2

Mathematical Foundations Behind Latent Semantic Analysis

Dian I. Martin
Small Bear Technical Consulting, LLC

Michael W. Berry
University of Tennessee

Latent semantic analysis (LSA)is based on the concept of vector space models, an approach using linear algebra for effective yet automated information retrieval. The vector space model (VSM) was developed to handle text retrieval from a large information database where the text is heterogeneous and the vocabulary varies. One of the first systems to use a traditional VSM was the System for the Mechanical Analysis and Retrieval of Text (SMART; Buckley, Allan, & Salton, 1994; Salton & McGill, 1983). Among the notable characteristics of the VSM, used by SMART, is the premise that the meaning of documents can be derived from its components or types. The underlying formal mathematical model of the VSM defines unique vectors for each type and document, and queries are performed by comparing the query representation to the representation of each document in the vector space. Query-document similarities are then based on concepts or similar semantic content (Salton, Buckley, & Allan, 1992).

LSA is considered a truncated vector space model that represents types and documents in a particular high-dimensional type-document vector space. The truncated VSM, VSM_k, used in LSA, uncovers the underlying or "latent" semantic structure in the pattern of type usage to define documents in a collection. As mentioned in chapter 1 (Landauer, this volume), a document is defined as the sum of the meaning of its types. By using the truncated singular value decomposition, LSA exploits the meaning of types by removing "noise" that is present due to the variability in type choice. Such noise is evidenced by polysemy (multiple meanings for one word) and synonymy (many words describing the same idea) found in documents (Deerwester, Dumais, Furnas, Landauer, & Harshman, 1990; Furnas, Landauer, Gomez, & Dumais, 1987). As a result, the similarity of documents is no longer dependent on the types they contain, but on the semantic content; therefore, documents deemed relevant to a given query do not necessarily contain the types in the query (Dumais, 1991; Letsche & Berry, 1997).

This chapter describes the vector space model and the mathematical foundations that LSA is based on both in broad concepts for readers seeking general understanding and finer detail for readers interested in the specifics. Creating the vector space model for latent semantic analysis is discussed first, which is a theoretical explanation of how a type-by-document input matrix derived from a document collection is decomposed into a vector space of type and document vectors by the singular value decomposition (SVD). It is by using the truncated SVD that LSA obtains the meanings of types and documents. In the next section, the actual computation of the vector space for LSA is described in fine detail. The computation required to convert an input matrix into vectors is complex, and this section is for those readers interested in understanding the sophisticated mathematical calculations of the "Lanczos algorithm with selective reorthogonalization" that is used to build the type and document vectors. Then, use of the truncated vector space model for LSA is covered. This section describes the basic manipulations of the vector space. Subsequent chapters of this volume describe, in detail, the many interesting and important applications that use the vector space model of LSA.

CREATING THE VECTOR SPACE MODEL FOR LATENT SEMANTIC ANALYSIS

The Input Matrix

To create a vector space model for latent semantic analysis, a type-by-document matrix must first be constructed. The rows of the input matrix are comprised of types, which are the individual components that make up a

document. Typically, these individual components are terms, but they can be phrases or concepts depending on the application. The columns of the input matrix are comprised of documents, which are of a predetermined size of text such as paragraphs, collections of paragraphs, sentences, book chapters, books, and so on, again depending on the application. A document collection composed of n documents and m types can be represented as an m by n type-by-document matrix **A**. Often $m >> n$, the number of types is greater than the number of documents, however, there are cases where this is reversed and $n >> m$, for example, when collecting documents from the Internet (Berry & Browne, 2005; Berry, Drmac, & Jessup, 1999). Initially, each column of the matrix **A** contains zero and nonzero elements, a_{ij}. Each nonzero element, a_{ij}, of the matrix **A** is the frequency of ith type in the jth document. A small example of a document collection and its corresponding input type-by-document matrix with type frequencies can found in Tables 2.1 and 2.2 (Witter & Berry, 1998). In this example, documents are the actual title, consisting only of italicized keywords. Documents labeled M1–M5 are music-related titles, documents labeled B1–B4 are baking-related titles, and no document has more than one occurrence of a type or keyword.

A weighting function is generally applied to the each nonzero (type frequency) element, a_{ij}, of **A** to improve retrieval performance (Berry & Browne, 2005; Dumais, 1991). In retrieval, the types that best distinguish particular documents from the rest of the documents are the most important; therefore, a weighting function should give a low weight to a high-frequency type that occurs in many documents and a high weight to types that occur in some documents but not all (Salton & Buckley, 1991). LSA applies

TABLE 2.1
Titles for Topics on Music and Baking

Label	*Titles*
M1	*Rock* and *Roll Music* in the 1960's
M2	Different *Drum Roll*s, a Demonstration of Techniques
M3	*Drum* and Bass *Composition*
M4	A Perspective of *Rock Music* in the 90's
M5	*Music* and *Composition* of Popular Bands
B1	How to Make *Bread* and *Roll*s, a *Demonstration*
B2	*Ingredients* for Crescent *Roll*s
B3	A *Recipe* for Sour*dough Bread*
B4	A Quick *Recipe* for Pizza *Dough* using Organic *Ingredients*

Note. Keywords are in italics.

TABLE 2.2
The 10 x 9 Type-by-Document Matrix With Type Frequencies Corresponding to the Titles in Table 2.1

Types	*Documents*								
	M1	M2	M3	M4	M5	B1	B2	B3	B4
Bread	0	0	0	0	0	1	0	1	0
Composition	0	0	1	0	1	0	0	0	0
Demonstration	0	1	0	0	0	1	0	0	0
Dough	0	0	0	0	0	0	0	1	1
Drum	0	1	1	0	0	0	0	0	0
Ingredients	0	0	0	0	0	0	1	0	1
Music	1	0	0	1	1	0	0	0	0
Recipe	0	0	0	0	0	0	0	1	1
Rock	1	0	0	1	0	0	0	0	0
Roll	1	1	0	0	0	1	1	0	0

both a local and global weighting function to each nonzero element, a_{ij}, in order to increase or decrease the importance of types within documents (local) and across the entire document collection (global). The local and global weighting functions for each element, a_{ij}, are usually directly related to how frequently a type occurs within a document and inversely related to how frequently a type occurs in documents across the collection, respectively. So, a_{ij} = local(i, j) * global(i), where local(i, j) is the local weighting for type i in document j, and global(i) is the type's global weighting (Dumais, 1991; Letsche & Berry, 1997). Local weighting functions include using type frequency, binary frequency (0 if the type is not in the document and 1 if the type is in the document), and log of type frequency plus 1. Global weighting functions include normal, gfidf, idf, and entropy, all of which basically assign a low weight to types occurring often or in many documents. A common local and global weighting function is log-entropy. Dumais found that log-entropy gave the best retrieval results, 40% over raw type frequency (Dumais, 1991). The local weighting function of log (type frequency + 1) decreases the effect of large differences in frequencies. Entropy, defined as $1 + \sum_j \frac{p_{ij} \log_2 (p_{ij})}{\log_2 n}$ where $p_{ij} = \frac{tf_{ij}}{gf_i}$, tf_{ij} = type frequency of type i in document j, and gf_i = the total number of times that type i appears in the entire collection of n documents, gives less weight to types occurring frequently in a document collection, as well as taking into account the distribution of types

over documents (Dumais, 1991). A more detailed description of the local and global weighting functions can be found in Berry and Browne (2005) and Dumais (1991). Table 2.3 has the local and global weighting function log-entropy applied to each nonzero type frequency in the type-by-document matrix given previously in Table 2.2.

