Subspace Tracking for Latent Semantic Analysis

Radim Řehůřek

NLP lab, Masaryk University in Brno
radimrehurek@seznam.cz

Abstract. Modern applications of Latent Semantic Analysis (LSA) must deal with enormous (often practically infinite) data collections, calling for a single-pass matrix decomposition algorithm that operates in constant memory w.r.t. the collection size. This paper introduces a *streamed distributed algorithm for incremental SVD updates*. Apart from the theoretical derivation, we present experiments measuring numerical accuracy and runtime performance of the algorithm over several data collections, one of which is the whole of the English Wikipedia.

1 Introduction

The purpose of *Latent Semantic Analysis (LSA)* is to find hidden (*latent*) structure in a collection of texts represented in the Vector Space Model [10]. LSA was introduced in [4] and has since become a standard tool in the field of Natural Language Processing and Information Retrieval. At the heart of LSA lies the *Singular Value Decomposition* (SVD) algorithm, which makes LSA (sometimes also called Latent Semantic Indexing, or LSI) really just another member of the broad family of applications that make use of SVD's robust and mathematically well-founded approximation capabilities[1]. In this way, although we will discuss our results in the perspective and terminology of LSA and Natural Language Processing, our results are in fact applicable to a wide range of problems and domains across much of the field of Computer Science.

In this paper, we will be dealing with the practical issues of computing SVD efficiently in a distributed manner. For a more gentle introduction to SVD and LSA, its history, pros and cons and comparisons to other methods, see elsewhere [4,5,7].

1.1 SVD Characteristics

In terms of practical ways of computing SVD, there is an enormous volume of literature [12,3,5,14]. The algorithms are well-studied and enjoy favourable numerical properties and stability, even in the face of badly conditioned input. They differ in their focus on what role SVD performs—batch algorithms vs. online updates, optimizing FLOPS vs. number of passes, accuracy vs. speed etc.

[1] Another member of that family is the *discrete Karhunen–Loève Transform*, from Image Processing; or Signal Processing, where SVD is commonly used to separate signal from noise. SVD is also used in solving shift-invariant differential equations, in Geophysics, in Antenna Array Processing, ...

Table 1. Selected SVD algorithms for truncated (partial) factorization and their characteristics. In the table, "—" stands for *no/not found.* See text for details.

Algorithm	**Distributable**	**Incremental**		**Matrix structure**	**Subspace tracking**	**Implementations**
		docs	terms			
Krylov subspace methods (Lanczos)	yes	—	—	sparse	—	PROPACK, ARPACK, SVDPACK, MAHOUT,
Halko et al. [8]	yes	—	—	sparse	—	redsvd, our own
Gorrell & Webb [6]	—	—	—	sparse	—	LingPipe, our own
Zha & Simon [14]	—	yes	yes	dense	yes	—, our own
Levy & Lindenbaum [9]	—	yes	—	dense	yes	—, our own
Brand [2]	—	yes	yes	dense	—	—, our own
this paper	yes	yes	—	sparse	yes	our own, open-sourced

In Table 1, we enumerate several such interesting characteristics, and evaluate them for a selected set of known algorithms.

Distributable. Can the algorithm run in a distributed manner (without major modifications or further research)? Here, we only consider distribution of a very coarse type, where each computing node works autonomously. This type of parallelization is suitable for clusters of commodity computers connected via standard, high-latency networks, as opposed to specialized hardware or supercomputers.

Incremental Updates. Is the algorithm capable of updating its decomposition as new data arrives, without recomputing everything from scratch? The new data may take form of new observations (documents), or new features (variables). Note that this changes the shape of the A matrix. With LSA, we are more interested in whether we can efficiently add new documents, rather than new features. The reason is that vocabulary drift (adding new words; old words acquiring new meanings) is a relativaly slow phenomena in natural languages, while new documents appear all the time.

Matrix Structure. Does the algorithm make use of the structure of the input matrix? In particular, does the algorithm benefit from sparse input? Algorithms that can be expressed in terms of *Basic Linear Algebra Subprograms (BLAS)* routines over the input matrix are relatively easily adapted to any type of input structure.

