A Semi-Discrete Matrix Decomposition for Latent Semantic Indexing in Information Retrieval

Tamara G. Kolda* and Dianne P. O'Leary†

December 5, 1996

Abstract

The vast amount of textual information available today is useless unless it can be effectively and efficiently searched. In information retrieval, we wish to match queries with relevant documents. Documents can be represented by the terms that appear within them, but literal matching of terms does not necessarily retrieve all relevant documents. Latent Semantic Indexing represents documents by approximations and tends to cluster documents on similar topics even if their term profiles are somewhat different. This approximate representation is usually accomplished using a low-rank singular value decomposition (SVD) approximation. In this paper, we use an alternate decomposition, the semi-discrete decomposition (SDD). In our tests, for equal query times, the SDD does as well as the SVD and uses less than one-tenth the storage. Additionally, we show how to update the SDD for a dynamically changing document collection.

1 Introduction

The vast amount of textual information available today is useless unless it can be effectively and efficiently searched. In *information retrieval*, we wish to match user information requests, or *queries*, with relevant information items, or *documents*. Examples of information retrieval systems include electronic library catalogs, the `grep` string-matching tool in Unix, and search engines such as Alta Vista for the World Wide Web.

Oftentimes, users are searching for documents about a particular concept that may not be accurately described by the list of keywords. For example, a search on a term such as "Mark Twain" is unlikely to find all documents about

*Applied Mathematics Program, University of Maryland, College Park, MD 20742. The work of this author was supported by the NSA, NPSC, and CCS. (`kolda@math.umd.edu`)

†Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742. The work of this author was supported by the National Science Foundation under Grant CCR-95-03126 (`oleary@cs.umd.edu`)

"Samuel Clemens." *We* might know that these are the same people, but the information retrieval systems have no way of knowing. *Latent semantic indexing* (LSI) is an approach to retrieval that attempts to represent such information and thus find *latent* relationships in the information stored in its database.

In the *vector space model for information retrieval*, discussed in Section 2, the database of documents is represented by an $m \times n$ *term-document matrix* where m is the number of terms and n is the number of documents. This matrix is typically less than 1% dense. Queries are represented as m-vectors, and a matrix-vector product produces an n-vector of scores that is used to rank the documents in relevance.

LSI is based on the vector space model, but the $m \times n$ term-document matrix is replaced by a low-rank approximation generated by the singular value decomposition (SVD). The SVD approximation is the sum of k rank-1 outer products of m-vectors $\mathbf{u}_i$ with n-vectors $\mathbf{v}_i$, weighted by scalars σ_i:

$$\sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^{\mathrm{T}}.$$

This approximation to the term-document matrix is optimal in the sense of minimizing the distance between that matrix and all rank-k matrices. LSI has performed well in both large and small tests; see, for example, Dumais [5, 6]. LSI is described in Section 3.

Thus far, only the singular value decomposition and its relatives, the ULV and URV decompositions [3], have been used in LSI. We propose using a very different decomposition, originally developed for image compression by O'Leary and Peleg [10]. In this decomposition, which we call the semi-discrete decomposition (SDD), the matrix is approximated by summing outer products just as in the SVD, but the m-vectors and n-vectors only have entries in the set $\{-1, 0, 1\}$. This decomposition is constructed via a greedy algorithm and is not an optimal decomposition; however, for equal query times, the SDD does as well as the SVD method and requires approximately one-tenth the storage. The trade-off is that the SDD takes substantially longer to compute for sparse matrices, but this is only a one-time expense. The SDD is discussed in Section 4, and computational comparisons with the SVD are given in Section 5.

In many information retrieval settings, the document database is constantly being updated. Much work has been done on updating the SVD approximation to the term-document matrix [2, 9], but it can be as expensive as computing the original SVD. Efficient algorithms for updating the SDD are given in Section 6.

2 The Vector Space Model

Both the SVD- and the SDD-based LSI models are built on the vector space model, which we describe in this section.

2.1 Creating the Term-Document Matrix

We begin with a collection of textual documents. We determine a list of keywords or *terms* by

1. creating a list of all words that appear in the documents,
2. removing words void of semantic content such as "of" and "because" (using the *stop word* list of Frakes and Baeza-Yates [7]), and
3. further trimming the list by removing words that appear in only one document.

The remaining words are the terms, which we number from 1 to m.

We then create an $m \times n$ *term-document matrix*

$$\mathbf{A} = [a_{ij}],$$

where a_{ij} represents the *weight* of term i in document j.

The most natural choice of weights is to set $a_{ij} = f_{ij}$, the number of times that term i appears in document j. Choosing the term weights properly is critical to the success of the vector space model, so more elaborate schemes have been devised.

A term weight has three components: local, global, and normalization. We let

$$a_{ij} = g_i \; t_{ij} \; d_j,$$

where t_{ij} is the term component (based on information in the jth document only), g_i is the global component (based on information about the use of the ith term throughout the collection), and d_j is the normalization component, specifying whether or not the columns are normalized. Various formulas for each component are given in Tables 1 – 3. In these formulas, χ represents the signum function

$$\chi(t) = \begin{cases} 1 & \text{if } t > 0, \\ 0 & \text{if } t = 0, \\ -1 & \text{if } t < 0. \end{cases}$$

The weight formula is specified by a three letter string whose letters represent the local, global, and normalization components respectively; for example, using weight `lxn` specifies that

$$a_{ij} = \frac{\log(f_{ij}+1)}{\sqrt{\sum_{k=1}^{m} \left(\log(f_{kj}+1)\right)^2}},$$

i.e., log local weights, no global weights, and column normalization.