Typically, the type-by-document input matrix **A** is considered sparse because it contains many more zero entries than nonzero entries. Each document in the collection tends to only use a small subset of types from the type set. Usually, only about 1% or less of the matrix entries are populated or nonzero (Berry & Browne, 2005). In the small example in Tables 2.2 and 2.3, approximately 25% of the matrix entries are nonzero.

Decomposition of Input Matrix Into Orthogonal Components

Once the input matrix **A** is created, it is transformed into a type and document vector space by orthogonal decompositions in order to exploit truncation of the vectors. Transforming a matrix by using orthogonal decompositions, or orthogonal matrices, preserves certain properties of the matrix, including the norms, or vector lengths and distances, of the row and column vectors that form the $m \times n$ type-by-document input matrix **A**. Specifically, orthogonal matrix decompositions preserve the 2-norm and the Frobenius norm of matrix **A** (Golub & Van Loan, 1989; Larson & Edwards, 1988).

TABLE 2.3
The 10 × 9 Weighted Type-by-Document Matrix Corresponding to the Titles in Table 2.1

Types	*Documents*								
	M1	M2	M3	M4	M5	B1	B2	B3	B4
Bread	0	0	0	0	0	.474	0	.474	0
Composition	0	0	.474	0	.474	0	0	0	0
Demonstration	0	.474	0	0	0	.474	0	0	0
Dough	0	0	0	0	0	0	0	.474	.474
Drum	0	.474	.474	0	0	0	0	0	0
Ingredients	0	0	0	0	0	0	.474	0	.474
Music	.347	0	0	.347	.347	0	0	0	0
Recipe	0	0	0	0	0	0	0	.474	.474
Rock	.474	0	0	.474	0	0	0	0	0
Roll	.256	.256	0	0	0	.256	.256	0	0

What is an orthogonal matrix? An orthogonal matrix is one with the property of $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$, where $\mathbf{Q}$ is an orthogonal matrix, $\mathbf{Q}^T$ is the transpose of matrix $\mathbf{Q}$ (the rows and columns of $\mathbf{Q}$ are the columns and rows of $\mathbf{Q}^T$), and $\mathbf{I}$ is the identity matrix:

$$Q^T Q = \begin{bmatrix} 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & & 0 \\ \vdots & 0 & \ddots & \ddots & \vdots \\ 0 & & \ddots & 1 & 0 \\ 0 & 0 & \dots & 0 & 1 \end{bmatrix} \qquad \textbf{2.1}$$

If $\mathbf{Q}$ is comprised of n column vectors, that is, $\mathbf{Q} = [\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_n]$, then for every pair of vectors $(\mathbf{q}_i, \mathbf{q}_j)$, taking the dot product $\mathbf{q}_i^T \mathbf{q}_j = 0$ if $i \neq j$ and $\mathbf{q}_i^T \mathbf{q}_j \neq 0$ if $i = j$. If two vectors satisfy the property that the dot product $\mathbf{q}_i^T \mathbf{q}_j = 0$ if i $\neq j$, then they are said to be orthogonal. Taking this a step further, an orthogonal matrix is also orthonormal. Each vector $\mathbf{q}_i$ is orthonormal if the length of $\mathbf{q}_i = 1$, denoted by $\| \mathbf{q}_i \| = 1$, which means $\mathbf{q}_i^T \mathbf{q}_i = 1$. Because the vectors $[\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_n]$ in matrix $\mathbf{Q}$ are orthonormal, they point in totally different directions, forming 90-degree angles between each and every vector. Moreover, the vectors in $\mathbf{Q}$ from an orthonormal basis for a vector space, meaning every vector in the vector space can be written as a linear combination of vectors $[\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_n]$. More specifically, the vectors $[\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_n]$ span the vector space and are linearly independent, that is, $c_1\mathbf{q}_1 + c_2\mathbf{q}_2 + \dots + c_n\mathbf{q}_n = 0$ if and only if the scalars $c_i = 0$ (Golub & Van Loan, 1989).

There is more than one method for decomposing the type-by-document matrix $\mathbf{A}$ into orthogonal components. One method is the QR factorization, which is described in detail in Berry and Browne (2005) and Berry et al. (1999), another method called the ULV low-rank orthogonal decomposition is described in detail in Berry and Fierro (1996), and yet another method called the semi-discrete decomposition (SDD) is described in detail in Kolda and O'Leary (1998). Whereas all these methods are viable options, the most popular method used by LSA to decompose the type-by-document input matrix $\mathbf{A}$ is the singular value decomposition (SVD). The SVD is generally the chosen orthogonal matrix decomposition of input matrix $\mathbf{A}$ for various reasons. First, the SVD decomposes $\mathbf{A}$ into orthogonal factors that represent both types and documents. Vector representations for both types and documents are achieved simultaneously. Second, the SVD sufficiently captures the underlying semantic structure of a collection and allows for adjusting the representation of types and documents in the vector space by choosing the number of dimensions (more on this later). Finally, the computation of the SVD is manageable for large datasets, especially

now with current computer architectures (including clusters and symmetric multiprocessors; Berry & Martin, 2005).

The SVD for a $m \times n$ type-by-document input matrix $\mathbf{A}$ as described earlier, with the rank (number of vectors in the basis of the column space or the vector subspace spanned by the column vectors) of $\mathbf{A} = r$, is defined as follows:

$$\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T \qquad \textbf{2.2}$$

where $\mathbf{U}$ is an orthogonal matrix ($\mathbf{U}^T\mathbf{U} = \mathbf{I}_m$), $\mathbf{V}$ is an orthogonal matrix ($\mathbf{V}^T\mathbf{V} = \mathbf{I}_n$), and Σ is a diagonal matrix ($\Sigma =$ diagonal$(\sigma_1, \sigma_2, \ldots, \sigma_n)$ with the remaining matrix cells all zeros [Golub & Van Loan, 1989]). The first r columns of the orthogonal matrix $\mathbf{U}$ contain r orthonormal eigenvectors associated with the r nonzero eigenvalues[1] of $\mathbf{A}\mathbf{A}^T$. The first r columns of the orthogonal matrix $\mathbf{V}$ contain r orthonormal eigenvectors associated with the r nonzero eigenvalues of $\mathbf{A}^T\mathbf{A}$. The first r diagonal entries of Σ are the nonnegative square roots of the r nonzero eigenvalues of $\mathbf{A}\mathbf{A}^T$ and $\mathbf{A}^T\mathbf{A}$. The rows of matrix $\mathbf{U}$ are the type vectors and are called left singular vectors. The rows of $\mathbf{V}$ are the document vectors and are called right singular vectors. The nonzero diagonal elements of Σ are known as the singular values (Berry, Dumais, & O'Brien, 1995).