Subspace Tracking. In online streaming environments, new observations come in asynchronously and the algorithm cannot in general store all the input documents in memory (not even out-of-core memory). The incoming observations must be immediate processed and then discarded[2].

Being online has implication on the decomposition itself, because we cannot even afford to keep the truncated right singular vectors V in memory. The size of $V_{n \times m}$ is $O(n)$, linear in the number of input documents, which is prohibitive. Therefore, only the U, S matrices are retained and the decomposition is used as a *predictive* (rather than descriptive) model. We call the

[2] This is in contrast to offline, *batch* algorithms, where the whole dataset is presented at once and the algorithm is allowed to go back and forth over the dataset many times.

$P_{m \times m} \equiv S^{-1}U^T$ matrix the *projection matrix*, and the projection process $V_x = P \cdot x$ is called *folding-in* in the context of LSA[3].

In such subspace tracking scenario, the input data stream can be assumed to be non-stationary. This allows us to introduce an explicit factor for "forgetting" old observations and adjusting the decomposition in favour of new data. This is realized by introducing a parameter $\gamma \in \langle 0.0, 1.0 \rangle$, called the *decay factor*, which dictates the rate of discounting the relevancy of old observations.

Available Implementations. While secondary from a theoretical point of view, we consider the availability of a real, executable reference implementation critical for a method's adoption. The application of LSA is relevant to a wider audience who simply do not possess the time or motivation to disentangle terse mathematical equations into functional programs. We also observe that most of the existing SVD implementations (with the notable exceptions of the Apache MAHOUT project and LingPipe) are written in somewhat opaque, FORTRANish style of coding, even when implemented in other languages.

2 Distributed LSA

In this section, we derive a novel algorithm for distributed online computing of LSA over a cluster of computers, in a single pass over the input matrix.

2.1 Overview

Distribution will be achieved by column-partitioning the input matrix A into several smaller submatrices, called *jobs*, $A^{m \times n} = \left[A_1^{m \times c_1}, A_2^{m \times c_2}, \cdots, A_j^{m \times c_j}\right]$, $\sum c_i = n$. Since columns of A correspond to documents, each job A_i amounts to processing a chunk of c_i input documents. The sizes of these chunks are chosen to fit available resources of the processing nodes: bigger chunks mean faster overall processing but on the other hand consume more memory.

Jobs are then distributed among the available cluster nodes, in no particular order, so that each node will be processing a different set of column-blocks from A. The nodes need not process the same number of jobs, nor process jobs at the same speed; the computations are completely asynchronous and independent. Once all jobs have been processed, the decompositions accumulated in each node will be merged into a single, final decomposition $P = (U, S)$ (see section 1.1 on subspace tracking for where V^T disappeared). As a reminder, U and S are respectively an orthonormal and a diagonal matrix such that $A = USV^T$, or equivalently and perhaps more naturally for avoiding mentioning the unused

[3] Note that folding-in is different to updating the decomposition: during folding-in, the U, S matrices stay intact and an existing model is only used to predict positions of documents in the latent space. In particular, $V^T_{m \times n} = S^{-1}U^T A = P_{m \times m} A_{m \times n}$, so that even though we cannot store the right singular vectors V^T during computations, they can still be recovered in a streaming fashion if needed, provided one has access to the projection matrix P and the original collection A.

V^T, such that $AA^T = US^2U^T$. The former factorization is called the Singular Value Decomposition, the latter is its related eigen decomposition.

What is needed are thus two algorithms:

1. **Base decomposition.** In main memory, find $P_i = (U_i^{m\times c_i}, S_i^{c_i\times c_i})$ eigen decomposition of a single job $A_i^{m\times c_i}$ such that $A_iA_i^T = U_iS_i^2U_i^T$.
2. **Merge decompositions.** Merge $P_i = (U_i, S_i)$, $P_j = (U_j, S_j)$ of two jobs A_i, A_j into a single decomposition $P = (U, S)$ such that $\left[A_i, A_j\right]\left[A_i, A_j\right]^T = US^2U^T$.

