Seminar für Computerlinguistik
Institut für allgemeine und angewandte Sprachwissenschaft
Ruprecht-Karls-Universität Heidelberg

Magisterarbeit

Latent Semantic Indexing and Information Retrieval

A quest with BosSE

Johanna Geiß
Neugasse 12
69117 Heidelberg
E-Mail: j-o-g@web.de

18 January 2006

Supervisors:
PD Dr. Karin Haenelt
and
Prof. Dr. Peter Hellwig

to Anni Reemda Elisabeth

Abstract

This master thesis deals with the implementation of a search engine using Latent Semantic Indexing (LSI) called *BoSSE*.
Four different search types were implemented which allow a search for documents or terms similar to a given term, query or document. These search types are evaluated and the importance of term weighting, exclusion of non content words and the right selection of k for the reduction of dimension are discussed.
Furthermore, an introduction to Latent Semantic Indexing (LSI) and an explanation of the Singular Value Decomposition (SVD) is given.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Most common search engines have serious problems returning all the documents which are important to the query because they can not disambiguate ambiguous terms or find documents which only include synonyms of the query terms.
A promising approach to overcoming these shortcomings gives Latent Semantic Indexing (LSI).
This indexing scheme uses Singular Value Decomposition (SVD) to find the underlying latent semantic structure.

In this master thesis the implementation of a search engine called *BosSE*, which uses LSI is described and evaluated.
The goal of the implementation was to design a local search engine, which searches through a larger document collection consisting of articles written in everyday language with an unlimited domain (see chapter 4). Four different search types were implemented allowing users to search for documents and terms.
In chapter 5 the performance of *BosSE* is discussed. Together with an examination of the different search types, a report is given of the performance development when changing the values for the SVD and for indexing. This includes a discussion of the importance of exclusion of stop words and term weighting for LSI.
The remaining problems of the implementation and possible improvements are described in chapters 6 and 7.

But initially a short introduction to LSI is given in chapter 2 and an explanation and guide to SVD in chapter 3.

Chapter 2

Introduction to Latent Semantic Indexing

LSI [Deerwester et al., 1990] is a special vector space approach to conceptual Information Retrieval (IR). It attempts to overcome the common problems of search engines. These problems are:

1. Common search engines do not allow users to retrieve documents which are terms missing from the query although they might be relevant.

2. Synonymy: the same meaning can be expressed by two or more different terms.
   But the user will only enter one of these words in a query and most of the search engines do not expand the query by synonyms. Therefore the number of retrieved documents may be fewer than the optimal number. In other words the synonymy problem can decrease the recall of a search engine.
   Even if there is an automatic term expansion, it might do more bad than good, because it may add terms that have a different meaning from that intended by the user [Deerwester et al., 1990, p. 392], which leads to another typical problem: polysemy.

3. Polysemy: one term may have two or more different meanings.
   In a query the meaning can not be disambiguated so the search engine will retrieve documents that are not relevant to the meaning intended by the user. That is to say: the polysemy problem can decrease the precision of an information retrieval system.

How will LSI solve this problems?
LSI is designed to uncover the latent semantic structure of a document collection by

building a semantic space. It therefore uses the word usage patterns that exist in the document collection, namely, the word co-occurrences.
Within the semantic space created by LSI, terms and documents are represented. This is desired because "then a query can be placed at the centroid of its term points" [Deerwester et al., 1990, p. 394] and at the same time it can be used as a pseudo-document.

By reducing the term-document space to fewer dimensions, SVD reveals the underlying relationships between terms and documents in all possible combinations, while 'noise' (differences in word usage, terms that do not help distinguish documents, etc.) [Letsche, 1996, S.16] is reduced. In other words "the patterns of word usage across the entire document collection" [Letsche, 1996, p.16] are analyzed and similarities between terms and documents, between terms and between documents are shown within the reduced space.
The patterns of word usage are build up on word co-occurrences. Terms that often co–occur get high similarity values. For example *school* and *pupil* often occur together since these terms are semantically related.
There are several grades of relationships between terms. When two terms occur together in a document it is a first order co–occurrence. If *pupil* co–occurs with *school* and *school* co–occurs with *teacher*, *teacher* and *pupil* are of second order co–occurrence.
The correspondence between high order co-occurrence and the values produced by LSI were analysed in [Kontostathis and Pottenger, 2004]. In conclusion this analysis shows that term co-occurrence plays a crucial role in LSI. "We have explicitly shown use of higher orders of co-occurrence in the Singular Value Decomposition algorithm and, by inference, on the systems that rely on SVD, such as LSI.". "LSI emphasises important semantic distinctions while de-emphasising terms that co-occur frequently with many other terms (reduces noise)" [Kontostathis and Pottenger, 2004, p.21, p. 6].
Thus *school* and *pupil* will be placed close to each other as will *school* and *teacher.* That is why *pupil* and *teacher* will be close to each other as well, whereas terms that occur very often together with many other terms like *is* or *are* (supposed they are not deleted during stop-word elimination) will be de-emphasised and not put near *teacher*, *pupil* or *school.*

Now even documents that do not share the same terms might be placed near each other in the reduced term–document space when they are semantically related, that is to say their words have higher order co–occurrence.
If the user searches for *teacher* and *pupil* he will also get documents which include only the term *school*.