Truncation of Orthogonal Components

A pictorial representation of the SVD of input matrix $\mathbf{A}$ and the best rank-k approximation to $\mathbf{A}$ can be seen in Figure 2.1 (Berry et al., 1995; Witter & Berry, 1998). Given the fact that $\mathbf{A}$ can be written as the sum of rank 1 matri-

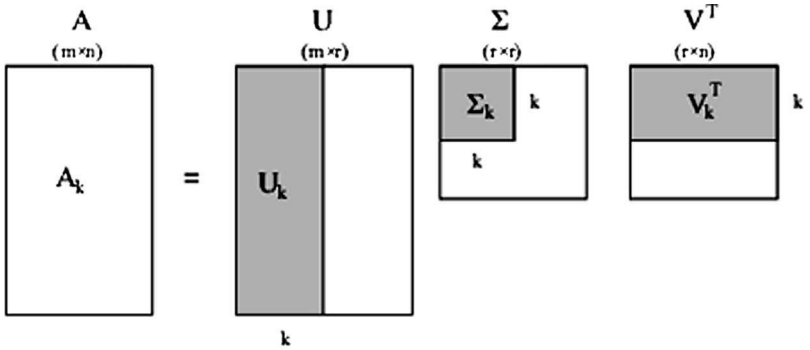

Figure 2.1. Diagram of the truncated SVD.

[1]As defined in Larson and Edwards (1988), an eigenvector and eigenvalue of the matrix $\mathbf{A}$ satisfy $\mathbf{Ax} = \lambda\mathbf{x}$, where $\mathbf{x}$ is an eigenvector and λ is its corresponding eigenvalue. Two properties of eigenvectors and eigenvalues that play a role in the SVD computation are that all eigenvalues of a symmetric matrix ($\mathbf{B}^T = \mathbf{B}$) are real, and the eigenvectors associated with each distinct eigenvalue for a symmetric matrix are orthogonal.

ces (Björck, 1996): $A = \sum_{i=1}^{r} u_i \sigma_i v_i^T$, r can be reduced to k to create $A_k = \sum_{i=1}^{k} u_i \sigma_i v_i^T$. The matrix $\mathbf{A}_k$ is the best or closest (distance is minimized) rank k approximation to the original matrix $\mathbf{A}$ (Björck, 1996; Berry & Browne, 2005; Berry et al., 1995; Berry et al., 1999). The matrix $\mathbf{A}_k$ ($\mathbf{A}_k = \mathbf{U}_k \Sigma_k \mathbf{V}_k^T$) is created by ignoring or setting equal to zero all but the first k elements or columns of the type vectors in $\mathbf{U}$, the first k singular values in Σ, and the first k elements or columns of the document vectors in $\mathbf{V}$. The first k columns of $\mathbf{U}$ and $\mathbf{V}$ are orthogonal, but the rows of $\mathbf{U}$ and $\mathbf{V}$, the type and document vectors, consisting of k elements are not orthogonal. By reducing the dimension from r to k, extraneous information and variability in type usage, referred to as "noise," which is associated with the database or document collection is removed. Truncating the SVD and creating $\mathbf{A}_k$ is what captures the important underlying semantic structure of types and documents. Types similar in meaning are "near" each other in k-dimensional vector space even if they never co-occur in a document, and documents similar in conceptual meaning are near each other even if they share no types in common (Berry et al., 1995). This k-dimensional vector space is the foundation for the semantic structures LSA exploits.

Using the small document collection from Table 2.1 and its corresponding type-by-document matrix in Tables 2.2 and 2.3, the SVD can be computed and truncated to a two-dimensional vector space by reducing the rank to $k = 2$. Table 2.4 shows the SVD of the example type-by-document matrix given in Table 2.3. The values in the boldface cells in the type matrix $\mathbf{U}$, the document matrix $\mathbf{V}$, and the diagonal matrix of singular values Σ are used to encode the representations of types and documents in the two-dimensional vector space.

Figure 2.2 shows a rank-2, $k = 2$, plot of the types, represented by squares, and documents, represented by triangles, in the music and baking titles collection. Each point represents a type or document vector, a line starting at the origin and ending at a defined type or document point. The (x, y) pair is defined by x = first dimension or column of matrix $\mathbf{U}$ or $\mathbf{V}$ multiplied by the first singular value and y = second dimension or column of matrix $\mathbf{U}$ or $\mathbf{V}$ multiplied by the second singular value for type and document points, respectively. Looking at the vectors for the types and documents, the types most similar to each other and the documents most similar to each other are determined by the angles between vectors (more on this in the Querying subsection). If two vectors are similar, then they will have a small angle between them. In Figure 2.2, the documents M4, "A Perspective of Rock Music in the 90's," and M1 "Rock and Roll Music in the 1960's" are the closest documents to document M3, "Drum and Bass Composition," and yet they share no types in common. Similarly, the type vector for "music" is closest

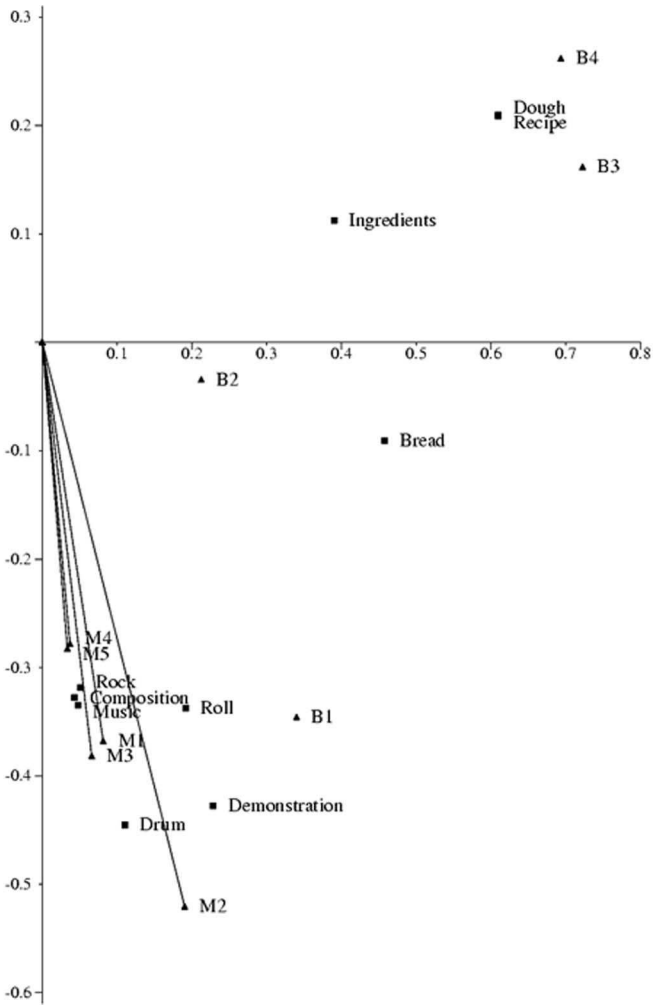

Figure 2.2. The rank-2 LSA vector space for the music/baking titles collection.

to type vectors "rock" and "composition," however, the next closest type vector corresponds to the type vector for "drum." This similarity is notable because "music" and "drum" never co-occur in the same document.

