular uppercase letters to denote matrices and boldface lowercase letters to denote vectors. For example, $A \in \mathbb{R}^{+^{m\times n}}$ is a m-by-n nonnegative real matrix, whose element (i, j) is denoted by $A_{i,j}$. We use $\mathbb{O}$ to denote the set of indices of nonzero values in A, *i.e.*, $\mathbb{O} = \{(i, j)|A_{i,j} > 0\}$, and similarly define $\mathbb{O}_i = \{j|A_{i,j} > 0\}$ and $\mathbb{O}^j = \{i|A_{i,j} > 0\}$.

Definition 1 (Nonnegative Matrix Factorization). *Given $A \in \mathbb{R}^{+^{m\times n}}$ and a positive integer $k <= min\{m, n\}$, find $W \in \mathbb{R}^{+^{m\times k}}$ and $H \in \mathbb{R}^{+^{k\times n}}$ such that a divergence function $\mathcal{D}(A||\widetilde{A})$ is minimized, where $\widetilde{A} = WH$ is the reconstructed matrix from the factorization.*

A probabilistic interpretation of NMF is to take $A_{i,j}$ as an observation from a distribution whose mean is parameterized by $\widetilde{A}_{i,j}$. In the following, we briefly review the two most popular NMFs in this framework (Section 2.1), and discuss the case using the Exponential distribution for Web lifetime data in Section 2.2.

2.1 Gaussian and Poisson NMF

When we take

$$A_{i,j} \sim \mathcal{G}aussian(\widetilde{A}_{i,j}, \sigma^2),$$

maximizing the likelihood of observing A w.r.t. W and H under the i.i.d. assumption

$$L(W, H|A) = \prod_{(i,j)} \frac{1}{\sqrt{2\pi}\sigma} exp\{-\frac{(A_{i,j} - \widetilde{A}_{i,j})^2}{2\sigma^2}\}$$

is equivalent to minimizing

$$\mathcal{D}(A||\widetilde{A}) = \sum_{(i,j)} (A_{i,j} - \widetilde{A}_{i,j})^2 = ||A - WH||^2,$$

Table 1: A Variety of NMFs based on Different Probabilistic Assumptions

(a) Gaussian NMF (GNMF)

$$A_{i,j} \sim \mathcal{G}aussian(\widetilde{A}_{i,j}, \sigma^2)$$

$$H \leftarrow H. * \frac{W^T A}{W^T W H} \quad (1)$$

$$W \leftarrow W. * \frac{A H^T}{W H H^T} \quad (2)$$

(b) Poisson NMF (PNMF)

$$A_{i,j} \sim \mathcal{P}oisson(\widetilde{A}_{i,j})$$

$$H \leftarrow H. * \frac{W^T \frac{A}{WH}}{W^T E^1} \quad (3)$$

$$W \leftarrow W. * \frac{\frac{A}{WH} H^T}{E H^T} \quad (4)$$

(c) Exponential NMF (ENMF)

$$A_{i,j} \sim \mathcal{E}xponential(\widetilde{A}_{i,j})$$

$$H \leftarrow H. * \frac{W^T [A./(WH)^2]}{W^T [1./WH]} \quad (5)$$

$$W \leftarrow W. * \frac{[A./(WH)^2] H^T}{[1./WH] H^T} \quad (6)$$

Note: $E \in \mathbb{R}^{+\,m \times n}, E_{i,j} = 1$

which is the Euclidean distance that leads to the most popular form of NMF [29]. We call it the Gaussian NMF or GNMF in short.

Similarly, when the Poisson distribution is used to model count data (*e.g.*, click counts), *i.e.*,

$$A_{i,j} \sim \mathcal{P}oisson(\widetilde{A}_{i,j}),$$

then maximizing the likelihood of observing A

$$L(W, H|A) = \prod_{(i,j)} exp\{-\widetilde{A}_{i,j}\} \frac{(\widetilde{A}_{i,j})^{A_{i,j}}}{A_{i,j}!}$$

becomes to minimizing

$$\mathcal{D}(A||\widetilde{A}) = \sum_{(i,j)} (\widetilde{A}_{i,j} - A_{i,j} log(\widetilde{A}_{i,j})),$$

which is the generalized KL-divergence as used in [28]. The resulting NMF is called Poisson NMF or PNMF in short.

Lee and Seung present a multiplicative algorithm that iteratively find the solution W and H for both GNMF and PNMF [28, 29]. The update formulae are reproduced in Table 1(a) and Table 1(b), respectively. Throughout this paper, we use ".∗" and "./" (or equivalently "−") to denote the element-wise matrix multiplication and division.

2.2 Exponential NMF

Besides count and Gaussian data, another important kind of measurements on dyads is the lifetime data. A good example of lifetime data in the Web context is the dwell time of a user on a webpage: the time until the user navigates away from the page. Proper modeling of the dwell time can help improving Web relevance and fighting Web spams [32].

Lifetime is usually modeled by the Weibull distribution

$$f(x|\gamma, \beta) = \frac{\gamma}{\beta} x^{\gamma - 1} e^{-x^\gamma / \beta}.$$

But because its mean $E(X) = \beta^{1/\gamma} \Gamma(1 + \frac{1}{\gamma})$ involves two parameters and hence cannot be parameterized by a single value $\widetilde{A}_{i,j}$, we instead consider the Exponential distribution, which is a special case of Weibull with $\gamma = 1$ and $E(X) = \beta$. The previous work on BrowseRank [32] also adopts the same simplification while achieving reasonable results.