And synonymy? Imagine a long article about pupils learning experience in grammar school. The author would not always use the term *pupil* because it is not good writing style to use the same word throughout the whole text. Thus he might use carefully chosen synonyms such as *learner*, *schoolchild* or *student*. *Pupil* would then occur with these terms in first order and they would not co-occur only in one article but in several. Therefore they will end up close to each other in the semantic space of LSI.
But which solution does LSI give to the polysemy problem?
Let us have a look at an example of polysemy taken from [Penguin, 2001]:

> **mouse** 1. ANY OF NUMEROUS SPECIES OF SMALL RODENTS WITH A POINTED SNOUT, GREY TO BROWN FUR, AND A LONG SLENDER ALMOST HAIRLESS TAIL [...]
> 2. IN COMPUTING, A SMALL BOX, WITH A MOVABLE BALL UNDER IT, THAT IS CONNECTED TO A COMPUTER [...]
> 3. A TIMID PERSON, ESPECIALLY A SHY OR VERY QUIET GIRL OR WOMAN

Here *mouse* has got three distinct meanings. Let us assume, in the document collection used, there is one article about computers with the content words *[screen, mouse, trackball, printer]* and one about rodents with the index terms *[rat, squirrel, mousetrap, mouse, cheese, gnawing]*. In the LSI space *screen, printer, trackball* will build one cluster and *rat, squirrel, mousetrap, cheese, gnawing* another. *Mouse* will be placed between them.
If the user now enters a query containing *mouse and cheese* LSI will retrieve the article about rodents with a much higher similarity value than the one about computer. But if the user enters only the word *mouse* not even a LSI search engine can guess what the user means, thus it will retrieve both documents with equal similarity.

The ability of LSI depends on the quality of the corpus (a document collection). Good retrieval results can only be achieved when the term distribution is ‘natural’ and there are no ‘nonsense’ texts within the collection. That is to say simple listings of things should not be included in a document collection that is used for an IR application working with word use patterns.

2.1. The LSI algorithm

A search engine using LSI consists of two parts: preprocessing and search.

2.1.1. The Preprocessing

In the preprocessing phase the term–document space of a document collection is built. Usually this has to be done only once, or when significant changes to the corpus or the document collection have been made. It is independent of the actual search/es, which the user performs. Because of that, the time it takes is not crucial to the efficiency of the system.
The preprocessing steps are:

- Given a corpus (a document collection), the LSI search engine first of all indexes all terms in the corpus or in every document in the collection. Within that procedure the ‘stop-words’ (all terms considered common) might be eliminated. The result is a table of frequencies of occurrence of terms in each document (see section 4.2.2).
- Local and global weighting functions might be applied to estimate the relative importance of a term within the document and within the whole collection (see section 4.2.2).
- The values of the index are written to a Term Document Matrix (TDM) $A = [a_{i,j}]$ wherein each row represents an unique term and each column a document where $a_{i,j}$ is the (weighted) frequency of term i in document j. Usually this is a high dimensional sparse $m \times n$ matrix. That means most of the cells are filled with 0 because normally not every word appears in each document. The number of terms (m) is significantly greater than n (number of documents). In this work

the dimension of a TDM will be denoted as *terms* $\times$ *documents* ($t \times d$) for better comprehension.

- SVD is a mathematical method for the factorisation of any matrix into three matrices T, S and D. In LSI it is used to reduce the dimension of the semantic space. It is applied to A to reduce the rank of the matrix. The rank (r) of a matrix is the smaller of the number of linearly independent rows and the number of linearly independent columns in this matrix: $r = min(t, d)$. Typically in IR it is equal to the number of documents.
  Thus SVD gives a rank-k approximation A_k of A. k stands for the number of remaining dimensions in the semantic space (see chapter 3).

2.1.2. The Search

The efficiency of this part of the search engine with which the user interacts, is very important. The following steps are executed for each search.

1. First the user decides which type of search to perform. The types are:
   - search for terms similar to an entered term (term to terms search)
   - search for documents similar to an entered document (document to documents search)
   - search for documents similar to an entered term (term to documents search)
   - search for documents similar to an entered query (query to documents search)
2. The user enters the query i.e. a term, a document or a sequence of words.
3. According to the search type, the term and/or document vectors are scaled for the search (see section 3.2.3).
4. Depending on what was entered
   - the query is processed - the terms that were not found in the index are taken from the query and this is projected into the SVD semantic space or
   - the entered word is checked against in the index and the corresponding vector is scaled for the search or

- the vector for the chosen document is scaled to prepare it for searching.

5. The similarities between the query, the entered word or entered document and each vector in the search space are calculated.

6. The results of the similarity calculation are ranked and returned to the user.

Chapter 3

Singular Value Decomposition

SVD is a method from linear algebra for the factorization of any rectangular (and any square) matrix A with the dimensions of $t \times d$ into three matrices. It is related to the Eigenvalue decomposition of square matrices [Golub and Van Loan, 1996].

3.1. The purpose of SVD

SVD is used to reduce the rank of a matrix without losing important content and to eliminate all noise, that is to say all data that obscure the content. But what does that mean? The following two examples were chosen to illustrate the use of SVD.
In [Persson, 2005] a good example from the field of image compression can be found:

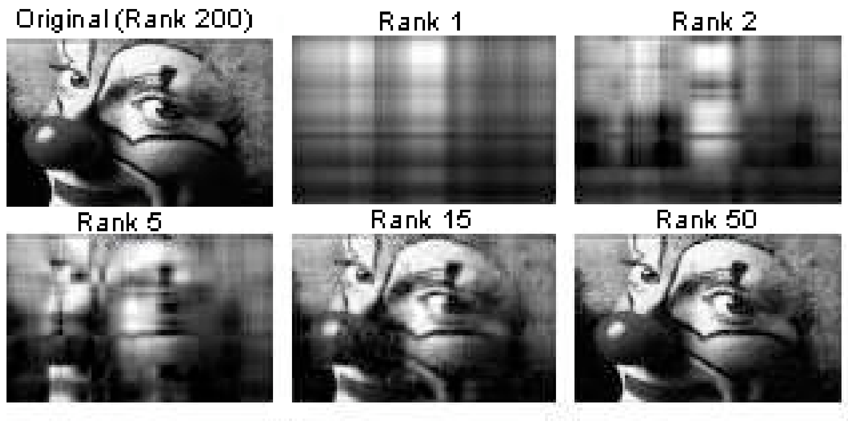


Figure 3.1.: SVD in image compression: View the $m \times n$ image as a matrix. The rank of this matrix was reduced from $r = 200$ to $k = 1, 2, 5, 15, 50$. Hardly any difference is visible between the rank $k = 50$ approximation of the image and the original, but the computer storage is reduced from $m \times n$ to $k(m + n)$[Persson, 2005].