Symbol	Formula for t_{ij}	Brief Description	Ref.
`b`	$\chi(f_{ij})$	Binary	[11]
`t`	f_{ij}	Term Frequency	[11]
`c`	$.5\,\chi(f_{ij}) + .5\left(\frac{f_{ij}}{\max_k f_{kj}}\right)$	Augmented Normalized Term Frequency	[7, 11]
`l`	$\log(f_{ij}+1)$	Log	[7]

Table 1: Local Term Weight Formulas

Symbol	Formula for g_i	Brief Description	Ref.
`x`	1	No change	[11]
`f`	$\log\left(\frac{n}{\sum_j \chi(f_{ij})}\right)$	Inverse Document Frequency (IDF)	[11]
`p`	$\log\left(\frac{n - \sum_j \chi(f_{ij})}{\sum_j \chi(f_{ij})}\right)$	Probabilistic Inverse	[7, 11]

Table 2: Global Term Weight Formulas

Symbol	Formula for d_j	Brief Description	Ref.
`x`	1	No Change	[11]
`n`	$\left(\sum_{i=1}^{m}(g_i t_{ij})^2\right)^{-1/2}$	Normal	[11]

Table 3: Normalization Formulas

2.2 Query Creation and Processing

A query is represented as an m-vector

$$\mathbf{q} = [q_i],$$

where q_i represents the weight of term i in the query. In order to rank the documents, we compute

$$\mathbf{s} = \mathbf{q}^{\mathrm{T}}\mathbf{A},$$

where the jth entry in $\mathbf{s}$ represents the score of document j. The documents can then be ranked according to their scores, highest to lowest, for relevance to the query.

We must also specify a term weighting for the query. This need not be the same as the weighting for the documents. Here

$$q_i = g_i \, \hat{t}_i,$$

where g_i is computed based on the frequencies of terms in the *document* collection, and $\hat{t}_i$ is computed using the same formulas as for t_{ij} given in Table 1 with f_{ij} replaced by $\hat{f}_i$, the frequency of term i in the query. Normalizing the query vector has no effect on the document rankings, so we never do it. This means the last component of the three-letter query weighting string is always `x`. So, for example, the weighting `cfx` means

$$q_i = \left(.5 \, \chi(\hat{f}_i) + .5 \left(\frac{\hat{f}_i}{\max_k \hat{f}_k} \right) \right) \log \left(\frac{n}{\sum_{j=1}^{n} f_{ij}} \right).$$

A six-letter string, e.g. `lxn.cfx`, specifies the document and query weights. We will use various weighting in our LSI experiments.

3 LSI via the SVD

3.1 Approximating the Term-Document Matrix

In LSI, we can use a matrix approximation of the term-document matrix generated by the SVD. The SVD decomposes $\mathbf{A}$ into a set of n triplets of left ($\mathbf{u}_i$) and right ($\mathbf{v}_i$) singular vectors and scalar singular values (σ_i):

$$\mathbf{A} = \sum_{i=1}^{n} \sigma_i \mathbf{u}_i \mathbf{v}_i^{\mathrm{T}}.$$

The vectors $\mathbf{u}_i$ are mutually orthogonal and have norm one, the vectors $\mathbf{v}_i$ are mutually orthogonal with norm one, and the non-negative scalars σ_i are ordered from greatest to least. The SVD is more commonly seen in matrix notation as

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}}$$

where the columns of $\mathbf{U}$ are the left singular vectors, the columns of $\mathbf{V}$ are the right singular vectors, and $\mathbf{\Sigma}$ is a diagonal matrix containing the singular values.

The SVD can be used to build a rank-k approximation to $\mathbf{A}$ by only using the first k triplets; i.e.,

$$\mathbf{A} \approx \mathbf{A}_k \equiv \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^{\mathrm{T}}.$$

In matrix form, this is written as

$$\mathbf{A} \approx \mathbf{A}_k \equiv \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^{\mathrm{T}},$$

where $\mathbf{U}_k$ and $\mathbf{V}_k$ consist of the first k columns of $\mathbf{U}$ and $\mathbf{V}$ respectively, and $\mathbf{\Sigma}_k$ is the leading $k \times k$ principal submatrix of $\mathbf{\Sigma}$. It can be shown that $\mathbf{A}_k$ is the best rank-k approximation to $\mathbf{A}$ in the Frobenius norm and in the Euclidean norm [8].

3.2 Query Processing

We can process queries using our approximation for $\mathbf{A}$:

$$\begin{aligned}
\mathbf{s} &= \mathbf{q}^{\mathrm{T}}\mathbf{A} \approx \mathbf{q}^{\mathrm{T}}\mathbf{A}_k \\
&= \mathbf{q}^{\mathrm{T}}\mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^{\mathrm{T}} \\
&= (\mathbf{q}^{\mathrm{T}}\mathbf{U}_k \mathbf{\Sigma}_k^{\alpha})(\mathbf{\Sigma}_k^{1-\alpha}\mathbf{V}_k^{\mathrm{T}}) \\
&\equiv \tilde{\mathbf{q}}^{\mathrm{T}}\tilde{\mathbf{A}}.
\end{aligned}$$

The scalar α controls the splitting of the $\mathbf{\Sigma}_k$ matrix and has no effect unless we re-normalize the columns of $\tilde{\mathbf{A}}$. We will experiment with various choices for α and re-normalization in Section 5.2.