Specifically, when $A_{i,j}$ is assumed to come from an Exponential distribution with $\beta = \widetilde{A}_{i,j}$, *i.e.*,

$$A_{i,j} \sim \mathcal{E}xponential(\widetilde{A}_{i,j}),$$

maximizing the likelihood of observing A w.r.t. W and H

$$L(W, H|A) = \prod_{(i,j)} \frac{1}{\widetilde{A}_{i,j}} exp\{-\frac{A_{i,j}}{\widetilde{A}_{i,j}}\}$$

becomes to minimizing

$$\mathcal{D}(A||\widetilde{A}) = \sum_{(i,j)} (log(\widetilde{A}_{i,j}) + \frac{A_{i,j}}{\widetilde{A}_{i,j}}).$$

We use gradient-descent algorithm to find the solution. Some matrix calculus reveals that the gradient of $\mathcal{D}(A||\widetilde{A})$ w.r.t. H is

$$\frac{\partial \mathcal{D}}{\partial H} = W^T [\frac{1}{WH} - \frac{A}{(WH).^2}],$$

which leads to the following update formula

$$H \leftarrow H + \mu. * W^T [\frac{A}{(WH)^2} - \frac{1}{WH}], \quad (7)$$

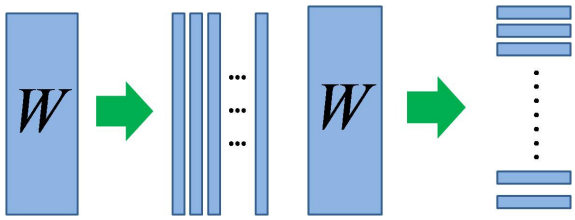
and $\mu > 0$ is the step-size. When μ takes $\frac{H}{W^T [1./WH]}$, we obtain the multiplicative updating rule for the Exponential NMF (ENMF in short) as

$$H \leftarrow H. * \frac{W^T [A./(WH)^2]}{W^T [1./WH]} \quad (8)$$

which, together with the formula for W, is summarized in Table 1(c) for comparison with GNMF and PNMF. The proof of convergence using Eqns. 5 and 6 is similar to that for GNMF in [29], and it is skipped here due to limited space.

3. SCALING-UP NMF ON MAPREDUCE

MapReduce [14] is a programming model and associated infrastructure that provide automatic and reliable parallelization once a computation task is expressed as a series of `Map` and `Reduce` operations. Specifically, the `Map` function reads a <key, value> pair, and emits one or many intermediate <key, value> pairs. The MapReduce infrastructure then groups together all values with the same intermediate key, and constructs a <key, ValueList> pair with ValueList containing all values associated with the same key. The `Reduce` function takes a <key, ValueList> pair and emits one or many new <key, value> pairs. As both `Map` and `Reduce` operate on <key, value> pairs, a series of mapper and reducers are usually streamlined for complicated tasks. With the MapReduce infrastructure, a user can fully focus on the logic of mappers and reducers, and lets the infrastructure deal with messy issues about distributed computing. Open-source implementations of MapReduce infrastructure are readily available such as the Apache Hadoop project.



(a) Parallel NMF (b) Distributed NMF

Figure 1: Different Partition Schemas for NMF

Even with the updating formulae laid out in Table 1, it is still a nontrivial task to distribute NMF on MapReduce clusters. In the first place, the giant matrices A, W and H need to be carefully partitioned so that each partition can be efficiently shuffled across machines when needed. In the second place, we must arrange the computation properly such that most computation can be carried out locally and in parallel.

In the following, we first discuss how to partition the matrices in Section 3.1, and then explain how to scale up GNMF on MapReduce in Section 3.2. We finally illustrate how to adapt the scaling-up scheme for GNMF to PNMF and ENMF in Section 3.3. Because the updating formulae are symmetric between W and H, we will limit our discussion to the update of H.

3.1 Matrix Partition Schemes

Because matrix A is sparse, it is naturally represented as $(i, j, A_{i,j})$ tuples that are spread across machines. For dense matrices W and H, how to partition them will significantly affect the final scalability.

Previous work on parallel NMF [23, 36] chooses to partition W and H along the long dimension as illustrated in Figure 1(a). This is a sensible choice because it conforms to the conventional thinking of matrix multiplication in the context of computing W^TA and W^TW (Eqn 1). By partitioning W and H along the long dimension and assuming A is in the shared memory, different threads can compute corresponding rows of W^TA on different cores. Similarly, as all columns of W are held in the shared memory, W^TW can be also calculated in parallel.

Unfortunately, partitioning W and H along the long dimension does not prevail for distributed NMF. First, each column of W can be simply too large to be manipulated in memory, and it is also too big to be passed around across machines. Second, partitioning along the long dimension unnecessarily limits the maximum parallelism to the factorization dimensionality k as there are only k columns in W. Finally, when partitioning W along the long dimension, W^TA and W^TW can no longer be computed in parallel because we can no longer assume A and all columns of W can be accessible with low overhead.

To address these limitations, we propose to partition W and H along the short dimension as illustrated in Figure 1(b). As will be seen in the rest of this section, this way of partitioning not only enables the parallel computation of both W^TA and W^TW but also maximizes the data locality to minimize the communication cost. To be precise, this partition renders the following view of W and H

$$W = \begin{pmatrix} \mathbf{w_1} \\ \mathbf{w_2} \\ \vdots \\ \mathbf{w_m} \end{pmatrix} \text{ and } H = \left(\mathbf{h_1} \mathbf{h_2} \dots \mathbf{h_n}\right), \tag{9}$$

where $\mathbf{w_i}$'s $(1 \le i \le m)$ and $\mathbf{h_j}$'s $(1 \le j \le n)$ are k-dimensional row and column vectors, respectively. Consequently, W and H are stored as sets of $< i, \mathbf{w}_i >$ and $< j, \mathbf{h}_j >$ key-value pairs.

3.2 GNMF on MapReduce

The updating formula for H (Eqn. 1) is composed of three components: $X = W^TA$, $Y = W^TWH$, and $H \leftarrow H. * X./Y$, where X and Y are auxiliary matrices for notation convenience. The three components are discussed in the following three subsections, and Figure 2 depicts the entire flowchart of updating H on MapReduce clusters.

3.2.1 Computing $X = W^TA$

Let $\mathbf{x}_j$ denote the jth column of X, then

$$\mathbf{x}_j = \sum_{i=1}^{m} A_{i,j}\mathbf{w}_i^T = \sum_{i \in \mathbb{O}^j} A_{i,j}\mathbf{w}_i^T.$$

It indicates that $\mathbf{x}_j$ is a linear combination of $\{\mathbf{w}_i^T\}$ over the nonzero cells on the jth column of A, which can be implemented by the the following two sets of MapReduce operations.