If viewing this $m \times n$ sized image in figure 3.1 as a matrix, SVD can be applied. The rank of the original image was reduced. The rank $k = 1$ approximation shows only vertical and horizontal lines. "This checkerboard-like structure is typical of low rank component approximations to images" [Moler, 2004, S.26]. The rank k=50 approximation of the image requires less computer storage ($k(m + n)$ instead of $m \times n$) while still having a good resolution. Hardly any visual difference to the original rank k=200 image can be recognized.

SVD helps to find a better (and smaller) representation of a given space. Therefore it first analyses the space and then returns a better basis. What does that mean?
Imagine an owner of an aquarium, who wants to take a photo of his fishes. In this photo all fishes are to be represented and nicely presented, that means not bunched together in one big swarm. From the owner's current point of view a few fishes are in front of each other or hidden by plants. So he takes his camera and searches for the best angle to see as many fishes as possible.
That is exactly the same as what SVD does when it factorizes the matrix. It looks for the best basis coordinate system within the space in which the document and term vectors can be shown.
The owner of the fish tank takes a photo. Now he has got a two dimensional picture of his three dimensional aquarium. By taking the picture he reduced the space, still has a good representation of his fishes and most of them are clearly visible.
Exactly the same happens when the semantic space is reduced by truncating the three matrices given by SVD. When multiplying these three truncated matrices we get a rank reduced matrix which is an approximation of the original matrix.
The terms and documents are projected into a semantic space with smaller dimension. The latent semantic structure, that is to say the relationships between documents and terms in the document collection, is revealed.

3.2. SVD in Information Retrieval

In a common Vector-Space Model only terms *or* documents are represented in the Vector-Space.
What does that mean? What is a term space or a document space?

This will be explained by means of a very simple example Term Document Matrix (TDM) Y, which will be used throughout this section.
For this example I chose a square 3×3 matrix, because it can be graphically represented and the rank can be reduced to 2. The TDM Y is given in 3.1.

Table 3.1.: Example TDM Y of size 3×3

	d1	d2	d3
t1	2	1	1
t2	1	0	1
t3	0	2	1

3.2.1. The term space

In a term space each document forms a vector and each term represents a dimension. In other words the three documents from this example are displayed as three vectors in a chart with three orthogonal axes, one for each term in figure 3.2.

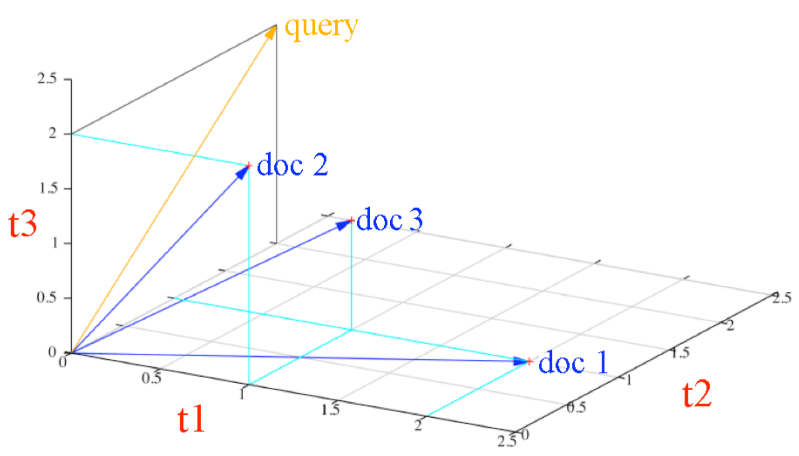


Figure 3.2.: Term space of the TDM Y: the documents d1, d2, d3 from the example TDM (see 3.1) are shown as blue vectors in the term space of the three terms t1, t2, t3. The query $\langle 0, 2, 2 \rangle$ is depicted by the yellow vector.

The direction and the longitude of a vector are determined by the number of occurrences of each of the three terms in the document.
This example uses three terms, so the space has got three dimensions. If a matrix of a real document collection were used, the space would have thousands of dimensions, one

for each content word and as many vectors as documents in the document collections would be displayed in it.
To reduce the dimension of such a space one would have to reduce the number of content words, but in Information Retrieval one does not wish to miss possible important terms just to make the search faster. Furthermore, this kind of reduction would not cause a better representation of the documents.
In a term space only similarities between documents can be calculated. A query is represented as a pseudo-document. It is noted how often a term that represents a dimension occurs in the query. Here the query ”t2, t2, t3, t3, t4” was used with the vector $\langle 0, 2, 2 \rangle$ (the yellow vector in figure 3.2).
The only way to calculate similarities between a term and a document is by using it as a query. LSI, therefore provides an extra calculation scheme (see section 3.2.3).
The axes which represent the terms are orthogonal to each other, so there is no information concerning the similarity between content words.

Cosine similarity

To calculate the similarities between the documents and the query the cosine similarity measurement is used. With this measurement the angle between two vectors can be estimated, an angle being a good measurement of similarity [Letsche, 1996]. The smaller the angle the nearer the vectors are to each other, the more similar the vectors are and in vector space IR models the more similar the documents are.
The calculation of an angle between two vectors $\vec{a}$ and $\vec{b}$ is taken from the inner product (also called dot product or scalar product) described in equation 3.1.

$$\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| \cdot cos(\alpha) \tag{3.1}$$

This states that the product of two vectors is given by the product of their norms (in geometric terms, the length of the vector) multiplied by the cosine of the angle α between them. For the area of application here it means that we can easily calculate the angle respectively the cosine value for the angle between them by converting equation 3.1:.

$$cos(\alpha) = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|} \tag{3.2}$$

The values of $cos(\alpha)$ are in the range of -1 which corresponds to $180°$ or 0% similarity to 1 which stands for $0°$ or $360°$ which is equal to 100% similarity. So a $cos(\alpha)$ of 0

corresponds to a similarity of 50%, which means that the vectors are rectangular to each other.
To compute the percentage p_{sim} of similarity the following equation 3.3 is used.

$$p_{sim} = 50 \cdot cos(\alpha) + 50 \tag{3.3}$$

The similarities between the documents from the example TDM Y are shown in table 3.2. The query and the document $d3$ have got the highest similarity value, in this example 0.82, that corresponds to an angle of 35° between them. $\vec{d3}$ is the nearest vector to the *query*. They are similar to a degree of 91%, whereas the document vector $\vec{d1}$ is only 66% similar to the query vectors, which is shown by the smaller cosine value of 0.32 corresponding to an angle of 71°. The lowest value possible would be of course -1 which is equal to an angle of 180°, which means that the two vectors would lie in opponent directions.

