Information Management Tools for Updating an SVD-Encoded Indexing Scheme

Gavin W. O'Brien

Computer Science Department

CS-94-258 October 1994

Information Management Tools for Updating an SVD-Encoded Indexing Scheme

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Gavin W. O'Brien

December 1994

Acknowledgments

I thank my thesis advisor, Dr. Michael Berry, for his support and guidance. I appreciate his patience and advice in all aspects of creating this thesis. I am grateful to Dr. Susan Dumais from Bellcore and Sowmini Varadhan for their technical advice. I also thank my committee members Dr. Brad Vander Zanden and Dr. David Straight. Finally, I must thank the people who have help motivate me in completing my Masters degree in a timely manner, my parents, Dr. William and Margaret O'Brien and Michelle Johnson.

This research has been supported in part by Apple Computer through Contract No. C24-9100120 and by the National Science Foundation under Grant No. NSF-ASC-92-03004.

Abstract

Lexical-matching methods for information retrieval can be inaccurate when they are used to match a user's queries. Typically, information is retrieved by literally matching terms in documents with those of the query. The problem is that users want to retrieve on the basis of conceptual topic or meaning of a document. There are usually many ways to express a given concept (synonymy), so the literal terms in a user's query may not match those of a relevant document. In addition, most words have multiple meanings (polysemy), so terms in a user's query will literally match terms in irrelevant documents. The implicit high-order structure of associating terms with documents can be exploited by the singular value decomposition (SVD). Latent Semantic Indexing (LSI) is a conceptual indexing technique which uses the SVD to estimate the underlying latent semantic structure of the word to document association. By computing a lower-rank approximation to the original term-document matrix, LSI dampens the effects of word choice variability by representing terms and documents using the (orthogonal) left and right singular vectors.

Current methods for adding new text to an LSI database can have deteriorating effects on the orthogonality of the vectors used to represent terms and documents in high-dimensional subspaces. Updating the SVD so as to preserve the orthogonality among document vectors corresponding to the new term-document matrix is one remedy. Computing the SVD of the new term-document matrix can be avoided by using SVDPACKC routines for appropriate submatrices constructed from existing term and document vectors and similar vectors corresponding to the new text. The cost of the numerical computations needed to update the SVD versus the potential inaccuracy of simply folding-in text presents an interesting tradeoff for LSI database management.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Typically, information is retrieved by literally matching terms in documents with those of a query. However, lexical-matching methods for information retrieval can be inaccurate when they are used to match a user's query. Since there are usually many ways to express a given concept (synonymy), the literal terms in a user's query may not match those of a relevant document. In addition, most words have multiple meanings (polysemy), so terms in a user's query will literally match terms in irrelevant documents. A better approach would allow users to retrieve information on the basis of a conceptual topic or meaning of a document.

Latent semantic indexing (LSI) [DDF$^+$90] tries to overcome the problems of lexical-matching by assuming there is some underlying latent semantic structure of word usage data that is partially obscured by the variability of word choice. LSI is a conceptual-indexing technique which uses the singular value decomposition (SVD) [GL89] to estimate the underlying latent semantic structure of word to document as-

sociation. This implicit high-order structure of associating terms with documents can be exploited by the SVD. A number of software tools have been developed to perform operations such as: parsing, query matching, and adding additional terms or documents to an existing LSI-generated database. The bulk of the LSI process involves computing a truncated SVD of sparse term by document matrices which are used to model the latent semantic structure, and the creation of a keyword database. The SVD combined with the software tools mentioned above comprise an LSI-generated database.

Current methods for adding new text to an LSI-generated database can have deteriorating effects on both the SVD and retrieval of information. This thesis develops a new technique for adding text which maintains the integrity of the SVD and improves information retrieval.

1.2 Statement of Problem

Folding-in text, the current method for updating an LSI-generated database, is an inexpensive but potentially inaccurate way of updating. Folding-in does not recompute the latent semantic model, via the SVD, but instead applies the existing model to the new terms and documents. This can have deteriorating effects on retrieval if the new text uses words differently than existing text. The new word usage data is potentially lost or misrepresented. SVD-updating, the method developed in this thesis, properly updates the current SVD-generated model allowing new word usage data to perturb the existing model.

1.3 Overview

Chapter 2 is a review of basic concepts needed to understand LSI. Chapter 3 uses a constructive example to illustrate how LSI represents terms and documents in the same semantic space, how a query is represented, how additional documents are folded-in, and how SVD-updating represents additional documents. Chapter 4 comprises a detailed discussion of the algorithm used in SVD-updating, while Chapter 5 compares both methods of updating, folding-in and SVD-updating, with regard to robustness of query matching and computational complexity.

Chapter 2

Preliminaries

2.1 Singular Value Decomposition

The singular value decomposition is commonly used in the solution of unconstrained linear least squares problems, matrix rank estimation, and canonical correlation analysis [Ber92a]. Given an $m \times n$ matrix A, where without loss of generality $m \geq n$ and $\text{rank}(A) = r$, the singular value decomposition of A, denoted by $\text{SVD}(A)$, is defined as

$$A = U\Sigma V^T \tag{2.1}$$

where $U^TU = V^TV = I_n$ and $\Sigma = \text{diag}(\sigma_1, \cdots, \sigma_n), \sigma_i > 0$ for $1 \leq i \leq r, \sigma_j = 0$ for $j \geq r+1$. The first r columns of the orthogonal matrices U and V define the orthonormal eigenvectors associated with the r nonzero eigenvalues of AA^T and A^TA, respectively. U and V are referred to as the left and right singular vectors, respectively. The singular values of A are defined as the diagonal elements of Σ which are the nonnegative square roots of the n eigenvalues of AA^T [GL89].

The following two theorems illustrate how the SVD can reveal important informa-

tion about the structure of a matrix.

Theorem 1. Let the SVD of A be given by Equation (2.1) and

$$\sigma_1 \geq \sigma_2 \cdots \geq \sigma_r > \sigma_{r+1} = \cdots = \sigma_n = 0$$

and let $R(A)$ and $N(A)$ denote the range and null space of A, respectively. Then

1. rank property: $\text{rank}(A) = r$, $N(A) \equiv \text{span}\{v_{r+1}, \cdots, v_n\}$, and $R(A) \equiv \text{span}\{u_1, \cdots, u_r\}$, where $U = [u_1 u_2 \cdots u_m]$ and $V = [v_1 v_2 \cdots v_n]$.
2. dyadic decomposition: $A = \sum_{i=1}^{r} u_i \cdot \sigma_i \cdot v_i^T$.
3. norms: $\|A\|_F^2 = \sigma_1^2 + \cdots + \sigma_r^2$, and $\|A\|_2^2 = \sigma_1$.

Theorem 2. [Eckart and Young] Let the SVD of A be given by Equation (2.1) with $r = \text{rank}(A) \leq p = \min(m, n)$ and define

$$A_k = \sum_{i=1}^{k} u_i \cdot \sigma_i \cdot v_i^T \tag{2.2}$$

$$\min_{rank(B)=k} \|A - B\|_F^2 = \|A - A_k\|_F^2 = \sigma_{k+1}^2 + \cdots + \sigma_p^2.$$

(For a proof, see [GR71].) In other words, A_k, which is constructed from the k-largest singular triplets of A, is the closest rank-k matrix to A [GL89]. In fact, A_k is the best approximation to A for any unitarily invariant norm [Mir60]. Hence,

$$\min_{rank(B)=k} \|A - B\|_2 = \|A - A_k\|_2 = \sigma_{k+1}. \tag{2.3}$$

2.2 Latent Semantic Indexing

In order to implement Latent Semantic Indexing [DDF+90] a matrix of terms by documents must be constructed. The elements of the term-document matrix are the

occurrences of each word in a particular document, i.e.,

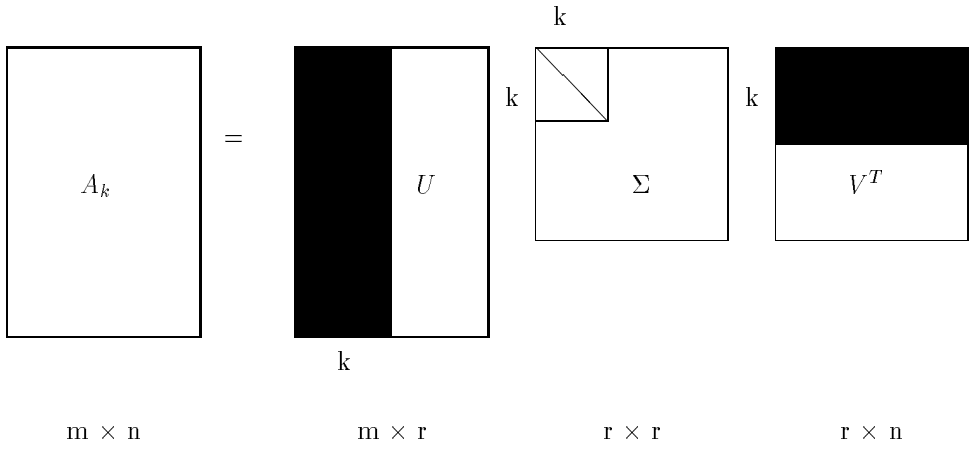
$$A = [a_{ij}], \tag{2.4}$$

where a_{ij} denotes the frequency in which term i occurs in document j. Since every word does not normally appear in each document, the matrix A is usually sparse. In practice, local and global weightings are applied [Dum91] to increase/decrease the importance of terms within or among documents. Specifically, we can write

$$a_{ij} = L(i,j) \times G(i), \tag{2.5}$$

where $L(i,j)$ is the local weighting for term i in document j, and $G(i)$ is the global weighting for term i. The matrix A is factored into the product of 3 matrices (Equation (2.1)) using the singular value decomposition (SVD). The SVD derives the latent semantic structure model from the orthogonal matrices U and V containing left and right singular vectors of A, respectively, and the diagonal matrix, Σ, of singular values of A. These matrices reflect a breakdown of the original relationships into linearly-independent vectors or *factor values*. The use of k factors or k-largest singular triplets is equivalent to approximating the original (and somewhat unreliable) term-document matrix by A_k in Equation (2.2). In some sense, the SVD can be viewed as a technique for deriving a set of uncorrelated indexing variables or factors, whereby each term and document is represented by a vector in k-space using elements of the left or right singular vectors.

In using A_k as an approximation to the original matrix A it is possible for documents with different sets of terms to be mapped into the same vector. These vectors represent extracted, common *meaning* relationships of many different terms and documents. Each term and document is represented by a vector of weights indicating the magnitude of association with each of the underlying concepts. The *meaning* of a



A_k	= Best rank-k approximation to A	m	= number of terms
U	= Term Vectors	n	= number of documents
Σ	= Singular Values	k	= number of factors
V	= Document Vectors	r	= rank of A

Figure 2.1: Mathematical representation of the matrix A_k from Equation (2.2).

particular term, query, or document can be expressed as a vector. The meaning representation is a reduction from the original n vectors to the new $k < n$ best surrogates by which they can be approximated.

It is important for the method that the derived k-dimensional factor space not reconstruct the original term-document matrix A perfectly, because we believe the original term-document matrix A to contain noise due to word choice. Rather, we want a derived structure, that expresses what is reliable and important in the underlying use of terms as document referents.

Figure 2.1 is a mathematical representation of the singular value decomposition. U and V are considered the term and document vectors respectively, and Σ represents the singular values. The shaded regions in U and V and the diagonal line in Σ represent A_k from Equation (2.2).

2.3 Queries

A query (or a set of words) can be considered as just another document, which can be represented as a vector in the k-dimensional space. As a *pseudo-document*, a query is simply a weighted sum of its component term vectors, which can be compared against all existing documents. A query vector (q_v) is defined as a vector of its weighted terms. For example, the query vector (q_v) can be represented mathematically as a pseudo-document (q) via

$$q = q_v^T U_k \Sigma_k^{-1}. \tag{2.6}$$

The (conceptually) nearest document vectors to the pseudo-document can be returned. Since an ordered list of all the documents can be returned, a measure for closeness must be set. One measure of nearness is the cosine. Two types of ordered listings exist, either a threshold is set for the cosine and all documents above it are returned or the z closest documents are returned [DDF+90].

2.4 Folding-In

Folding-in has been the only updating technique considered thus far for LSI. Suppose an LSI-generated database already exists. That is, a body of text has been parsed, a term-document matrix has been generated, and the SVD of the term-document matrix has been computed. If more terms and documents must be added, two alternatives for incorporating them currently exist: recomputing the SVD of a new term-document matrix or *folding-in* the new terms and documents.

Three terms are defined below to avoid confusion when discussing updating. *Updating* refers to the general process of adding new terms and/or documents to an exist-

ing LSI-generated database. Updating can mean either folding-in or SVD-updating. *SVD-updating* is the new method of updating developed in this thesis. *Recomputing the SVD* is not an updating method, but a way of creating an LSI-generated database with new terms and/or documents from scratch which can be compared to either updating method.

Recomputing the SVD of a larger term-document matrix requires more computation time and, for large problems, may be impossible due to memory constraints. Folding-in requires less time and memory but can have deteriorating effects on the representation of the new terms and documents. Recomputing the SVD allows the new t terms and d documents to directly affect the latent semantic structure by creating a new term-document matrix $A^{(m+t)\times(n+d)}$, computing the SVD of the new term-document matrix, and generating a different A_k matrix. In contrast, folding-in is based on the existing latent semantic structure, (the current A_k), and hence new terms and documents have no effect on the representation of the pre-existing terms and documents.

Folding-in documents is essentially the process described in Section 2.3 for pseudo-document representation of queries. Each new document is represented as a weighted sum of its component term vectors. Once a new document vector has been computed it is appended to the set of existing document vectors or columns of V_k (see Figure 2.2).

Similarly, new terms can be represented as a weighted sum of their component document vectors. Once the term vector has been computed it is appended to the set of existing term vectors or columns of U_k (see Figure 2.3).

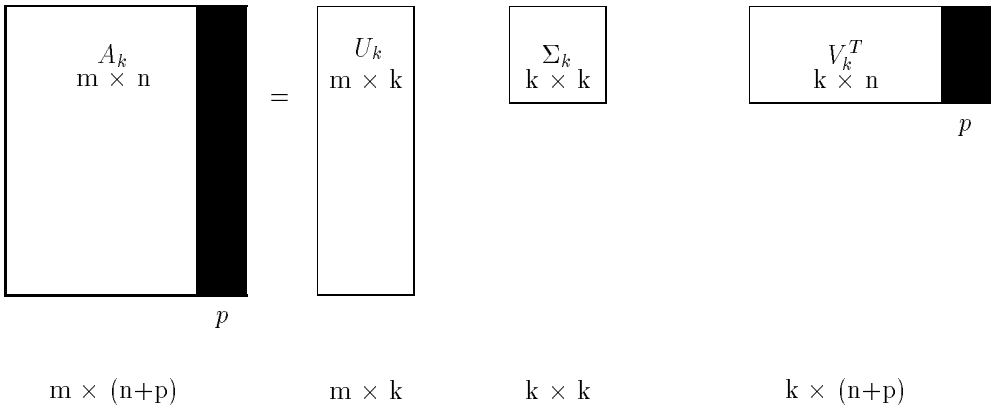


Figure 2.2: Mathematical representation of folding-in p documents.