The SVD has been used quite effectively for information retrieval, as documented in numerous reports. We recommend the original LSI paper [4], a paper reporting the effectiveness of the LSI approach on the TREC-3 dataset [5], and a more mathematical paper [2] for further information on the SVD for LSI.

4 LSI via a Semi-Discrete Decomposition

4.1 Approximating the Term-Document Matrix

The SVD produces the best rank-k approximation to a matrix, but generally, even a small SVD approximation requires more storage than the original matrix if the original matrix is sparse. To save storage and query time, we propose replacing the SVD by the semi-discrete decomposition (SDD). We write the matrix approximation as a sum of triplets,

$$\mathbf{A}_k = \sum_{i=1}^{k} d_i \mathbf{x}_i \mathbf{y}_i^{\mathrm{T}} = \mathbf{X}_k \mathbf{D}_k \mathbf{Y}_k^{\mathrm{T}},$$

where the m-vector $\mathbf{x}_i$ and the n-vector $\mathbf{y}_i$ have entries taken from the set $\{-1, 0, 1\}$, the scalar d_i is any positive number, and the matrices $\mathbf{X}_k$, $\mathbf{Y}_k$, and $\mathbf{D}_k$ are formed from the vectors and scalars as before. This decomposition does not reproduce $\mathbf{A}$ exactly, even if $k = n$, but the rank-k approximation can use substantially less storage. The SDD requires only the storage of $2k(n + m)$ values from the set $\{-1, 0, 1\}$ and k scalars. An element of the set $\{-1, 0, 1\}$ can be expressed using $\log_2 3$ bits, although our implementation uses two bits per element for simplicity. Furthermore, the SDD requires only single precision scalars because it is a self-correcting algorithm; on the other hand, the SVD has been computed in double precision accuracy for numerical stability. Assuming that double precision scalars require 8 bytes and single precision scalars require 4, and packing 8 bits in a byte, we obtain the following storage comparison between a rank-k SVD and SDD approximation to an $m \times n$ matrix:

Method	Component		Total Bytes
SVD	$\mathbf{U}$	km double precision numbers	$8k(m + n + 1)$
	$\mathbf{V}$	kn double precision numbers	
	$\mathbf{\Sigma}$	k double precision numbers	
SDD	$\mathbf{X}$	km numbers from $\{-1, 0, 1\}$	$4k + \frac{1}{4}k(m + n)$
	$\mathbf{Y}$	kn numbers from $\{-1, 0, 1\}$	
	$\mathbf{D}$	k single precision numbers	

The SDD approximation is constructed via a greedy algorithm, converging monotonically to $\mathbf{A}$:

Algorithm. (O'Leary and Peleg [10]) Let $k_{\max}$ be the rank of the desired approximation. This value can be determined ahead of time, or we can iterate until we obtain a desired accuracy. Set $k = 1$. Set the residual matrix $\mathbf{A}^{(c)} = \mathbf{A}$. Choose `tolerance`; we use 0.01.

For $k = 1, \ldots, k_{\max}$

1. Choose a non-zero vector $\mathbf{y} \in \{-1, 0, 1\}^n$. We use a vector of zeros with every hundredth element set to 1.
2. Inner iteration:

   Set `improvement` $= 1$, `change` $= 1$.
   While `improvement` $\geq$ `tolerance`,

(a) Hold the current $\mathbf{y}$ fixed and solve

$$\min_{\substack{\mathbf{x}\in\{-1,0,1\}^m \\ d\in\Re}} \|\mathbf{A}^{(c)} - d\mathbf{x}\mathbf{y}^{\mathrm{T}}\|_F.$$

(b) Hold the current $\mathbf{x}$ fixed and solve

$$\min_{\substack{\mathbf{y}\in\{-1,0,1\}^n \\ d\in\Re}} \|\mathbf{A}^{(c)} - d\mathbf{x}\mathbf{y}^{\mathrm{T}}\|_F.$$

(c) Evaluate the change:

$$\begin{aligned} \texttt{new\_change} &= \|\mathbf{A}^{(c)} - d\mathbf{x}\mathbf{y}^{\mathrm{T}}\|_F - \|\mathbf{A}^{(c)}\|_F, \\ \texttt{improvement} &= \left|\frac{\texttt{new\_change} - \texttt{change}}{\texttt{change}}\right|, \\ \texttt{change} &= \texttt{new\_change}. \end{aligned}$$

End While.

3. Set $\mathbf{x}_k = \mathbf{x}$, $\mathbf{y}_k = \mathbf{y}$, $d_k = d$. Set $\mathbf{A}^{(c)} = \mathbf{A}^{(c)} - d_k\mathbf{x}_k\mathbf{y}_k^{\mathrm{T}}$.

End For.

O'Leary and Peleg showed that the subproblems in steps (2a) and (2b) can be solved optimally. Specifically, the subproblem in step (2a) is solved as follows:

1. Let
$$s_i = x_i \sum_{j=1}^{n} a_{ij}^{(c)} y_j,$$
where $x_i = \pm 1$ is chosen so that $s_i \geq 0$ for all $i = 1, \ldots, m$.
2. Order the s_i's so that $s_{i_1} \geq s_{i_2} \geq \cdots \geq s_{i_m}$.
3. For $j = 1, \ldots, m$, let
$$h_j = \frac{1}{j}\left(\sum_{k=1}^{j} s_{i_k}\right)^2.$$
4. Choose J such that $h_J = max_j h_j$.
5. For $k = J+1, \ldots, m$, set $x_{i_k} = 0$.
6. Let
$$d = \sum_{i=1}^{m} s_i \Big/ \left(J \sum_{j=1}^{n} |y_j|\right).$$