- `Map-I`: Map $< i, j, A_{i,j} >$ and $< i, \mathbf{w}_i >$ on i such that tuples with the same i are shuffled to the same machine in the form of $< i, \{\mathbf{w}_i, (j, A_{i,j})\ \forall j \in \mathbb{O}_i\} >$.
- `Reduce-I`: Take $< i, \{\mathbf{w}_i, (j, A_{i,j})\ \forall j \in \mathbb{O}_i\} >$ and emit $< j, A_{i,j}\mathbf{w}_i^T >$ for each $j \in \mathbb{O}_i$.
- `Map-II`: Map $< j, A_{i,j}\mathbf{w}_i^T >$ on j such that tuples with the same j are shuffled to the same machine in the form of $< j, \{A_{i,j}\mathbf{w}_i^T\}\ \forall i \in \mathbb{O}^j >$.
- `Reduce-II`: Take $< j, \{A_{i,j}\mathbf{w}_i^T\}\ \forall i \in \mathbb{O}^j >$, and emit $< j, \mathbf{x}_j >$, where $\mathbf{x}_j = \sum_{i \in \mathbb{O}^j} A_{i,j}\mathbf{w}_i^T$.

The output from `Reduce-II` is the matrix X. In fact, as one would have noticed, this scheme of using two MapReduce operations can be used to multiply any two giant matrices when one is sparse and the other narrow. Multiplying two giant and dense matrices is usually uncommon because the result will take too much storage.

3.2.2 Computing $Y = W^TWH$

It is wise to compute Y by first computing $C = W^TW$ and then $Y = CH$ because it maximizes the parallelism while requiring fewer multiplications than $Y = W^T(WH)$. In fact, it is unrealistic to compute WH because the result is a giant dense matrix that will easily overrun the storage.

With the partition of W along the short dimension, calculation of W^TW can be fully parallelized because

$$W^TW = \sum_{i=1}^{m} \mathbf{w}_i^T\mathbf{w}_i.$$

It means that each machine can first compute $\mathbf{w}_i^T\mathbf{w}_i$ (a small $k \times k$ matrix) for all the $\mathbf{w}_i$'s it hosts, and then send them over for a global summation, as implemented by

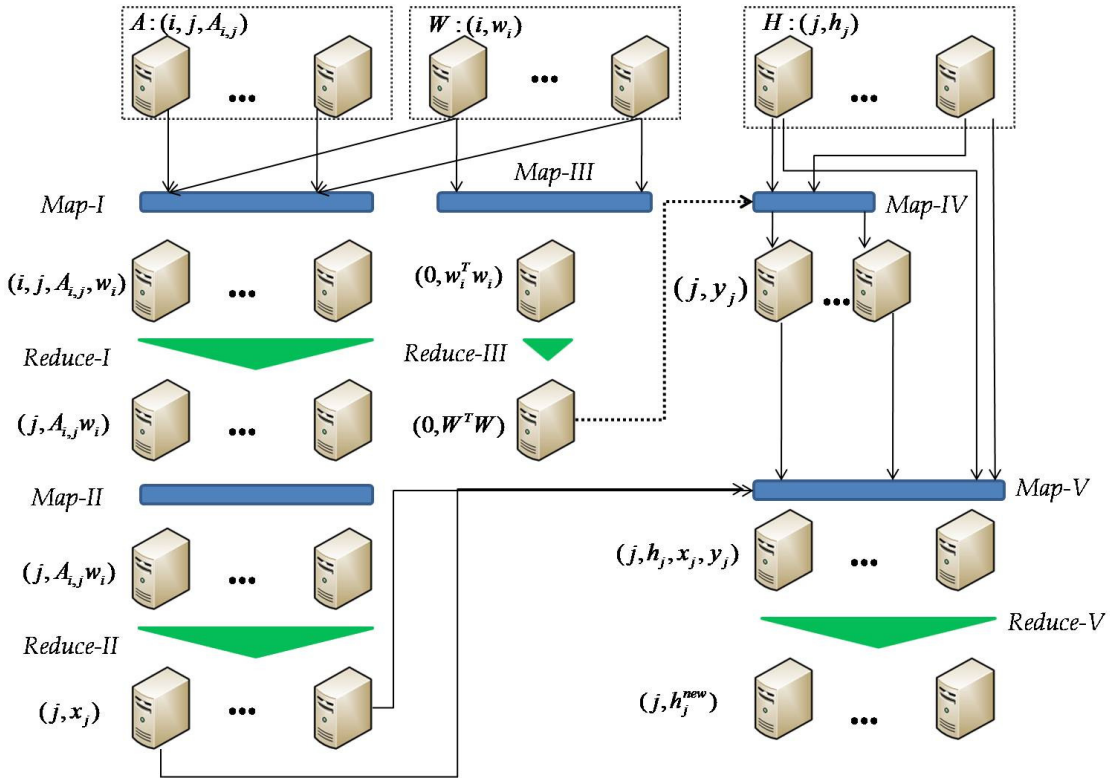


Figure 2: Computing $H \leftarrow H. * \frac{W^T A}{W^T W H}$ on MapReduce

- `Map-III`: Map $< i, \mathbf{w}_i >$ to $< 0, \mathbf{w}_i^T \mathbf{w}_i >$ where 0 is a dummy key value for data shuffling.
- `Reduce-III`: Take $< 0, \{\mathbf{w}_i^T \mathbf{w}_i\}_{i=1}^m >$, and emit $\sum_{i=1}^m \mathbf{w}_i^T \mathbf{w}_i$, which is the $W^T W$.

As summation is both associative and commutative, a combiner can be used to compute the partial sum of $\mathbf{w}_i^T \mathbf{w}_i$ on each machine and then passes the partial sum to the reducer to reduce network traffic.

Now that $C = W^T W$ is calculated, computing $Y = CH$ becomes as trivial as run through the following mapper with no data shuffled except copying the $k \times k$ matrix C to all the machines that host $\mathbf{h}_j$'s (as indicated by the dotted line in Figure 2).

- `Map-IV`: Map $< j, \mathbf{h}_j >$ to $< j, \mathbf{y}_i = C\mathbf{h}_j >$.

3.2.3 Updating $H = H. * X./Y$

Updating $H \leftarrow H. * X./Y$ is parallelized through the following MapReduce operation.