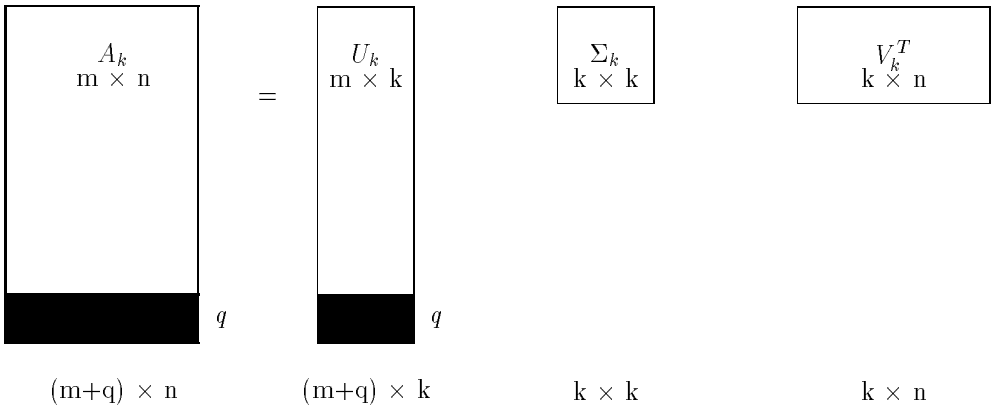


Figure 2.3: Mathematical representation of folding-in q terms.

To fold-in a new $m \times 1$ document vector, d, into an existing LSI model, the corresponding term vector d_t is first determined. Like a query vector, d_t is defined as a vector of its weighted terms. Then, a projection, d_p, of d onto the span of the current document vectors (columns of V_k) is computed by

$$d_p = d_t V_k \Sigma_k^{-1}. \tag{2.7}$$

Similarly, to fold-in a new $n \times 1$ term vector, t, into an existing LSI model, the corresponding document vector t_d must be derived. t_d is defined as a vector of its weighted documents. Then, a projection of t, t_q, onto the span of the current term vectors (columns of U_k) is determined by

$$t_q = t_d U_k^T \Sigma_k^{-1}. \tag{2.8}$$

Chapter 3

Latent Semantic Indexing and Updating Example

3.1 Latent Semantic Indexing

In this chapter an example of LSI and the folding-in process is presented. The titles (see Table 3.1) used to demonstrate the LSI process are extracted from the test case in [DDF+90]. These titles are based on two topics of memorandum from Bellcore: the "c" documents refer to human-computer interaction and the "m" documents refer to group theory. All underlined words are considered significant if they appear in more than one title.

Corresponding to the text in Table 3.1 is the 12 × 9 term-document matrix shown in Table 3.2. The elements of this matrix are the frequencies in which a term occurs in a document or title (see Section 2.4). For example, in title **c1**, the first column of the term-document matrix, *human*, *interface*, and *computer* all occur once. The SVD of the 12 × 9 term-document matrix is then computed. Selecting k=2, the best

Table 3.1: Original Data

Document Id	Titles
c1	Human Machine Interface for Lab ABC Computer Applications
c2	A Survey of User Opinion of Computer Systems Response Time
c3	The EPS User Interface Management Systems
c4	Systems and Human Systems Engineering Testing of EPS-2
c5	Relation of User-Perceived Response Time to Error Measurement
m1	The Generation of Random, Binary, Unordered Tree
m2	Intersection Graph of Paths in a Tree
m3	Graph Minors IV: Tree-Width and Well-Quasi-Ordering
m4	Graph Minors - A Survey

Table 3.2: Term-Document Matrix of Original Data

Terms	Documents								
	c1	c2	c3	c4	c5	m1	m2	m3	m4
human	1	0	0	1	0	0	0	0	0
interface	1	0	1	0	0	0	0	0	0
computer	1	0	0	0	0	0	0	0	0
user	0	1	1	0	1	0	0	0	0
system	0	1	1	2	0	0	0	0	0
response	0	1	0	0	1	0	0	0	0
time	0	1	0	0	1	0	0	0	0
EPS	0	0	1	1	0	0	0	0	0
survey	0	1	0	0	0	0	0	0	1
trees	0	0	0	0	0	1	1	1	0
graph	0	0	0	0	0	0	1	1	1
minors	0	0	0	0	0	0	0	1	1

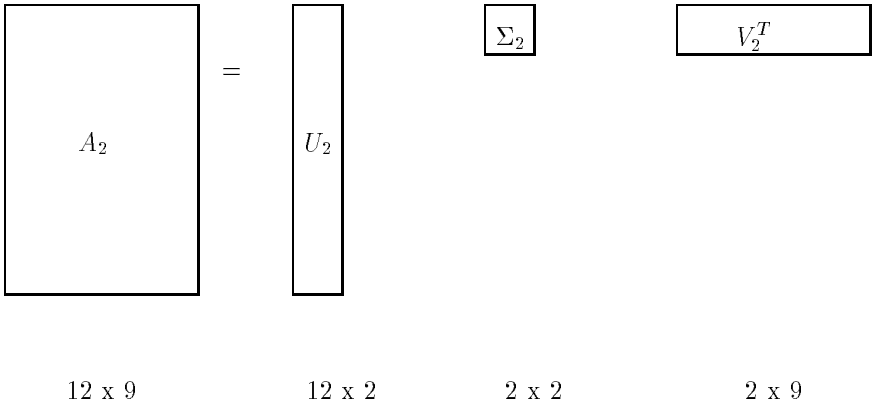


Figure 3.1: Best rank-2 approximation to A using 2-largest singular triplets.

rank-2 approximation to A is illustrated in Figure 3.1.

Using the first column of U_2 multiplied by the first singular value, σ_1, for the x-coordinates and the second column of U_2 multiplied by the second singular value, σ_2, for the y-coordinates, the terms can be represented on the Cartesian plane. Similarly, the first column of V_2 scaled by σ_1 are the x-coordinates and the second column of V_2 scaled by σ_2 are the y-coordinates for the documents. Figure 3.2 is a Two-dimensional plot of the terms and documents for the 12×9 example.

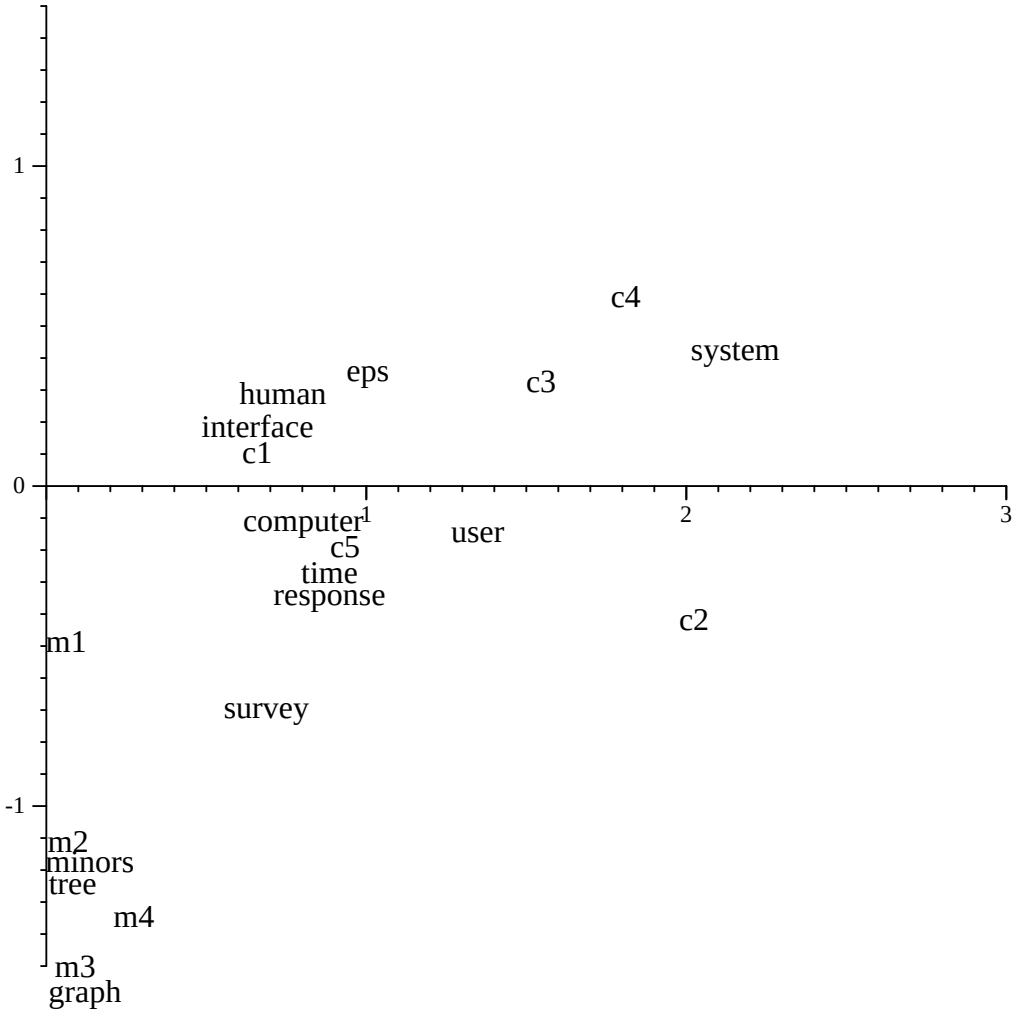


Figure 3.2: Two-dimensional plot of terms and documents for the 12 $\times$ 9 example.

$$\begin{pmatrix} 0.1383 & 0.0275 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}^T \begin{pmatrix} 0.2405 & -0.0432 \\ 0.3008 & 0.1413 \\ 0.0361 & -0.6228 \\ 0.2214 & 0.1132 \\ 0.1976 & 0.0721 \\ 0.0318 & -0.4505 \\ 0.2650 & -0.1072 \\ 0.2059 & -0.2736 \\ 0.6445 & -0.1673 \\ 0.0127 & -0.4902 \\ 0.2650 & -0.1072 \\ 0.4036 & -0.0571 \end{pmatrix} \begin{pmatrix} 3.3409 & 0 \\ 0 & 2.5417 \end{pmatrix}^{-1}$$

Figure 3.3: Derived coordinates for the query of *human computer*.

Notice the documents and terms pertaining to human computer interaction are clustered around the x-axis and the graph theory-related terms and documents are clustered around the y-axis. Such groupings suggest that documents **c1** through **c5** are similar in meaning and that documents **m1** through **m4** also have similar meaning.

3.2 Queries

Suppose we are interested in the documents that pertain to *human computer interaction*. Recall that a query vector (q_v) is represented as a pseudo document (q) via $q = q_v^T U_k \Sigma_k^{-1}$ (see Equation (2.6)). *Interaction* is not an indexed term in the database so it is omitted from the query leaving *human computer*. Mathematically, the Cartesian coordinates of the query are determined according to Figure 3.3.

This query vector is then compared (in the Cartesian plane) to all the documents in the LSI-generated database. All documents whose cosine with the query vector is greater than .90 is illustrated in the shaded region of Figure 3.4.

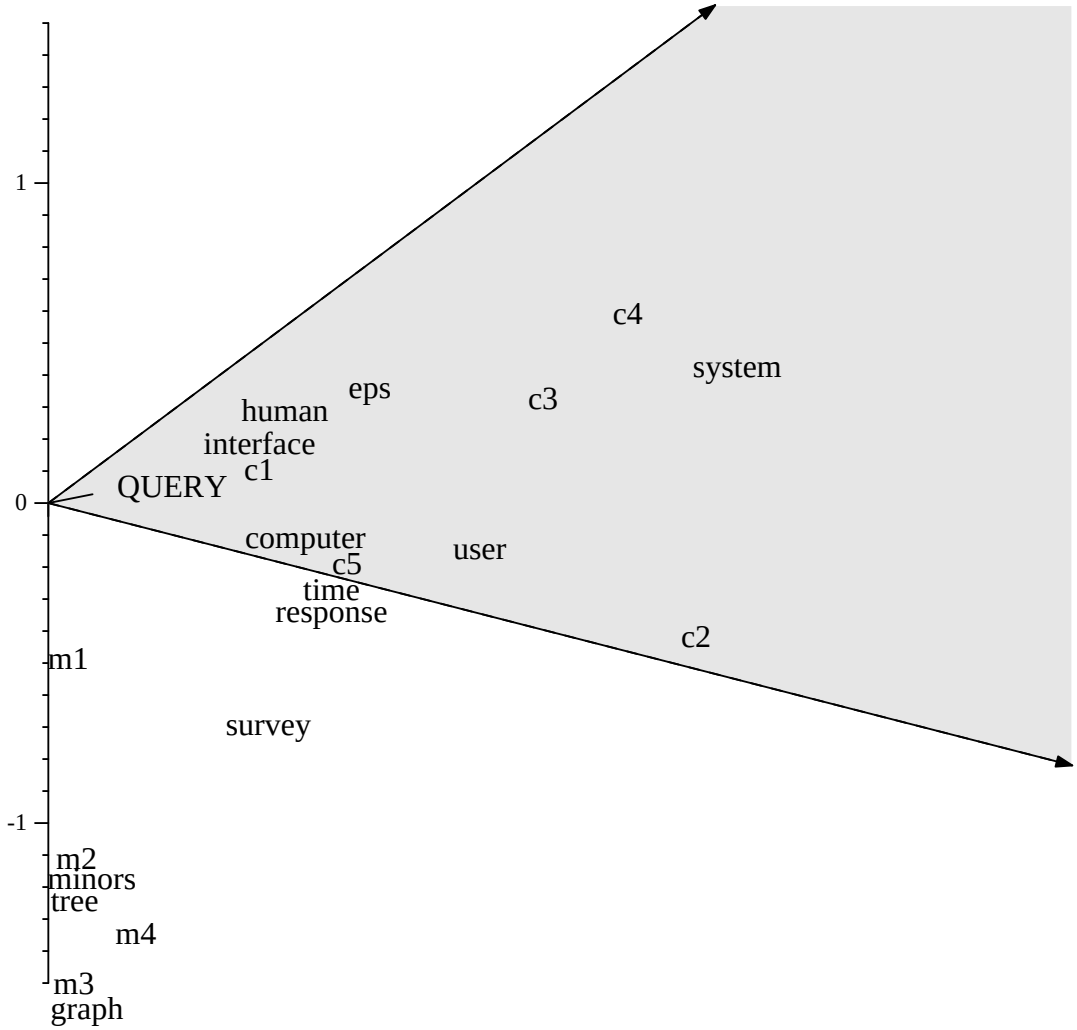


Figure 3.4: A Two-dimensional plot of terms and documents along with the query *human computer*.