The most expensive parts of the subproblem solution are the matrix-vector multiply in step (1) and the sort in step (2). Because $\mathbf{A}$ is sparse, we never form (the dense matrix) $\mathbf{A}^{(c)}$ explicitly; instead, in step (1) we calculate $\mathbf{Ay}$ and $\mathbf{X}_{k-1}\mathbf{D}_{k-1}\mathbf{Y}_{k-1}^{\mathrm{T}}\mathbf{y}$. The calculation of $\mathbf{Ay}$ requires approximately the same number of multiplies and additions as there are nonzeros in $\mathbf{A}$, and the computation of $\mathbf{X}_{k-1}(\mathbf{D}_{k-1}(\mathbf{Y}_{k-1}^{\mathrm{T}}\mathbf{y}))$ requires k multiplies and no more than $k^2n + km^2$ additions or subtractions. The m-long sort requires approximately $O(m \log m)$ operations.

Note that step (6) does not need to be done when solving the first subproblem since d is immediately re-calculated in the next subproblem. For the second subproblem, we do a transpose multiplication in the first step and an n-long sort in the second step. Thus, for each inner iteration, we have a multiplication by $\mathbf{A}^{(c)}$, a multiplication by $(\mathbf{A}^{(c)})^{\mathrm{T}}$, an m-long sort and an n-long sort. The number of inner iterations is controlled by the `tolerance` threshold.

4.2 Query Processing

We evaluate queries in much the same way as we did for the SVD, by computing $\mathbf{s} = \tilde{\mathbf{q}}^{\mathrm{T}}\tilde{\mathbf{A}}$, with

$$\tilde{\mathbf{A}} = \mathbf{D}_k^{1-\alpha}\mathbf{Y}_k^{\mathrm{T}}, \quad \tilde{\mathbf{q}} = \mathbf{D}_k^{\alpha}\mathbf{X}_k^{\mathrm{T}}\mathbf{q}.$$

Again, we generally re-normalize the columns of $\tilde{\mathbf{A}}$.

For decompositions of equal rank, processing the query for the SDD requires significantly fewer floating-point operations than processing the query for the SVD:

Operation	SDD	SVD
Additions	$k(m+n)$	$k(m+n)$
Multiplications	k	$k(1+m+n)$

If we re-normalize the columns of $\tilde{\mathbf{A}}$ then each each method requires n additional multiplies and storage of n additional floating point numbers.

5 Computational Comparison of the SDD- and SVD-Based LSI Methods

In this section, we present computational results comparing the SDD- and SVD-based LSI methods. All tests were run on a Sparc 20. Our code is in C, with the SVD taken from SVDPACKC [1].

5.1 Methods of Comparison

We will compare the SDD- and SVD-based LSI methods using three standard test sets. Each test set comes with a collection of documents, a collection of queries, and *relevance judgments* for each query. The relevance judgments are lists of the documents relevant to each query. The test sets, each with over 1000 documents, are described in Table 4.

	MEDLINE	**CRANFIELD**	**CISI**
Number of Documents:	1033	1399	1460
Number of Queries:	30	225	35
Number of (Indexing) Terms:	5526	4598	5574
Avg. No. of Terms/Document:	48	57	46
Avg. No. of Documents/Term:	9	17	12
% Nonzero Entries in Matrix:	0.87	1.24	0.82
Storage for Matrix (MB):	0.4	0.6	0.5
Avg. No of Terms/Query:	10	9	7
Avg. No. Relevant/Query:	23	8	50

Table 4: Characteristics of the test sets.

We will compare the systems by looking at *mean average precision*, a standard measure used by the information retrieval community.

When we evaluate a query, we return a ranked list of documents. Let r_i denote the number of relevant documents among the top i documents. The *precision* for the top i documents, p_i, is then defined as

$$p_i = \frac{r_i}{i},$$

i.e., the proportion of the top i documents that are relevant.

The N-point (interpolated) *average precision* for a single query is defined as

$$\frac{1}{N} \sum_{i=0}^{N-1} \tilde{p} \left(\frac{i}{N-1} \right).$$

where

$$\tilde{p}(x) = \max_{\frac{r_i}{r_n} \geq x} p_i.$$

Typically, 11-point interpolated average precision is used. Each of our data sets has multiple queries, so we compare the mean average precision and the median average precision, expressed as percentages. In other papers, average precision generally refers to mean average precision.

5.2 Parameter Choices

We have two parameter choices to make for the SDD and SVD methods: the choice of the splitting parameter α, and the choice of whether or not to renormalize the columns of $\tilde{\mathbf{A}}$.

	SDD		SVD	
	Re-Normalize?		Re-Normalize?	
α	Yes	No	Yes	No
0	62.1	61.2	65.1	64.2
0.5	62.6	61.2	64.7	64.2
-0.5	57.9	61.2	64.7	64.2
1.0	61.7	61.2	64.2	64.2
-1.0	48.6	61.2	62.3	64.2

Table 5: Mean average precision for the SDD and SVD methods with different parameter choices on the MEDLINE data set with k=100 and weighting `lxn.bpx`.

We experimented with the SVD and SDD methods on the MEDLINE data set using the weighting `lxn.bpx`. The results are summarized in Table 5. In all further tests, we will use $\alpha = 0.5$ with re-normalization for the SDD method and $\alpha = 0$ with re-normalization for the SVD method. We experimented using other weighings and other data sets and confirmed that these parameter choices are always best or very close to it.