- `Map-V`: Map $< j, \mathbf{h}_j >$, $< j, \mathbf{x}_j >$ and $< j, \mathbf{y}_j >$ on j such that tuples with the same j are shuffled to the same machine in the form of $< j, \{\mathbf{h}_j, \mathbf{x}_j, \mathbf{y}_j\} >$.
- `Reduce-V`: Take $< j, \{\mathbf{h}_j, \mathbf{x}_j, \mathbf{y}_j\} >$ and emit $< j, \mathbf{h}_j^{new} >$, where $\mathbf{h}_j^{new} = \mathbf{h}_j. * \mathbf{x}_j./\mathbf{y}_j$.

This finishes the update of H, and updating W can be carried out in the same fashion. In the following, we will examine how the above scaling-up scheme carries over to PNMF and ENMF.

3.3 PNMF and ENMF on MapReduce

Since the updating formulae of PNMF and ENMF share the same structure as GNMF, the challenges in distributed PNMF and ENMF still lie on how to compute the nominator X and the denominator Y. Once X and Y are computed, the same `Map-V` and `Reduce-V` can be re-used for the final update.

3.3.1 Distributed PNMF

Computing the nominator $X = W^T[A./(WH)]$ for PNMF is similar to GNMF because once $\widehat{A} = A./[WH]$ is computed, it becomes $X = W^T \widehat{A}$. Furthermore, since $\widehat{A}_{i,j} = 0$ if $A_{i,j} = 0$, $\widehat{A}$ can be computed through two sets of MapReduce operations: the first gets $< i, j, A_{i,j}, \mathbf{h}_j >$ and the second obtains $< i, j, A_{i,j}/(\mathbf{w}_i \mathbf{h}_j) >$.

In computing $W^T \widehat{A}$, we no longer need two more MapReduce operations because we have already joined A with W in the last step. We can instead output $< j, [A_{i,j}/\mathbf{w}_i \mathbf{h}_j]\mathbf{w}_i >$ from the last step and streamline the output directly to `Map-II`. Not only does this save some time, but it also reduces the network traffic.

The denominator $Y = WE$ appears formidable because it seems to multiply two giant dense matrices. But since all elements of E is 1, all the columns of Y are the same, *i.e.*, $\mathbf{y}_j = \sum_{i=1}^m \mathbf{w}_i^T$, $\forall j \in [1, m]$. So we only need to calculate any column $\mathbf{y}_j$, possibly in parallel, and copy it to all the machines that host $\mathbf{h}_j$'s for the update of H. Fortunately, calculating $\mathbf{y}_j$ is simply a sum of all rows of W, which can be done is a similar way as calculating $W^T W$. In conclusion, distributed PNMF can be implemented on MapReduce.

3.3.2 Distributed ENMF

The computation of the nominator for ENMF is essentially the same as that for PNMF, and the same optimization to save one set of MapReduce operations applies as well. But unfortunately, its denominator presents a challenge because it explicitly asks for the giant dense matrix $1./(WH)$. To circumvent that, we can approximate it by only keeping the cells corresponding to nonzero values of A.

Approximation is common (and necessary for some cases) in scaling up algorithms to Web scale on distributed clusters. For example, the parallel LDA [41] approximates the Gibbs sampling, which is essentially sequential, by parallel sampling in a batch mode, and it still achieves very good results as shown in [8]. We will leave a full exploration of how to approximate ENMF on MapReduce and its applications to Web lifetime data to the future work.

4. EXPERIMENTAL STUDY

This section reports on the experimental evaluation based on GNMF because of its popularity in both literature and practice. In Section 4.1, we examine how the performance varies w.r.t. different factors using a dedicated sandbox cluster. Then we demonstrate the effectiveness of NMF on website recommendation and its scalability on real data in Section 4.2.

4.1 Sandbox Experiments

In order to collect detailed execution statistics and prevent the interference of other jobs on a shared computer cluster, we construct a dedicated Hadoop cluster that hosts up to 8 worker machines as our sandbox. These machines are not in the same configuration, but all have a Pentium 4 CPU with 1 or 2 cores, 1 to 2 GB memory, and a hard drive with more than 150 GB free space.

We wrote a random matrix generator, which generates a matrix $A \in \mathbb{R}^{+m \times n}$ with sparsity δ on given the parameters m, n and δ. By default, $m = 2^{17}$, $n = 2^{16}$, $\delta = 2^{-7}$ and $k = 2^3$. We put these parameters in the exponentials of 2 in order to see how the performance varies when a factor doubles while covering a large parameter spectrum. We use V to denote the number of worker machines in the cluster, and it varies from 1 to 8.

In the following, we first examine the computation breakdown among the three components in Section 4.1.1, then present how the performance varies w.r.t. δ, k and V in Section 4.1.2. Finally, we report on our experience on implementing NMF using a distributed matrix library in Section 4.1.3. This set of experiments is designed for a clear understanding of the algorithms but not for showcasing the scalability. All the reported time is for one iteration and it is in minute.

4.1.1 Computation Cost Breakdown

Table 2 lists the computation cost of each component in terms of both the amount of shuffled data (in MB) and the elapse time, when k is varied on two matrices with sparsity 2^{-7} and 2^{-10}.

The first thing to notice is that $X = W^T A$ dominates the computation cost in terms of both shuffled data and elapse time. The reason is that its computation involves two sets of MapReduce operations as discussed in Section 3.2.1. Since A is usually larger than W and H, we expect that the cost will significantly drop when A becomes sparser, and this is verified by the right half of Table 2. This is encouraging for practice because the sparsity of real-world data is usually much smaller than 2^{-7}.

Second, we see that $Y = W^T WH$ does not throttle the computation even though a single reducer is used to perform the sum. This is attributed to the high locality and parallelism in computing $W^T W$ as discussed in Section 3.2.2.

Finally, we note that analyzing the performance of a distributed job is a non-trivial task. Even with a dedicated sandbox cluster, there are still many factors out of our control, such as data allocation and network communication. But nevertheless Table 2 provides valuable insights into the cost breakdown, which will help guide further optimization. For example, knowing $X = W^T A$ is the dominant factor, we would endeavor to save the set of MapReduce operations for distributed PNMF and ENMF as discussed in Section 3.3.

