EDLSI with PSVD Updating

April Kontostathis* Erin Moulding and Raymond J. Spiteri†

Abstract

This paper describes the results obtained from the merging of two techniques that provide improvements to search and retrieval using Latent Semantic Indexing (LSI): Essential Dimensions of LSI (EDLSI) and partial singular value decomposition (PSVD) updating. EDLSI utilizes an implementation of LSI that requires the use of only a few dimensions in the LSI space. The PSVD updating and folding-up algorithms can be used to incrementally maintain the integrity of the LSI space as new content is added. In this paper, we show that EDLSI works as expected with these updating techniques, maintaining retrieval performance and dramatically improving runtime performance.

Folding-in is another technique for incorporating documents into an LSI space; however, it does not deliberately maintain the orthogonality of the document and term vectors, and thus retrieval performance usually suffers. Interestingly we find that combining EDLSI with folding-in results in retrieval performance that is very similar to that of standard LSI but at a dramatically reduced cost.

1 Introduction

Since Latent Semantic Indexing (LSI) was introduced in 1998, many groups have studied its effectiveness on a variety of collections and applications [8], [10], [27], [11], [17], whereas other papers have studied its effectiveness and efficiency [2], [3], [6], [13], [7]. Recently, Kontostathis has identified an effective way to utilize LSI using only a few LSI dimensions [15], and Mason (née Tougas) and Spiteri have proposed an efficient approach for handling updating of the LSI space [18], [25]. In this paper, we show that the combination of Essential Dimensions of LSI (EDLSI) [15] with different updating strategies, e.g., [19], [5], [28], [18], [25], is efficient, effective, and consistently meets or exceeds LSI in terms of retrieval performance.

We begin with a brief overview of LSI in Section 2, including methods for updating the semantic space as documents are added. We provide an overview of EDLSI in Section 3 and describe our approach for testing the effectiveness of EDLSI in the context of updating in Section 4. We present the results in Section 5, and offer our conclusions in Section 6.

*Department of Mathematics and Computer Science, Ursinus College, Collegeville PA 19426, akontostathis@ursinus.edu.

†Department of Computer Science, Univ. of Saskatchewan, Saskatchewan, S7N 5C9, Canada, erm764@mail.usask.ca, spiteri@cs.usask.ca.

2 Overview of Latent Semantic Indexing

In this section we begin with a description of vector-space retrieval, and then provide a brief overview of LSI [8]. We also describe the updating methods used in our study: recomputing the partial singular value decomposition (PSVD), folding-in, PSVD updating, percentage-based folding-up, and adaptive folding-up.

2.1 Vector-Space Retrieval In traditional vector-space retrieval, a document is represented as a vector in t-dimensional space, where t is the number of terms in the lexicon being used. If there are d documents in the collection, then the vectors representing the documents can be represented by a matrix $A \in \Re^{t \times d}$, called the *term-document matrix*. Entry $a_{i,j}$ of matrix A indicates how important term i is to document j, where $1 \leq i \leq t$ and $1 \leq j \leq d$.

The entries in A can be binary numbers (1 if the term appears in the document and 0 otherwise), raw term frequencies (the number of times the term appears in the document), or weighted term frequencies. Weighting can be done using either local weighting, global weighting, or a combination of both. Because document sizes often vary, weights are also usually normalized; otherwise, long documents are more likely to be retrieved. See, e.g., [1], [21] for a comprehensive discussion of local and global weighting techniques. In our experiments we used the log-entropy weighting scheme because it has been shown to work well with LSI [9].

Common words, such as *and*, *the*, *if*, etc., are considered to be *stop-words* [1] and are not included in the term-document matrix. We used a standard English stop list containing approximately 500 words. Words that appear infrequently are often excluded to reduce the size of the lexicon. We retain all terms that appear in at least 2 documents.

Like the documents, queries are represented as t-dimensional vectors, and local and global term weighting is applied. Documents are retrieved by projecting $\mathbf{q}$ into the row (document) space of the term-document matrix, A:

$$\mathbf{w} = \mathbf{q}^T A.$$

where $\mathbf{w}$ is a d-dimensional row vector, entry j of which

is a measure of how relevant document j is to query $\mathbf{q}$. In a traditional search-and-retrieval application, documents are sorted based on their relevance score (i.e., vector $\mathbf{w}$) and returned to the user with the highest-scoring document appearing first. The order in which a document is retrieved is referred to as the *ranking* of the documents with respect to the query. The experiments in this paper run multiple queries against a given dataset, so in general the query vectors $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_n$ are collected into a matrix $Q \in \Re^{t \times n}$ and their relevance scores are computed as

$$W = Q^T A,$$

where entry $w_{i,j}$ in $W \in \Re^{n \times d}$ is a measure of how relevant document j is to query i.