Table 3.2.: Document similarities of TDM Y in the vector space: the similarities were calculated with the cosine similarity measurement (see equation 3.2).

	d1	d2	d3	query
d1	1 (100%)	0.4 (70%)	0.77 (88.5%)	0.32 (66%)
d2	0.4 (70%)	1 (100%)	0.77 (88.5%)	0.63 (81.5%)
d3	0.77 (88.5%)	0.77 (88.5%)	1 (100%)	0.82 (91%)
query	0.32 (66%)	0.63 (81.5%)	0.82 (91%)	1 (100%)

3.2.2. The document space

In a document space the terms are represented as vectors (the rows of the TDM contain the components of the vectors) and the documents are the axes around which the vectors are grouped. The three terms from the example are displayed as three vectors in figure 3.3 with three orthogonal axes, one for each document.
The size and the direction of the vectors demonstrate how often a term occurs in each document.
Corresponding to what was explained before, if a matrix of a real corpus were used, the space would have as many dimensions as documents in the corpus. To reduce the dimensions of this space the number of documents would have to be declined, which would

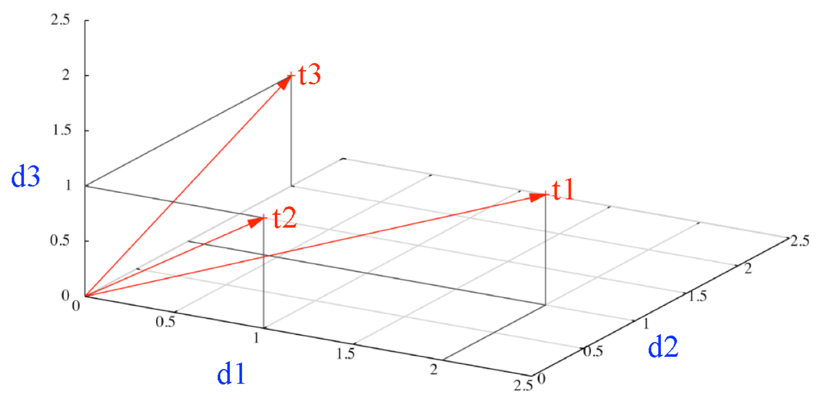


Figure 3.3.: Document Space of the TDM Y: The terms t1, t2 and t3 from the example TDM (see 3.1) are shown as red vectors in the document space of the three documents d1, d2, d3.

decrease the usability of an Information Retrieval system.
In this space only the similarities between terms and not between terms and documents can be discussed. The documents are like the terms in the term space represented by orthogonal axes.
The similarities between the terms are shown in table 3.3. In this example the terms t1 and t2 have got the highest similarity, 93.5%, which was calculated from the cosine value of 0.87%, which corresponds to an angle of 29.5°. The most dissimilar terms in this example are t2 and t3 whose vectors $\vec{t2}$ and $\vec{t3}$ form an angle of 71.3°, which is equal to a cosine of 0.32.

Table 3.3.: Term similarities of TDM Y in the vector space model: the similarities were calculated with the cosine similarity measurement (equation 3.2).

	t1	t2	t3
t1	1 (100%)	0.87 (93.5%)	0.55 (77.5%)
t2	0.87 (93.5%)	1 (100%)	0.32 (66%)
t3	0.55 (77.5%)	0.32 (66%)	1 (100%)

3.2.3. The SVD semantic space

The goal of LSI is to represent the terms together with the documents in one space. That makes it possible to calculate similarities between documents, between terms, between terms and documents and to place queries at the centroid of their terms in order to compare them to documents.

To calculate the new basis for that space and the vectors, SVD is needed.
Moreover, we aim to reduce the dimensions or the rank of the TDM to unveil the latent semantic structure, this is also done by SVD.

The Singular Value Decomposition splits the $t \times d$ matrix A where $t \geq d$ and $r = d$ into three matrices:

$$A = U\Sigma V^T \tag{3.4}$$

U is an rectangular matrix of the size $t \times r$ and contains in its rows the term vectors scaled to the new basis. It will therefore be called T. The other rectangular matrix is V^T, which is the transposed matrix V. To transpose a matrix means to interchange its rows with its corresponding columns. Those matrices are marked by ^{T}. V^T has the size $r \times d$ and contains the new vectors for the documents, which is why it will be called D^T. The square matrix Σ is a diagonal matrix, which means that only the cells in the main diagonal (from top left corner to bottom right corner) are non-zero. It contains the singular values σ_1, σ_2, ...,σ_n where $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_n \geq 0$ and will be named S.
All matrices can be factorized this way [Strang, 2003], shown in equation 3.4 and depicted in figure 3.4.

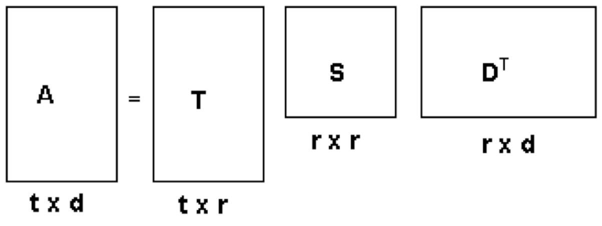


Figure 3.4.: Schema of SVD for a matrix A: The term document matrix A is factorized into three matrices: T of size $t \times r$, the diagonal matrix S of size $r \times r$ and the $r \times d$ matrix D^T.