4.1.2 Performance w.r.t. δ, k and V

Figure 3 plots how the performance varies w.r.t. the sparsity δ, the dimensionality k, and the number of worker machines in cluster V. Figure 3(a) plots the elapse time vs. the number of nonzero cells in A when the sparsity goes from 2^{-8} to 2^{-4}, which exhibits an expected linear relationship.

Figure 3(b), on the other hand, reveals the linearity between the elapse time and the dimensionality k. Specifically, it shows how the elapse time changes when k gradually quadruples from 8 to 512. The figure shows that the slope for $\delta = 2^{-10}$ is much smaller than that for $\delta = 2^{-7}$. This means that although the elapse time increases linearly w.r.t. k, the normalized slope is much smaller than 1, and the sparser the matrix A is, the smaller the slope will be. For example, when k goes from 8 to 512 (64X), the elapse time for $\delta = 2^{-7}$ is 33X while that for $\delta = 2^{-10}$ is merely 19X.

Finally, we examine how the number of worker machines affects the performance. Figure 3(c) plots the speedup when more and more machines are enabled in the cluster. The ideal speedup is along the diagonal, which upper-bounds the practical speedup because of the Amdahl's Law. On the matrix with $\delta = 2^{-7}$, the speedup is nearly 5 when 8 workers are enabled. The gap between the practical and the ideal speedup is due to many factors such as the overhead of shuffling data across machines and logistics on job balancing and check-pointing. Interestingly, we notice that when the matrix becomes sparser, the speedup actually drops. This suggests that the overhead actually outweighs real computation for these small datasets that can be processed with a single powerful machine; in other words, the cluster is not yet saturated.

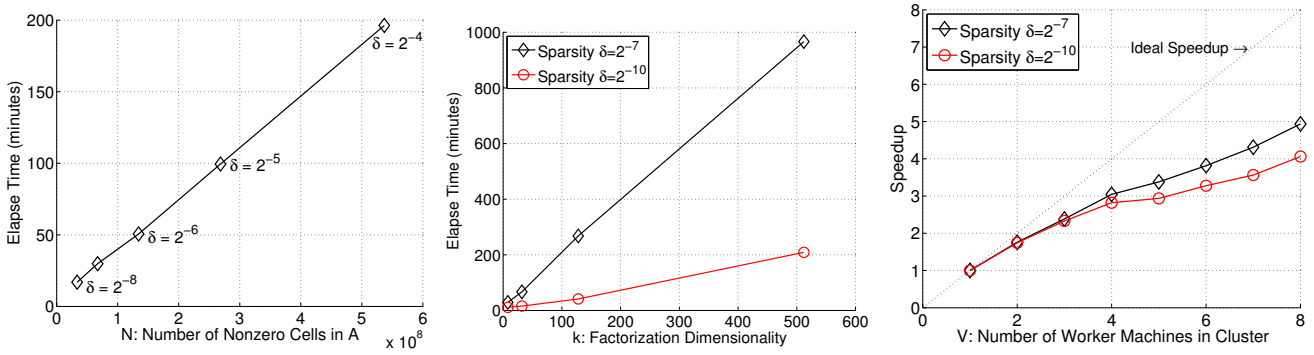
4.1.3 GNMF using a Distributed Matrix Library

We also implemented GNMF based on a distributed matrix library, called Hama, which aims at serving as a "distributed scientific package on Hadoop for massive matrix and graph data."[2] Since Hama implements a set of generic matrix operations, we could easily build the GNMF on top of it and successfully ran through some small examples. Unfortunately, the implementation failed when the matrix becomes $2^{15} \times 2^{14}$, which is a 16th of our default data set, with error messages reporting "exhausted the java heap space."

[2] `http://incubator.apache.org/hama/` on 10/25/2009

Component	Sparsity $\delta = 2^{-7}$						Sparsity $\delta = 2^{-10}$					
	k = 8		k = 32		k = 128		k = 8		k = 32		k = 128	
	Shuffle	T_{elapse}	Shuffle	T_{elapse}	Shuffle	T_{elapse}	Shuffle	T_{elapse}	Shuffle	T_{elapse}	Shuffle	T_{elapse}
$X = W^T A$	2206	11.64	6539	27.3	44799	121	327.3	3.25	921.1	5.24	5733.8	16.6
$Y = W^T WH$	5.16	1.25	17.24	1.44	66.37	2.69	5.13	1.14	17.5	1.24	66.9	2.77
$H = H. * X./Y$	16.97	1.02	52.3	1.04	193.6	1.15	16.29	0.97	53.3	1.02	197.5	1.18

Table 2: Computation Breakdowns with Different k and δ



(a) Elapse Time w.r.t. δ (b) Elapse Time w.r.t. k (c) Speed-up w.r.t. V

Figure 3: Performance w.r.t. δ, k and V

A preliminary investigation reveals the following two factors that limit the scalability of Hama-based GNMF. First, since Hama aims at a generic matrix library, its implementation divides the matrices into blocks and invokes a Mapper and a Reducer for each block pair. This results in a huge number of MapReduce operations and consequently a lot of data is unnecessarily shuffled around. Second, Hama is built on top of a random-access data system, called HBase, which can be very costly when data becomes huge. This observation reaffirms our belief that specific design is needed to fully exploit the algorithm-specific data access patterns for the optimal scalability.

4.2 NMF on Website Recommendation

An important feature shipped in the recent Internet Explorer release is the "Suggested Site," which recommends related sites according to the site the user is browsing, as shown in Figure 5. This feature can be implemented through NMF-based collaborative filtering on user browsing logs. In this section, we investigate the effectiveness of this approach (Section 4.2.1), with a particular focus on its scalability (Section 4.2.2), because a method that does not scale well will be of limited utility in practice.