5.3 Comparisons

We tried the SDD and SVD methods with a number of weighings. We selected these particular weighings for testing in LSI based on their good performance for the vector space method on these datasets. We present mean average precision results in Table 6 using a rank $k = 100$ approximation in each method; this table also includes vector space (VS) results for comparison.

To continue our comparisons, we select a "best" weighting for each data set. In Table 6 we have highlighted the "best" results for each data set in boldface type. We will use only the corresponding weighings for the remainder of the paper, although further experiments show similar results for other weighings.

In Figures 1 – 3, we compare the SVD and SDD methods on the data sets.

In Figure 1, we present results for the MEDLINE data. The upper right graph plots the mean average precision vs. query time, and the upper left graph plots the median average precision vs. query time. (The query time is the total time required to execute all queries associated with the data set.) Observe that the SDD method has maximal precision at a query time of 3.4 seconds, corresponding to $k = 140$, a mean average precision of 63.6 and a median average precision of 71.4. The SVD method reaches its peak at 8.4 seconds, corresponding to $k = 110$, and mean and median average precisions of 65.5 and 71.7 respectively. The performance of the SDD method is on par with

	MEDLINE			CRANFIELD			CISI		
Weight	SDD	SVD	VS	SDD	SVD	VS	SDD	SVD	VS
`lxn.bfx`	62.6	64.6	54.6	35.7	**40.4**	45.5	15.6	16.6	17.7
`lxn.bpx`	**62.6**	**65.1**	**54.6**	35.6	39.9	45.5	15.2	16.9	17.8
`lxn.lfx`	61.2	64.0	53.7	**35.8**	40.3	45.6	16.0	16.6	18.2
`lxn.lpx`	61.3	64.3	53.8	35.5	40.1	**45.7**	15.5	16.9	18.3
`lxn.tfx`	60.9	63.5	53.2	35.7	40.2	45.6	16.3	16.9	**18.4**
`lxn.tpx`	60.9	63.8	53.4	35.4	39.9	45.6	15.7	17.0	18.3
`cxx.bpx`	57.9	59.6	53.6	32.9	38.9	43.4	17.1	**17.9**	17.5
`cxn.bfx`	58.4	62.5	53.6	33.1	38.7	44.1	17.8	16.5	17.4
`cxn.bpx`	58.4	63.0	53.6	32.6	38.7	43.4	**18.1**	17.6	17.5
`cxn.tfx`	56.8	61.5	52.5	33.3	38.8	43.9	17.1	16.9	18.2
`cxn.tpx`	57.0	61.8	52.6	32.7	38.2	43.3	17.1	17.7	18.2

Table 6: Mean average precision results for the SDD and SVD methods with k=100.

the SVD method except for somewhat worse behavior on queries 26 and 27. We have no explanation for the SDD behavior on these two queries.

In terms of storage, the SDD method is extremely economical. The middle left graph plots mean average precision vs. decomposition size (in megabytes (MB)), and the middle right graph plots median average precision vs. the decomposition size. Note that a significant amount of extra storage space is required in the computation of the SVD; this is not reflected in these numbers. From these plots, we see that even a rank-30 SVD takes 50% more storage than a rank-600 SDD, and each increment of 10 in rank adds approximately 0.5 MB of additional storage to the SVD. The original data takes only 0.4 MB, but SVD requires over 1.5 MB before it even begins to come close to what the SDD can do in less than 0.2 MB.

The lower left graph illustrates the growth in required storage as the rank of the decomposition grows. For a rank-600 approximation, the SVD requires over 30 MB of storage while the SDD requires less than 1 MB.

It is interesting to see how good these methods are at approximating the matrix. The lower right graph shows the Frobenius norm (F-norm) of the residual, divided by the Frobenius norm of the original matrix, as a function of storage (logarithmic scale). The SVD eventually forms a better approximation to the term-document matrix, making it behave more like the vector space method. This is not necessarily desirable.

The CRANFIELD dataset is troublesome for LSI techniques; they do not do as well as the vector space method. From the upper two graphs in Figure 2 we see that, for equal query times, the SDD method does as well as the SVD

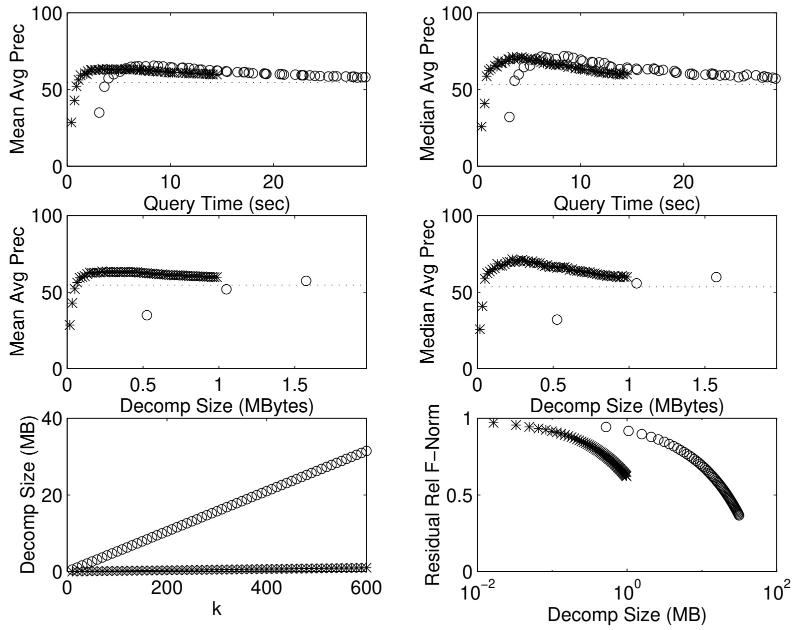

Figure 1: A comparison of the SVD ($*$) and SDD ($\circ$) on the MEDLINE data set. We plot 60 data points for each graph, corresponding to $k = 10, 20, \ldots, 600$. The dotted lines show the corresponding data for the vector space method.