For the TDM Y from above the factorization is:

$$\mathbf{Y} = \begin{pmatrix} -0.75 & 0.41 & 0.52 \\ -0.36 & 0.41 & -0.84 \\ -0.56 & -0.82 & -0.16 \end{pmatrix} \begin{pmatrix} 3.11 & 0 & 0 \\ 0 & 1.73 & 0 \\ 0 & 0 & 0.56 \end{pmatrix} \begin{pmatrix} -0.6 & -0.6 & 0.54 \\ 0.71 & -0.71 & 0 \\ 0.38 & -0.38 & -0.84 \end{pmatrix}$$

The multiplication of these matrices returns the original matrix Y.
Now the terms and documents can be shown in the same space. Let us call this new space the SVD space because new axes were found by SVD along which the terms and documents can be grouped.
By now the different searches, I mentioned about could be performed by scaling the new term and document vectors according to what is being looked for, but the latent semantic structure has not yet been revealed. Therefore the dimension of the new SVD space has to be reduced. This is done by truncating the three matrices as shown in figure 3.5. Depending on k (number of remaining dimensions) from T the last $r - k$ columns will

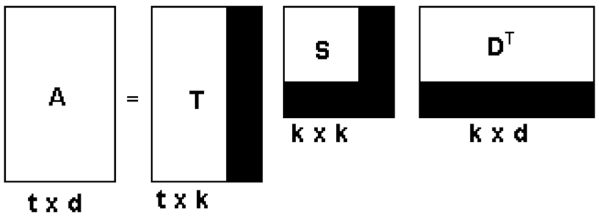


Figure 3.5.: Truncation of the SVD space

be deleted. These contained the components of the term vectors for the $k+1$, $k+2$,..., $k+n$, r dimensions resulting in T_k. The $r-k$ last rows of D^T, which contained the components of the document vectors for dimension $k+1$ to dimension r which leads to D_k^T, are truncated . To get S_k from S the last $r-k$ values from the diagonal are taken.
For the example above the truncated SVD is:

$$\mathbf{Y_{k=2}} = \begin{pmatrix} -0.75 & 0.41 \\ -0.36 & 0.41 \\ -0.56 & -0.82 \end{pmatrix} \begin{pmatrix} 3.11 & 0 \\ 0 & 1.73 \end{pmatrix} \begin{pmatrix} -0.6 & -0.6 & -0.54 \\ 0.71 & -0.71 & 0 \end{pmatrix} =$$

	d1	d2	d3
t1	1.90	0.90	1.26
t2	1.18	0.17	0.6
t3	0.04	2.05	0.94

The rank of the matrix Y has been reduced from $r = 3$ to $r = 2$, that means $Y_{k=2}$ is the best rank 2 approximation of Y.

In $Y_{k=2}$ there are no zeros remaining and the values have changed slightly. For example $y_{k,(2,2)} = 0.17$ whereas $y_{(2,2)} = 0$. This is because the latent semantic structure was revealed and shows that t2 and d2 are somehow related, even if t2 does not occur in d2. "It is important for the method that the derived k dimensional factor space *not* reconstruct the original term space perfectly, because we believe the original term space to be unreliable. Rather we want a derived structure that expresses what is reliable and important in the underlying use of terms as document referents." [Deerwester et al., 1990, p. 395].
Generically the rank k approximation of A is the product of the three truncated matrices T_k, S_k and D_k^T which is shown in equation 3.5.

$$A_k = T_k S_k D_k^T \tag{3.5}$$

With these truncated matrices the new vectors have to be scaled according to the search type.

Term to term similarity

To compare terms with each other or, in other words, to look for similar words to a given one, the dot product (see equation 3.1) of two row vectors of A_k is calculated. The multiplication $A_k A_k^T$ as in equation 3.6 results in a matrix containing these dot products.

$$A_k A_k^T = T_k S_k D_k^T (T_k S_k D_k^T)^T = T_k S_k D_k^T D_k S_k^T T_k^T = T_k S_k S_k T_k^T = T_k S_k^2 T_k^T = T_k S_k (T_k S_k)^T \tag{3.6}$$

D_k is orthonormal which is is why $D_k^T D_k = E$ where E is the identity matrix (a diagonal matrix with the values in the diagonal equal to 1). S_k is a diagonal matrix thus $S_k = S_k^T . a_{i,j}$ of this matrix can be obtained by taking the dot product of the ith and jth row of TS. But the $cos(\alpha)$ of these vectors can also be calculated for the equation since the $cos(\alpha)$ descends from the dot product see equation 3.2.
Thus all that is to be done is to scale the term vectors in T_k by the values from S_k as seen in equation 3.7.

$$TT = T_k S_k \tag{3.7}$$

The result is the $t \times r$ matrix TT which contains in its rows the new scaled term vectors now ready for comparing against each other and to show to what extent two terms have a similar pattern of occurrence across the corpus. For Y this calculation is:

$$\mathbf{TT} = \begin{pmatrix} -0.75 & 0.41 \\ -0.36 & 0.41 \\ -0.56 & -0.82 \end{pmatrix} \begin{pmatrix} 3.11 & 0 \\ 0 & 1.73 \end{pmatrix} = \begin{pmatrix} -2.33 & 0.71 \\ -1.12 & 0.71 \\ -1.74 & -1.42 \end{pmatrix}$$

Now the similarities between the new term vectors $t1$ $\langle -2.33, 0.71\rangle$, $t2$ $\langle -1.12, 0.71\rangle$ and $t3$ $\langle -1.7, -1.42\rangle$ can be calculated and are listed in table 3.4. These new term vectors are shown by red arrows in figure 3.6.