The best selection of rank or number of dimensions to use in the space remains an open question. In practice, the choice for k depends on empirical testing. For large datasets, empirical testing shows that the optimal choice for the number of dimensions ranges between 100 and 300 (Berry et al., 1999; Jessup & Martin, 2001; Lizza & Sartoretto, 2001). As stated previously, whatever the choice for k, the rank-k matrix, $\mathbf{A}_k$, constructed by the truncated SVD factors,

TABLE 2.4
The SVD of the Weighted Type-by-Document Matrix Represented in Table 2.3

Matrix U-Type Vectors									
Bread	**.42**	**–.09**	–.20	.33	–.48	–.33	.46	–.21	–.28
Composition	**.04**	**–.34**	.09	–.67	–.28	–.43	.02	–.06	.40
Demonstration	**.21**	**–.44**	–.42	.29	.09	–.02	–.60	–.29	.21
Dough	**.55**	**.22**	.10	–.11	–.12	.23	–.15	.15	.11
Drum	**.10**	**–.46**	–.29	–.41	.11	.55	.26	–.02	–.37
Ingredients	**.35**	**.12**	.13	–.17	.72	–.35	.10	–.37	–.17
Music	**.04**	**–.35**	.54	.03	–.12	–.16	–.41	.18	–.58
Recipe	**.55**	**.22**	.10	–.11	–.12	.23	–.15	.15	.11
Rock	**.05**	**–.33**	.60	.29	.02	.33	.28	–.35	.37
Roll	**.17**	**–.35**	–.05	.24	.33	–.19	.25	.73	.22
Matrix Σ-Singular Values									
	1.10	0	0	0	0	0	0	0	0
	0	**.96**	0	0	0	0	0	0	0
	0	0	.86	0	0	0	0	0	0
	0	0	0	.76	0	0	0	0	0
	0	0	0	0	.66	0	0	0	0
	0	0	0	0	0	.47	0	0	0
	0	0	0	0	0	0	.27	0	0
	0	0	0	0	0	0	0	.17	0
	0	0	0	0	0	0	0	0	.07
	0	0	0	0	0	0	0	0	0
Matrix V-Document Vectors									
M1	**.07**	**–.38**	.53	.27	.08	.12	.20	.50	.42
M2	**.17**	**–.54**	–.41	.00	.28	.43	–.34	.22	–.28
M3	**.06**	**–.40**	–.11	–.67	–.12	.12	.49	–.23	.23
M4	**.03**	**–.29**	.55	.19	–.05	.22	–.04	–.62	–.37
M5	**.03**	**–.29**	.27	–.40	–.27	–.55	–.48	.21	–.17
B1	**.31**	**–.36**	–.36	.46	–.15	–.45	.00	–.32	.31
B2	**.19**	**–.04**	.06	–.02	.65	–.45	.41	.07	–.40
B3	**.66**	**.17**	.00	.06	–.51	.12	.27	.25	–.35
B4	**.63**	**.27**	.18	–.24	.35	.10	–.35	–.20	.37

produces the best approximation to the type-by-document input matrix **A**, always. Keep in mind that this does not equate to the optimal number dimensions to use in certain applications; meaning one should not always use the first 100–300 dimensions. For some applications it is better to use a subset of the first 100 or 300 dimensions or factors (Landauer & Dumais, 1997).

COMPUTING THE VECTOR SPACE MODEL FOR LATENT SEMANTIC ANALYSIS

Solving an Eigenproblem

Computing the reduced dimensional vector space for a given type-by-document input matrix is a nontrivial calculation. Given a realistic, large $m \times n$ type-by-document matrix **A** where $m \geq n$, computing the SVD becomes a problem of finding the k largest eigenvalues and eigenvectors of the matrix $\mathbf{B} = \mathbf{A}^{\mathrm{T}}\mathbf{A}$. Finding the eigenvectors of **B** produces the document vectors (recall from the Decomposition of Input Matrix into Orthogonal Components subsection the columns of orthogonal matrix **V** in the SVD are the eigenvectors of **B**), and finding the eigenvalues of **B** produces the singular values (the nonnegative square roots of the eigenvalues of **B**). The type vectors are produced by back multiplying, $\boldsymbol{U}_k = \boldsymbol{A}\boldsymbol{V}_k\Sigma_k^{-1}$. If $n > m$, there are more documents than types, then computing the SVD is reduced to finding the k largest eigenvalues and eigenvectors of $\mathbf{B} = \mathbf{A}\mathbf{A}^{\mathrm{T}}$. In this case, finding the eigenvectors of **B** produces the type vectors (recall from the Decomposition of Input Matrix into Orthonal Components subsection the columns of orthogonal matrix **U** in the SVD are the eigenvectors of **B**), and again finding the eigenvalues of **B** produces the singular values (the nonnegative square roots of the eigenvalues of **B**). As with type vectors in the previous case, the document vectors are produced by back multiplying, $\boldsymbol{V}_k = \boldsymbol{A}^T\boldsymbol{U}_k\Sigma_k^{-1}$. To summarize, given the symmetric matrix **B**, which is created from the sparse input matrix **A**, the objective is to find the k largest eigenvalues and eigenvectors of **B**. Thus, the SVD computation is based on solving a large, sparse symmetric eigenproblem (Berry, 1992; Golub & Van Loan, 1989).

Basic Lanczos Algorithm

The Lanczos algorithm is proven to be accurate and efficient for large, sparse symmetric eigenproblems where only a modest number of the largest or smallest eigenvalues of a matrix are desired (Golub & Van Loan, 1989; Parlett & Scott, 1979). The Lanczos algorithm, which is an iterative method

and most often used to compute the k largest eigenvalues and eigenvectors of **B**, actually approximates the eigenvalues of **B** (Berry & Martin, 2005).

The Lanczos algorithm involves the partial tridiagonalization of matrix **B**, where a tridiagonal matrix is defined as follows:

$$\boldsymbol{T} = \begin{bmatrix} \alpha_1 & \beta_1 & 0\ldots & 0\ldots \\ \beta_1 & \alpha_2 & \beta_2 & 0\ldots \\ \ldots & \ldots & \ldots & \ldots \\ 0\ldots & 0 & \beta_{v-1} & \alpha_v \end{bmatrix} \qquad \textbf{2.3}$$

where $\mathbf{T}_{ij} = 0$ whenever $|i - j| > 1$ (Parlett, 1980). A tridiagonal matrix **T** is considered unreduced if $\beta_i \neq 0$, $i = 1, \ldots, v - 1$. A sequence of symmetric tridiagonal matrices $\mathbf{T}_j$ is generated by the algorithm with the property that eigenvalues of $\mathbf{T}_j$ are progressively better estimates of **B**'s largest (or smallest) eigenvalues (Berry, 1992). These eigenvalues emerge long before tridiagonalization is complete, before the complete v × v tridiagonal matrix is produced. There are important reasons for transforming matrix **B** to an unreduced tridiagonal matrix. First, eigenvalues and eigenvectors of **T** can be found with significantly fewer arithmetic operations than for **B**. Second, every symmetric matrix **B** can be reduced to **T** by a finite number of elementary orthogonal transformations ($\mathbf{Q}^T\mathbf{B}\mathbf{Q} = \mathbf{T}$). Finally, as long as **T** is unreduced, **T**'s eigenvalues are distinct; there are not multiple eigenvalues with the same value (Parlett, 1980).