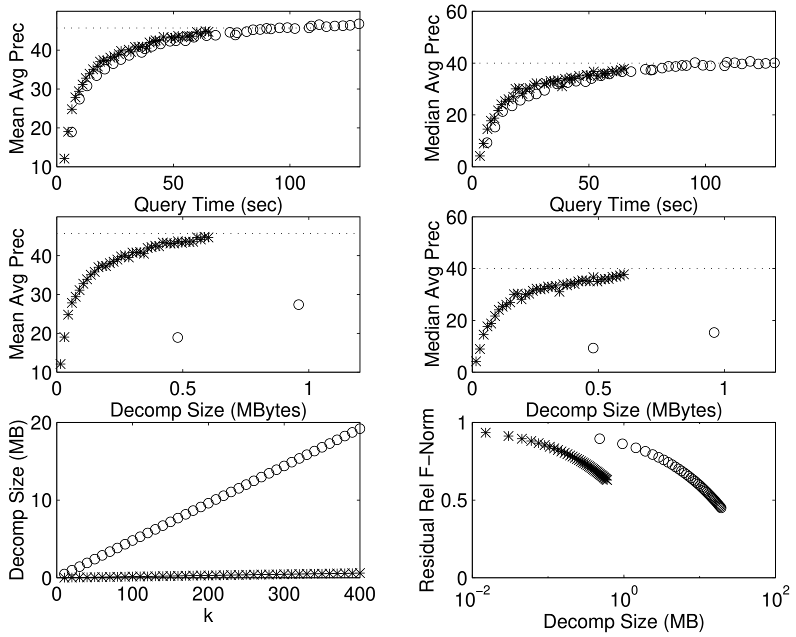

Figure 2: A comparison of the SVD ($*$) and SDD ($\circ$) on the CRANFIELD data set. We plot 40 data points for each graph, corresponding to $k = 10, 20, \ldots, 400$. The dotted lines show the corresponding data for the vector space method.

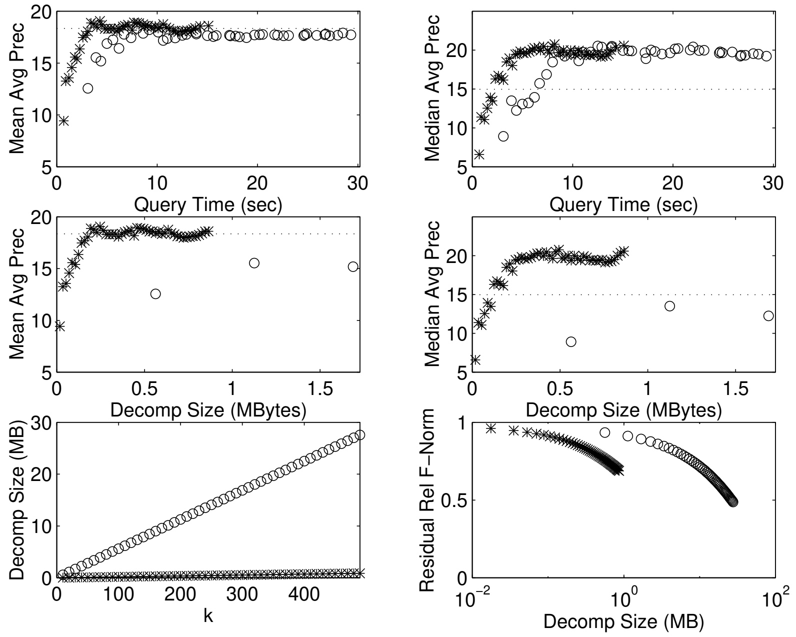

Figure 3: A comparison of the SVD ($*$) and SDD ($\circ$) on the CISI data set. We plot 49 data points for each graph, corresponding to $k = 10, 20, \ldots, 490$. The dotted lines show the corresponding data for the vector space method.

method. The other graphs show that, as in the MEDLINE test, the SDD is much more economical in terms of storage and achieves a somewhat less accurate approximation of the matrix.

In Figure 3 we compare the SVD and SDD methods on the CISI data. The SDD method is better overall than the SVD method in terms of query time, and its mean average precision peaks higher than the SVD method — 19.1 versus 18.3. Again, the storage differences are dramatic.

	MEDLINE		CRANFIELD		CISI	
	SDD	SVD	SDD	SVD	SDD	SVD
Query Time (Sec)	3.4	3.6	63.8	77.3	4.3	4.4
Dimension (k)	140	20	390	210	140	30
Mean Avg Prec	63.6	51.8	44.9	44.5	19.1	15.2
Median Avg Prec	71.4	55.7	37.3	37.4	19.4	12.2
Decomp Storage (MB)	0.2	1.1	0.6	10.1	0.2	1.7
Decomp Time (Sec)	245.4	4.7	1313.8	91.5	279.0	13.0
Rel. F-Norm of Resid	0.85	0.90	0.63	0.59	0.85	0.89

Table 7: Comparison of the SDD and SVD methods at the query time where the SDD has the highest mean average precision.