From table 3.3 where the similarities of the terms in the original document space are

Table 3.4.: Term similarities for the TDM Y in SVD space calculated using the cosine similarity measurement from equation 3.2.

	t1	t2	t3
t1	1 (100%)	0.96 (98%)	0.56 (78%)
t2	0.96 (98%)	1 (100%)	0.32 (66%)
t3	0.56 (78%)	0.32 (66%)	1 (100%)

shown, to table 3.4 the values have changed. $t1$ and $t2$ get a similarity of 98% in the SVD space of in contrast to 93.5% in the document space. Here the similarity increased but it can also decrease during SVD and the revealing of the latent semantic structure.

As described above in equation 3.6 there is the possibility of calculating the dot products for discovering the similarities between terms. But by using this method the values in TT are not normalized, the highest value is not known and therefore it's hard to assess the value. To overcome this shortcoming one would have to normalize the rows first before calculating the matrix product. It is also not practicable for actual application, because for term to term similarity the matrix would be of size $t \times t$ which is acceptable when there are three terms in the collection but not for thousands of words. This matrix would have to be stored and to be read in, which would take more time then calculating the cosine between each row from a matrix of size $t \times r$.

Document to document similarity

To be able to calculate the similarities among documents, dot products (see equation 3.1) between two columns of A_k have to be calculated. The product of $A_k^T A_k$ from equation

3.8 contains these similarities.

$$A_k^T A_k = (T_k S_k D_k^T)^T T_k S_k D_k^T = D^T S_k^T T_k^T T_k S_k D_k T_k^T = D_k S_k^2 D_k^T = D_k S_k (D_k S_k)^T \quad (3.8)$$

S_k is a diagonal matrix so $S_k = S_k^T$. T_k is orthonormal so $T_k^T T_k = E$ is valid. $a_{i,j}$ of this matrix can be obtained by taking the dot product of the ith and jth row of DS, but the $cos(\alpha)$ of these vectors can be also calculated, as the equation for the $cos(\alpha)$ descends from the dot product see equation 3.2.
In other words, the document vectors in D_k have to be scaled by S_k as in equation 3.9.

$$DD = D_k S_k \quad (3.9)$$

Note that here D_k and not D_k^T is used.
This calculation is analogous to the calculation for the term to term similarity. The calculation for Y is:

$$\mathbf{DD} = \begin{pmatrix} -0.6 & 0.71 \\ -0.6 & -0.71 \\ -0.54 & 0 \end{pmatrix} \begin{pmatrix} 3.11 & 0 \\ 0 & 1.73 \end{pmatrix} = \begin{pmatrix} -1.87 & 1.23 \\ -1.87 & -1.23 \\ -1.68 & 0 \end{pmatrix}$$

In the rows of DD the new document vectors are written. Now the similarities between the documents can be calculated and are shown in table 3.5. The new document vectors are shown in figure 3.6 by dark grey arrows.
Here also an increase of similarity can be noted in contrast to the term space (for the term space similarities in the common vector space model see table 3.2). The document vector $\vec{d1}$ and $\vec{d3}$ now forms a smaller angle of 32.86° than in the term space where the angle was 39.65°.

Table 3.5.: Document similarities of TDM, Y in SVD space calculated using the cosine similarity measurement from equation 3.2

	d1	d2	d3
d1	1 (100%)	0.4 (70%)	0.84 (92%)
d2	0.4 (70%)	1 (100%)	0.84 (92%)
d3	0.84 (92%)	0.84 (92%)	1 (100%)

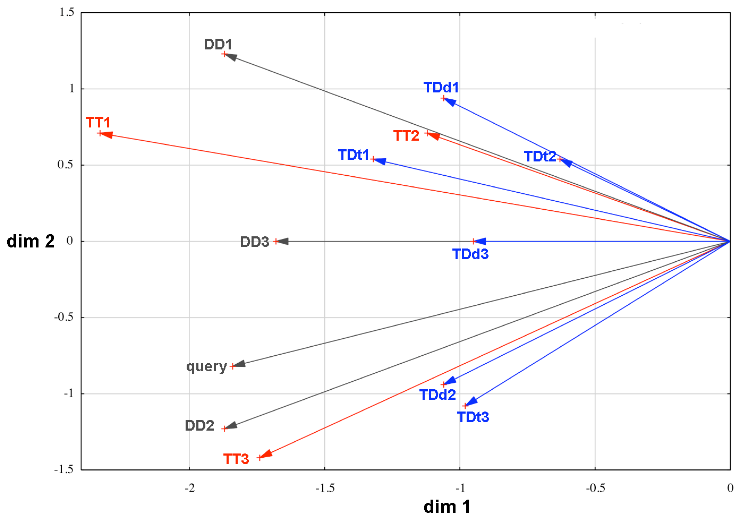


Figure 3.6.: Different scaled term and document vectors in the reduced SVD space. The red vectors depict the term vectors scaled for term to term search. The document vectors scaled for the document to document search as well for the query to document search are shown by dark grey arrows. The query was folded into the SVD space for comparison to the document vectors and is shown by the dark grey vector *query*. The blue vectors show the term and document vectors scaled for term to document search.

Term to document similarity

To get values for the similarity between terms and documents the rank k approximation of A has to be calculated. That is shown in equation 3.5. This equation can be changed to equation 3.10.

$$A_k = T_k S_k D_k^T = T_k S_k^{\frac{1}{2}} S_k^{\frac{1}{2}} D_k^T = T_k S_k^{\frac{1}{2}} (D_k S_k^{\frac{1}{2}})^T \tag{3.10}$$

The matrix S_k is split into $S_k^{\frac{1}{2}} S_k^{\frac{1}{2}}$ which is equal to $\sqrt{S_k}\sqrt{S_k}$ which results in S_k. To calculate the square root of a diagonal matrix, the square roots of the values have to be extracted.