4.2.1 Effectiveness

We sample from one-week's log data from a popular browser in the English (US) market. The log contains the browsed URLs for opted-in users. Each log entry contains an anonymized user id (UID) and a browsed URL, together with some meta-information such as the timestamp. As we are interested in recommending websites (instead of URLs), each URL is trimmed to the website level, *e.g.*, `http://www.cnn.com/SPECIALS/2009/health.care/` is truncated to `cnn.com`. We record a UID-by-website count matrix involving 17.8 million distinct UIDs and 7.24 million distinct websites, and

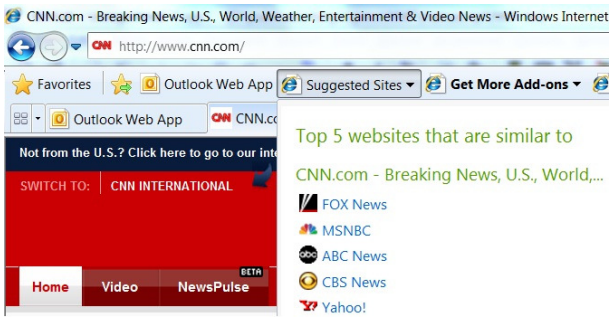
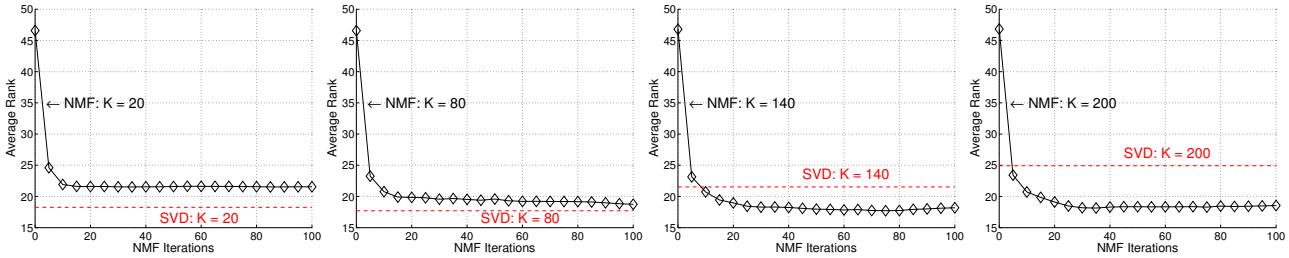


Figure 5: Snapshot of the IE8 Suggested Site

apply the TF-IDF transformation as in common practice. The idea is to first factorize the matrix, and then recommend websites based on the re-constructed matrix from the factorization.

For effectiveness test, we take the top-1000 UIDs that have the largest number of associated websites, and this gives us 346,040 distinct (UID, website) pairs. To measure the effectiveness, we randomly hold out a visited website for each UID, and mix it with another 99 un-visited sites. The 100 websites then constitute the test case for that UID. In the end, there are 1000 test cases, one for each UID. The goal is to check the rank of the holdout visited website among the 100 websites for each UID, and the overall effectiveness is measured by the average rank across the 1000 test cases, the smaller the better. The same metric is used in a recent study of community recommendation [8]. As expected, a random algorithm will end up with a score around 50.

We run GNMF on the matrix with the holdout website for each UID marked as 0, and Figure 4 plots the effectiveness w.r.t. the number of iterations for different k's. For compari-



(a) k = 20 (b) k = 80 (c) k = 140 (d) k = 200

Figure 4: Effectiveness of GNMF on Website Recommendation

(a) Execution w.r.t. Iterations

Total Iter.	T_{elapse}	T_{sum}	Read	Written	Shuffle
1	89	3252	791	642	338
2	180	5772	1090	856	576
3	233	8905	1392	1072	812
6	419	16045	2301	1723	1495

(b) Execution w.r.t. k

k	T_{elapse}	T_{sum}	Read	Written	Shuffle
10	89	3252	791	642	338
40	168	6438	1357	1179	576
70	254	13753	1925	1719	1495
100	373	26012	2497	2261	1518

(c) Execution w.r.t. Sampling Time Period

	Data				Execution				
	m	n	N	Size on Disk	T_{elapse}	T_{sum}	Read	Written	Shuffle
1 week	17.8M	7.24M	297M	38.8 GB	89	3252	791	642	338
2 weeks	26.4M	10.0M	540M	70.9 GB	94	3651	1495	1229	614
3 weeks	34.6M	12.0M	769M	101.1 GB	110	4498	2088	1711	839
4 weeks	42.7M	13.5M	996M	131.0 GB	129	6153	2712	2225	1124
5 weeks	51.0M	14.9M	1219M	160.5 GB	183	6551	3265	584	1334

Table 3: Scalability of Distributed GNMF on Real-world Data Sets

son, recommendations based on SVD are also plotted in each figure. As can be seen, both NMF and SVD are much more effective than random recommendation. Secondly, SVD and NMF are comparable for this application (depending on the choice of k) in terms of effectiveness. In the following, we examine how well GNMF scales to real-world data.

4.2.2 *Scalability*

Table 3 reports on the scalability of GNMF on real-world data sets, which are the same user browsing log but in a much larger scale. Five statistics are reported for each job: the elapse time T_{elapse}, the sum of the elapse time on all machines in the cluster T_{sum}, the IO "Read" and "Write" and the amount of data shuffled across machines denoted by "Shuffle." The time is measured in minutes and data is in GB. T_{sum} approximately characterizes the total computation load, and the amount of shuffled data reflects the communication cost. For example, the last row of Table 3(c) shows that by sampling from 5-weeks' log, we obtain a 51M-by-14.9M matrix with 1.219 billion nonzero values that takes 160GB on disk. By setting k to the default value 10, it takes 183 minutes to run one iteration of the distributed GNMF on top of the matrix. The overall time spent on all machines for this job is 6551 minutes and 1334 GB data is shuffled across machines for this job.

To be specific, Table 3(a) shows how the algorithm scales w.r.t. iterations. On the one hand, it shows that the elapse time increases linearly with the number of iterations, but on the other hand, we note that the average time per iteration becomes smaller when more iterations are executed in one job. For example, running 6 iterations in one job takes much shorter time than running 6 jobs one iteration each. Quantitatively, if we normalize the elapse time by that of 1 iteration, and regress the normalized time to the number of iterations, we see that the resulting slope is 0.72. The slope of 0.72 actually comes from the cross-iteration optimizations that are enabled by our distributed algorithm. For example, when certain columns of H are updated, they can be immediately used to compute corresponding part of HH^T and AH^T for the update of W. As indicated by the slope of 0.72, the cross-iteration optimization can significantly improve the performance when multiple iterations are executed in one job.