Table 3.3: Returned documents based on different numbers of LSI factors.

Number of Factors					
$k = 2$		$k = 4$		$k = 9$	
c3	.99	c1	.99	c1	.88
c1	.99	c3	.39	c4	.31
c4	.98	c2	.30	c2	.31
c2	.93	c4	.22		
c5	.90				

A different cosine threshold, of course, could have been used so that a larger or smaller set of documents would be returned. The cosine is merely used to rank-order documents and its explicit value is not always an adequate measure of relevance. This phenomenon will be illustrated in the next section.

3.3 Comparison with Lexical Matching

In this example, LSI has been applied using 2 factors (i.e. A_2 is used to approximate the original 12×9 term-document matrix). Using a cosine threshold of .90, all five documents related to *human computer interaction* were returned: documents **c1**, **c2**, **c3**, **c4**, and **c5**. With lexical-matching, only three documents (**c1**, **c2**, **c4**) are returned. Hence, the LSI approach extracts two additional documents (**c3** and **c5**) which are relevant to the query yet share no common terms. Table 3.3 lists the LSI-ranked documents with different numbers of factors (k). The documents returned in Table 3.3 satisfy a cosine threshold of .20, i.e. returned documents are within a cosine of .20 of the pseudo-document used to represent the query.

Table 3.4: Additional titles for updating.

Document Id	Titles
n1	System Time to Traverse a B-Tree Graph
n2	Interface Graph Tools
n3	Graph Minors Implemented on Computer Systems
n4	Systems Tree
n5	Computer Graph
n6	Survey of Computer Time
n7	A Survey of Human Interface Computer Systems

3.4 Folding-In

Suppose the fictitious titles listed in Table 3.4 are to be updated to the original set of titles in Table 3.1. While some titles in Table 3.4 use terms from both the human computer interaction and group theory categories, others use only terms from each separate category. As with Table 3.1, all underlined words in Table 3.4 are considered significant since they appear in more than one title (across all 16 titles from Tables 3.1 and 3.4). Folding-in (see Section 2.4) is one approach for updating the original LSI-generated database with the 7 new titles. Figure 3.5 demonstrates how these titles are folded-into an LSI-generated database based on $k = 2$ factors. The new documents are denoted on the graph by their document id's. Notice that the coordinates of the original titles stay fixed, and hence the new data has no effect on the clustering of existing terms or documents.

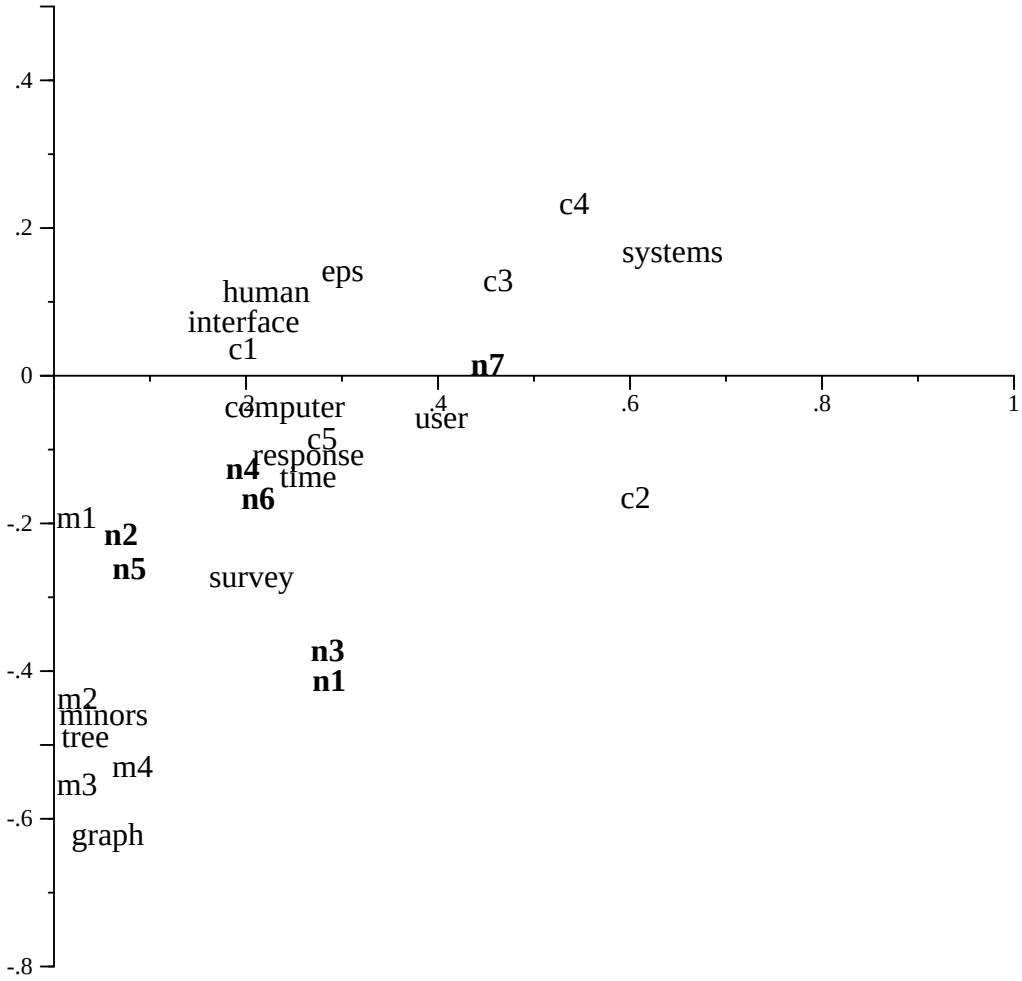


Figure 3.5: Two-dimensional plot of folded-in documents.

3.5 Recomputing the SVD

Ideally, the most robust way to produce the best rank-k approximation (A_k) to an LSI-generated database which has been added new terms and documents is to simply compute the SVD of a reconstructed term-document matrix, say $\tilde{A}$. Updating methods which can approximate the SVD of the larger term-document matrix $\tilde{A}$ become attractive in the presence of memory or time constraints. Therefore, the accuracy of SVD-updating approaches will be compared to that obtained when the SVD of $\tilde{A}$ is explicitly computed.

Suppose the titles from Table 3.4 are combined with those of Table 3.1 in order to create a new 12 $\times$ 16 term-document matrix $\tilde{A}$. Following Figure 3.1, we then construct the best rank-2 approximation to $\tilde{A}$,

$$\tilde{A}_2 = \tilde{U}_2 \tilde{\Sigma}_2 \tilde{V}_2^T. \tag{3.1}$$

Figure 3.6 is a 2-dimensional plot of the 12 terms and 16 documents using the elements of $\tilde{U}_2$ and $\tilde{V}_2$ for term and document coordinates, respectively. Notice the difference in term and document positions between Figures 3.5 and 3.6. Clearly, the new terms and documents from Table 3.4 have helped redefine the underlying latent structure when the SVD of $\tilde{A}$ is computed. Folding-in the 7 new documents based on the existing rank-2 approximation to A (defined by Table 3.2) may not accurately reproduce the true LSI representation of the new LSI-generated database.

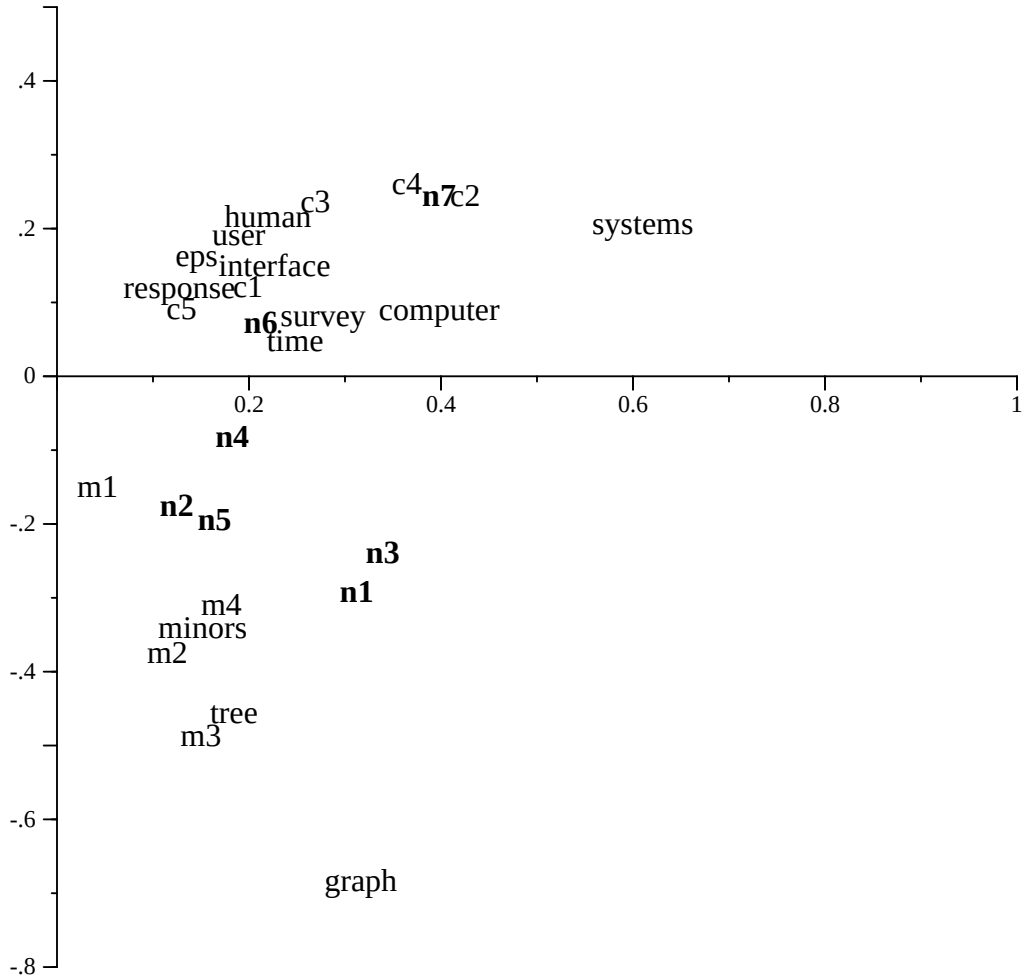


Figure 3.6: Two-dimensional plot of terms and documents using the SVD of a reconstructed term-document matrix.

Chapter 4

SVD-Updating

Chapter 4 describes SVD-updating using titles from Tables 3.1 and 3.4. The three steps required to perform a complete SVD-update involve adding new documents, adding new terms, and correction for changes in term weightings. These steps are not restricted to the ordering presented in the section below, but it is a logical approach to SVD-updating.

4.1 Overview of SVD-updating

This section is an overview of the three steps used to SVD-update an LSI-generated database. Let D denote the d new document vectors to process, then D is an $m \times d$ sparse matrix since most terms (as was the case with the original term-document matrix A) do not occur in each document. D is appended to the columns of the rank-k approximation of the $m \times n$ matrix A, i.e., from Equation (2.2), A_k so that the k-largest singular values and corresponding singular vectors of

$$B = (A_k \mid D) \tag{4.1}$$

are computed. This is almost the same process as recomputing the SVD, only A is replaced by A_k.

Let T denote a collection of $t \times 1$ term vectors for SVD-updating. Then T is a $t \times n$ sparse matrix, since each term rarely occurs in every document. T is then appended to the rows of A_k so that the k-largest singular values and corresponding singular vectors of

$$B = \begin{pmatrix} A_k \\ \hline T \end{pmatrix} \tag{4.2}$$

are computed.

The correction step for incorporating changes in term weights is performed after any terms or documents have been SVD-updated and the term weightings of the original matrix have changed. For a change of weightings in j terms, let Y_j be an $m \times j$ matrix comprised of rows of zeros or rows of the j-th order identity matrix, I_j, and let Z_j be an $n \times j$ matrix whose columns specify the actual differences between old and new weights for each of the j terms. Computing the SVD of the following rank-j update to A_k defines the correction step.

$$B = A_k + Y_j Z_j^T. \tag{4.3}$$

4.2 SVD-Updating Example

To illustrate SVD-updating, suppose the fictitious titles in Table 3.4 are to be added to the original set of titles in Table 3.1. In this example, only documents are added and weights are not adjusted, hence only the SVD of the matrix B in Equation (4.1) is computed.

Initially, a 12×7 term-document matrix, D, corresponding to the fictitious titles

in Table 3.4 is generated and then appended to A_2 to form a 12×16 matrix B of the form given by Equation (4.1). Following Figure 3.1, the best rank-2 approximation (B_2) to B is given by

$$B_2 = \hat{U}_2 \hat{\Sigma}_2 \hat{V}_2^T,$$

where the columns of $\hat{U}_2$ and $\hat{V}_2$ are the left and right singular vectors, respectively, corresponding to the two largest singular values of B.

Figure 4.1 is a two-dimensional plot of the 12 terms and 16 documents using the elements of $\hat{U}_2$ and $\hat{V}_2$ for term and document coordinates, respectively. Notice the similar clustering of terms and documents in Figures 4.1 and 3.6 (recomputing the SVD) and the difference in clustering with Figure 3.5 (folding-in).