There are two immediate deficiencies of vector-space retrieval. First, W might pick up documents that are not relevant to the queries in Q but contain some of the same words. Second, Q may overlook documents that are relevant but that do not use the exact words being queried. The PSVD that forms the heart of classical LSI systems is used to capture term relationship information in the term-document space. Documents that contain relevant terms but perhaps not exact matches ideally end up close to the queries in the LSI space [8].

2.2 LSI and the PSVD The PSVD, also known as the truncated SVD, is derived from the SVD. The (reduced) SVD decomposes the term-document matrix into the product of three matrices: $U \in \Re^{t \times r}$, $\Sigma \in \Re^{r \times r}$, and $V \in \Re^{d \times r}$, where r is the rank of the matrix A. The columns of U and V are orthonormal, and Σ is a diagonal matrix, the diagonal entries of which are the r (positive) non-zero singular values of A, customarily arranged in non-increasing order. Thus A is factored as

$$A = U \Sigma V^T.$$

The PSVD produces an optimal rank-k ($k < r$) approximation to A [12] by truncating Σ after the first (and largest) k singular values. The corresponding columns from $k+1$ to r of U and V are also truncated, leading to matrices $U_k \in \Re^{t \times k}$, $\Sigma_k \in \Re^{k \times k}$, and $V_k \in \Re^{d \times k}$. A is then approximated by

$$A \approx A_k = U_k \Sigma_k V_k^T.$$

In the context of LSI, there is evidence to show that A_k provides a better model of the semantic structure of the corpus than the original term-document matrix was able to provide for some collections [8, 10, 11, 5, 4].

Choosing an optimal LSI dimension k for a given collection remains elusive. Traditionally, the optimal k has been chosen by running a set of queries with known relevant document sets for multiple values of k. The k that results in the best retrieval performance is chosen as the optimal k for each collection. We use this approach to determine the optimal k in our experiments. Optimal k values in a traditional LSI system are typically in the range of 100–300 dimensions [10], [17]. The k values required by EDLSI are much smaller (see Section 5).

2.3 Adding New Information to an LSI Space The PSVD is useful in information retrieval and other contexts, but its computation is expensive [28]. This expense is offset by the fact that the LSI space for a given set of documents can be used for many queries. However, if terms or documents are added to the initial dataset, then either the PSVD must be recomputed for the new A or the new information must be incorporated into the current PSVD. In LSI, the two traditional ways of incorporating information into an existing PSVD are folding-in [19], [5] and PSVD updating [28].

2.3.1 Folding-in Suppose $D \in \Re^{t \times p}$ is a matrix whose columns are the p new documents to be added. In order to fold-in new document vectors, the documents are first projected into the reduced k-dimensional LSI space and then appended to the bottom of V_k. The projection is

$$D_k = D^T U_k \Sigma_k^{-1},$$

and D_k is then appended to the bottom of V_k, so the augmented term-document matrix is now approximated by

$$[\ A, \quad D\ ] \approx U_k\, \Sigma_k\, [\ V_k^T, \quad D_k^T\ ].$$

Note that U_k and Σ_k are not modified when using folding-in. A similar procedure can be applied to U_k to fold-in terms [5]. Folding-in may be useful if only a few documents are to be added to the dataset or if word usage patterns do not vary significantly, as may be the case in a restricted lexicon. However, in general, each application of folding-in corrupts the orthogonality of the columns of U_k or V_k, eventually reducing the effectiveness of LSI [5], [19]. It is not generally advisable to rely exclusively on folding-in when the dataset changes frequently [23].

2.3.2 PSVD Updating Another alternative to recomputing the PSVD is PSVD updating. PSVD updating is faster than recomputing and more accurate than folding-in for LSI [19], [5], [23], [28]. Updating the PSVD gives the same result (to within rounding er-

rors) as recomputing the PSVD but with significantly less computational expense.

In what follows, we describe the updating approach by Zha and Simon, who have shown it to produce better results for LSI than that of O'Brien [28], [19].

Let $D \in \Re^{t \times p}$ be the matrix whose columns are the p new documents to be added (as above). Define

$$\hat{A} = \left[\begin{array}{cc} A_k, & D \end{array} \right].$$

The following method updates the PSVD of A to give the PSVD of $\hat{A}$. Let $\hat{D} \in \Re^{t \times p}$ be defined by

$$\hat{D} = \left[I_t - U_k U_k^T \right] D,$$

where I_t is the identity matrix of size t.

The matrix $I_t - U_k U_k^T$ is an orthogonal projection that maps the columns of D into the subspace that is orthogonal to the space spanned by the columns of U_k. Note that an orthogonal projection is a matrix P such that $P^2 = P$ and $P = P^T$. Having formed the projection $\hat{D}$, we now form the reduced QR decomposition of $\hat{D}$ such that

$$Q_D R_D = \hat{D}.$$

Recall that with such a decomposition, $Q_D \in \Re^{t \times p}$ has orthonormal columns and $R_D \in \Re^{p \times p}$ is upper triangular. Then,

$$\hat{A} = \left[\begin{array}{cc} U_k, & Q_D \end{array} \right] \left[\begin{array}{cc} \Sigma_k & U_k^T D \\ 0_{pk} & R_D \end{array} \right] \left[\begin{array}{cc} V_k^T & 0_{kp} \\ 0_{pd} & I_p \end{array} \right],$$

where I_p is the identity matrix of size p and 0_{mn} denotes the zero matrix of size $m \times n$.