Table 3(b), on the other hand, shows the execution w.r.t. k, when k varies from 10 to 100 with incremental of 30. The table shows the same linear relationship as observed in Figure 3(b).

Finally, Table 3(c) lists how the algorithm scales with increasingly larger data sampled from increasingly longer time periods. As can be seen, the algorithm scales smoothly with the data, and it takes around 3 hours to run one iteration on a 51M-by-14.9M matrix containing 1.2 billion nonzero values. Considering the cross-iteration optimization and assuming that 20 iterations are enough as observed from Figure 4, the overall running time would be less than 60 hours. In other words, if we launch the job in Friday evening, the job would have already finished by the time we step back into the office on Monday morning. In order to further push the limit and make sure no hidden issues will explode the algorithm, we test the distributed GNMF on a 43.9M-by-

769M matrix with 4.38 billion nonzero values, and it takes less than 7 hours to finish one iteration. To the best of our knowledge, this is the largest factorization reported in literature, and it assures the scalability for practical usage.

5. RELATED WORK

Parallel operations on matrices (*e.g.*, multiplication, inversion, *etc.*) have been studied for decades because of the ubiquitousness of matrices in many areas, and mature libraries are readily available, *e.g.*, PSBLAS [19]. Because these general-purpose libraries are designed for multi-thread parallel computers, they cannot be easily ported to computer clusters that hold terabyte to petabyte data. Unlike on a parallel machine, data sharing and communication are no longer lightweight on distributed clusters. It was not until recently when data explodes with the booming of the Web that people began to look for scalable approaches to manipulating matrices that are too large to reside in memory, and the ongoing Hama project is a representative of this effort. While it is exciting to target a general solution, we believe that dedicated design could better exploit the data locality and parallelism that would otherwise be ignored in a general solution, as we have demonstrated for the NMF case in Section 4.1.3. On the other hand, experience gleaned from scaling up different algorithms would help build a general library. For example, the two MapReduce-based scheme in computing $W^T A$ can be generalized for multiplying two giant matrices when one is dense but narrow and the other is big but sparse.

Given the parallel library for matrices and the usefulness of NMF, it comes with no surprise to find parallel NMF [23, 36] in literature. As discussed in Section 3.1, in response to the characteristics of distributed clusters, we partition W and H in the opposite direction to that adopted in [23, 36]. In general, this study reminds us that one needs to carefully re-evaluate the parallelism scheme in order to port an algorithm to distributed clusters for Web-scale data.

Distributed NMF is by no means the only data analysis algorithm that is ported to MapReduce clusters. The generality of MapReduce computation model lends itself easily to many interesting applications (*e.g.*, see [7]). Das et al. distributed the probabilistic latent semantic indexing (which uses EM) on MapReduce, and showcased its effectiveness in building up personalized news service [11]. Nallapati et al. investigate how to perform variational EM for the application of learning text topics [33]. While the E-step can be easily distributed, the M-step is still centralized, which could potentially become a bottleneck. To over the bottleneck, Kowalczyk and Vlassis proposed the Newscast EM which decentralizes the M-step through gossip-based communication model [27]. Wolfe et al. also decentralized the M-step, and further showed that the decentralized M-step can further take advantages of the network topology for improved scalability [42]. Because the general applicability of EM algorithm, success on distributed EM effectively enables many real-world applications (*e.g.*, [11, 8]).

Recently, Chu et al. show that many popular machine learning algorithms, including locally weighted linear regression (LWLR), k-means, logistic regression (LR), naive Bayes (NB), SVM, ICA, PCA, gaussian discriminant analysis (GDA), Expectation maximization (EM), and backpropagation (NN) can all be implemented within the MapReduce paradigm [10]. Google, in response to the demand of analyzing huge amount of data in the Web era, has developed parallel SVM (PSVM) [6] and parallel LDA (PLDA) [41] on MapReduce clusters. Readers interested in a comprehensive review of large scale data analysis through parallelism are referred to their recent tutorial [5]. The distributed NMF as developed in this paper adds to the arsenal of scalable tools in analyzing Web-scale dyadic data.

There are numerous factorization techniques, and each of them has many extensions to include additional constraints like sparsity or orthogonality [39]. We here choose to scale up NMF simply because of its respect to the nonnegativity that is intrinsic to the Web data and the numerous successes of NMF as reported in literature [30, 43, 38, 4, 40]. Besides exploiting parallelism, many researchers have also tried to scale up factorization from algorithmic aspects (*e.g.*, [44, 35]). These algorithms can effectively factorize tens of thousands by tens of thousands matrices with millions of nonzero values. While these algorithms are not comparable to ours in terms of the data scales, their algorithmic design could be exploited to further boost the scalability on distributed clusters.

6. CONCLUSIONS

Confronted with huge amount of Web dyadic data and lured by the usefulness of NMF, we were determined to scale up NMF. In this paper, we showed that by carefully partitioning the data and arranging the computation, factorizing million-by-million matrices with billions of nonzero values becomes feasible on distributed MapReduce clusters.

There are many future work down the road. On the algorithmic side, the first priority is to regularize the factorization with additional constraints, such as sparsity [22] and orthogonality [17]. While these additional constraints will lead to different updating formulae as shown in [22, 17], the multiplicative update structure is not altered, which renders incorporating these constraints possible, albeit non-trivial. On the application side, we will explore other applications that are enabled by distributed NMF beyond website recommendations.

Acknowledgements

The authors would like to thank the following friends and colleagues for their help on algorithm design, literature survey, implementation and proofreading: Chris J.C. Burges, Chris Ding, Susan Domais, Paul Hsu, Emre Kıcıman, Tao Li, Xiaolong Li, Ethan Tu, Lin Xiao and Xiaoxin Yin. The authors also appreciate the anonymous reviewers who not only offered us detailed review comments but also insightful suggestions on future work.

7. REFERENCES

[1] D. Agarwal and S. Merugu. Predictive discrete latent factor models for large scale dyadic data. In *KDD*, 2007.

[2] E. Agichtein, E. Brill, and S. Dumais. Improving web search ranking by incorporating user behavior information. In *SIGIR*, pages 19–26, 2006.

[3] R. Baeza-Yates, C. Hurtado, and M. Mendoza. Query recommendation using query logs in search engines. *EDBT Workshops*, pages 588–596, 2004.