The ith and jth cell of this matrix A_k can be obtained by calculating the dot product of the ith row of the matrix TD_T resulting from equation (3.11) and the jth row from the matrix TD_D specified in (3.12).

$$TD_T = T_k S_k^{\frac{1}{2}} \tag{3.11}$$

$$TD_D = D_k S_k^{\frac{1}{2}} \tag{3.12}$$

I have calculated TD_T which contains the scaled term vectors in its rows and TD_D wherein the scaled document vectors are written (also in rows) for Y:

$$\mathbf{TD_T} = \begin{pmatrix} -0.75 & 0.41 \\ -0.36 & 0.41 \\ -0.56 & -0.82 \end{pmatrix} \begin{pmatrix} 1.76 & 0 \\ 0 & 1.32 \end{pmatrix} = \begin{pmatrix} -1.32 & 0.54 \\ -0.63 & 0.54 \\ -0.98 & -1.08 \end{pmatrix}$$

$$\mathbf{TD_D} = \begin{pmatrix} -0.6 & 0.71 \\ -0.6 & -0.71 \\ -0.54 & 0 \end{pmatrix} \begin{pmatrix} 1.76 & 0 \\ 0 & 1.32 \end{pmatrix} = \begin{pmatrix} -1.06 & 0.94 \\ -1.06 & -0.94 \\ -0.95 & 0 \end{pmatrix}$$

Now the similarities between terms and documents can be calculated using the cosine similarity measurement. The results for the example TDM Y are shown in table 3.6. These similarities could not be easily calculated in the common Vector Space Model unless queries consisting of only one term were used. Here it can be seen that $t3$ has got

Table 3.6.: The term to document similarities of TDM Y in SVD space calculated using the cosine similarity measurement from equation 3.2.

	d1	d2	d3
t1	0.94 (97%)	0.44 (72%)	0.93 (96.5%)
t2	0.999 (99.95%)	0.13 (56.5%)	0.75 (87.5%)
t3	0.01 (50.5%)	0.994 (99.7%)	0.67 (83.5%)

the highest similarity to $d2$, 99.7% which is logical because this term occurs in $d2$ twice whereas it can be found only once in $d2$, and $d1$ does not contain the term at all. These similarities can also be recognized in figure 3.6 where these term and document vectors scaled for the term to document search are depicted by blue arrows and are named TDt1, TDt2 and TDt3 for the terms and TDd1, TDd2 and TDd3 for the documents.

The query to document similarity

To search for a document that is similar to an entered set of words, this query has first to be folded into the SVD space to become a pseudo–document $\vec{d_{qu}}$. Then this pseudo–document can be compared to the other documents as in a document to document similarity calculation.

This folding in is done by multiplying the query vector with the corresponding rows of T_k

as shown in equation 3.13. Usually this product has to be scaled by S_k^{-1}, but because it will be compared to the document vectors and these will be scaled by S_k (equation 3.9), it must also be multiplied by S_k. But $S_k^{-1}S_k = E$ so this step can be omitted.

$$\vec{d_q} = \vec{qu}^T T_{qu} S_k^{-1} S_k = \vec{qu}^T T_{qu} \tag{3.13}$$

$$\vec{d_q} = \begin{pmatrix} 0 & 2 & 2 \end{pmatrix} \begin{pmatrix} -0.75 & 0.41 \\ -0.36 & 0.41 \\ -0.56 & -0.82 \end{pmatrix} = \begin{pmatrix} -1.84 & -0.81 \end{pmatrix}$$

Now this vector $\vec{d_{qu}}$ can be compared with the vectors from DD from equation 3.9. The results are shown in table 3.7.

The document $d2$ has got the highest similarity, 99%, which means that there is only an angle of approximately 8° between the two vectors. This can also be verified by considering figure 3.6 and the dark grey vectors.

Table 3.7.: Query to document similarity for TDM Y and query $\vec{qu}$ in SVD space

	query
d1	0.54 (77%)
d2	0.99 (99.5%)
d3	0.91 (95.5%)

3.3. The mathematics behind SVD

In this section the mathematics behind SVD and the calculation of SVD are illustrated. The information where taken from [Lang and Pucker, 1998] and [Strang, 2003].

The purpose of SVD is to diagonalize any $t \times d$ matrix A. The diagonalization corresponds to a transit to a new coordinate system [Lang and Pucker, 1998, p. 419]. This is what we require to obtain the latent semantic structure of the corpus, that is to say the concepts which are hidden in the TDM.

Orthonormal bases have to be found, to which the matrix A can be diagonalized. A is an arbitrary matrix, so it need not to be symmetric. Thus two bases are required, one for the columns space, and one for the row space both subspaces of the matrix A [Strang, 2003,

p.368]. The column space lies in $\mathbb{R}^t$ (a column has got t cells), the row space in $\mathbb{R}^d$ (a row consists of d cells) and they are both r dimensional.
An orthonormal basis consists of orthogonal unit vectors.
These basis vectors for the two bases are written in the columns of U (basis for the column space) and V (basis for the row space) in equation (3.4) or T and D in equation 3.14 and Σ or S is the diagonalization of A.

$$A = U\Sigma V^T \tag{3.14}$$

T consists of r column vectors with t cells and D of r column vectors with d components.

We do not simply wish to obtain an orthonormal basis vectors, we are searching for stable vectors which are characteristic of A, which in our case represent the underlying concepts of the document collection. So orthogonal unit vectors are needed, which do not change their directions when multiplying them by A as other vectors do. In other words we are looking for eigenvectors $\vec{x}$ which lie in the same direction as Ax.
When multiplying an eigenvector $\vec{x}$ with A the result is $\lambda\vec{x}$ (equation 3.15). Where λ is an eigenvalue. Its value determines the scaling of the eigenvector, whether it is racked, clinched or turned.

$$A\vec{x} = \lambda\vec{x} \tag{3.15}$$

But how can this equation be solved? How do we get the eigenvalues/eigenvectors? By subtracting $\lambda\vec{x}$ equation 3.15 becomes equation 3.16.