Let $\tilde{A} \in \Re^{(k+p) \times (k+p)}$ be defined by

$$\tilde{A} = \left[\begin{array}{cc} \Sigma_k & U_k^T D \\ 0_{pk} & R_D \end{array} \right].$$

Furthermore, let $\tilde{A} = \tilde{U} \tilde{\Sigma} \tilde{V}^T$ be the SVD of $\tilde{A}$, and partition it to give

$$\tilde{A} = \left[\begin{array}{cc} \tilde{U}_k, & \tilde{U}_p \end{array} \right] \left[\begin{array}{cc} \tilde{\Sigma}_k & 0_{kp} \\ 0_{pk} & \tilde{\Sigma}_p \end{array} \right] \left[\begin{array}{cc} \tilde{V}_k, & \tilde{V}_p \end{array} \right]^T,$$

where $\tilde{U}_k \in \Re^{(k+p) \times k}$, $\tilde{U}_p \in \Re^{(k+p) \times p}$, $\tilde{\Sigma}_k \in \Re^{k \times k}$, $\tilde{\Sigma}_p \in \Re^{p \times p}$, $\tilde{V}_k \in \Re^{(k+p) \times k}$, $\tilde{V}_p \in \Re^{(k+p) \times p}$.

The rank-k PSVD of the updated term-document matrix $\hat{A} = [A_k, D]$ is

$$\hat{A}_k = \left(\left[\begin{array}{cc} U_k, & Q_D \end{array} \right] \tilde{U}_k \right) \tilde{\Sigma}_k \left(\left[\begin{array}{cc} V_k & 0_{dp} \\ 0_{pk} & I_p \end{array} \right] \tilde{V}_k \right)^T.$$

Note that only quantities from the rank-k PSVD of $\tilde{A}$ appear. A similar process can be followed for adding terms.

2.3.3 Folding-up In 2007, Tougas and Spiteri introduced the original folding-up algorithm and showed it to be an attractive alternative to either folding-in or PSVD updating alone [24].

Folding-up starts from the premise that new documents (or terms) to be added should first be folded-in, and the original document vectors stored for later use. After a certain amount of folding-in has been done, the changes that have been made by the folding-in process are discarded, and the PSVD is updated to represent the current semantic space using a PSVD updating method, such as that of Zha and Simon [28].

The goal of the folding-up method is to switch from folding-in to PSVD updating before the accuracy of the PSVD degrades significantly from the folding-in process. After the PSVD update has been performed, the cycle begins again; documents and/or terms are again folded-in until it is deemed necessary to update the PSVD. In folding-up, the PSVD updating process can be thought of as a correction step, and it is important to decide at what point this step becomes necessary in order to best exploit both the computational efficiency of folding-in and the computational accuracy of PSVD updating.

In [24], after the number of documents that have been folded-in reaches a pre-selected percentage of the current matrix, the vectors that have been appended to V_k during folding-in are discarded. The PSVD is then updated to reflect the addition of all the document vectors that have been folded-in since the last update. The process continues with folding-in until the next update. The choice of the percentage used is empirically based.

The results of a further study of measures to determine when to update during the folding-up process were reported in [18], which describes a process referred to as *adaptive folding-up*. In the adaptive folding-up method, a measure of the error in the PSVD produced by folding-in is monitored to determine when it is most advantageous to switch from folding-in to PSVD updating. This eliminates the need to choose the percentage of documents that may be folded-in before a PSVD update occurs; instead it computes a measure of the accumulated error, based on the loss of orthogonality in the right singular vectors of the PSVD produced by folding-in. The algorithm uses an empirically determined error threshold, τ, to determine whether or not a PSVD update is necessary. Before each new group of documents is folded-in, the adaptive folding-up method computes the error that would be introduced by folding-in these documents. If adding this error to the accumulated error causes it to exceed τ, then PSVD updating is performed instead of folding-in, and the changes from folding-in are again discarded.

The accumulated error is then reset to zero. The error threshold $\tau = 0.01$ was established through extensive empirical testing on the Medlars (MED), Cranfield (CRAN), and CISI datasets, which are freely available on the SMART web site at Cornell University [22], and we used this value in our experiments.

Folding-up incurs some additional overhead because the document vectors that are being folded-in between updates must be saved in their original form until the PSVD updating occurs. The complexity of the process is of the same order as PSVD updating, but the actual computation time depends on the number of iterations in which updating is replaced by folding-up. In [24] and [18], Mason (née Tougas) and Spiteri demonstrate that folding-up produces results that are not statistically different from those produced by recomputing the PSVD but at a significantly reduced computational cost (as measured by CPU time).