We would like to highlight the fact that the first algorithm will perform decomposition of a *sparse* input matrix, while the second algorithm will merge two *dense* decompositions into another dense decomposition. This is in contrast to incremental updates discussed in the literature [2,9,14], where the existing decomposition and the new documents are mashed together into a single matrix, losing any potential benefits of sparsity as well as severely limiting the possible size of a job. The explicit merge procedure also makes the distributed version of the algorithm straightforward, so that the computation can be split across a cluster of computers. Another volume of literature on efficient SVD concerns itself with Lanczos-based iterative solvers (see e.g. [11] for a large-scale batch approach). These are not applicable to the streaming scenario, as they require a large number of $O(k)$ passes over the input and are not incremental.

2.2 Solving the Base Case

There exist a multitude of partial sparse SVD solvers that work in-core. We view the particular implementation as "black-box" and note that the Lanczos-based implementations mentioned in Table 1 are particularly suitable for this task.

2.3 Merging Decompositions

No efficient algorithm (as far as we know) exists for merging two truncated eigen decompositions (or SVD decompositions) into one. We therefore propose our own, novel algorithm here, starting with its derivation and summing up the final version in the end.

The problem can be stated as follows. Given two truncated eigen decompositions $P_1 = (U_1^{m\times k_1}, S_1^{k_1\times k_1})$, $P_2 = (U_2^{m\times k_2}, S_2^{k_2\times k_2})$, which come from the (by now lost and unavailable) input matrices $A_1^{m\times c_1}$, $A_2^{m\times c_2}$, $k_1 \leq c_1$ and $k_2 \leq c_2$, find $P = (U, S)$ that is the eigen decomposition of $\left[A_1, A_2\right]$.

Our first approximation will be the direct naive

$$U, S^2 \xleftarrow{eigen} \left[U_1S_1, U_2S_2\right]\left[U_1S_1, U_2S_2\right]^T. \tag{1}$$

This is terribly inefficient, and forming the matrix product of size $m \times m$ on the right hand side is prohibitively expensive. Writing SVD$_k$ for truncated SVD that returns only the k greatest singular numbers and their associated singular vectors, we can equivalently write

$$U, S, V^T \xleftarrow{SVD_k} \left[\gamma U_1S_1, U_2S_2\right]. \tag{2}$$

This is more reasonable, with the added bonus of increased numerical accuracy over the related eigen decomposition. Note, however, that the computed right singular vectors V^T are not needed at all, which is a sign of further inefficiency. Also, the fact that U_1, U_2 are orthonormal is completely ignored. This leads us to break the algorithm into several steps:

Algorithm 1. Baseline merge

```
Input: Truncation factor $k$, decay factor $\gamma$, $P_1 = (U_1^{m\times k_1}, S_1^{k_1\times k_1})$, $P_2 = (U_2^{m\times k_2}, S_2^{k_2\times k_2})$
Output: $P = (U^{m\times k}, S^{k\times k})$
1 $Q, R \xleftarrow{QR} [\gamma U_1 S_1, U_2 S_2]$
2 $U_R, S, V_R^T \xleftarrow{SVD_k} R$
3 $U^{m\times k} \leftarrow Q^{m\times(k_1+k_2)} U_R^{(k_1+k_2)\times k}$
```

On line 1, an orthonormal subspace basis Q is found which spans both of the subspaces defined by columns of U_1 and U_2, $\text{span}(Q) = \text{span}(\left[U_1, U_2\right])$. Multiplications by S_1, S_2 and γ provide scaling for R only and do not affect Q in any way, as Q will always be column-orthonormal. Our algorithm of choice for constructing the new basis is QR factorization, because we can use its other product, the upper trapezoidal matrix R, to our advantage. Now we are almost ready to declare (Q, R) our target decomposition (U, S), except R is not diagonal. To diagonalize the small matrix R, we perform an SVD on it, on line 2. This gives us the singular values S we need as well as the rotation of Q necessary to represent the basis in this new subspace. The rotation is applied on line 3. Finally, both output matrices are truncated to the requested rank k. The costs are $O(m(k_1+k_2)^2)$, $O((k_1+k_2)^3)$ and $O(m(k_1+k_2)^2)$ for line 1, 2 and 3 respectively, for a combined total of $O(m(k_1+k_2)^2)$.