4.3 SVD-Updating Procedures

The mathematical computations required in each phase of the SVD-updating process are detailed in this section. SVD-updating incorporates new term or document information into an existing semantic model (A_k from Equation (2.2)) using sparse term-document matrices (D, T, and $Y_j Z_j^T$) discussed in Section 4.1. SVD-updating exploits the previous singular values and singular vectors of the original term-documents matrix A as an alternative to recomputing the SVD of $\tilde{A}$ in Equation (3.1). In general, the cost of computing the SVD of a sparse matrix [B+93] can be generally expressed as

$$I \times \text{ cost } (G^T G x) + trp \times \text{ cost } (Gx), \tag{4.4}$$

where I is the number of iterations required by a Lanczos-type procedure [Ber92a] to approximate the eigensystem of $G^T G$ and trp is the number of accepted singular triplets (i.e. singular values and corresponding left and right singular vectors). The

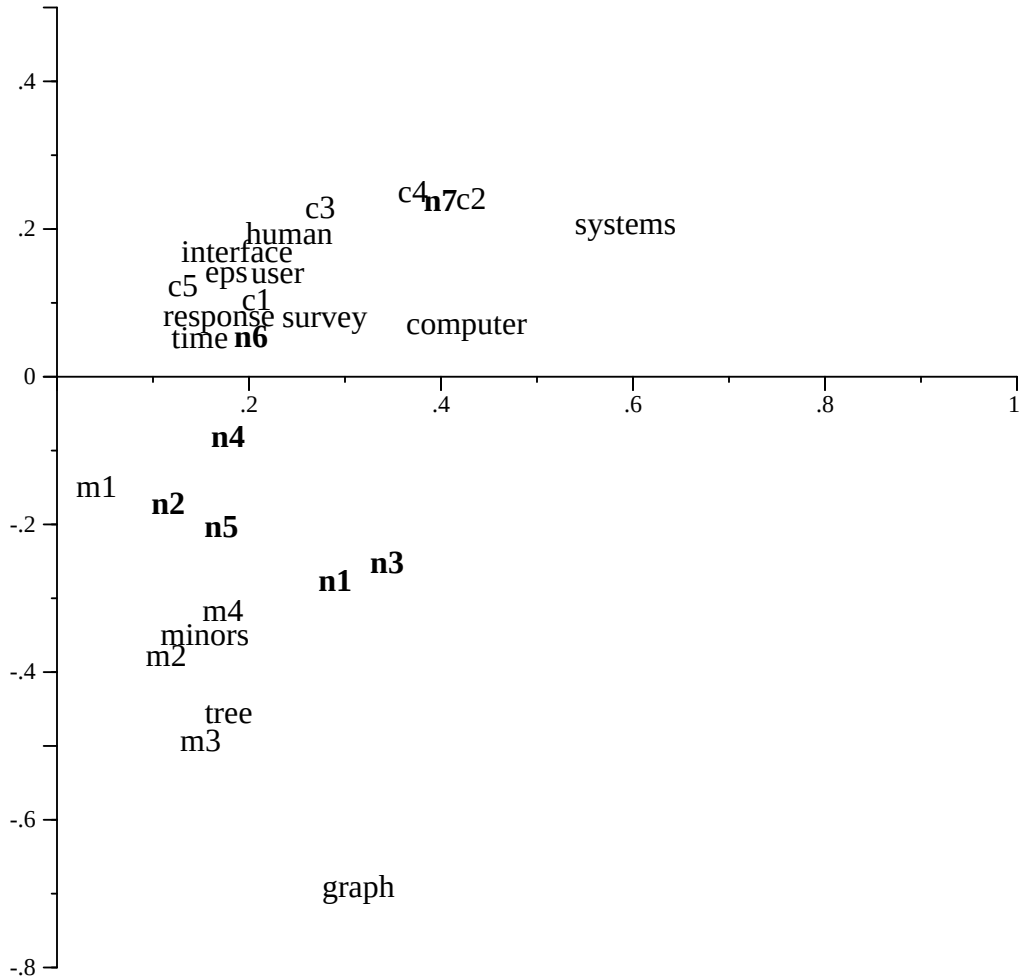


Figure 4.1: Two-dimensional plot of terms and documents using the SVD-updating process.

additional multiplication by G is required to extract the left singular vector given approximate singular values and their corresponding right singular vector approximations from a Lanczos procedure. The subsections below demonstrate how to update the existing rank-k approximation A_k using standard linear algebra along with an estimate of the number of floating-point operations required for each SVD-updating phase.

4.3.1 SVD-Updating of Documents

Let $B = (A_k \mid D)$ from Equation (4.1) and define SVD $(B) = U_B \Sigma_B V_B^T$. Then

$$U_k^T B \begin{pmatrix} V_k & \\ & I_d \end{pmatrix} = (\Sigma_k \mid U_k^T D),$$

since $A_k = U_k \Sigma_k V_k^T$. If $F = (\Sigma_k \mid U_k^T D)$ and SVD(F) $= U_F \Sigma_F V_F^T$, then it follows that

$$U_B = U_k U_F \text{ and}$$
$$V_B = \begin{pmatrix} V_k & \\ & I_d \end{pmatrix} V_F,$$

since $(U_F U_k)^T B \begin{pmatrix} V_k & \\ & I_d \end{pmatrix} = \Sigma_F = \Sigma_B$. Hence U_B and V_B are $m \times k$ and $(n+d) \times (k+d)$ dense matrices, respectively.

Table 4.1 contains a list of symbols, dimensions, and definitions of the variables used to express the computational cost of all the SVD-updating phases. Computational cost is defined in terms of floating-point operations (multiplications or additions) and denoted by *flops*.

Suppose G in Equation (4.4) is defined by $G = (\Sigma_k \mid U_k^T D)$ then the required flops

Table 4.1: Symbols used for computational complexity.

Symbol	Dimensions	Definition
A	$m \times n$	Original Term-document matrix
U_k	$m \times k$	Left singular vectors of A_k
Σ_k	$k \times k$	Singular values of A_k
V_k	$n \times k$	Right singular vectors of A_k
Z_j	$n \times j$	Adjusted weights
Y_j	$m \times j$	Permutation matrix
D	$m \times d$	New document vectors
T	$t \times n$	New term vectors
$nnz(X)$		Number of non zeros in X

for computing Gx are $2(nnz(D)+mk)-m$ and those for G^TGx are $4nnz(D)+4mk+k-2m-d$. Inserting these costs into Equation (4.4) we obtain,

$$I \times [4nnz(D) + 4mk + k - 2m - d] + trp \times [2nnz(D) + 2mk - m] \tag{4.5}$$

as an estimate of the number of flops required to compute SVD(G). The costs for computing U_B and V_B are $m[2k^2 - k]$ and $n[2k^2 - k]$, respectively. Therefore, the total cost for SVD-updating of documents is given by,

$$\begin{aligned} &[I \times [4nnz(D) + 4mk + k - 2m - d] + trp \times [2nnz(D) + 2mk - m]] \\ &+ [(2k^2 - k)(m + n)]. \end{aligned} \tag{4.6}$$

4.3.2 SVD-Updating of Terms

Let $B = \begin{pmatrix} A_k \\ \hline T \end{pmatrix}$ from Equation (4.2) and define SVD $(B) = U_B \Sigma_B V_B^T$. Then

$$\begin{pmatrix} U_k^T & \\ & I_t \end{pmatrix} BV_k = \begin{pmatrix} \Sigma_k \\ \hline TV_k \end{pmatrix}.$$

If $H = \left(\begin{array}{c} \Sigma_k \\ \hline TV_k \end{array} \right)$ and $\mathrm{SVD}(H) = U_H \Sigma_H V_H^T$ then it follows that

$$U_B = \begin{pmatrix} U_k & \\ & I_t \end{pmatrix} U_H \text{ and}$$

$$V_B = V_k V_H,$$

since $U_H^T \begin{pmatrix} U_k^T & \\ & I_t \end{pmatrix} B V_k V_H = \Sigma_H = \Sigma_B$. Hence U_B and V_B are $(m+t) \times (k+t)$ and $n \times k$ dense matrices, respectively.

Suppose H in Equation (4.4) is defined by $G = H = \left(\begin{array}{c} \Sigma_k \\ \hline TV_k \end{array} \right)$ then the flops for computing Gx are $2nnz(T) + 2nk + k - n - t$ and those for $G^T Gx$ are $4nnz(T) + 4nk + k - 2n - t$. Inserting these costs into Equation (4.4) we obtain,

$$I \times [4nnz(T) + 4kn + k - 2n - t] + trp \times$$
$$[2nnz(T) + 2kn + k - 2n - t] \tag{4.7}$$

as an estimate of the number of flops required to compute $\mathrm{SVD}(G) = \mathrm{SVD}(H)$. The costs for computing U_B and V_B are $m[2k^2 + k]$ and $n[2k^2 - k]$ flops respectively. Therefore, the total cost for SVD-updating of terms is given by,

$$[I \times [4nnz(T) + 4kn + k - 2n - t)] + trp \times [2nnz(T) + 2kn + k - 2n - t]]$$
$$+ [(2k^2 + k)(m + n)]. \tag{4.8}$$

4.3.3 SVD-Updating with Term Weight Corrections

Let $B = A_k + Y_j Z_j^T$, where Y_j is $(m+t) \times j$ and Z_j is $(n+d) \times j$ from Equation (4.3). Then

$$U_k^T B V_k = (\Sigma_k + U_k^T Y_j Z_j^T V_k).$$

If $Q = (\Sigma_k + U_k^T Y_j Z_j^T V_k)$ and SVD$(Q) = U_Q \Sigma_Q V_Q^T$, then it follows that

$$U_B = U_k U_Q \text{ and}$$

$$V_B = V_k V_Q.$$

Since $(U_Q U_k)^T B V_k V_Q = \Sigma_Q = \Sigma_B$. Hence U_B and U_Q are $m \times k$ and $n \times k$ dense matrices, respectively.

Suppose G in Equation (4.4) is defined by $G = Q = (\Sigma_k + U_k^T Y_j Z_j^T V_k)$ then the flops for computing Gx are $2nk + 2mk + 2nnz(Z_j) - j - n + k$ and those for $G^T Gx$ are $4nnz(Z_j) + 4km + 2mj + 2kn + 3k - 2n - 2j - m$. Inserting these costs into Equation (4.4) we obtain,

$$\begin{aligned} & I \times [4nnz(Z_j) + 4km + 2mj + 2kn + 3k - 2n - 2j - m] + trp \times \\ & [2nk + 2mk + 2nnz(Z_j) - j - n + k], \end{aligned} \tag{4.9}$$

where j is the number of terms whose weights have changed. The costs for computing U_B and V_B are $m(2k^2 - k)$ and $n(2k^2 - k)$ flops, respectively. Therefore, the total cost of SVD-updating with term weight corrections is given by,

$$\begin{aligned} & [I \times [4nnz(Z_j) + 4km + 2mj + 2kn + 3k - 2n - 2j - m] + \\ & trp \times [2nk + 2mk + 2nnz(Z_j) - j - n + k]] \\ & + (2k^2 - k)(m + n). \end{aligned} \tag{4.10}$$

4.4 LSI-Updating Example with Term Weight Corrections

This section describes SVD-updating using the same titles from Table 3.1 and Table 3.4. However, this example differs from that of Section 4.2 because different local and global weightings (see Equation 2.4) have been applied. Let a_{ij} denote the frequency of

term i in document j (Equation (2.5)), df_i denote the number of documents containing the term i, gf_i denote the frequency of term i in the entire collection, and $ndocs$ denote the number of documents in the collection. Suppose local (logarithmic) and global (entropy) weightings are applied so that from Equation (2.5) we have

$$L(i,j) = log(a_{ij}+1) \text{ and } G(i) = 1 - \sum_j \frac{p_{ij} log(p_{ij})}{log(ndocs)}, \text{ where } p_{ij} = \frac{a_{ij}}{df_i}.$$

We will refer to this particular weighting scheme as the log-entropy weighting [DDF+90].

Suppose an LSI-generated database has already been created by applying the log-entropy weighting to the titles from Table 3.1. SVD-updating first updates the existing database with the documents from Table 3.4, as in the Section 4.3.1. However, A_k has old weightings compared to D because new documents have changed both the local and global weightings of the original term-document matrix A defined in Table 3.2. In other words, the variables $ndocs, df_i$, and gf_i have all changed due to the addition of more documents. Figure 4.2 is a two-dimensional plot of the terms and documents of SVD-updating before the weighting correction step from Section 4.3.3 has been applied. The dotted circles in Figure 4.2 are used to compare clusterings of terms and documents in later Figures.

After the documents have been updated, the term weight correction step can be applied. The $m \times j$ matrix Y_j discussed in Section 4.1 denotes exactly which terms (or rows of the matrix A) have changed. Let e_i denote the ith canonical vector and $\vec{0}$ denote the $j \times 1$ null vector, then

$$Y_j = [e_1, \vec{0}, e_2, e_3, e_4, e_5, \vec{0}, e_6, e_7, e_8, e_9, \vec{0}]^T. \tag{4.11}$$

Z_j^T (see Table 4.2) reflects a change of weights for 9 out of 12 terms. Each row of the $j \times n$ matrix is computed by subtracting the new term weights from the

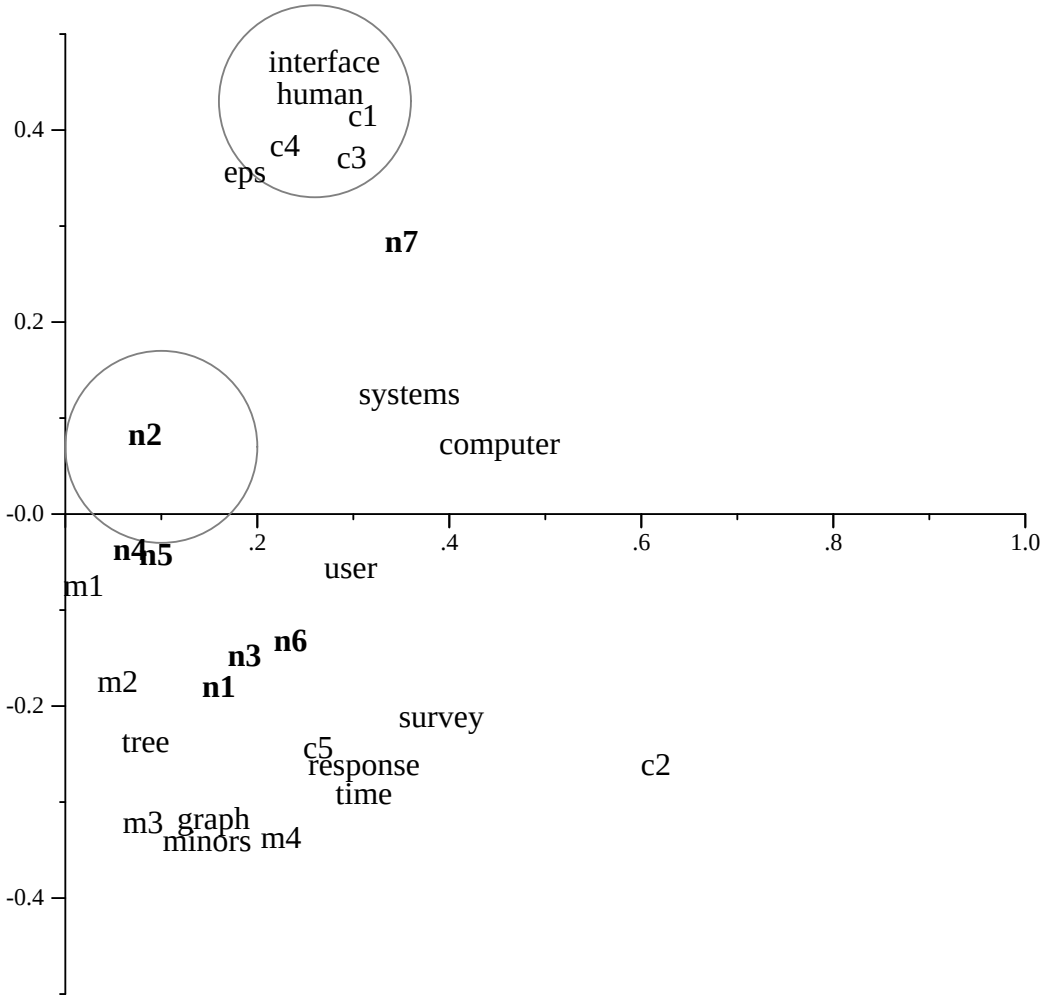


Figure 4.2: Two-dimensional plot of SVD-updating (before the term weight correction method) with log-entropy weighting.