3 Essential Dimensions of LSI (EDLSI)

As the dimension of the LSI space k approaches the rank r of the term-document matrix A, LSI approaches vector-space retrieval. In particular, vector-space retrieval is equivalent to LSI when $k = r$. However, vector-space retrieval outperforms LSI on some collections, even for relatively large values of k [16] and [14].

In [15], Kontostathis hypothesizes that optimized traditional LSI systems capture the term relationship information within the first few dimensions and then continue to capture the traditional vector space retrieval data as k approaches r. Her experiments show that improvements to both retrieval and runtime performance can be obtained by combining the term relationship information captured in the first few SVD vectors with vector-space retrieval, a technique referred to as Essential Dimensions of Latent Semantic Indexing (EDLSI).

Kontostathis demonstrated good performance on a variety of collections by using only the first 10 dimensions of the SVD [15]. The model obtains final document scores by computing a weighted average of the traditional LSI score using a small value for k and the vector-space retrieval score. The result vector computation is

$$W = x(Q^T A_k) + (1 - x)(Q^T A),$$

where x is a weighting factor ($0 \leq x \leq 1$) and k is small.

In [15], parameter settings of $k = 10$ and $x = 0.2$ were shown to provide consistently good results across a variety of large and small collections studied. On average, EDLSI improved retrieval performance an average of 12% over vector-space retrieval. All collections showed significant improvements, ranging from 8% to 19%. Significant improvements over LSI were also noted in most cases. LSI outperformed EDLSI for $k = 10$ and $x = 0.2$ on only two small datasets, MED and CRAN. It is well known that LSI performs particularly well on these datasets. Optimizing k and x for these specific datasets restored the outperformance of EDLSI.

Furthermore, computation of only a few singular values and their associated singular vectors has a significantly reduced cost when compared to the usual 100–300 dimensions required for traditional LSI. EDLSI also requires minimal extra memory during query run time when compared to vector-space retrieval and much less memory than LSI [15].

4 Combining EDLSI with Updating

In this section, we apply five updating methods to an EDLSI system to determine the optimal retrieval and run-time approach for using LSI for search and retrieval when documents are added to a collection. The datasets used in our experiments are described in Section 4.1, and the metrics used to measure performance are described in Section 4.2. The experimental approach is described in Section 4.3.

4.1 Datasets The first set of datasets we chose were the small SMART datasets that have been often used in LSI research. These corpora are well understood and provide a sensible baseline. The list of datasets and their basic properties appear in Table 1. LSI is known to perform well with the first three of these datasets (MED, CISI, CRAN). LSI is known to not perform well on the fourth dataset, CACM. Standard queries and relevance judgment sets exist for all of these corpora.

For our second set of experiments, we chose two subsets of the TREC AQUAINT corpus [26]. These subsets, subsequently referred to as HARD-1 and HARD-2, contain 15,000 and 30,000 documents respectively. They were created by randomly selecting documents from the TREC AQUAINT track text collection with no document duplication within the subsets. Further details are also given in Table 1. The TREC HARD queries (1-50) from TREC 2003 were used for testing.

For our experiments we used the log-entropy weighting scheme, cosine normalization of the document vector, a standard English stop-list to prune frequently occurring words, and the Porter stemmer algorithm to reduce words to their roots [20].

4.2 Metrics The standard metrics for evaluating the effectiveness of an information retrieval algorithms are *precision* and *recall*.

The *precision*, P, is defined as the number of retrieved documents that are relevant divided by the

Table 1: Summary of collections tested.

Identifier	Description	Documents	Terms	Queries
MED	Medical abstracts	1,033	5,735	30
CISI	Information science abstracts	1,460	5,544	35
CRAN	Cranfield collection	1,398	4,563	225
CACM	Communications of the ACM	3,204	4,863	64
HARD-1	15,000 documents from TREC HARD	15,000	77,250	50
HARD-2	30,000 documents from TREC HARD	30,000	112,113	50

total number of documents that are retrieved [1]. This is expressed as

$$P = \frac{d_{rr}}{d_{tr}},$$

where d_{rr} is the number of retrieved documents that are relevant, and d_{tr} is the total number of retrieved documents.

The *recall*, R, is defined as the number of relevant document retrieved divided by the total number of relevant documents in the dataset [1]. This is expressed as

$$R = \frac{d_{rr}}{d_r},$$

where d_r is the total number of relevant documents.

When measuring recall during the updating process, we only count documents that have been 'seen' by the system. For example, even though MED has 1,033 documents, we may want to measure precision and recall after only 700 have been processed. A relevant document that has not yet been added to the LSI space is not counted in the denominator of the recall calculation.

We define the 11-point precision for each query set as follows. We measure the precision for each of the queries at 11 standard recall levels (0%, 10%, ... , 100%) and average the precision across recall levels. Precision at a given recall level is obtained by retrieving documents until the requested recall level is exceeded. The Mean Average Precision (MAP) is computed by averaging the 11-point precision across all test queries.