Table 7 compares the two methods for the query time at which the SDD method peaks on mean average precision. On all three data sets, the SDD has higher mean and median precisions than the SVD. Since all the methods have similar performance in terms of the mean and median average precision, observe that the trade-off is in the decomposition computation time and the decomposition storage requirement; the SVD is much faster to compute, but the SDD is much smaller.

The results on the three data sets can be summarized as follows: the SDD method is competitive with the SVD method for information retrieval. For equal query times, the SDD method generally has a better mean and median average precision. The SDD requires much less storage and may be the only choice when storage is at a premium. The only disadvantage is the long time required for the initial decomposition, but this is generally a one-time-only expense. Further research should be done on improving the decomposition algorithm.

6 Modifying the SDD when the Document Collection Changes

Thus far we have discussed the usefulness of the SDD on a fixed document collection. In practice, it is common for the document collection to be dynamic:

new documents are added, and old documents are removed. Thus, the list of terms might also change. In this section, we will focus on the problem of modifying a SDD decomposition when the document collection changes.

SVD-updating has been studied by O'Brien [9]. He reports that updating the SVD takes almost as much time as re-computing it, but that it requires less memory. His methods are similar to what we do in Method 1 in the next section.

6.1 Adding or Deleting Documents or Terms

Suppose that we are adding new documents to the collection. For now we will assume that this does not affect the set of terms and that no global weighting was used on the matrix. Let

$$\mathbf{A} = \left[\begin{array}{cc} \mathbf{A}^{(1)} & \mathbf{A}^{(2)} \end{array} \right]$$

represent the updated collection of documents where $\mathbf{A}^{(1)}$ is the original matrix and $\mathbf{A}^{(2)}$ is the matrix representing the new documents and is weighted in the same way as $\mathbf{A}^{(1)}$.

Assume that $\mathbf{X}^{(1)}$, $\mathbf{D}^{(1)}$, and $\mathbf{Y}^{(1)}$ are the components of the SDD decomposition for $\mathbf{A}^{(1)}$. We propose two methods for updating this decomposition.

Method 1: Append rows to $\mathbf{Y}^{(1)}$. The simplest update method is to append new rows to $\mathbf{Y}^{(1)}$. In other words, keeping $\mathbf{X}^{(1)}$, $\mathbf{D}^{(1)}$, and $\mathbf{Y}^{(1)}$ fixed, we wish to compute $\mathbf{Y}^{(2)}$ such that

$$\mathbf{A} \approx \mathbf{X}^{(1)}\mathbf{D}^{(1)} \left[\begin{array}{c} \mathbf{Y}^{(1)} \\ \mathbf{Y}^{(2)} \end{array} \right]^T .$$

Let $k_{\max}$ be the rank of the decomposition desired; generally this is the same as the rank of the original decomposition. For each value of $k = 1, \ldots, k_{\max}$, we must find the vector $\mathbf{y}$ that solves

$$\min ||\mathbf{A}^{(c)} - d\mathbf{x}\mathbf{y}^T||_F ,$$

where $\mathbf{A}^{(c)} = \mathbf{A}^{(2)} - \mathbf{X}^{(1)}_{k-1}\mathbf{D}^{(1)}_{k-1}(\mathbf{Y}^{(2)}_{k-1})^T$, $\mathbf{x}$ is the kth column of $\mathbf{X}^{(1)}$, and d is the kth diagonal element of $\mathbf{D}^{(1)}$. We never access $\mathbf{A}^{(1)}$, and this may be useful in some situations. The solution $\mathbf{y}$ becomes the kth column of $\mathbf{Y}^{(2)}$. The procedure to optimally solve this problem is the same as that used on the subproblem discussed in Section 3.1, except that here d is fixed, so the definition of h_j in the third step is changed to

$$h_j = 2d \sum_{k=1}^{j} s_{i_k} - jd^2 \sum_{k=1}^{m} |x_k|.$$

Method 2: Re-Compute D and Y. Another possible method is to completely re-compute $\mathbf{D}$ and $\mathbf{Y}$, keeping $\mathbf{X}^{(1)}$ fixed.

Let $k_{\max}$ be the rank of the decomposition desired. For each $k = 1, \ldots, k_{\max}$, we must find the d and $\mathbf{y}$ that solve

$$\min \|\mathbf{A}^{(c)} - d\mathbf{x}\mathbf{y}^T\|_F,$$

where $\mathbf{A}^{(c)} = \mathbf{A} - \mathbf{X}^{(1)}_{k-1}\mathbf{D}_{k-1}\mathbf{Y}^T_{k-1}$ and $\mathbf{x}$ is the kth column of $\mathbf{X}^{(1)}$. The solutions d and $\mathbf{y}$ become the kth diagonal element of $\mathbf{D}$ and the kth column of $\mathbf{Y}$ respectively.

Neither method has any inner iterations, and so both are fast. We tried each update method on a collection of tests derived from the MEDLINE data. We split the MEDLINE document collection into two groups. We did a decomposition on the first group of documents with $k = 100$, then added the second group of documents to the collection, and updated the decomposition via each of the two update methods. The results are summarized in Table 8. The second method is better, as should be expected since we are allowing more to change. For the second method, the decrease in mean average precision is not very great when we add only a small number of documents. As the proportion of new documents to old documents grows, however, performance worsens.