Table 4.2: Example of Z_j^T

$$\begin{pmatrix}
-.330 & 0 & -.0807 & -.184 & 0 & 0 & 0 & 0 & 0 \\
-.330 & 0 & 0 & 0 & 0 & -.184 & -.214 & -.184 & 0 \\
0 & 0 & 0 & -.184 & 0 & 0 & -.214 & 0 & 0 \\
0 & 0 & -.0807 & 0 & 0 & 0 & -.214 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -.184 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -.0804 \\
0 & -.201 & 0 & 0 & 0 & 0 & 0 & 0 & -.0804 \\
0 & -.201 & 0 & 0 & -.0807 & 0 & 0 & 0 & -.0804 \\
0 & -.201 & 0 & 0 & -.0807 & -.184 & 0 & 0 & 0
\end{pmatrix}$$

old term weights for each document (see Figure 4.2). Once Y_j and Z_j^T have been constructed the SVD of the matrix Q from Section 4.3.3 is then computed. Figure 4.3 is a two-dimensional plot of the terms and documents after the term weight correction method has been applied. In comparison with the recomputed SVD (using log-entropy weightings) shown in Figure 4.4, we can see that the clustering of terms and documents with the term weight correction method (Figure 4.3) more closely resembles that of recomputing the SVD (Figure 4.4) as opposed to SVD-updating without term weight correction (Figure 4.2).

The major differences between SVD-updating with and without the term weight correction step (Figures 4.3 and 4.2) and recomputing the SVD (Figure 4.4) is due to the approximation of the original term-document matrix A by A_2 rather than $\tilde{A}_2$. In order to numerically quantify the differences, matrix norms can be applied. Let $\bar{A}$ be the first 9 columns of the 12×16 new term-document matrix and let A_2 be the rank-2 approximation of the original 12×9 term-document matrix A. An estimate of the error in approximating the new term-document matrix with a rank-2 approximation of the original term-document matrix is given by

$$\|\bar{A} - A_2\|_2 = 1.1805. \tag{4.12}$$

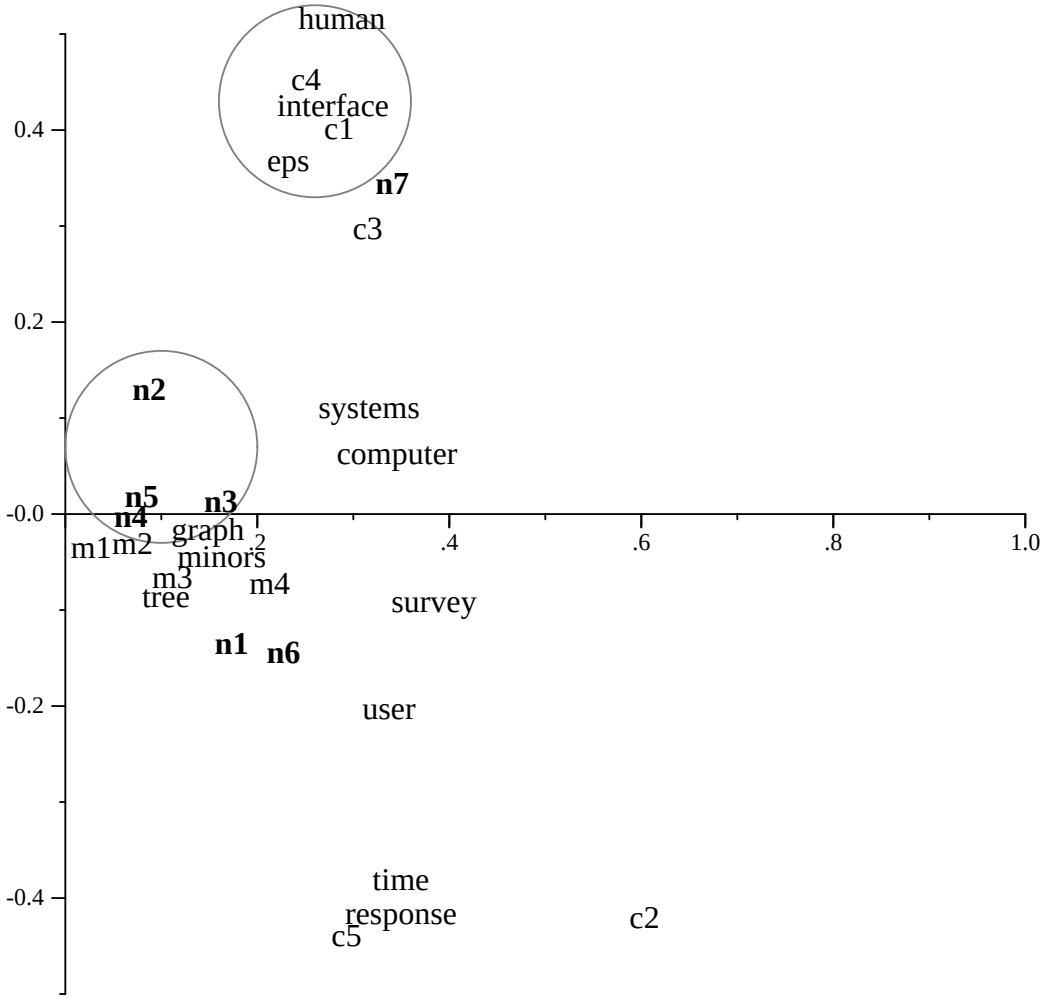


Figure 4.3: Two-dimensional plot of SVD-updating (after the term weight correction method) with log-entropy weighting.

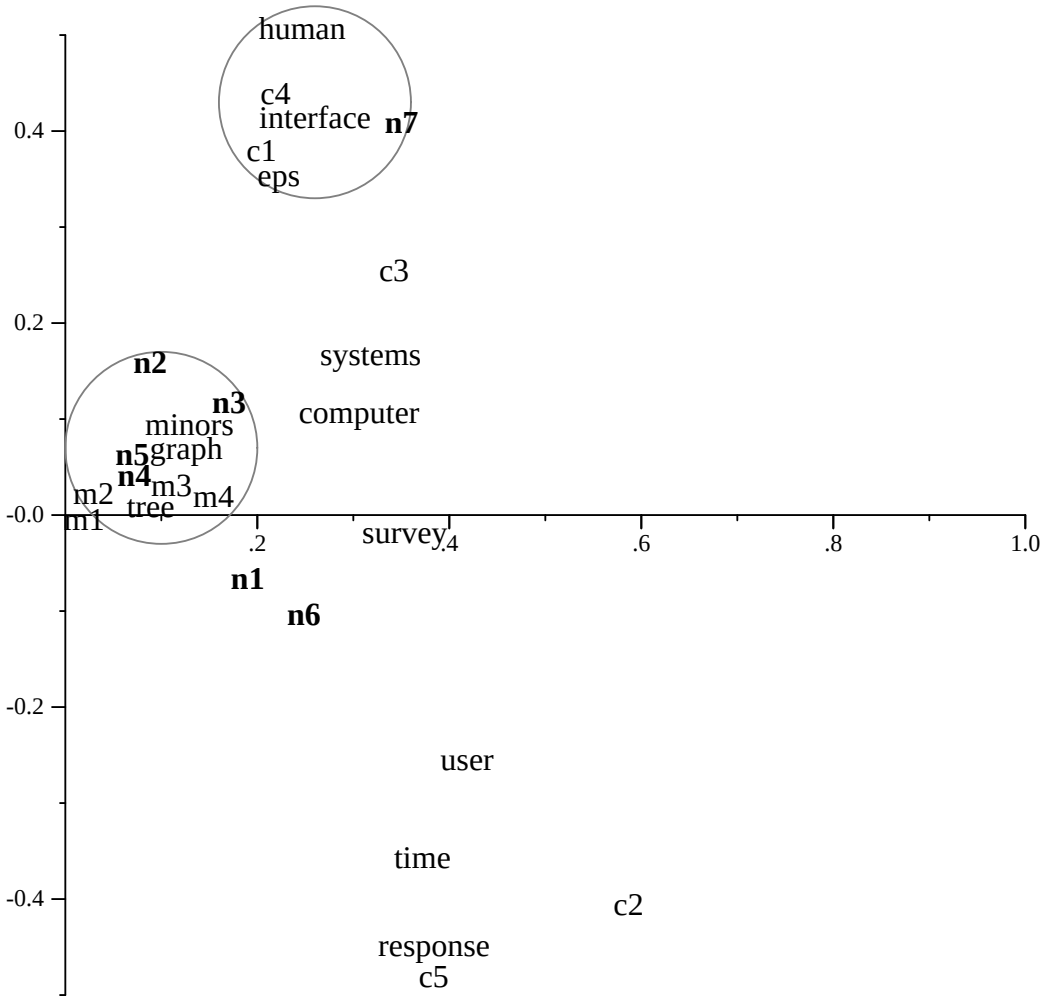


Figure 4.4: Two-dimensional plot of new SVD with log-entropy weighting.

When the term weight correction is applied to A_2, the error in representing $\bar{A}$ is reduced to

$$\|\tilde{A} - A_2 + Y_j Z_j^T\|_2 = 1.0665. \tag{4.13}$$

If the true rank-2 approximation to $\bar{A}$ is computed ($\bar{A}_2$) then

$$\|\tilde{A} - \tilde{A}_2\|_2 = 1.102 = \bar{\sigma_3}, \tag{4.14}$$

where $\bar{\sigma_3}$ is the third-largest singular value of $\bar{A}$ (see Mirsky's result in Equation (2.3)). Notice that the norm in Equation (4.13) more closely resembles the error in (4.14) than does the norm in (4.12).

4.5 Orthogonality

Orthogonality is a property maintained in the SVD by the left and right singular vectors. An $m \times n$ orthogonal matrix Q satisfies $Q^T Q = I_n$, where I_n is the n-th order identity matrix. Let D_p be the collection of all folded-in documents where each column of the $p \times k$ matrix is a document vector of the form d_p from Equation (2.7). Similarly, let T_q be the collection of all folded-in terms such that each column of the $q \times k$ matrix is a term vector of the form t_q from Equation (2.8). Then, all term vectors and document vectors associated with folding-in can be represented as $\hat{U}_k = \left(U_k^T \mid T_q^T\right)^T$ and $\hat{V}_k = \left(V_k^T \mid D_p^T\right)^T$, respectively.

The folding-in process corrupts the orthogonality of $\hat{U}_k$ and $\hat{V}_k$ by appending non-orthogonal submatrices T_q and D_p to U_k and V_k, respectively. Computing $\hat{U}_k^T \hat{U}_k$ and $\hat{V}_k^T \hat{V}_k$, the loss of orthogonality in $\hat{U}_k$ and $\hat{V}_k$ can be measured by

$$\|\hat{U}_k^T \hat{U}_k - I_k\|_2 \tag{4.15}$$

and

$$\|\hat{V}_k^T \hat{V}_k - I_k\|_2. \tag{4.16}$$

Table 4.3 illustrates the orthogonality of $\hat{V}_k$ using three different updating methods for adding new documents to the original 12×9 example. SVD-updating and recomputing the SVD maintain the orthogonality of U_k and V_k to working precision (single precision or 32-bit arithmetic for Table 4.3). Folding-in does not maintain the orthogonality of $\hat{U}_k$ or $\hat{V}_k$ since arbitrary vectors of weighted terms or documents are appended to U_k or V_k, respectively. However, the amount by which the folding-in method perturbs the orthogonality of $\hat{U}_k$ or $\hat{V}_k$ does indicate how much distortion has occurred due to the addition of new terms or documents.

Table 4.3: Loss of orthogonality in $\hat{V}_k$ for folding-in and in V_k for SVD-updating and recomputing the SVD using the 12×16 example.

Folding-in	SVD-Updating	Recomputing the SVD
1.445E+1	2.206E-4	2.472E-4

4.6 Memory Considerations

One of the major concerns in updating LSI databases is memory conservation. Recomputing the SVD requires memory to store all the nonzeros of the new $(m+t) \times (n+d)$ term-document matrix $\tilde{A}$. While SVD-updating requires only the nonzeros of the matrices defining new terms or documents, folding-in requires no sparse input matrices. Table 4.4 contains a list of estimated memory usages for each updating approach. Clearly, folding-in is the most memory conservative while recomputing the SVD can easily exhaust available memory if the number of current documents (n) becomes

Table 4.4: Lanczos memory constraints.

Method	Memory
SVD-updating (any phase)	$I \times k + trp \times (1 + m + n)$
Folding-in documents	$d + k + n$
Folding-in terms	$t + k + m$
Recomputing the SVD	$I \times n + trp \times (1 + m + n)$

quite large. Each of the SVD-updating phases can be applied so that memory is better conserved ($k \ll m$).

4.7 Computational Complexity for SVD-Updating

This section compares the computational complexities of folding-in and SVD-updating. Whereas the computational cost of recomputing the SVD can be considerably less than that of SVD-updating, the memory constraints (see Section 4.6) of modest computing environments (e.g., workstations) may preclude its applicability (see Chapter 5). Table 4.5 contains the complexities for folding-in terms and documents, recomputing the SVD, and the three phases of SVD-updating. Using the complexities in Table 4.5 the required flops for each method is graphed for varying numbers of added documents or terms.

Assumptions needed in constructing the graphs in Figures 4.5 through 4.8 are listed in Table 4.6. Using parameters obtained from the letter A of the *Columbia Condensed Encyclopedia*, (CCE), let the number of terms $m = 5119$ and the number of documents $n = 1063$. For an alternative scenario in which relatively few terms are used in many documents, let $m = 1000$ and $n = 6000$. The sparse matrices A, D, T, and Z_j^T are all considered to be 0.1% dense. Term-document matrices corresponding

Table 4.5: Computational complexity of updating methods.