Although more elegant than the direct decomposition given by Equation 2, the baseline algorithm is only marginally more efficient than the direct SVD. This comes as no surprise, as the two algorithms are quite similar and SVD of rectangular matrices is often internally implemented by means of QR in exactly this way. Luckily, we can do better.

First, we observe that the QR decomposition makes no use of the fact that U_1 and U_2 are already orthogonal. Capitalizing on this will allow us to represent U as an update to the existing basis U_1, $U = \left[U_1, U'\right]$, dropping the complexity of the first step to $O(mk_2^2)$. Secondly, the application of rotation U_R to U can be rewritten as $UU_R = \left[U_1, U'\right] U_R = U_1 R_1 + U' R_2$, dropping the complexity of the last step to $O(mkk_1 + mkk_2)$. Plus, the algorithm can be made to work by modifying the existing matrices U_1, U_2 in place inside BLAS routines, which is a considerable practical improvement over Algorithm 1, which requires allocating additional $m(k_1+k_2)$ floats.

The first two lines construct the orthonormal basis U' for the component of U_2 that is orthogonal to U_1; $\text{span}(U') = \text{span}((I - U_1 U_1^T)U_2) = \text{span}(U_2 - U_1(U_1^T U_2))$.

As before, we use QR factorization because the upper trapezoidal matrix R will come in handy when determining the singular vectors S.

Algorithm 2. Optimized merge

```
Input: Truncation factor $k$, decay factor $\gamma$, $P_1 = (U_1^{m \times k_1}, S_1^{k_1 \times k_1})$, $P_2 = (U_2^{m \times k_2}, S_2^{k_2 \times k_2})$
Output: $P = (U^{m \times k}, S^{k \times k})$
1 $Z^{k_1 \times k_2} \leftarrow U_1^T U_2$
2 $U', R \xleftarrow{QR} U_2 - U_1 Z$
3 $U_R, S, V_R^T \xleftarrow{SVD_k} \begin{bmatrix} \gamma S_1 & ZS_2 \\ 0 & RS_2 \end{bmatrix}^{(k_1+k_2) \times (k_1+k_2)}$
4 $\begin{bmatrix} R_1^{k_1 \times k} \\ R_2^{k_2 \times k} \end{bmatrix} = U_R$
5 $U \leftarrow U_1 R_1 + U' R_2$
```

Line 3 is perhaps the least obvious, but follows from the requirement that the updated basis $\left[U, U'\right]$ must satisfy

$$\left[U_1 S_1, U_2 S_2\right] = \left[U_1, U'\right] X, \tag{3}$$

so that

$$X = \left[U_1, U'\right]^T \left[U_1 S_1, U_2 S_2\right] = \begin{bmatrix} U_1^T U_1 S_1 & U_1^T U_2 S_2 \\ U'^T U_1 & U'^T U_2 S_2 \end{bmatrix}. \tag{4}$$

Using the equalities $R = U'^T U_2$, $U'^T U_1 = 0$ and $U_1^T U_1 = I$ (all by construction) we obtain

$$X = \begin{bmatrix} S_1 & U_1^T U_2 S_2 \\ 0 & U'^T U_2 S_2 \end{bmatrix} = \begin{bmatrix} S_1 & ZS_2 \\ 0 & RS_2 \end{bmatrix}. \tag{5}$$

Line 4 is just a way of saying that on line 5, U_1 will be multiplied by the first k_1 rows of U_R, while U' will be multiplied by the remaining k_2 rows. Finally, line 5 seeks to avoid realizing the full $\left[U_1, U'\right]$ matrix in memory and is a direct application of the equality

$$\left[U_1, U'\right]^{m \times (k_1+k_2)} U_R^{(k_1+k_2) \times k} = U_1 R_1 + U' R_2. \tag{6}$$

As for complexity of this algorithm, it is again dominated by the matrix products and the dense QR factorization, but this time only of a matrix of size $m \times k_2$. The SVD of line 3 is a negligible $O(k_1 + k_2)^3$, and the final basis rotation comes up to $O(mk \max(k_1, k_2))$. Overall, with $k_1 \approx k_2 \approx k$, this is an $O(mk^2)$ algorithm.