Step 1 of the Lanczos algorithm is to tridiagonalize the symmetric matrix **B**. To compute $\mathbf{T}_j = \mathbf{Q}^T\mathbf{B}\mathbf{Q}$, define the orthogonal matrix $\mathbf{Q} = [\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_j]$ where the vectors $\mathbf{q}_i$, known as the Lanczos vectors, are orthonormal vectors. To reduce **B** to a tridiagonal matrix step by step, the vectors for $\mathbf{Q}_j$ and the entries of $\mathbf{T}_j$ are computed one column at a time by the following the basic Lanczos recursion (Parlett, 1980; Parlett & Scott, 1979):

1. Generate a random vector $\mathbf{q}_1$ such that it is orthonormal ($\|\mathbf{q}_1\| = 1$).
2. Define $\beta_0 \equiv 0$ and $\mathbf{q}_0 \equiv 0$.
3. Recursively do the following for $j = 1, 2, \ldots, l$:
   $\boldsymbol{r}_j = \mathbf{B}\mathbf{q}_j - \mathbf{q}_{j-1}\beta_{j-1}$;
   $\alpha_j = \mathbf{q}_j^T\boldsymbol{r}_j = \mathbf{q}_j^T \times (\mathbf{B}\mathbf{q}_j - \mathbf{q}_{j-1}\beta_{j-1}) = \mathbf{q}_j^T\mathbf{B}\mathbf{q}_j$,
   where $\mathbf{q}_j^T\,\mathbf{q}_{j-1}\beta_{j-1} = 0$ due to orthogonality;
   $\boldsymbol{r}_j = \boldsymbol{r}_j - \mathbf{q}_j\alpha_j$;
   $\beta_j = \|\boldsymbol{r}_j\|$, where $\|\boldsymbol{r}_j\|$ is the vector length of $\boldsymbol{r}_j$;
   $\boldsymbol{q}_{j+1} = \dfrac{\boldsymbol{r}_j}{\beta_j} = \dfrac{\boldsymbol{r}_j}{\|\boldsymbol{r}_j\|}$, where $\mathbf{q}_{j+1}$ becomes orthonormal.

To determine α_j, $\mathbf{r}_j$, β_j and $\mathbf{q}_{j+1}$, at each step only β_j, $\mathbf{q}_{j-1}$, and $\mathbf{q}_j$ are needed. There are two things to observe about the computations in the Lanczos algorithm. First, by mathematical induction, it can be proven that the vectors $\mathbf{q}_i$ are uniquely determined by $\mathbf{B}$ and $\mathbf{q}_1$ (Parlett, 1980). Second, the vector $\mathbf{q}_{j+1}$ (= $\mathbf{B}\mathbf{q}_j - \beta_{j-1}\mathbf{q}_{j-1} - \alpha_j\mathbf{q}_j$), before normalizing or dividing by $\beta_j = \| \mathbf{r}_j \|$ to make its vector length equal to one, is built on the previous vectors $\mathbf{q}_{j-1}$ and $\mathbf{q}_j$; therefore, the next Lanczos vector $\mathbf{q}_{j+1}$ is obtained by orthogonalizing $\mathbf{B}\mathbf{q}_j$ with respect to $\mathbf{q}_{j-1}$ and $\mathbf{q}_j$ (Berry, 1992; Berry & Martin, 2005). This guarantees the orthogonality of vectors $\mathbf{q}_j$ in the matrix $\mathbf{Q} = [\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_j]$, and the normalizing of each Lanczos vector guarantees that the vectors $\mathbf{q}_j$ are orthonormal. Given these two observations, the vectors in $\mathbf{Q}_j$ form the orthonormal basis for a subspace known as a Krylov subspace (Golub & Van Loan, 1989; Parlett, 1980). A Krylov subspace is defined by the span of $\{\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_j\}$ = span of $\{\mathbf{q}_1, \mathbf{B}\mathbf{q}_1, \ldots, \mathbf{B}^{j-1}\mathbf{q}_1\}$. Therefore, the span of $\{\mathbf{Q}_j\}$ is a Krylov subspace for the matrix $\mathbf{B}$. The Lanczos procedure can be viewed as a technique for computing orthonormal bases for the Krylov subspaces at different j steps and for computing an orthogonal projection of $\mathbf{B}$ onto these subspaces (Berry, 1992).

Step 2 of the Lanczos algorithm is to examine the eigenvalues for $\mathbf{T}_j$ at various steps j to see if they are good approximations to the eigenvalues of $\mathbf{B}$. This is done by diagonalizing the tridiagonal matrix $\mathbf{T}_j$ and finding its eigenvalues and eigenvectors (remember an eigenvalue λ_k and its corresponding eigenvector $\mathbf{x}_k$ associated with $\mathbf{T}_j$ satisfy $\mathbf{T}_j\mathbf{x}_k = \lambda_k\mathbf{x}_k$). Let $\mathbf{T}_j = \mathbf{S}_j\Theta_j\mathbf{S}^T_j$, where $\mathbf{S}_j = [\mathbf{s}_1, \ldots, \mathbf{s}_j]$ is a $j \times j$ orthogonal matrix and $\Theta_j = \text{diag}(\theta_1, \ldots, \theta_j)$ is a diagonal matrix. The eigenvalues, $\lambda_k = \theta_k$ ($k = 1, 2, \ldots, j$), and eigenvectors, $\mathbf{x}_k = \mathbf{Q}_j\mathbf{s}_k$ ($k = 1, 2, \ldots, j$), of $\mathbf{T}_j$ are produced by the following derivation:

$$\begin{aligned} &\text{since } \mathbf{Q}^T{}_j\mathbf{B}\mathbf{Q}_j = \mathbf{T}_j = \mathbf{S}_j\Theta_j\mathbf{S}^T_j \\ &\text{then } (\mathbf{Q}_j\mathbf{S}_j)^T\mathbf{B}(\mathbf{Q}_j\mathbf{S}_j) = \Theta_j \text{ or} \\ &\mathbf{B}(\mathbf{Q}_j\mathbf{S}_j) = \Theta_j(\mathbf{Q}_j\mathbf{S}_j). \end{aligned} \qquad \textbf{2.4}$$