			Method 1		Method 2	
Documents		Decomp	Time	Mean	Time	Mean
Old	New	Time (Sec)	(Sec)	Avg Prec	(Sec)	Avg Prec
929	104	134.5	5.0	59.72	5.6	61.48
826	207	129.5	4.9	55.29	5.4	60.06
723	310	125.9	5.1	53.49	5.5	60.84
619	414	113.5	5.0	46.47	5.5	56.98
516	517	107.9	5.0	37.77	5.5	55.04
413	620	95.5	5.3	35.96	5.5	54.51
309	724	83.6	5.1	19.33	5.5	46.56
206	827	64.3	5.1	21.25	5.4	49.36
103	930	63.0	5.2	11.17	5.2	39.91

Table 8: Comparison of two update methods on the MEDLINE data set with $k = 100$.

If we want to incorporate additional terms, we would be adding additional rows to $\mathbf{A}$. The two update methods discussed above can also be used in this situation. If we want to add both new terms and new documents, we can add one and then the other.

If we wish to delete terms or documents, we simply delete the corresponding rows in the $\mathbf{X}$ and $\mathbf{Y}$ matrices.

6.2 Iterative Improvement of the Decomposition

If we have an existing decomposition, perhaps resulting from adding and/or deleting documents and/or terms, we may wish to improve on this decomposition without actually re-computing it. We consider two approaches.

Method 1: Partial Re-Computation In order to improve on this decomposition, we could reduce its rank by deleting 10% of the vectors and then re-compute them using our original algorithm. This method's main disadvantage is that it can be expensive in time. If performed on the original decomposition, it has no effect.

Method 2: Fix and Compute. This method is derived from the second update method. We fix the current $\mathbf{Y}$ and re-compute $\mathbf{X}$ and $\mathbf{D}$; we then fix the current $\mathbf{X}$ and re-compute the $\mathbf{Y}$ and $\mathbf{D}$. This method is very fast because there are no inner iterations. This can be repeated to further improve the results. If applied to an original decomposition, it would change it.

We took the decompositions resulting from the second update method in the last subsection and applied the improvement methods to them. We have a rank-100 decomposition. For the first improvement method, we re-computed 10 dimensions. For the second improvement method, we applied the method once. The results are summarized in Table 9. If we have added only a few documents, the first method improves the precision while the second method worsens it. On the other hand, if we have added many documents, then the second method is much better. The first method could be improved by re-computing more dimensions, but this would quickly become too expensive. The second method greatly improves poor decompositions and is relatively inexpensive. It can be applied repeatedly to further improve the decomposition.

7 Conclusions

We have introduced a semi-discrete matrix decomposition for use in LSI. For equal query times, the SDD-LSI method performs as well as the original SVD-LSI method. The advantage of the SDD method is that the decomposition takes very little storage, and the disadvantage is that the initial time to form the decomposition is large. Since decomposition is a one-time expense, we believe that the SDD-LSI algorithm will be quite useful in application.

We have also introduced methods to dynamically change the SDD decomposition if the document collection changes and methods to improve the decomposition if it is found to be inadequate.

			Method 1		Method 2	
Documents		Prev Mean	Time	Mean	Time	Mean
Old	New	Avg Prec	(Sec)	Avg Prec	(Sec)	Avg Prec
929	104	61.48	18.0	60.83	13.8	61.24
826	207	60.06	19.3	59.81	13.9	61.55
723	310	60.84	20.8	60.76	13.7	61.92
619	414	56.98	21.5	58.60	14.4	60.49
516	517	55.04	20.4	56.76	13.5	58.19
413	620	54.51	22.9	56.04	13.7	59.41
309	724	46.56	20.1	47.90	13.4	55.03
206	827	49.36	20.8	53.44	13.5	56.20
103	930	39.91	18.9	45.36	13.3	52.08

Table 9: Comparison of two improvement methods on the MEDLINE data set with $k = 100$.

Acknowledgments

We are grateful to Duncan Buell, John Conroy, Ken Kolda, Steve Kratzer, Joe McCloskey, and Doug Oard for helpful comments.

References

[1] Michael Berry, Theresa Do, Gavin O'Brien, Vijay Krishna, and Sowmini Varadhan. SVDPACKC (Version 1.0) Users' Guide. Technical Report CS-93-194, Computer Science Department, University of Tennessee, Knoxville, TN 37996-1301, 1993.

[2] Michael W. Berry, Susan T. Dumais, and Gavin W. O'Brien. Using linear algebra for intelligent information retrieval. *SIAM Review*, 37:573–595, 1995.

[3] M.W. Berry and R.D. Fierro. Low-rank orthogonal decompositions for information retrieval applications. *Numerical Linear Algebra with Applications*, 1:1–27, 1996.

[4] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the Society for Information Science*, 41:391–407, 1990.

[5] Susan Dumais. Improving the retrieval of infomation from external sources. *Behavior Research Methods, Instruments, & Computers*, 23:229–236, 1991.

[6] Susan Dumais. Latent sematic indexing (LSI):TREC-3 report. In D.K. Harman, editor, *Proceedings of the Third Text REtrieval Conference (TREC-3)*, NIST Special Publication 500-225, pages 219–230, April 1995. (http://potomac.ncsl.nist.gov/TREC).

[7] William B. Frakes and Ricardo Baeza-Yates. *Information Retrieval: Data Structures and Algorithms.* Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[8] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* Johns Hopkins Press, 2nd edition, 1989.

[9] Gavin O'Brien. Information management tools for updating an SVD-encoded indexing scheme. Master's thesis, University of Tennessee, Knoxville, TN 37996-1301, 1994.

[10] Dianne P. O'Leary and Shmuel Peleg. Digital image compression by outer product expansion. *IEEE Transactions on Communications*, 31:441–444, 1983.

[11] Gerald Salton and Chris Buckley. Term weighting approaches in automatic text retrieval. *Information Processing and Management*, 24:513–523, 1988.