4.3 Methodology We partitioned each collection so that we have an initial set of documents to form the initial term-document matrix, with several batches of documents to be added incrementally. The collections were partitioned into an initial set containing 50% of the total documents and then augmented with increments containing 3% of the total documents (for a total of 17 updates). For each corpus and each method, we determined the optimal k for LSI and the optimal k and x for EDLSI. These values appear in Table 2.

The optimal parameters were determined as follows. For EDLSI, we tested values of k from 5 to 50 for the small collections and from 5 to 100 for HARD-1, both in increments of 5. We tested values of x from 0.1 to 0.9 in increments of 0.1. The weighting $x = 0$ is equivalent to vector space retrieval, and $x = 1.0$ is traditional LSI. For LSI, we used values of k from 25 to 200 for the small collections and from 25 to 500 for HARD-1, both in increments of 25. Due to the large size of the HARD-2 dataset, the optimal parameters for HARD-2 were simply taken to be those of the HARD-1 run. In all cases, if multiple runs resulted in the maximum MAP value, the run with the lowest k and/or x value was chosen as optimal.

Five different updating methods for dealing with increasing collection size were tested on each dataset for both LSI and EDLSI: recomputing the PSVD with each document set update, folding-in, PSVD updating, folding-up (percentage based), and adaptive folding-up.

5 Results

EDLSI with each of the five updating methods generally matched or exceeded the retrieval performance, as measured by MAP, when compared to LSI. Furthermore, EDLSI always exceeded the performance of LSI for a given updating method in terms of run time and memory requirements.

5.1 Retrieval Performance Table 3 reports the retrieval results for EDLSI and LSI for each of the collections and each of the updating methods. Negative numbers in the percent improvement column indicate that the retrieval performance of optimal LSI was better than that of optimal EDLSI. The only collection on which LSI outperforms EDLSI is MED, and this slight edge is not present when the folding-in or recomputing methods were used. On the CISI collection, EDLSI slightly outperforms LSI for all updating methods except folding-in, where there was no measurable differ-

Table 2: Empirically determined optimal k and x for all updating methods.

Method	**Optimal EDLSI k value**	**Optimal EDLSI x value**	**Optimal LSI k value**
MED			
Recomputing the PSVD	50	0.3	75
Folding-In	20	0.1	125
PSVD Updating	25	0.2	125
Percentage-Based Folding-Up	20	0.2	125
Adapting Folding-Up	25	0.2	125
CISI			
Recomputing the PSVD	25	0.1	50
Folding-In	5	0.1	75
PSVD Updating	25	0.1	50
Percentage-Based Folding-Up	35	0.1	50
Adaptive Folding-Up	25	0.1	50
CRAN			
Recomputing the PSVD	5	0.1	175
Folding-In	5	0.1	100
PSVD Updating	5	0.1	175
Percentage-Based Folding-Up	5	0.1	200
Adaptive Folding-Up	5	0.1	175
CACM			
Recomputing the PSVD	10	0.1	200
Folding-In	5	0.1	200
PSVD Updating	15	0.1	200
Percentage-Based Folding-Up	15	0.1	175
Adaptive Folding-Up	15	0.1	175
HARD-1			
Recomputing the PSVD	45	0.1	475
Folding-In	20	0.1	425
PSVD Updating	20	0.1	325
Percentage-Based Folding-Up	35	0.1	350
Adaptive Folding-Up	20	0.1	325

ence in retrieval performance (to two decimal places). EDLSI outperforms LSI, sometimes significantly, on the remaining four collections in each of the five updating methods, with MAP improvements ranging from 7.5 to 125 percent. The greatest MAP improvement was relative to folding-in, emphasizing the fact that although it is by far the least computationally inexpensive updating method, it is very reliable when EDLSI is used, but not reliable for LSI.

5.2 Run-time Performance The experiments were performed using Matlab 7.7.0.471 (R2008b) on a computer with 8 Intel Xeon L5420 2.5 GHz processors and 32GB RAM running 64-bit RedHat Linux.

We recall from Section 2 that the PSVD updating procedure uses the rank-k PSVD of $\tilde{A} \in \Re^{(k+p)\times(k+p)}$, where p is the number of documents added to the current term-document matrix. These PSVD components $\tilde{U}_k$, $\tilde{\Sigma}_k$, and $\tilde{V}_k$ of $\tilde{A}$ can be computed in Matlab in two ways. The first is to use the `svd` function, which computes the full SVD; the relevant PSVD information is then extracted from the full SVD. The second is to use the `svds` function, which computes the rank-k PSVD directly. The function `svds` is generally recommended when the PSVD of a large, sparse matrix is desired for small k. Beyond this heuristic, it is difficult to predict which function is more efficient for a given matrix. The results from either method to compute the PSVD of $\tilde{A}$ are identical in principle; the differences are only in run-time and memory required.