An eigenpair $(\lambda_k, \mathbf{x}_k)$ for $\mathbf{T}_j$ is known as a Ritz pair, where the eigenvalue λ_k is called a Ritz value, and the eigenvector $\mathbf{x}_k$ is called a Ritz vector (Golub & Van Loan, 1989; Parlett, 1980). These Ritz pairs are the best set of approximations to the desired eigenpairs of $\mathbf{B}$ (Parlett, 1980). The accuracy of these approximations has been studied in great depth by Kaniel and Saad (Berry, 1992; Parlett, 1980), and they conclude that the eigenvalues of $\mathbf{T}_j$ are good approximations to the extreme, largest or smallest, eigenvalues of $\mathbf{B}$. This accuracy is determined by examining the residual norm, $\|\mathbf{B}\mathbf{x}_k - \lambda_k\mathbf{x}_k\|$, of each Ritz pair. The residual norm can be calculated without constructing $\mathbf{x}_k$ ($k = 1, 2, \ldots, j$) by simply taking the bottom elements of the normalized

eigenvectors, $\mathbf{s}_k$ (k = 1, 2, … , j), of $\mathbf{T}_j$ (Parlett, 1980). If the accuracy of an eigenvalue of $\mathbf{T}_j$ is sufficient, then step 3 of the Lanczos algorithm is to compute the eigenvectors, $\mathbf{x}_k$ (k = 1, 2, … , j), of $\mathbf{T}_j$. Once an accepted eigenpair of **B** is determined, a singular vector and singular value have been found. This is considered step 4 of the Lanczos algorithm. If $\mathbf{B} = \mathbf{A}^T\mathbf{A}$, then a right singular vector has been found, and if $\mathbf{B} = \mathbf{A}\mathbf{A}^T$ then a left singular vector has been found. Table 2.5 outlines the basic steps of the Lanczos algorithm for computing the sparse SVD of input matrix **A**.

Lanczos Algorithm With Selective Reorthogonalization

Theoretically, it is easy to prove (and as already explained) that the basic Lanczos recursion guarantees orthogonality among all vectors in matrix $\mathbf{Q} = [\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_j]$ by only orthogonalizing the current Lanczos vector with the previous two Lanczos vectors (Parlett, 1980). However, in using finite-precision arithmetic, the Lanczos procedure suffers from the loss of orthogonality in the Lanczos vectors. This leads to "numerically multiple" eigenvalues, the same eigenvalue calculated multiple times, and unwanted eigenvalues of $\mathbf{T}_j$. There are a few options for dealing with these problems. One is to use total reorthogonalization where every Lanczos vector is orthogonalized against all previously generated Lanczos vectors. This approach requires a great deal of storage and many arithmetic operations and can complicate the Lanczos process. Another option is to disregard the loss in orthogonality among Lanczos vectors and deal with these problems directly, but in practice it is difficult to keep track of unwanted or "numerically multiple" eigenvalues. Currently, the best option for remedying these problems is to use selective reorthogonalization of the Lanczos vectors, which is known as the "Lanczos algorithm with selective reorthogonalization" (Berry, 1992; Parlett, 1980; Parlett & Scott, 1979). In

TABLE 2.5
Lanczos Algorithm for Computing the Sparse SVD

1. Use the basic Lanczos recursion to generate a sequence of symmetric tridiagonal matrices, $\boldsymbol{T}_i$ (i = 1, 2, …, p).

2. For some $j \leq p$, compute the eigenvalues of $\boldsymbol{T}_j$. Determine which eigenvalues of $\boldsymbol{T}_j$ are good approximations to the eigenvalues of $\boldsymbol{B}$.

3. For each accepted eigenvalue, or Ritz value, λ_k compute its corresponding eigenvector, or Ritz vector, $\mathbf{x}_k$, where $\mathbf{x}_k = Q_j\mathbf{s}_k$. The set of Ritz pairs are used as an approximation to the desired eigenvalues and eigenvectors of matrix $\boldsymbol{B}$.

4. For all accepted eigenpairs (λ_k, $\mathbf{x}_k$) compute the corresponding singular vectors and values for type-by-document input matrix $\boldsymbol{A}$.

this method, the Lanczos vectors are periodically reorthogonalized against the previous ones whenever a threshold for mutual orthogonality is exceeded (Berry & Martin, 2005). In a comparison study among several algorithms, it was found that the Lanczos algorithm using selective reorthogonalization was the fastest method for computing k of the largest singular vectors and corresponding singular values (Berry, 1992). With a recent parallel/distributed implementation of the Lanczos algorithm using selective reorthogonalization, the capacity to handle larger data collections, and thus input matrices, is now feasible (Berry & Martin, 2005). The time and storage burden of transforming a type-by-document input matrix into singular vectors and values has been reduced, increasing the suitability of this algorithm for computing the SVD.

USING THE VECTOR SPACE MODEL FOR LATENT SEMANTIC ANALYSIS

Querying

Once a reduced rank, or k-dimensional, vector space for types and documents has been produced, finding types and documents close to a given query becomes a simple process. A query is represented in the k-dimensional vector space much like a document; therefore, it is referred to as a *pseudo-document* (Deerwester et al., 1990). A query is the weighted sum of its type vectors scaled by the inverse of the singular values, this individually weights each dimension in the k-dimensional type-document vector space. A query can be represented by

$$query = \boldsymbol{q}^T \boldsymbol{U}_k \Sigma_k^{-1} \qquad \textbf{2.5}$$

where $\mathbf{q}^T$ is vector containing zeros and weighted type frequencies corresponding to the types specified in the query. The weighting is determined by applying one of the weighting functions described in the Input Matrix subsection.

Once a pseudo-document is formed and projected into the type-document space, a similarity measure is used to determine which types and documents are closest to the query. The cosine similarity measure is commonly used, and the cosine angle between the query, or pseudo-document, and each of the documents or types is computed. The cosines and the corresponding documents or types are then ranked in descending order; thus, the document or type with the highest cosine with the query is given first. Once the ranked list is produced, documents or types above a certain threshold are then deemed relevant (Letsche & Berry, 1997).

Referring back to the small music/baking titles collection example given in the Creating the Vector Space Model for Latent Semantic Analysis section, the query "Recipe for White Bread" can be computed as a pseudo-document and projected into the rank-2 space in Figure 2.2. Calculating the cosines of the query vector with each document vector, a ranked list of the largest cosines is shown in Table 2.6. Given a cosine threshold of greater than .80, documents B2, B3, B1, and B4 are the most relevant. Document B2, "Ingredients for Crescent Rolls," is found to be the most relevant to the query "Recipe for White Bread," even though this document has no types in common with the query.

One way to enhance a query and thereby retrieve relevant documents is to apply a method known as relevance feedback (Salton & Buckley, 1990). Given the query and the initial ranked list of documents and their corresponding cosines with the query, relevance feedback allows users to indicate which documents they think are relevant from the initial list. The query with the incorporation relevance feedback is represented by

$$query = \boldsymbol{q}^T \boldsymbol{U}_k \Sigma_k^{-1} + \boldsymbol{d}^T \boldsymbol{V}_k \qquad \textbf{2.6}$$

where $\mathbf{d}^T$ is a vector whose elements specify which documents to add to the query (Letsche & Berry, 1997). Again, this query is matched against the documents to obtain a ranked list of documents.