Method	**Complexity**
SVD-updating documents	$[I \times [4nnz(D) + 4mk + k - 2m - d] + trp \times [2nnz(D) + 2mk - m]] + [(2k^2 - k)(m + n)]$
SVD-updating terms	$[I \times [4nnz(T) + 4kn + k - 2n - t] + trp \times [2nnz(T) + 2kn + k - 2n - t]] + [(2k^2 - k)(m + n)].$
SVD-updating correction step	$[I \times [4nnz(Z_j) + 4km + 2mj + 2kn + 3k - 2n - 2j - m] + trp \times [2nnz(Z_j) + 2km + 2kn + k - j - n]] + [(2k^2 - k)(m + n)]$
Folding-in documents	$2mkd$
Folding-in terms	$2nkt$
Recomputing the SVD	$I \times [4nnz(A) - (m + t) - (n + d)] + trp \times 2nnz(A) - (m + t)$

Table 4.6: Assumptions for graphing computational complexities.

Symbol	Definition	Value when $m = 5119, n = 1063$	Value when $m = 1000, n = 6000$
A	$m \times n \times 0.1\%$	5441	6000
D	$m \times d \times 0.1\%$	$5.119d$	d
T	$t \times n \times 0.1\%$	$1.063t$	$6t$
Z_j^T	$j \times n \times 0.1\%$	$0.532t$	$3t$
I	$k \times 3$	124	124
trp	$> k$	110	110
j	$t \times 50\%$	$0.5t$	$0.5t$
k		108	108

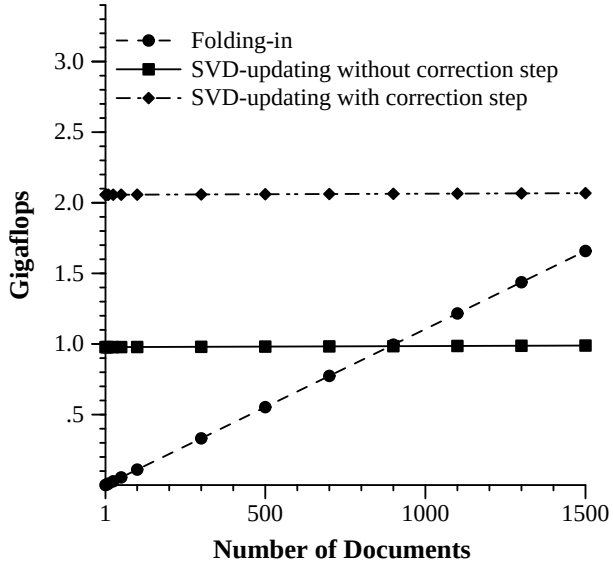


Figure 4.5: Complexity of adding documents with $m \ll n$.

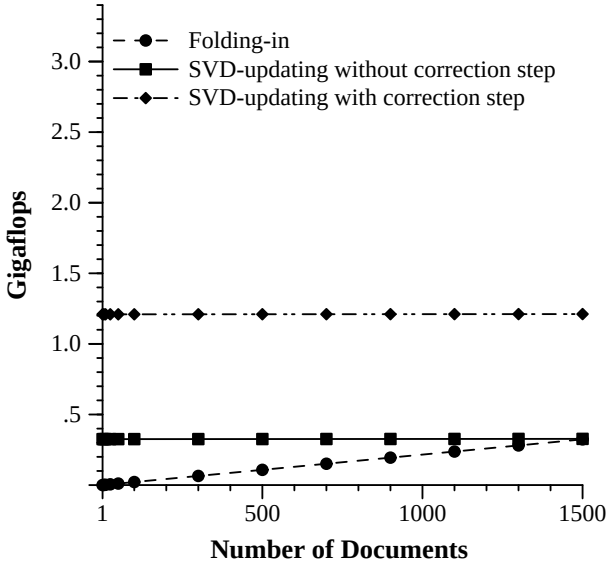


Figure 4.6: Complexity of adding documents with $m \gg n$.

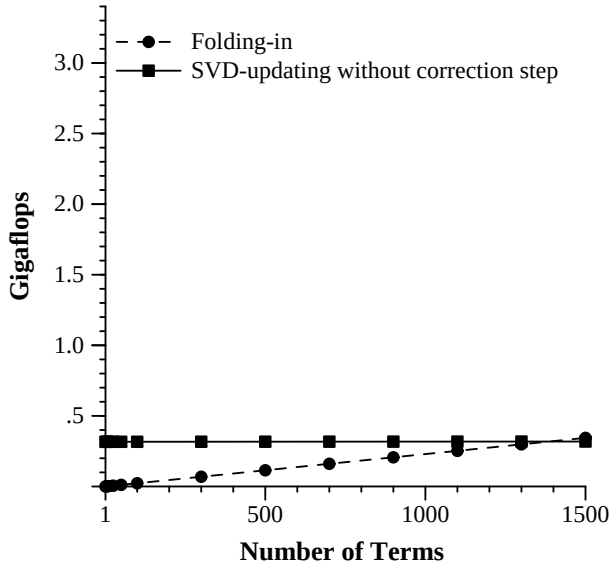


Figure 4.7: Complexity of adding terms with $m \ll n$.

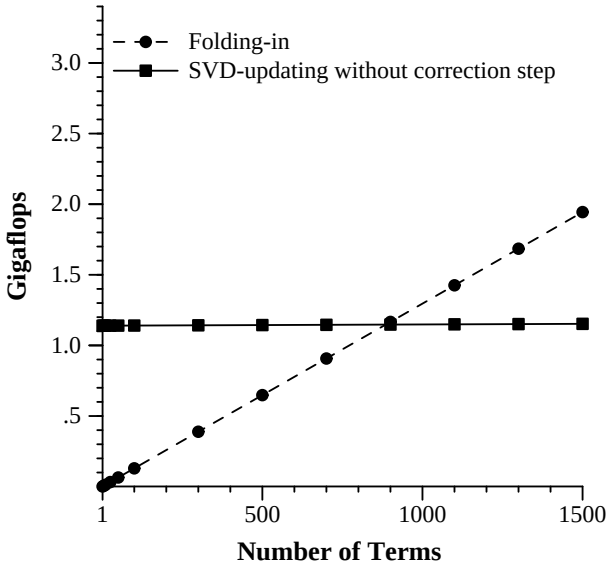


Figure 4.8: Complexity of adding terms with $m \gg n$.

to letters of the Columbia Condensed Encyclopedia were typically 0.1% dense. Table 4.6 also contains the values of I, trp, j, and k based on the two possible choices for m and n.

Figures 4.5 and 4.6 vary the range of documents, d, being added to an LSI-database. When $m \ll n$, i.e., the number of documents being updated is relatively small, folding-in requires fewer flops than SVD-updating (see Figure 4.5), but as the number of documents being updated becomes relative large (greater than 900) SVD-updating requires fewer computations than folding-in. When $m \gg n$, i.e. the number of documents being updated is very large relative to the number of terms, the number of documents needed for SVD-updating to perform better than folding-in has grown to greater than 1500 documents (see Figure 4.6). The folding-in method for documents is dependent on the number of documents (n) and not on the number of terms (m) (see Table 4.5), this explains the linearity of the two folding-in curves in Figures 4.5 and 4.6. The term weight correction step requires twice the required flops of SVD-updating documents.

Figures 4.7 and 4.8 vary the range of terms, t, added to the LSI-database. When $m \ll n$, SVD-updating performs better than folding-in after 1500 terms have been updated (see Figure 4.7). When $m \gg n$, folding-in requires fewer flops than SVD-updating for $t < 900$, otherwise SVD-updating requires fewer flops (see Figure 4.8). Since adding new terms has no effect on the original weightings (see Section 4.4), the term weight correction step is not illustrated in Figures 4.7 and 4.8.

Chapter 5

Performance Benchmarks

5.1 Overview

In this chapter, we discuss the performance of SVD-updating and folding-in compared to recomputing the SVD. Three performance metrics are used: 1) retrieval accuracy, 2) memory usage, and 3) speed. The comparisons are based on experiments using articles from the Condensed Columbia Encyclopedia (CCE).

5.2 Retrieval Accuracy

Relevance feedback [SB90] is a retrieval technique which initially uses a query to obtain a set of relevant documents from which selected documents are used to retrieve even more relevant documents. This section uses relevance feedback to measure retrieval accuracy. Specifically, the returned documents of the relevance feedback queries of folding-in and SVD-updating are compared with the documents returned from relevance feedback queries associated with recomputing the SVD.

Using the 12×16 example with log-entropy weighting described in Section 4.4,

Table 5.1: Relevance feedback results using the 12×16 example and log-entropy weightings.

Query Document	Method	Returned Documents															
c2	Folding-in	c2	n1	n3	n5	n6	c5	m4	n4	n7	n2	m3	m2				
	SVD-updating	c2	n1	n6	m1	c5	m2	m3	m4	n5	n4	n3					
	Recomputing the SVD	c2	n6	c5	n1	m4	m1	n5	m2	m3	n3	n4	c3				
m2	Folding-in	m2	m3	m1	m4	c5	n6	n2	n1	c2							
	SVD-updating	m2	m3	m4	n5	n4	n3	m1	n6	n1	c2	c3	n7	c5	n2	c1	c4
	Recomputing the SVD	m2	n5	m3	n3	n4	m1	c3	m4	n7	n2	n1	c1	c4	n6	c2	
n4	Folding-in	n4	n7	n2	c1	c3	c4	n3	c2	n1	n5	n6	c5				
	SVD-updating	n4	n5	m4	m3	n3	m2	c3	m1	n7	n6	n1	c2	n2	c1	c4	c5
	Recomputing the SVD	n4	n3	c3	m3	m2	n5	n7	m1	m4	n2	c1	c4	n1	n6	c2	
n7	Folding-in	n7	n4	n2	c1	c3	c4	n3	c2	n1	n5	n6	c5				
	SVD-updating	n7	n2	c3	c1	c4	n3	n4	n5	m4	m3	m2					
	Recomputing the SVD	n7	c3	n2	n4	c1	c4	n3	m3	m2	n5	m1	m4	n1			

relevance feedback was performed with documents **c2, m2, n4** and **n7**, i.e., these documents were used as new queries to obtain more relevant information. Table 5.1 shows the returned documents from the relevance feedback queries for each method. To compare the results of the queries, documents returned by recomputing the SVD were compared with documents returned by folding-in and SVD-updating. A document returned by folding-in or SVD-updating is counted as a hit if that document is also returned by recomputing the SVD. Otherwise, a document is counted as a miss. Hits and misses were computed as a function of window size, where window size is defined as the first n documents in the LSI rank-ordered list associated with a query. For example, a window size of two would compare the top two returned documents from recomputing the SVD with the top two documents from folding-in and SVD-updating. Table 5.2 contains the hits and misses of folding-in and SVD-updating at even window sizes for each document used in the queries (**c2, m2, n4, n7**). Figures

Table 5.2: Relevance feedback results of misses and hits.

Query Document	Method	Window Size (2)		Window Size (4)		Window Size (6)	
		Misses	Hits	Misses	Hits	Misses	Hits
n2	Folding-in	1	1	2	2	2	4
	SVD-updating	1	1	1	3	1	5
m2	Folding-in	1	1	2	2	3	3
	SVD-updating	1	1	1	3	1	5
n4	Folding-in	1	1	3	1	4	2
	SVD-updating	1	1	2	2	1	5
n7	Folding-in	1	1	1	3	0	6
	SVD-updating	1	1	1	3	1	5

Query Document	Method	Window Size (8)		Window Size (10)		Window Size (12)	
		Misses	Hits	Misses	Hits	Misses	Hits
c2	Folding-in	2	6	3	7	2	10
	SVD-updating	1	7	1	9	1	11
m2	Folding-in	4	4	5	5	7	5
	SVD-updating	1	7	3	7	2	10
n4	Folding-in	4	4	5	5	5	7
	SVD-updating	1	7	1	9	3	9
n7	Folding-in	1	7	2	8	4	8
	SVD-updating	1	7	1	9	2	10

5.1 through 5.4 plot hits from Table 5.2 for folding-in and SVD-updating.

For the 12×16 example, SVD-updating has more hits than folding-in at each window size using relevance feedback with documents **c2, m2** and **n4**. This superior retrieval accuracy indicates that SVD-updating returns documents more similar to that of recomputing the SVD than folding-in. Relevance feedback using SVD-updating with document **n7** (Figure 5.4) does not always have more hits than folding-in, but neither method performed convincingly better than the other. Folding-in's accuracy is dependent on the term-document relationships remaining the same before and after updating. In the case of **n7**, the usage of terms in the document was similar to the previous usage of the terms in documents **c1** and **c2**.

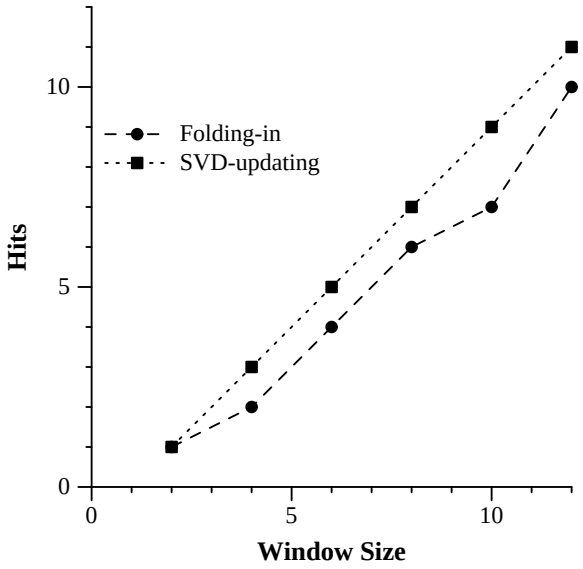


Figure 5.1: **c2** relevance feedback hits.

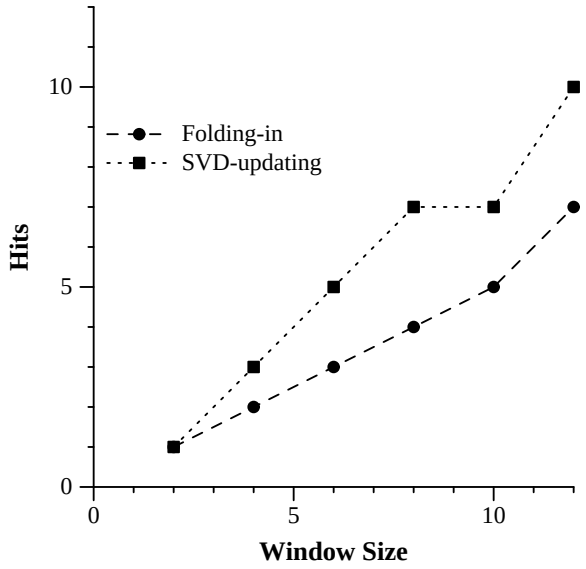


Figure 5.2: **m2** relevance feedback hits.