We ran tests with both `svds` and `svd` and report the minimum run times in Table 3. For the small collections, we found that using `svd` was faster for LSI. In this case k and p are comparable, so that k is significant relative to $k+p$ and $k+p$ is not large. Using `svds` is in general faster for EDLSI, where k is small. For the large collections, where p is larger than k, so that k is not significant relative to $k+p$ and $k+p$ is large, in all but one case using `svds` was faster for both LSI and EDLSI. Run times in Table 3 for which `svd` was faster than `svds` are denoted with an asterisk.

We use *speed-up* as our run-time performance measure. Speed-up is computed by dividing the LSI run time by the corresponding EDLSI run time. A value greater than 1 indicates that the use of EDLSI led to an improvement in run time. Optimal run times for a given experiment were determined using the minimum of the measured CPU times over multiple runs on a dedicated machine. Small collections were run 10 times; larger collections were run 3 times. In all cases, the timings measured showed good consistency within a given experiment. Because folding-in is effectively instantaneous on the small collections, the minimum CPU time observed was often 0.00; thus we report < 0.01 for this circumstance. We see that the use of EDLSI always reduced CPU time. The largest run-time improvements were obtained by using EDLSI with recomputing the PSVD, with a speed-up of as much as 610. The speed-up for folding-in was measured to be as high as 28.0. The speed-up for PSVD updating was as high as 16.8. Speed-ups for folding-up were as high as 8.8 for percentage-based folding-up and 10.9 for adaptive folding-up.

Significantly, EDLSI with folding-in achieved the best run-time performance, and it ran very quickly with almost no degradation in MAP compared with recomputing the PSVD.

In all cases, EDLSI with folding-up uses fewer CPU resources than traditional LSI (with recomputing) or LSI with folding-up. In fact, using EDLSI with adaptive folding-up instead of LSI with recomputing on the HARD-1 collection reduced CPU time required from 61,555.32 seconds to 456.84 seconds, a 99.3% reduction.

It appears that most of the benefit comes from the use of EDLSI (i.e., a reduction from 61,555.32 seconds to 937.55 seconds) because EDLSI requires only $k = 45$ dimensions. However the use of EDLSI with adaptive folding-up allows a further reduction to only $k = 20$ dimensions, further reducing CPU time to 456.84 seconds.

5.3 Memory Considerations The RAM requirements with EDLSI are also significantly smaller than for LSI. For example, EDLSI with updating or adaptive folding-up on HARD-2 at the optimal MAP level of $k = 20$ requires approximately 9 MB of RAM for the documents (30,000 documents $\times$ 20 dimensions $\times$ 16 bytes per double precision number) and approximately 34 MB for the terms. On the other hand, LSI for the same methods at the MAP optimal level of $k = 325$ requires a total of approximately 705 MB for the documents and terms. Hence the use of EDLSI requires only about 6% of the memory required by LSI.

5.4 Sensitivity of EDLSI to k and x Figures 1 and 2 show the retrieval performance of EDLSI and LSI respectively when k is varied across a range of values for HARD-1. The graphs show that EDLSI reaches its optimal retrieval performance at a much smaller value of k than LSI and that the retrieval performance does not change dramatically once the optimal level is passed. The retrieval performance of LSI, on the other hand, continues to rise significantly until it reaches its optimal level at $k = 325$, after which point it stabilizes. Noting in particular the scale of each graph, we see that even at the lowest k values, the retrieval performance of EDLSI

is quite stable and higher than LSI retrieval performance with its optimal k.

The corresponding graph when x is varied near its optimal value is shown in Figure 3. Once again we see that the retrieval performance of EDLSI does not vary significantly as x changes near the MAP optimal level of $x = 0.1$.

Figure 1: Retrieval performance of EDLSI with adaptive folding-up with varying k ($x = 0.1$)

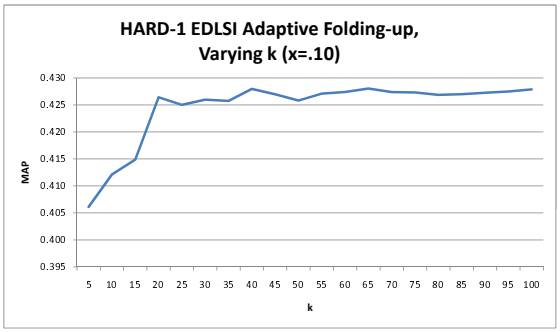


Figure 2: Retrieval performance of LSI with adaptive folding-up with varying k

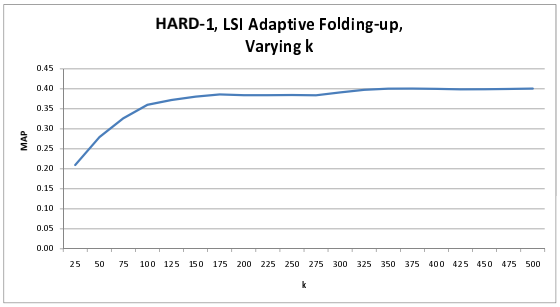
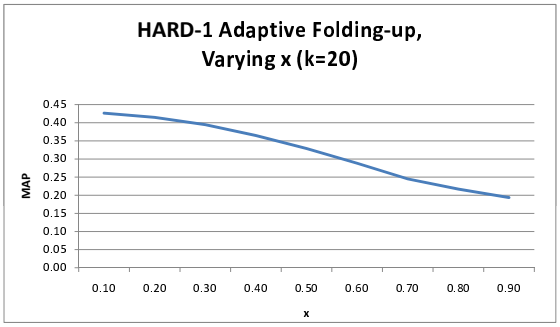