In Section 3, we will compare the runtime speed of both these proposed merge algorithms on real corpora.

2.4 Effects of Truncation

While the equations above are exact when using matrices of full rank, it is not at all clear how to justify truncating all intermediate matrices to rank k in each update. What effect does this have on the merged decomposition? How do these effects stack up as we perform several updates in succession?

In [15], the authors did the hard work and identified the conditions under which operating with truncated matrices produces exact results. Moreover, they show by way of perturbation analysis that the results are stable (though no longer exact) even if the input matrix only approximately satisfies this condition. They show that matrices coming from natural language corpora do indeed possess the necessary structure and that in this case, a rank-k approximation of A can be expressed as a combination of rank-k approximations of its submatrices without a serious loss of precision.

3 Experiments

In this section, we will describe two sets of experiments. The first set concerns itself with numerical accuracy of the proposed algorithm, the second with its performance.

In all experiments, the decay factor γ is set to 1.0, that is, there is no discounting in favour of new observation. The number of requested factors is $k = 200$ for the small and medium corpus and $k = 400$ for the large Wikipedia corpus.

3.1 Algorithms

We will be comparing four implementations for partial Singular Value Decomposition:

SVDLIBC. A direct sparse SVD implementation due to Douglas Rohde[4]. SVDLIBC is based on the SVDPACK package by Michael Berry [1]. We use its LAS2 routine to retrieve only the k dominant singular triplets.

ZMS. implementation of the incremental one-pass algorithm from [13]. All the operations involved can be expressed in terms of Basic Linear Algebra Subroutines (BLAS). For this reason we use the NumPy library, which makes use of whatever LAPACK library is installed in the system, to take advantage of fast blocked routines. The right singular vectors and their updates are completely ignored so that our implementation of their algorithm also realizes subspace tracking.

DLSA. Our proposed method. We will be evaluating two different versions of the merging routine, Algorithms 1 and 2, calling them $DLSA_1$ and $DLSA_2$ in the tables. We will also observe effects of varying the job sizes c and the number of cluster nodes p (see Section 2). Again, NumPy is used for dense matrix operations. The base case decomposition is realized by an adapted LAS2 routine from SVDLIBC.

HEBB. Streamed stochastic Hebbian algorithm from [6] which loops over the input dataset, in $k * epochs$ passes, to converge at the singular triplets. The straightforward implementation suffered from serious convergency issues that we couldn't easily fix, so we only include it in our comparisons for the smallest dataset. This algorithm internally updates the singular triplets with explicit array loops (no BLAS).

3.2 Datasets

For the experiments, we will be using three datasets.

[4] `http://tedlab.mit.edu/~dr/SVDLIBC/`

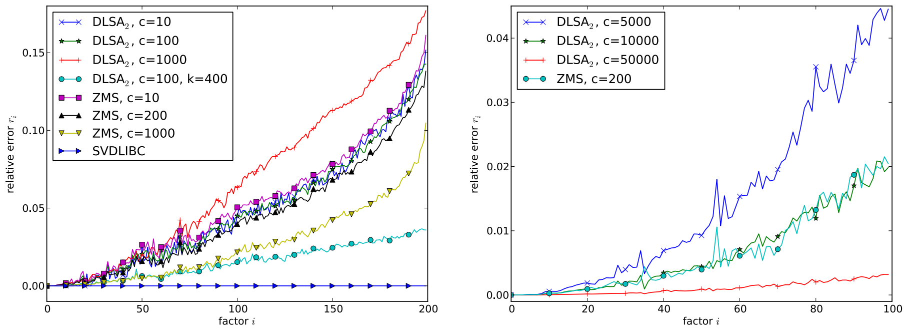

Fig. 1. Accuracy of singular values for various decomposition algorithms on the small corpus (left) and medium corpus (right)

Medium size corpus. A corpus of 61,293 mathematical articles collected from the digital libraries of NUMDAM, arXiv and DML-CZ. Together these comprise about 270 million corpus positions, with over 6 million unique word types (we parse out mathematical equations and use them as separate word types). After the standard procedure of pruning out word types that are too infrequent (hapax legomena, typos, OCR errors, etc.) or too frequent (stop words), we are left with 315,002 distinct features. The final matrix $A^{315,002 \times 61,293}$ has 33.8 million non-zero entries, with density less than 0.18%. This corpus was chosen so that it fits into core memory of a single computer and its decomposition can therefore be computed directly. This will allow us to establish the "ground-truth" decomposition and set an upper bound on achievable accuracy and speed.