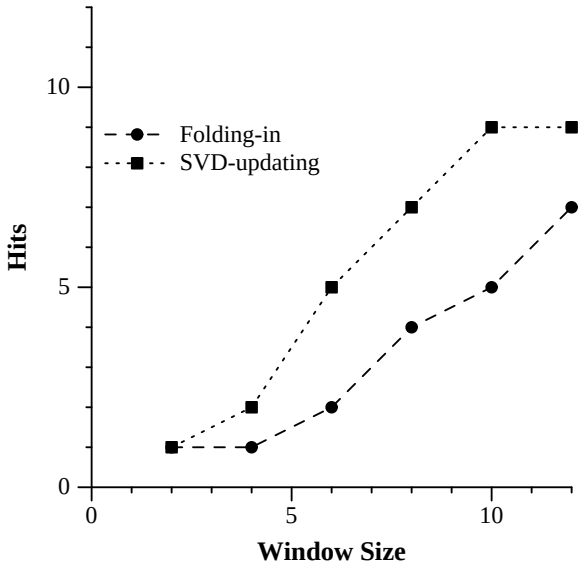


Figure 5.3: **n4** relevance feedback hits.

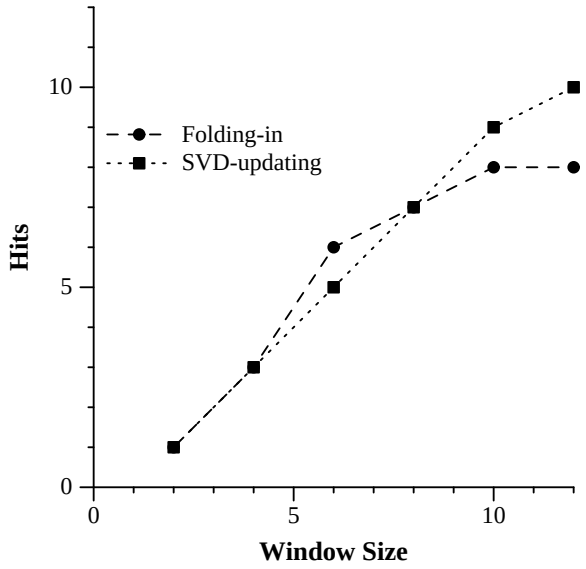


Figure 5.4: **n7** relevance feedback hits.

5.3 Memory Usage

Section 4.6 discussed memory conservation with respect to the three updating methods. This section provides examples of memory usage using articles comprising the letters A through F of the Columbia Condensed Encyclopedia (CCE). Memory usage for computing the SVD of sparse matrices was originally discussed in [Do93]. The memory allocation for SVDPACKC [Ber92b] and the single-vector Lanczos algorithm used in recomputing the SVD can be categorized according to the storage of Lanczos vectors, singular vectors, and temporary workspace. For an $m \times n$ term-document matrix A, the memory allocation by category can be expressed as

Lanczos Vectors	$k(lanmax + 2)$
Singular Vectors	$k(I) + m + I^2$
Temporary Workspace	$9k + 4lanmax + m + 1 + (n + nnz + 1)/2 + nnz,$

where k is the number of singular values and singular vectors to be computed, $lanmax$ is the maximum number of iterations allowed by the Lanczos algorithm, m is the largest dimension of A, n is the smallest dimension of A, nnz is the number of nonzeros of A, and I is the actual number of iterations taken.

The model for computing memory usage for SVD-updating is essentially the same as for computing the SVD, since extra memory is only needed for retrieving a previously-generated left or right singular vector from disk, and for a temporary work array whose size is equal to current number of LSI factors (k). However the number of nonzeros for the matrices D and T from Equation (4.1) and (4.2) is considerably smaller (in practice) than the number of nonzeros of the reconstructed term-document matrix $\tilde{A}$ see (Section 4.4). Hence, recomputing the SVD will require more in-core memory for

Table 5.3: Parameters of datasets from the Columbia Condensed Encyclopedia.

dataset	Size in bytes	Number of Documents	Number of Terms	Number of Factors	nnzeros
A	478,775	1,063	5,119	108	28,116
B	549,635	1,251	5,617	105	34,619
C	715,125	1,423	6,914	101	43,724
D	258,172	607	3,115	102	14,072
E	263,869	504	3,065	102	13,105
F	261,403	545	3,106	104	13,966
AB	1,028,410	2,314	8,905	100	67,264
ABC	1,743,517	3,737	12,660	100	117,187
ABCD	2,001,689	4,344	13,892	111	136,138
ABCDE	2,265,558	4,848	14,976	100	153,943
ABCDEF	2,526,961	5,393	16,030	100	172,684

computing singular values and singular vectors. The cost of memory for folding-in can be as small as

$$2m + k.$$

Actual memory usage was computed (see Table 5.4) using parameters associated with the letters A through F of the CCE (Table 5.3). Figure 5.5 is a graph of the memory usage (in megabytes) associated with the three updating methods. Folding-in was the most conservative method in terms of memory usage while recomputing the SVD was the most expensive; SVD-updating was modestly conservative in memory usage. Recomputing the SVD memory usage is most closely related to the size of the database and grows at a faster rate than SVD-updating or folding-in. Folding-in and SVD-updating memory usage, on the other hand, is more related to the sparsity associated with the term by document correlation of the incoming data rather than the sparsity of the completely updated term by document matrix.

Table 5.4: Actual memory usage for updating the Columbia Condensed Encyclopedia.

Letter(s) of CCE updated with	Letter(s) of CCE updated to	Folding-in bytes	SVD-updating bytes	Recomputing the SVD bytes
B	A	12,260	3,009,060	5,150,952
C	AB	13,112	3,235,824	8,230,048
D	ABC	54,696	1,694,496	9,289,188
E	ABCD	53,072	1,161,948	10,372,960
F	ABCDE	54,056	1,221,028	11,536,860

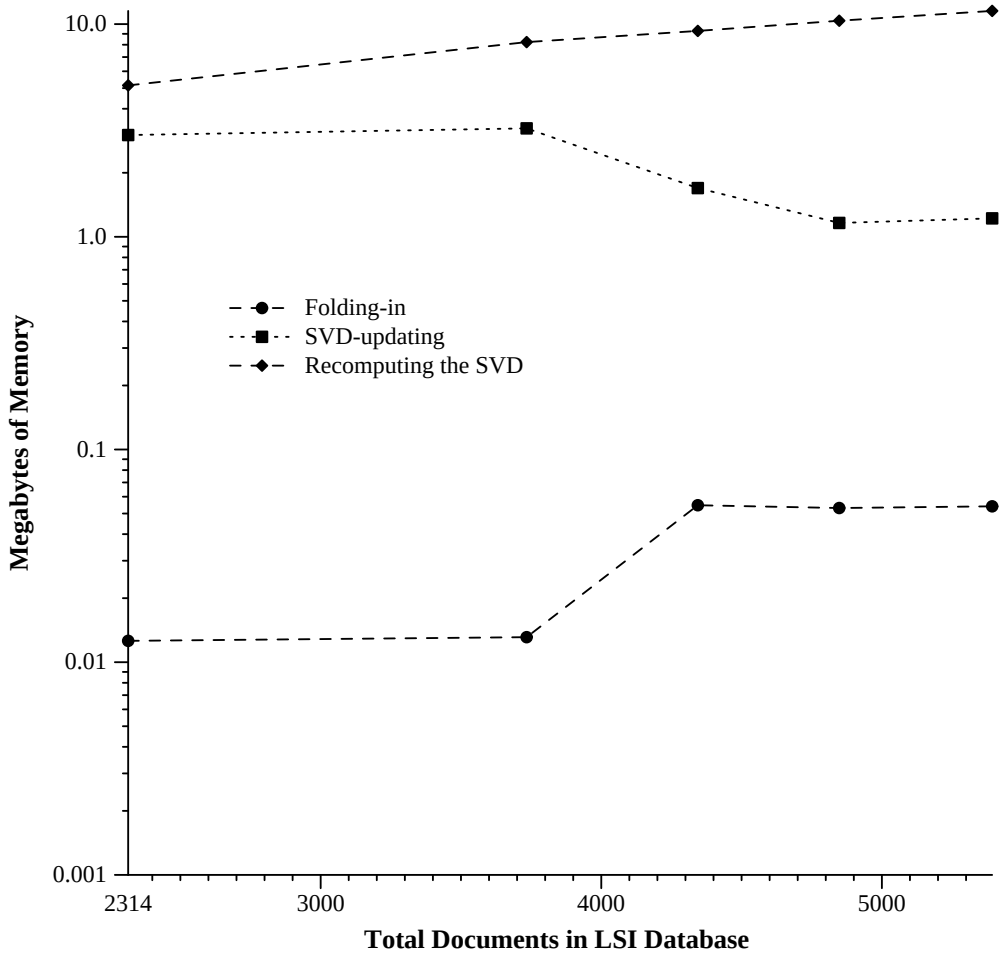


Figure 5.5: Memory usage for updating letters A-F of CCE.

Table 5.5: Timing benchmarks updating the letters A-F of CCE.

Letters in existing database	Letters updated to existing database	Folding-in (min:sec)	SVD-updating (min:sec)	Recomputing the SVD (min:sec)
A	B	8:31	6:40	6:29
AB	C	10:32	11:30	13:53
ABC	D	4:26	14:02	18:01
ABCD	E	3:59	15:06	17:47
ABCDE	F	4:07	15:10	16:11

5.4 Timing Benchmarks

Using the letters A through F of the Columbia Condensed Encyclopedia, elapsed wall-clock timings of the updating methods were recorded. Wall-clock times on a Sun SPARCstation2 (with accelerator chip) and 64 megabytes of memory were obtained. All three updating methods were timed for the entire LSI process. Each letter (except A) was updated to the set of letters that precedes it. For example, the letters A and B (AB) were updated with the letter C. Table 5.5 contains the wall-clock times of each updating run for the letters B through F. Folding-in's wall-clock times were dependent on the number of documents being added to the existing LSI-generated database. SVD-updating's times were dependent on both the sparsity of the updated documents and the size of the existing LSI-generated database. Recomputing the SVD was dependent on the sparsity of the entire LSI-generated database, the existing LSI-database combined with the new documents, and on the type of data (homogeneous or heterogeneous) being added to the database. Figure 5.5 plots the timings from Table 5.5. Clearly, folding-in and SVD-updating times increased as new letters of the CCE were added. In general, recomputing the SVD increased as new letters of the CCE were added but decreases in times did occur when the number of terms did not

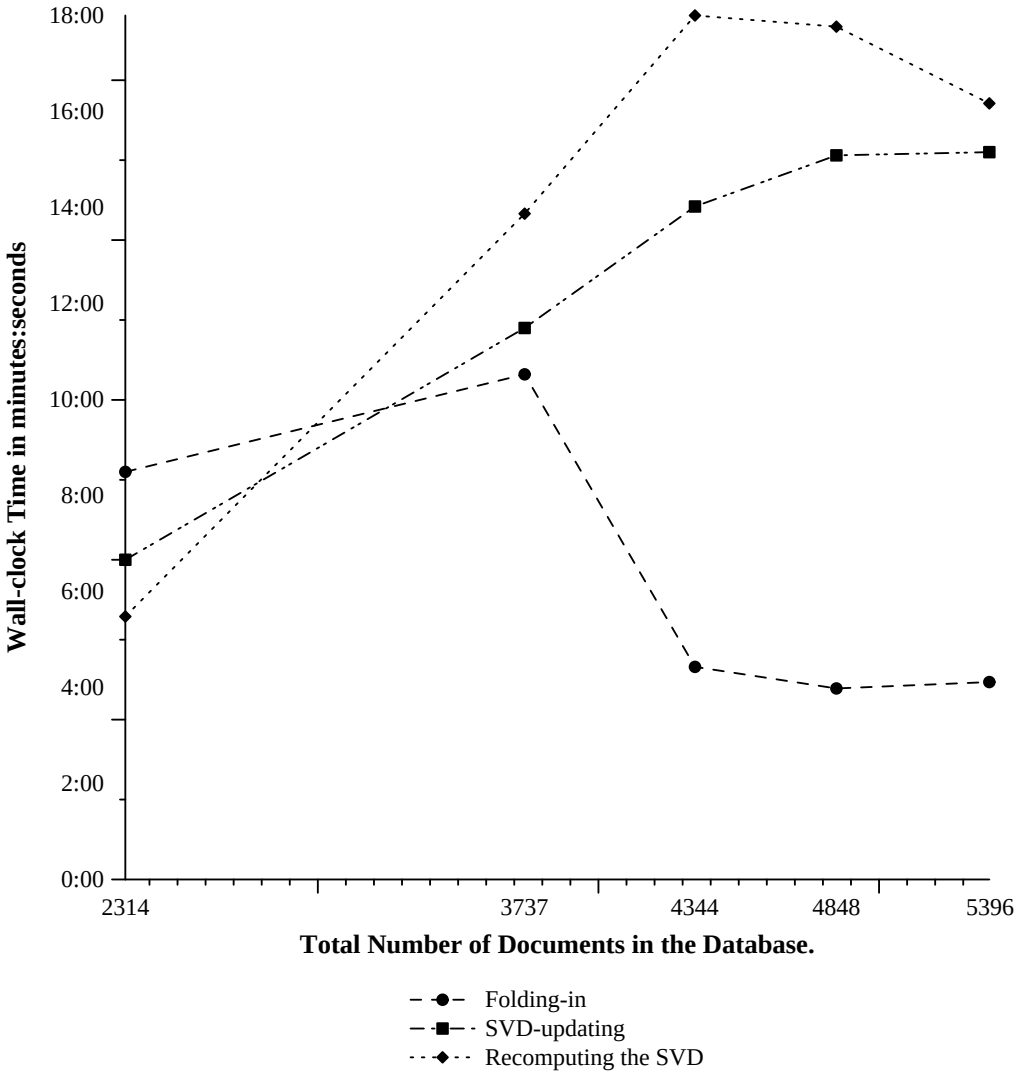


Figure 5.6: Wall-clock time in seconds for updating documents of the Columbia Condensed Encyclopedia.

grow significantly.