There are three sorts of comparisons that can be done in the vector space: comparing two types, comparing two documents, and comparing a type to a document. By definition, a query is considered a document, a pseudo-document; therefore, the comparison between two documents is the same as comparing a pseudo-document and a document. The same is true when comparing a document and a type. The analysis on these three types of comparisons was first described in Deerwester et al. (1990) and is presented here.

To find the degree of similarity between type *i* and type *j*, the dot product between row vectors in $\mathbf{A}_k$ is examined. Given the reduced matrix $\mathbf{A}_k = \mathbf{U}_k \Sigma_k \mathbf{V}_k^T$, the dot product between any two type vectors reflects the similar-

TABLE 2.6
Results for the Query "Recipe for White Bread" Using a Cosine Threshold of .80

Document	*Cosine*
B2: Ingredients for Crescent Rolls	.99800
B3: A Recipe for Sourdough Bread	.90322
B1: How to make Bread and Rolls, a Demonstration	.84171
B4: A Quick Recipe for Pizza Dough using Organic Ingredients	.83396

ity of types in the document collection. Therefore, if the (square) similarity matrix to obtain all the type-to-type dot products is defined as

$$\mathbf{A}_k\mathbf{A}_k^T = \mathbf{U}_k\Sigma_k\mathbf{V}_k^T(\mathbf{U}_k\Sigma_k\mathbf{V}_k^T)^T = \mathbf{U}_k\Sigma_k\mathbf{V}_k^T\mathbf{V}_k\Sigma_k\mathbf{U}_k^T = \mathbf{U}_k\Sigma_k\Sigma_k\mathbf{U}_k^T \quad \textbf{2.7}$$

then the dot product between any two types is the dot product between row i and j of $\mathbf{U}_k\Sigma_k$. Therefore, to perform a comparison in k-dimensional vector space between any two types, the type vectors scaled by the singular values are used to compute the similarity measure regardless of whether the similarity measure used is the cosine, Euclidean distance, or some other measure.

The comparison of two documents follows the same analysis. To determine the degree of similarity between two documents, the dot product between column vectors is examined. Given the reduced matrix $\mathbf{A}_k = \mathbf{U}_k\Sigma_k\mathbf{V}_k^T$, the dot product between any two document vectors indicates the extent to which two documents have similar type patterns or type meanings. If the (square) similarity matrix to compute all the document-to-document dot products is created by

$$\mathbf{A}_k^T\mathbf{A}_k = (\mathbf{U}_k\Sigma_k\mathbf{V}_k^T)^T\mathbf{U}_k\Sigma_k\mathbf{V}_k^T = \mathbf{V}_k\Sigma_k\mathbf{U}_k^T\mathbf{U}_k\Sigma_k\mathbf{V}_k^T = \mathbf{V}_k\Sigma_k\Sigma_k\mathbf{V}_k^T \quad \textbf{2.8}$$

then the dot product between any two documents is the dot product between row i and j of $\mathbf{V}_k\Sigma_k$. Therefore, to do a comparison between any two documents, or a document and a pseudo-document, the document vectors, or pseudo-document vector, scaled by the singular values are used to compute the similarity measure.

The comparison between a type and a document is different than the comparison between any two types or any two documents. In this case, the comparison is analyzed by looking at an element of $\mathbf{A}_k$. Remember that the reduced rank matrix is defined as $\mathbf{A}_k = \mathbf{U}_k\Sigma_k\mathbf{V}_k^T$, and the element a_{ij} of $\mathbf{A}_k$ is obtained by taking the dot product of row vector **i** in $\mathbf{U}_k$ scaled by $\sqrt{\Sigma_k}$ and the row vector **j** in $\mathbf{V}_k$ scaled by $\sqrt{\Sigma_k}$. Thus, following the same procedure as with types and documents, when computing a comparison, regardless of similarity measure used, between a type and a document or pseudo-document, the corresponding type vector and document vector scaled by the square root of the singular values are needed.

Updating

The ability to add new types and documents to the reduced rank type-document vector space is important because the original information in the document collection oftentimes needs to be augmented for different contextual or conceptual usages. One of three methods, "folding-in," recom-

puting the SVD, or SVD-updating, described in Berry et al. (1995) is generally used when updating or adding new types or documents to an existing vector space. However, to date, there is no optimal way to add information to an existing type-document space that is more accurate than recomputing the *k*-dimensional vector space with the added information while directly and accurately affecting the underlying latent structure in the document collection.

The simple way of handling the addition of types and documents is to "fold" types or documents into the *k*-dimensional vector space. The "folding-in" procedure is based on the existing type-document vector space. As with querying, to fold-in a document, a pseudo-document is built. A new document, $\boldsymbol{d}$, is folded into the existing *k*-dimensional vector space by projecting $\boldsymbol{d}$ onto the span of the current type vectors by computing $\boldsymbol{d}_{new} = \boldsymbol{d}^T \boldsymbol{U}_k \boldsymbol{\Sigma}_k^{-1}$. The vector $\mathbf{d}$, representing a document, contains zero and nonzero elements where the nonzero elements correspond to the type frequencies contained in the document adjusted by a weighting function described in the Input Matrix subsection. Similarly, to fold a new type vector, $\mathbf{t}$, into an existing *k*-dimensional vector space, a projection of $\mathbf{t}$ onto the space of the current document vectors is computed by $\boldsymbol{t}_{new} = \boldsymbol{t}\boldsymbol{V}_k \boldsymbol{\Sigma}_k^{-1}$. In this case, the vector $\mathbf{t}$ contains zero and nonzero elements where the nonzero elements are weighted elements corresponding to the documents that contain the type. Both the vectors of $\mathbf{d}_{new}$ and $\mathbf{t}_{new}$ are added to the vector space as another document and type, respectively. This method is a quick and easy way to add new types and documents to a vector space, but by no means does it change the existing type-document vector space. Essentially the new types and documents have no effect on the underlying semantic structure or meanings of types and documents.

The best way to produce a *k*-dimensional type-document vector space with new types and documents playing a role in the meanings of types and documents is to reproduce the type-by-document matrix with the added types and documents, recompute the SVD, and regenerate the reduced rank vector space. The exact effects of adding those specific types or documents are reflected in the vector space. Of course, the expense of this method is time. Although time efficient ways of calculating the SVD are being developed, recomputation is still computationally intensive (Berry & Martin, 2005).

One algorithm that has been proposed in literature as an alternative to folding in and recomputing the SVD is the SVD-updating algorithm (Berry et al., 1995). Performing the SVD-updating technique requires three steps: adding new documents, adding new types, and correcting for changes in type weightings. All steps use the reduced rank vector space, $\mathbf{A}_k$, to incorporate new types or documents into the existing type-document space by exploiting the previously computed singular vectors and values. This tech-

nique is definitely more difficult and more computationally complex than folding-in, but it does guarantee the orthogonality of vectors among the existing and new type and document vectors. Although the SVD-updating technique does try and mimic the effects of new types and documents on the underlying semantic structure, it will deviate somewhat from the actual recomputation of the reduced rank space using the same data because the update is based on $\mathbf{A}_k$ and not an original type-by-document matrix $\mathbf{A}$ (Berry et al., 1995). An updating method, which is guaranteed to match the results of a recomputed SVD, is given in Zha and Simon (1999).