Small corpus. A subset of 3,494 documents from the medium size corpus. It contains 39,022 features and the sparse $A^{39,022 \times 3,494}$ matrix has 1,446,235 non-zero entries, so that it is about 23 times smaller than the medium size corpus.

Large corpus. The last corpus is the English Wikipedia[5]. This corpus contains 3.2 million documents covering almost 8 million distinct word types. We clip the vocabulary size to the 100,000 most frequent word types, after discarding all words that appear in more than 10% of the documents ("stop-list"). This leaves us with a sparse term-documents matrix with 0.5G non-zero entries, or 14 times the size of the medium corpus.

3.3 Accuracy

Figure 1 plots the relative accuracy of singular values found by DLSA, ZMS, SVDLIBC and HEBB algorithms compared to known, "ground-truth" values S_G. We measure accuracy of the computed singular values $\overline{S}$ as $r_i = |\overline{s_i} - s_{Gi}|/s_{Gi}$, for $i = 1, \dots, k$. The ground-truth singular values S_G are computed directly with

[5] The latest static dump as downloaded from `http://download.wikimedia.org/enwiki/latest`, June 2010.

Table 2. Decomposition accuracy on the small corpus, measured by RMSE of document similarities based on ground truth vs. given algorithm

Algorithm	Job size	RMSE
SVDLIBC	3,494	0.0
ZMS	10	0.0204
ZMS	200	0.0193
ZMS	1,000	0.0162

Algorithm	Job size	RMSE
DLSA$_2$	10	0.0204
DLSA$_2$	100	0.0199
DLSA$_2$	1,000	0.0163
DLSA$_2$	100, $k = 400$	0.0094

LAPACK's DGESVD routine for the small corpus and with SVDLIBC's LAS2 routine for the medium corpus.

We observe that the largest singular values are practically always exact, and accuracy quickly degrades towards the end of the returned spectrum. This leads us to the following refinement: When requesting x factors, compute the truncated updates for $k > x$, such as $k = 2x$, and discard the extra $x - k$ factors only when the final projection is actually needed. This approach is marked as "DLSA$_2$, c=100, k=400" in Figure 3.3 and then used as default in the larger experiments on the medium corpus. The error is then below 5%, which is comparable to the ZMS algorithm (while DLSA is at least an order of magnitude faster, even without any parallelization).

Even with this refinement, the error is not negligible, so we are naturally interested in how it translates into error of the whole LSA application. This way of testing has the desirable effect that errors in decomposition which do not manifest themselves in the subsequent application do not affect our evaluation, while decomposition errors that carry over to the application are still correctly detected.

To this end, we conducted another set of accuracy experiments. In Latent Semantic Analysis, the most common application is measuring cosine similarity between documents represented in the new, "latent semantic" space. We will compute inter-document similarity of the entire input corpus, forming an $n \times n$ matrix C, where each entry $c_{i,j} = \text{cossim}(\overline{doc_i}, \overline{doc_j})$. We do the same thing for the corpus represented by the ground truth decomposition, obtaining another $n \times n$ matrix. Difference between these two $n \times n$ matrices (measured by Root Mean Square Error, or RMSE) then gives us a practical estimate of the error introduced by the given decomposition[6]. Note that in addition to testing the magnitude of singular values, this also tests accuracy of the singular vectors at the same time. In Table 2, we can observe that the error of DLSA$_2$ is around 2%, which is usually acceptable for assessment of document similarity and for document ranking.