Figure 3: Retrieval performance of EDLSI with adaptive folding-up with varying x ($k = 20$)



6 Conclusions

EDLSI and updating are two techniques that enhance the performance of LSI. EDLSI is a convex combination of LSI and vector retrieval. It is generally able to produce retrieval results that meet or exceed those of traditional LSI with a semantic space of much smaller dimension, hence leading to a substantial decrease in computing resources required. EDLSI is also less sensitive than LSI to the choice of the semantic space dimension k. Updating methods are able to handle large collections efficiently by incrementally maintaining the integrity of the semantic space as new content is added.

In this paper, we tested EDLSI on 6 different data sets with a range of sizes and known performances with LSI. We also used 5 different LSI updating procedures: recomputing, folding-in, PSVD updating, percentage-based folding-up, and adaptive folding-up.

We found that the optimal EDLSI MAP consistently matched or improved upon LSI, even on data sets that on which LSI was known to perform well. When a collection is growing, e.g., as documents are added, the use of EDLSI provides significant run-time and memory improvements without loss of retrieval accuracy for all 5 updating methods tested.

The results from using EDLSI with folding-in were particularly surprising and interesting. Only slightly degraded retrieval accuracy was measured when EDLSI with folding-in was used, and the run-time improvements were substantial. However, the issue of automatically detecting when folding-in provides reliable updating remains an open problem.

References

[1] R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[2] M. W. Berry. Large-scale sparse singular value computations. *The International Journal of Supercomputer Applications*, 6(1):13–49, Spring 1992.

[3] M. W. Berry, T. Do, G. O'Brien, V. Krishna, and S. Varadhan. SVDPACKC (version 1.0) User's Guide. Technical report, University of Tennessee, 1993.

[4] M. W. Berry, Z. Drmac, and E. R. Jessup. Matrices, vector spaces, and information retrieval. *SIAM Rev.*, 41(2):335–362, 1999.

[5] M. W. Berry, S. T. Dumais, and G. W. O'Brien. Using linear algebra for intelligent information retrieval. *SIAM Review*, 37(4):575–595, 1995.

[6] M. W. Berry and D. I. Martin. Principal component analysis for information retrieval. *Handbook of Parallel Computing and Statistics*, pages 399–413, 2005.

[7] C.-M. Chen, N. Stoffel, M. Post, C. Basu, D. Bassu, and C. Behrens. Telcordia LSI engine: implementation and scalability issues. In *Proceedings of the Eleventh International Workshop on Research Issues in Data Engineering (RIDE 2001)*, Heidelberg, Germany, Apr. 2001.

[8] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by Latent Semantic Analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[9] S. T. Dumais. Improving the retrieval of information from external sources. *Behavior Research Methods, Instruments, & Computers*, 23(2):229–236, 1991.

[10] S. T. Dumais. LSI meets TREC: A status report. In D. Harmon, editor, *The First Text REtrieval Conference (TREC1), National Institute of Standards and Technology Special Publication 500-207*, pages 137–152, 1992.

[11] S. T. Dumais. Latent semantic indexing (LSI) and TREC-2. In D. Harmon, editor, *The Second Text REtrieval Conference (TREC2), National Institute of Standards and Technology Special Publication 500-215*, pages 105–116, 1993.

[12] C. Eckart and G. Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, September 1936.

[13] J. Gao and J. Zhang. Sparsification strategies in latent semantic indexing. In M. W. Berry and W. M. Pottenger, editors, *Proceedings of the 2003 Text Mining Workshop*, May 2003.

[14] E. R. Jessup and J. H. Martin. Taking a new look at the latent semantic analysis approach to information retrieval. In *Computational Information Retrieval*, pages 121–144, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[15] A. Kontostathis. Essential dimensions of latent semantic indexing (LSI). *Proceedings of the 40th Hawaii International Conference on System Sciences*, 2007.

[16] A. Kontostathis, W. M. Pottenger, and B. D. Davison. Identification of critical values in latent semantic indexing. In T. Lin, S. Ohsuga, C. Liau, X. Hu, and S. Tsumoto, editors, *Foundations of Data Mining and Knowledge Discovery*, pages 333–346. Spring-Verlag, 2005.

[17] T. A. Letsche and M. W. Berry. Large-scale information retrieval with Latent Semantic Indexing. *Information Sciences*, 100(1-4):105–137, 1997.