[4] M. W. Berry, M. Browne, A. N. Langville, P. V. Pauca, and R. J. Plemmons. Algorithms and

applications for approximate nonnegative matrix factorization. *Computational Statistics & Data Analysis*, 52(1):155–173, September 2007.

[5] E. Y. Chang, K. Zhu, and H. Bai. Parallel algorithms for mining large-scale datasets. In *CIKM*, 2009.

[6] E. Y. Chang, K. Zhu, H. Wang, H. Bai, J. Li, Z. Qiu, and H. Cui. PSVM: Parallelizing support vector machines on distributed computers. In *NIPS*, 2007.

[7] D. Chen. Efficient geometric algorithms on the EREW PRAM. *IEEE Trans. Parallel Distrib. Syst.*, 1995.

[8] W.-Y. Chen, J.-C. Chu, J. Luan, H. Bai, Y. Wang, and E. Y. Chang. Collaborative filtering for orkut communities: discovery of user latent behavior. In *WWW*, 2009.

[9] Y. Chen, D. Pavlov, and J. F. Canny. Large-scale behavioral targeting. In *KDD*, pages 209–218, 2009.

[10] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, 2006.

[11] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *WWW*, pages 271–280, 2007.

[12] W. L. C. David A. Kenny, Deborah A. Kashy. Query clustering using user logs. *ACM Trans. Inf. Syst.*, 20(1):59–81, 2002.

[13] W. L. C. David A. Kenny, Deborah A. Kashy. *Dyadic Data Analysis*. The Guilford Press, 2006.

[14] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[15] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41:391–407, 1990.

[16] I. S. Dhillon. Co-clustering documents and words using bipartite spectral graph partitioning. In *KDD*, pages 269–274, 2001.

[17] C. Ding, T. Li, W. Peng, and H. Park. Orthogonal nonnegative matrix tri-factorizations for clustering. In *KDD*, pages 126–135, 2006.

[18] Q. Dong, X. Wang, and L. Lin. Application of latent semantic analysis to protein remote homology detection. *Bioinformatics*, 22(3):285–290, 2005.

[19] S. Filippone and M. Colajanni. PSBLAS: a library for parallel linear algebra computation on sparse matrices. *ACM Trans. Math. Softw.*, 26(4):527–550, 2000.

[20] D. Hanisch, A. Zien, R. Zimmer, and T. Lengauer. Co-clustering of biological networks and gene expression data. *Bioinformatics*, 18(1):145–154, 2002.

[21] T. Hofmann, J. Puzicha, and M. I. Jordan. Learning from dyadic data. In *NIPS*, pages 466–472, 1999.

[22] P. O. Hoyer. Non-negative matrix factorization with sparseness constraints. *J. Mach. Learn. Res.*, 5, 2004.

[23] K. Kanjani. Parallel non negative matrix factorization for document clustering. Technical report, Texas A & M University, May 2007.

[24] D. Kim, S. Sra, and I. S. Dhillon. Fast newton-type methods for the least squares nonnegative matrix approximation problem. In *SDM*, 2007.

[25] D. Kim, S. Sra, and I. S. Dhillon. Fast projection based methods for the least squares nonnegative matrix approximation problem. *Statistical Analysis and Data Mining*, 1(1), 2008.

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

[27] W. Kowalczyk and N. Vlassis. Newscast em. In *In NIPS 17*, pages 713–720. MIT Press, 2005.

[28] D. D. Lee and H. S. Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(6755):788–791, 1999.

[29] D. D. Lee and H. S. Seung. Algorithms for non-negative matrix factorization. In *NIPS*, 2000.

[30] S. Li, X. Hou, H. Zhang, and Q. Cheng. Learning spatially localized, parts-based representation. *CVPR*, 1:207, 2001.

[31] X. Li, C. Snoek, and M. Worring. Learning tag relevance by neighbor voting for social image retrieval. In *Proc. of the 1st ACM international conference on Multimedia information retrieval (MIR '08)*, pages 180–187, 2008.

[32] Y. Liu, B. Gao, T.-Y. Liu, Y. Zhang, Z. Ma, S. He, and H. Li. BrowseRank: letting web users vote for page importance. In *SIGIR*, pages 451–458, 2008.

[33] R. Nallapati, W. Cohen, and J. Lafferty. Parallelized variational em for latent dirichlet allocation: An experimental evaluation of speed and scalability. *Proc. of the 7th International Conference on Data Mining Workshops (ICDMW'07)*, pages 349–354, 2007.

[34] P. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1997.

[35] J. Rennie and N. Srebro. Fast maximum margin matrix factorization for collaborative prediction. In *ICML*, pages 713–719, 2005.

[36] S. A. Robila and L. G. Maciak. A parallel unmixing algorithm for hyperspectral images. In *Intelligent Robots and Computer Vision XXIV*, 2006.

[37] M. N. Schmidt, O. Winther, and L. K. Hansen. Bayesian non-negative matrix factorization. In *Independent Component Analysis and Signal Separation*, pages 540–547, 2009.

[38] F. Shahnaz, M. W. Berry, V. P. Pauca, and R. J. Plemmons. Document clustering using nonnegative matrix factorization. *Inf. Process. Manage.*, 42(2), 2006.

[39] A. P. Singh and G. J. Gordon. A unified view of matrix factorization models. In *PKDD*, pages 358–373, 2008.

[40] S. Sra and I. S. Dhillon. Nonnegative matrix approximation: Algorithms and applications. Technical report, CS Department, University of Texas, June 2006.

[41] Y. Wang, H. Bai, M. Stanton, W.-Y. Chen, and E. Y. Chang. Plda: Parallel latent dirichlet allocation for large-scale applications. In *ICAAIM*, 2009.

[42] J. Wolfe, A. Haghighi, and D. Klein. Fully distributed em for very large datasets. In *ICML*, 2008.

[43] W. Xu, X. Liu, and Y. Gong. Document clustering based on non-negative matrix factorization. In *SIGIR*, pages 267–273, 2003.

[44] K. Yu, S. Zhu, J. Lafferty, and Y. Gong. Fast nonparametric matrix factorization for large-scale collaborative filtering. In *SIGIR*, pages 211–218, 2009.