3.4 Performance

Performance was measured as wall-clock time on a cluster of four dual-core 2GHz Intel Xeons, each with 4GB of RAM, which share the same Ethernet segment

[6] This is a round-about sort of test—to see how accurate a decomposition is, we use it to solve a superordinate task (similarity of documents), then compare results of this superordinate task against a known result.

Table 3. Performance of selected partial decomposition algorithms on the small corpus, $A^{39,022 \times 3,494}$. Times are in seconds, on the serial setup.

(a) Serial algorithms

Algorithm	Job size	Time taken
HEBB	N/A	> 1h
SVDLIBC	3,494	16
ZMS	10	346
ZMS	100	165
ZMS	200	150
ZMS	500	166
ZMS	1,000	194
ZMS	2,000	out of memory

(b) DLSA variants

Job size	DLSA$_2$	DLSA$_1$
10	190	2,406
100	122	350
1,000	38	66
3,494	21	21

Table 4. Performance of selected partial decomposition algorithms on the medium corpus, $A^{315,002 \times 61,293}$. Times are in minutes.

(a) Serial algorithms, serial setup.

Algorithm	Job size c	Time taken [min]
SVDLIBC	61,293	9.2
ZMS	200	360.1

(b) distributed DLSA$_2$

	No. of nodes p			
Job size c	serial	1	2	4
1,000	55.5	283.9	176.2	114.4
4,000	21.8	94.5	49.6	38.2
16,000	15.5	29.5	32.0	23.0

and communicate via TCP/IP. The machines were not dedicated but their load was reasonably low during the course of our experiments. To make sure, we ran each experiment three times and report the best achieved time. These machines did not have any optimized BLAS library installed, so we also ran the same experiments on a "cluster" of one node, a dual-core 2.53GHz MacBook Pro with 4GB RAM and *vecLib*, a fast BLAS/LAPACK library provided by the vendor. This HW setup is marked as "serial" in the result tables, to differentiate it from the otherwise equivalent 1-node setup coming from our BLAS-less four-node cluster.

Table 3 summarizes performance results for the small corpus. For ZMS, the update time is proportional to *number of updates* $\cdot$ *cost of update* $\approx \lceil \frac{n}{c} \rceil \cdot m(k+c)^2$, so that the minimum (fastest execution) is attained by setting job size $c = k$. Overall, we can see that the direct in-core SVDLIBC decomposition is the fastest. The HEBB implementation was forcefully terminated after one hour, with some estimated eight hours left to complete. The speed of DLSA$_2$ approaches the speed of SVDLIBC as the job size increases; in fact, once the job size reaches the size of the whole corpus, it becomes equivalent to SVDLIBC. However, unlike SVDLIBC, DLSA can proceed in document chunks smaller than the whole corpus, so that corpora that do not fit in RAM can be processed, and so that different document chunks can be processed on different nodes in parallel.

For experiments on the medium corpus, we only included algorithms that ran reasonably fast during our accuracy assessment on the small corpus, that is, only ZMS, DLSA in its fastest variant *DLSA*$_2$ and SVDLIBC.

Performance of running the computation in distributed mode is summarized in Table 4(b). As expected, performance scales nearly linearly, the only overhead being sending and receiving jobs over the network and the final merges at the very end of the distributed algorithm. This overhead is only significant when dealing with a slow network and/or with extremely few updates. The faster the connecting network and the greater the number of updates, $\lceil \frac{n}{c} \rceil \gg p$, the more linear this algorithm becomes.

Another interesting observation comes from comparing DLSA$_2$ speed in the "serial setup" (fast BLAS) vs. "cluster setup" (no fast BLAS) in Table 4(b). The serial setup is about five times faster than the corresponding cluster version with $p = 1$, so that even spreading the computation over four BLAS-less nodes results in *slower* execution than on the single "serial" node. As installing a fast, threaded BLAS library[7] is certainly cheaper than buying five times as many computers to get comparable performance, we strongly recommend doing the former (or both).