Using the three updating methods for adding the letter C to the letters AB of CCE (i.e., 3,737 total documents after updating from Table 5.3), a code *profile* or graph of the significant features of the updating methods was constructed (Figure 5.7). SVD-updating and recomputing the SVD were fashioned from three basic components: 1) parsing, 2) las2, and 3) indexdoc. *Parsing*, parses the words from the text, eliminates common words, creates the database of terms, and generates a Compressed Column Storage (CCS) [B+94, DGL89] representation of the term-document matrix (see Appendix A). *Las2* computes the truncated SVD of the term-document matrix, and *indexdoc* creates a list for matching articles of the CCE with corresponding document vectors derived from the truncated SVD.

Folding-in is composed of two different codes, *make_vectors* and *vfold*. *Make_vectors* is a C code which creates a document or term vector for each document or term folded-in. *Vfold* appends the new vectors created by *make_vector* to the SVD contained in the *out* file (see Appendix B).

Analyzing Figure 5.7 we can see that the truncated SVD computation (*las2*) is faster for SVD-updating than for recomputing the SVD, since a smaller order matrix (D from Equation(4.1)) is processed in SVD-updating. However, *parsing* is slower in SVD-updating, negating the speed improvement by SVD-updating. The parsing algorithm for SVD-updating is not identical to that for recomputing the SVD. SVD-updating must check for the existence of terms before adding them to the database and this can explain the slower parsing. In addition, two different languages were used to implement the parsing. SVD-updating uses a script language (AWK) [AW88] as opposed to a C program for recomputing the SVD.

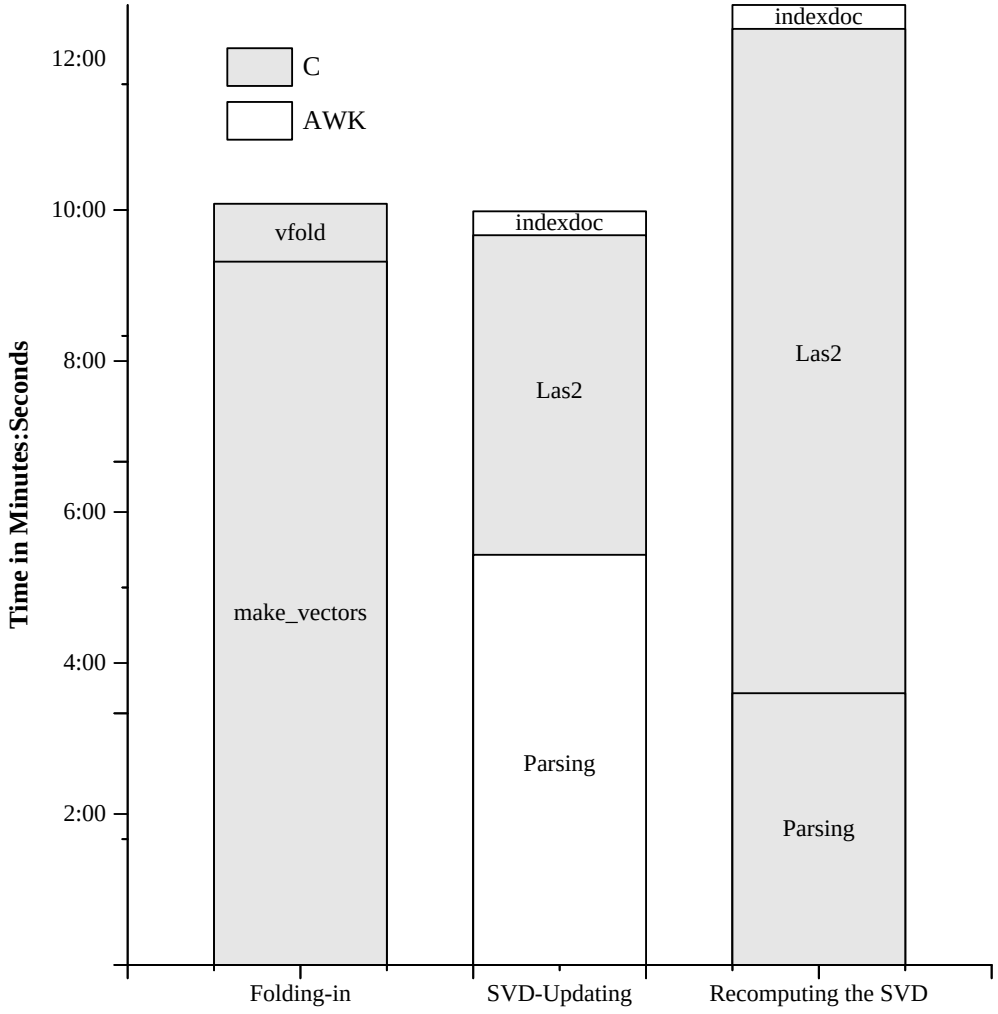


Figure 5.7: Updating documents profile.

Chapter 6

Summary and Future Work

6.1 Summary

We have produced an alternative updating method, SVD-updating, for LSI. Recomputing the SVD is the most accurate method of updating but memory and time constraints can eliminate it as a possibility. Folding-in is the fastest and most memory conservative but can be inaccurate especially as more documents are folded-in. SVD-updating is an alternative to folding-in and recomputing the SVD which adequately compromises the tradeoffs in memory usage, computing speed, and retrieval accuracy. Table 6.1 illustrates the attributes of the three updating methods. SVD-

Table 6.1: Attributes of updating methods.

Method	Memory	Computational Complexity	LSI Retrieval Accuracy
Recomputing the SVD	High	High	High
SVD-updating	Moderate	Moderate	Moderate to High
Folding-in	Low	Low	Low to High

updating and folding-in are implemented as software tools and are intended for an experienced user (LSI-database manager). Such LSI-database management tools facilitate term and/or document updating when resources (memory and computing time) are limited.

6.2 Future Work

Section 4.5 described a method for measuring the orthogonality of term and document vectors. A simple tool could be created using the *out* file format (see Appendix A) which could return the loss of orthogonality in Equations (4.15) and (4.16). Insights could be gained from monitoring the loss of orthogonality associated with folding-in and correlating it to the number of relevant documents returned.

Determining the optimal number of LSI factors is still an area of research. Currently anywhere between 100 to 300 factors are typically used. A tool for detecting the optimal number of factors should be investigated. Such a tool could read the *out* file and extract computed singular values in order to numerically assess the error in approximating the original term-document matrix (see Equation (4.14)). With regard to software issues, term parsing associated with the SVD-updating could be implemented in C rather than AWK. Also, the dense matrix multiplication routines used in SVD-updating could be modified to significantly reduce the amount of in-core memory allocated.

Bibliography

Bibliography

[AW88] Kernighan Aho and Weinberger. *The AWK Programming Language*. Addison-Wesley, New York, 1988.

[B+93] M. W. Berry et al. SVDPACKC: Version 1.0 User's Guide. Technical Report CS–93–194, University of Tennessee, Knoxville, TN, October 1993.

[B+94] R. Barrett et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, 1994.

[Ber92a] M. W. Berry. Large scale singular value computations. *International Journal of Supercomputer Applications*, 6(1):13–49, 1992.

[Ber92b] M. W. Berry. SVDPACK: A Fortran-77 Software Library for the Sparse Singular Value Decomposition. Technical Report CS–92–159, University of Tennessee, Knoxville, TN, June 1992.

[DDF+90] S. Deerwester, S. Dumais, G. Furnas, T. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[DGL89] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Software*, 15:1–14, 1989.

[Do93] T. Do. Sequential and Data-Parallel Implementations of a Lanczos Algorithm for the Singular Value Decomposition. Master's thesis, The University of Knoxville, Tennessee, Knoxville, TN, 1993.

[Dum91] S. T. Dumais. Improving the retrieval of information from external sources. *Behavior Research Methods, Instruments, & Computers*, 23(2):229–236, 1991.

[GL89] G. Golub and C. Van Loan. *Matrix Computations*. Johns-Hopkins, Baltimore, second edition, 1989.

[GR71] G. Golub and C. Reinsch. *Handbook for automatic computation II, linear algebra*. Springer-Verlag, New York, 1971.

[Mir60] L. Mirsky. Symmetric gage functions and unitarilly invariant norms. *Q. J. Math*, 11(1):50–59, 1960.

[SB90] G. Salton and C. Buckley. Improving retrieval performance by relevance feedback. *Journal of the American Society for Information Science*, 41(4):288–297, 1990.

Appendices

Appendix A

Compressed Column Storage

Figure A.1: Contents of `matrix.hb` file.

		`matrix.hb` file
Section 1 (Header)	Line 1	Title
	Line 2	#
	Line 3	rra rows columns nonzeroes 0
	Line 4	(10i8) (10i8) (8f10.3) (8f10.3)
Matrix Coordinate Data	Section 2	Column pointers
	Section 3	Row index
	Section 4	Nonzero values of the matrix.

Line 1	Any title up to 128 characters
Line 2	#
Line 3	rra, rows = number of rows in A, cols = number of columns in A, nonzeros = number of nonzeros in A.
Line 4	Fortran formatting lines (from Harwell-Boeing format, obsolete and can be ignored).
Section 2	The position indicates the column index of nonzero matrix elements. The number subtracted from the next value indicates how many nonzeros are in that column.
Section 3	Indicates the row index of each matrix nonzero element. There are as many row index values as there are nonzero elements.

`matrix.hb` file example

```
Title: using stdin
#
rra                         12              7              22              0
          (10i8)            (10i8)              (8f10.3)              (8f10.3)
   1       5       7       11       13       15       18       23
   3       9      10       11        3        5        1        3        6        9
   9      12        1        3        1        8       10        1        4        5
   8       9
 1.000     1.000     1.000     1.000     1.000     1.000     1.000     1.000
 1.000     1.000     1.000     1.000     1.000     1.000     1.000     1.000
 1.000     1.000     1.000     1.000     1.000     1.000
```

Actual Representation of Matrix A

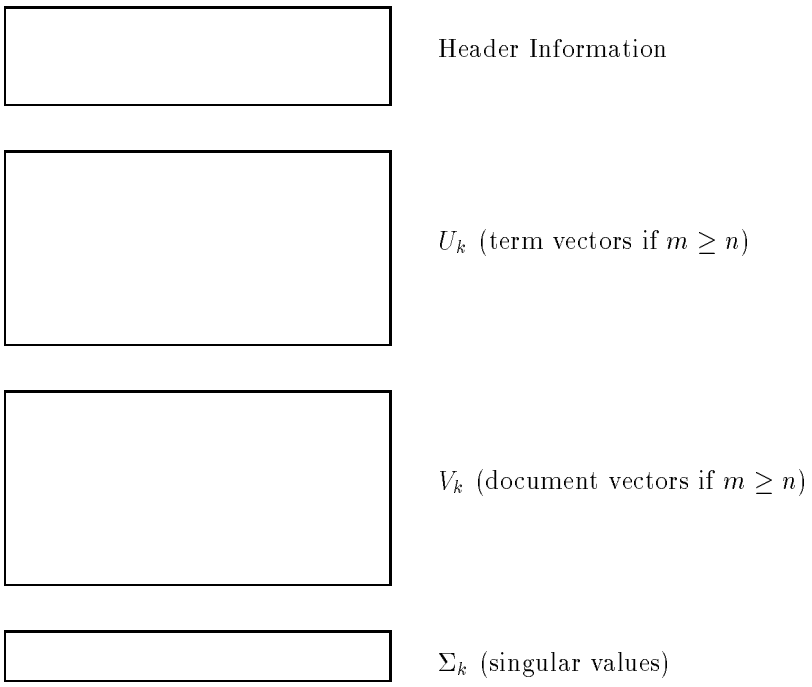
	1	2	3	4	5	6	7
1			1.000		1.000	1.000	1.000
2							
3	1.000	1.000	1.000		1.000		
4							1.000
5		1.000					1.000
6			1.000				
7							
8						1.000	1.000
9	1.000		1.000	1.000			1.000
10	1.000					1.000	
11	1.000						
12				1.000			

Appendix B

out File Format

The *out* file is a binary file consisting of a header (parameter list) and the matrix components of the truncated SVD (singular values and corresponding singular vectors). Figure B.1 shows a general overview of the *out* file.

Figure B.1: *out* file contents.



- *Out* file header information is a structure described in table B.1.

Table B.1: Header Information.

Label	Size	Description
header.size[COMMENT]	128 bytes	Time stamp information
header.size[TERM]	4 bytes	Number or terms
header.size[DOCUMENT]	4 bytes	Number of documents
header.size[FACTOR]	4 bytes	Number of LSI factors
header.folded[TERM]	4 bytes	Number of updated terms
header.folded[DOCUMENT]	4 bytes	Number of updated documents

- U_k comprises header.size[TERM] $\times$ header.size[FACTOR] floating-point numbers.
- V_k comprises header.size[DOCUMENT] $\times$ header.size[FACTOR] floating-point numbers.
- Σ_k header.size[TERMS] floating-point numbers.

Appendix C

Weightings

Define A = $[a_{ij}]$, where $a_{ij} \equiv L(i,j) \times G(i)$, $L(i,j) \equiv$ local weighting for term i in document j, and $G(i) \equiv$ global weighting for term i.

1. Local weights are used to stress overall importance in a document.

2. Global weights are used to stress overall importance to the collection of documents.

Table C.1: Popular local weightings.

Term Frequency	frequency with which a given term appears in a given document.
Binary weighting	replaces any term frequency ≥ 1 with 1.
Logarithmic weighting	log(term frequency +1) dampens effects of large variances in frequencies.

Definitions

tf_{ij}	$\equiv$	frequency of term i in document j.
df_i	$\equiv$	number of documents containing term i.
gf_i	$\equiv$	frequency of term i in collection (global).
$ndocs$	$\equiv$	number of documents in collection.

Table C.2: Popular global weightings.

Normal	$G(i) \equiv \sqrt{\frac{1}{\sum_j (tf_{ij})^2}}$
GfIdf	$G(i) \equiv \frac{gf_i}{df_i}$
Idf	$G(i) \equiv log_2 \left(\frac{ndocs}{df_i} \right) + 1$
1 - Entropy (Noise)	$G(i) \equiv 1 - \sum_j \frac{p_{ij} log(p_{ij})}{log(ndocs)}$, where $p_{ij} = \frac{tf_{ij}}{df_i}$

1. Global weighting schemes give less weight to terms that occur frequently or in many documents.

2. Entropy is based on information-theoretic ideas which takes distribution of terms over documents into account.

3. Combination of a local log weight $[log(tf_{ij} + 1)]$ and a global entropy weight (LogEntropy) typically yields best improvement in retrieval performance.

Vita

Gavin William O'Brien was born in Presque Isle, Maine on May 25, 1965. He graduated from Presque Isle High School in 1983 and received a Bachelor of Science degree in Mathematics from Bates College in May 1987. After living in Boston, MA from 1987-1991, he moved to Knoxville, TN where he received his Masters in Computer Science.