Downdating

Following the same argument as updating, the ability to remove types and documents from a reduced rank vector space is also important because there are times when the original information in a document collection needs to be diminished for different contextual and conceptual usages. Again there are three methods—"folding-out," recomputing the SVD, or "downdating the reduced model" method—used in downdating or removing a type or document from an existing vector space. However, as with updating, to date there is no way to remove information from an existing type-document space that accurately affects the underlying semantic structure of a document collection and is more expedient than recomputing the original k-dimensional vector space.

The simplest method for removing information from a type-document vector space is "folding-out" of types or documents. This technique simply ignores the unwanted types and documents as if they were absent from the vector space and thus the document collection. The types and documents are no longer used in comparisons.

Of course, the method of removing types or documents from a document collection, reproducing the type-by-document matrix, recomputing the SVD, and regenerating the k-dimensional vector space is always an option. Recreating the type-document vector space with certain types and documents removed is definitely the most accurate in showing the effects of the removed information on the underlying semantic structure of the original database. As stated in the Updating subsection, this requires computational complexity and time.

One other algorithm, described in Witter and Berry (1998), called downdating the reduced model (DRM), tries to reflect the change that removing types and documents has on the reduced rank vector space without recomputing the SVD as a new type-by-document vector space. This technique involves three steps: removing a type, removing a document, and updating type weights from the reduced rank vector space $\mathbf{A}_k$. The pre-

viously calculated singular vectors and values are used in downdating the vector space. This algorithm approximates the effects that removing types and documents have on the reduced rank vector space, and it maintains orthogonality among the existing and new type and document vector columns. The DRM method is not as accurate as recomputing the vector space, but it is more accurate than folding-out. But likewise it is also computationally more time efficient than recomputing the vector space but much slower that folding-out.

CONCLUSIONS

Latent semantic analysis (LSA) uses a reduced rank vector space model to exploit the latent semantic structure of type-document associations. As evidenced by this chapter, creating, calculating, and using the reduced rank vector space model is nontrivial and based on sophisticated numerical algorithms. The mathematical foundations laid out in this chapter are the basis for which the applications of LSA are built on. The remaining chapters of this volume describe the various LSA applications and manipulations that exploit the reduced rank vector space model and structure.

REFERENCES

Berry, M. W. (1992). Large sparse singular value computations. *International Journal of Supercomputer Applications*, *6*, 13–49.

Berry, M. W., & Browne, M. (2005). *Understanding search engines: Mathematical modeling and text retrieval* (2nd ed.). Philadelphia: SIAM.

Berry, M. W., & Drmac, Z., & Jessup, E. (1999). Matrices, vector spaces, and information retrieval. *SIAM Review*, *41*, 335–362.

Berry, M. W., Dumais, S., & O'Brien, G. (1995). Using linear algebra for intelligent information retrieval. *SIAM Review*, *37*, 573–595.

Berry, M. W., & Fierro, R. (1996). Low-rank orthogonal decompositions for information retrieval applications. *Numerical Linear Algebra With Applications*, *3*, 301–327.

Berry, M. W., & Martin, D. (2005). Principle component analysis for information retrieval. In E. Kontoghiorghes (Series Ed.), *Statistics: A series of textbooks and monographs: Handbook of parallel computing and statistics* (pp. 399–413). Boca Raton, FL: Chapman & Hall/CRC.

Björck, Å. (1996). *Numerical methods for least squares problems*. Unpublished manuscript, Linköping University, Linköping, Sweden.

Buckley, C., Allan, J., & Salton, G. (1994). Automatic routing and ad-hoc retrieval using SMART: TREC 2. In D. Harman (Ed.), *Proceedings of the Second Text Retrieval Conference TREC-2* (National Institute of Standards and Technology Special Publication No. 500–215, pp. 45–56). Gaithersburg, MD: NIST.

Deerwester, S., Dumais, S., Furnas, G., Landauer, T., & Harshman, R. (1990). Indexing by latent semantic analysis. *Journal of the American Society for Information Sciences*, *41*, 391–407.

Dumais, S. (1991). Improving the retrieval of information from external sources. *Behavior Research Methods, Instruments, and Computers*, *23*, 229–236.

Furnas, G., Landauer, T., Gomez, L., & Dumais, S. (1987). The vocabulary problem in human–system communication. *Communications of the ACM*, *30*, 964–971.

Golub, G., & Van Loan, C. F. (1989). *Matrix computations* (2nd ed.). Baltimore: Johns Hopkins Unversity Press.

Jessup, E., & Martin, J. (2001). Taking a new look at the latent semantic analysis approach to information retrieval. In M. W. Berry (Ed.), *Computational information retrieval* (pp. 121–144). Philadelphia: SIAM.

Kolda, T. G., & O'Leary, D. P. (1998). A semi-discrete matrix decomposition for latent semantic indexing in information retrieval. *ACM Transactions on Information Systems*, *16*, 322–346.

Landauer, T., & Dumais, S. (1997). A solution to Plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge. *Psychological Review*, *104*, 211–240.

Larson, R., & Edwards, B. (1988). *Elementary linear algebra*. Lexington, MA: Heath.

Letsche, T., & Berry, M. W. (1997). Large-scale information retrieval with latent semantic indexing. *Information Sciences*, *100*, 105–137.

Lizza, M., & Sartoretto, F. (2001). A comparative analysis of LSI strategies. In M. W. Berry (Ed.), *Computational information retrieval* (pp. 171–181). Philadelphia: SIAM.

Parlett, B. (1980). *The symmetric eigenvalue problem*. Englewood Cliffs, NJ: Prentice-Hall.

Parlett, B., & Scott, D. (1979). The Lanczos algorithm with selective reorthogonalization. *Mathematics of Computation*, *33*, 217–238.

Salton, G., & Buckley, C. (1990). Improving retrieval performance by relevance feedback. *Journal of the American Society for Information Sciences*, *41*, 288–297.

Salton, G., & Buckley, C. (1991). Automatic text structuring and retrieval—experiments in automatic encyclopedia searching. *Proceedings of the 14th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval,* pp. 21–30.

Salton, G., Buckley, C., & Allan, J. (1992). Automatic structuring of text files. *Electronic Publishing*, *5*, 1–17.

Salton, G., & McGill, M. (1983). *Introduction to modern information retrieval*. New York: McGraw-Hill.

Witter, D., & Berry, M. W. (1998). Downdating the latent semantic indexing model for conceptual information retrieval. *The Computer Journal*, *41*, 589–601.

Zha, H., & Simon, H. (1999). On updating problems in latent semantic indexing. *SIAM Journal of Scientific Computing*, *21*, 782–791.