English Wikipedia results. Since the Wikipedia corpus is too large to fit in RAM, we only ran the streamed ZMS and $DLSA_2$ algorithms, asking for 400 factors in each case. On the "serial setup" described above, ZMS took 109 hours, $DLSA_2$ 8.5 hours. We'd like to stress that these figures are achieved using a single commodity laptop, with a one-pass online algorithm on a corpus of 3.2 million documents, without any subsampling. In distributed mode with six nodes, the time of $DLSA_2$ drops to 2 hours 23 minutes[8].

4 Conclusion

In this paper we developed and presented a novel distributed single-pass eigen decomposition method, which runs in constant memory w.r.t. the number of observations. This method is suited for processing extremely large (possibly infinite) sparse matrices that arrive as a stream of observations, where each observation must be immediately processed and then discarded. The method is embarrassingly parallel, so we also evaluated its distributed version.

We applied this algorithm to the application of Latent Semantic Analysis, where observations correspond to documents and the number of observed variables (word types) is in the hundreds of thousands. The implementation achieves excellent runtime performance in serial mode (i.e., running on a single computer) and scales linearly with the size of the computer cluster [16]. The implementation is released as open source, under the OSI-approved LGPL licence, and can be downloaded from `http://nlp.fi.muni.cz/projekty/gensim/`.

Acknowledgements

This study was partially supported by the LC536 grant of MŠMT ČR.

[7] Options include vendor specific libraries (e.g. Intel's MKL, Apple's vecLib, Sun's Sunperf), ATLAS, GotoBLAS, ...

[8] For details and reproducibility instructions, see the *gensim* package documentation.

References

1. Berry, M.: Large-scale sparse singular value computations. The International Journal of Supercomputer Applications 6(1), 13–49 (1992)
2. Brand, M.: Fast low-rank modifications of the thin singular value decomposition. Linear Algebra and its Applications 415(1), 20–30 (2006)
3. Comon, P., Golub, G.: Tracking a few extreme singular values and vectors in signal processing. Proceedings of the IEEE 78(8), 1327–1343 (1990)
4. Deerwester, S., Dumais, S., Furnas, G., Landauer, T., Harshman, R.: Indexing by Latent Semantic Analysis. Journal of the American Society for Information Science 41(6), 391–407 (1990)
5. Golub, G., Van Loan, C.: Matrix computations. Johns Hopkins University Press, Baltimore (1996)
6. Gorrell, G., Webb, B.: Generalized hebbian algorithm for incremental Latent Semantic Analysis. In: Ninth European Conference on Speech Communication and Technology (2005)
7. Griffiths, T., Steyvers, M., Tenenbaum, J.: Topics in semantic representation. Psychological Review 114(2), 211–244 (2007)
8. Halko, N., Martinsson, P., Tropp, J.: Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions. arXiv 909 (2009)
9. Levy, A., Lindenbaum, M.: Sequential Karhunen–Loeve basis extraction and its application to images. IEEE Transactions on Image Processing 9(8), 1371 (2000)
10. Salton, G.: Automatic text processing: the transformation, analysis, and retrieval of information by computer. Addison-Wesley Longman Publishing Co., Inc., Boston (1989)
11. Vigna, S.: Distributed, large-scale latent semantic analysis by index interpolation. In: Proceedings of the 3rd International Conference on Scalable Information Systems, pp. 1–10. ICST (2008)
12. William, H., Teukolsky, S., Vetterling, W.: Numerical Recipes in C: The art of scientific computing. Cambridge University Press, New York (1988)
13. Zha, H., Marques, O., Simon, H.: Large-scale SVD and subspace-based methods for Information Retrieval. Solving Irregularly Structured Problems in Parallel, 29–42 (1998)
14. Zha, H., Simon, H.: On updating problems in Latent Semantic Indexing. SIAM Journal on Scientific Computing 21, 782 (1999)
15. Zha, H., Zhang, Z.: Matrices with low-rank-plus-shift structure: Partial SVD and Latent Semantic Indexing. SIAM Journal on Matrix Analysis and Applications 21, 522 (2000)
16. Řehůřek, R., Sojka, P.: Software Framework for Topic Modelling with Large Corpora. In: Proceedings of LREC 2010 Workshop on New Challenges for NLP Frameworks, pp. 45–50 (2010)