[18] J. E. Mason and R. J. Spiteri. A new adaptive folding-up algorithm for information retrieval. *Proceedings of the Text Mining Workshop 2007*, 2008.

[19] G. W. O'Brien. Information tools for updating an svd-encoded indexing scheme. Master's thesis, University of Tennessee, 1994.

[20] M. F. Porter. An Algorithm for Suffix Stripping. In *Readings in information retrieval*, pages 313–316. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[21] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Inf. Process Manage.*, 24(5):513–523, 1988.

[22] SMART. ftp://ftp.cs.cornell.edu/pub/smart/, 2009.

[23] J. E. Tougas and R. J. Spiteri. Updating the partial singular value decomposition in latent semantic indexing. *ScienceDirect, Computational Statistics & Data Analysis*, 52:174–183, 2006.

[24] J. E. Tougas and R. J. Spiteri. Two uses for updating the partial singular value decomposition in latent semantic indexing. *ScienceDirect, Applied Numerical Mathematics*, 58:499–510, 2007.

[25] J. E. B. Tougas. Folding-up: A hybrid method for updating the partial singular value decomposition in latent semantic indexing. Master's thesis, Dalhousie University, 2005.

[26] TREC HARD. http://ciir.cs.umass.edu/research/hard/guidelines.html, 2009.

[27] S. Zelikovitz and H. Hirsh. Using LSI for text classification in the presence of background text. In H. Paques, L. Liu, and D.Grossman, editors, *Proceedings of CIKM-01, 10th ACM International Conference on Information and Knowledge Management*, pages 113–118, 2001.

[28] H. Zha and H. D. Simon. On updating problems in latent semantic indexing. *SIAM J. Sci. Comput., 21(2)*, pages 782–791, 1999.

Table 3: Retrieval and run-time comparisons for all collections.

Method	Optimal EDLSI MAP	Optimal LSI MAP	Percent Improve-ment	Optimal EDLSI Run Time	Optimal LSI Run Time	EDLSI Speed Up
MED						
Recomputing the PSVD	0.43	0.42	2.3	27.15	93.30	3.44
Folding-In	0.40	0.30	33.3	< 0.01	0.08	> 8.00
PSVD Updating	0.42	0.43	-2.3	$*0.51$	$*4.14$	8.12
Percentage-Based Folding-Up	0.42	0.43	-2.3	$*0.71$	$*2.60$	3.66
Adaptive Folding-Up	0.42	0.43	-2.3	$*0.68$	$*2.69$	3.96
CISI						
Recomputing the PSVD	0.10	0.09	11.1	6.38	26.74	4.19
Folding-In	0.09	0.09	0	< 0.01	0.01	> 1.00
PSVD Updating	0.10	0.09	11.1	$*0.77$	$*1.31$	1.70
Percentage-Based Folding-Up	0.10	0.09	11.1	1.34	$*1.63$	1.22
Adaptive Folding-Up	0.10	0.09	11.1	1.13	$*1.60$	1.42
CRAN						
Recomputing the PSVD	0.12	0.11	9.1	1.19	386.84	325.08
Folding-In	0.12	0.09	33.3	0.01	0.07	7.00
PSVD Updating	0.12	0.11	9.1	$*0.43$	$*7.22$	16.79
Percentage-Based Folding-Up	0.12	0.11	9.1	0.76	$*6.66$	8.76
Adaptive Folding-Up	0.12	0.11	9.1	0.51	$*5.58$	10.94
CACM						
Recomputing the PSVD	0.19	0.15	26.7	3.66	1,040.11	284.18
Folding-In	0.18	0.08	125.0	0.01	0.28	28.00
PSVD Updating	0.19	0.14	35.7	2.18	$*22.89$	10.50
Percentage-Based Folding-Up	0.19	0.13	46.2	2.73	$*14.04$	5.14
Adaptive Folding-Up	0.19	0.13	46.2	2.58	$*12.45$	4.83
HARD-1						
Recomputing the PSVD	0.43	0.40	7.5	937.55	61,555.32	65.66
Folding-In	0.42	0.37	13.5	1.29	7.91	6.13
PSVD Updating	0.43	0.40	7.5	434.04	$*837.43$	1.93
Percentage-Based Folding-Up	0.43	0.40	7.5	691.63	1,314.26	1.90
Adaptive Folding-Up	0.43	0.40	7.5	456.84	1,204.04	2.64
HARD-2						
Recomputing the PSVD	0.34	0.29	17.2	1,636.86	99,106.30	610.38
Folding-In	0.33	0.28	17.9	2.45	15.42	6.29
PSVD Updating	0.34	0.29	17.2	1,723.05	2,905.00	1.69
Percentage-Based Folding-Up	0.34	0.30	13.3	3,475.91	4,956.75	1.43
Adaptive Folding-Up	0.34	0.29	17.2	1,857.21	4,395.03	2.37