$$(A - \lambda E)\vec{x} = 0 \tag{3.16}$$

E is the unity matrix, a diagonal matrix consisting of 1.
If this equation has a nontrivial solution, $A - \lambda I$ is not invertible, which means that no inverted $A - \lambda I$ (it will be called B^{-1}) exists which makes $B^{-1}B = BB^{-1} = E$. Thus the determinate has to be equal to 0 (equation 3.17). The inclusion of this derivation here would go beyond the scope of this chapter, for more information see [Strang, 2003, chapter 6].

$$det(A - \lambda E) = 0 \tag{3.17}$$

With this equation the eigenvalues λ can be derived.
But a determinant can only be calculated from square matrices. In IR most TDMs are

rectangular. To make a matrix A square it can be multiplied by A^T.
But $AA^T \neq A^TA$. So there are two ways of calculating a square matrix. As described above, two bases are needed, therefore two sets of eigenvectors are needed, so we need two square matrices. The eigenvectors for the column space which are written to T are calculated with $AA^T = A_T$ and the eigenvectors for the row space which are written to D with $A^TA = A_D$.
How can this be achieved?
For A_T and A_D the following steps have to be performed:

1. Calculate the determinant of $(A_{T/D} - \lambda E)$ from equation 3.17. This will result in a polynomial of r^{th} order.

2. Find the values for λ where the polynomial equals 0 (the roots or the zeros of the polynomial This will lead to the eigenvector $\vec{x_i}$ for the eigenvalue λ_i

3. The eigenvectors are not yet unit vectors. Its lengths has to be normalized. The length of a vector is its absolute value: $|\vec{x}| = \sqrt{\sum\limits_{i=1}^{n} x_i^2}$

4. Calculate the singular values by extracting the root of the eigenvalues, due to this only the positiv eigenvalues are taken into account.

5. Write the singular values in descending order into S.

6. Write the eigenvectors in corresponding order to S into T and D respectively.

Now the Singular Value Decomposition is calculated and the derived concepts or topics of the document collection are depicted in D, and the word distribution patterns in T. S is only used for scaling these matrices correspondingly to the searches.

Chapter 4

BosSE

4.1. What is BosSE?

BosSE is a GUI based local search engine using LSI. It is implemented in Python[1] and the user interface was designed using wxPython[2].
BosSE is optimized for the German release of Wikipedia [Wikipedia, 2005], but it can be used for every version of Wikipedia. Furthermore *BosSE* is easily adaptable to any other document collection.

4.1.1. The document collection

Wikipedia is a free internet encyclopedia to which everyone can contribute by changing articles or by adding new ones.
It is based on a ‘Wiki’ which is something like a content management system[3]. A Wiki is a website that allows users to edit and add content. The user does not need to know HTML. Each change will be recorded, thus an earlier version can be reconstructed easily. Wikipedia is multilingual, up to now there are 200 language editions available. “Ten editions have more than 50,000 articles each: English, German, French, Japanese, Polish, Italian, Swedish, Dutch, Portuguese, and Spanish.”[4]. In total there are more than 2,500,000 articles available. The project was started in 2001 and is operated since 2003 by the Wikimedia Foundation – a non-profit organisation.

[1]version 2.4.1 `http://www.python.org` last visited 2.11.2005
[2]version 2.6-mac-Unicode `http://www.wxpython.org/` last visited 2.11.2005
[3]for further information see `http://en.wikipedia.org/wiki/Wiki`
[4]`http://en.wikipedia.org/wiki/Wikipedia` last visited 10.12.05

The slogan of Wikipedia is "The Free Encyclopedia that anyone can edit". This might be a good thing and a bad thing. The bad thing is, that it is open to vandalism, inaccuracy and opinion although contributors are asked to keep a "neutral point of view". It has been criticized, that is lacks an authority and reliability. But every article is reviewed by thousands of users who can and will update the articles. The premise of Wikipedia is, that due to continuous improvement it will get more and more perfect[5].
But what is important for this IR project is: it is free (published under the GNU Free Documentation License (GFDL)[6]) and (living) natural language is used for it was actually written by people like you and me of today. This includes natural word patterns and actual use of synonyms.

For *BosSE* XML dumps of Wikipedia are used.
Different XML dumps of the encyclopedia are downloadable from `http://download.wikimedia.org/wikipedia/`. For each language there are several versions:

- `pages_full.xml` which includes all pages and all revisions
- `pages_current.xml` with only the current revision of all pages
- `pages_articles.xml` or `pages_public.xml` includes only the current versions of the articles, without discussions, talks or user pages.

For *BosSE* the files `http://download.wikimedia.org/wikipedia/de/20050903_pages_public.xml` and `http://download.wikimedia.org/wikipedia/nds/20051029_pages_articles.xml` were used. The latter is the release of Wikipedia in Plattdütsch (Low Saxon and East Low German) and it was used just for testing during implementation, because with 5.2 MB it is significantly smaller than the German version which is an XML file of 1.35 GB and includes approximately 280,000 articles.
It is also possible to export particular pages or a set of pages from Wikipedia to XML on `http://de.wikipedia.org/wiki/Spezial:Export`. For evaluating *BosSE* a corpus of 169 German articles was build (see chapter 5).
All these Wikipedia XML files, no matter how they were exported, have the same structure.

[5]for more on Wikipedia see `http://en.wikipedia.org/wiki/Wikipedia` last visited 10.12.2005
[6]`http://www.fsf.org/licensing/licenses/fdl.html`

The main element of each file is the element `<mediawiki>`. Within this tag the attribute for the XML name-space[7] (which is needed later on for the XML parser in *BosSE* to identify the tags), the attribute for the XML schema location[8], (which must be downloaded and saved in the directory of the corpus) and the attribute for the language of the XML dump, which helps to identify the several article collections and files needed for *BosSE*, is given.
After this opening some tags for site information are included and then a set of `<page>` elements follows. In these the actual articles are defined. For *BosSE* only the elements `<title>` and `<revision><text>` within the element `<page>` are important, because the title and the actual text is given there. For every page more additional information is available, but this is not needed for the purposes of the search engine.
The advantages of these XML dumps towards other document collections are:

- There is no encoding problem. The XML files are written in UTF8.
- There is no updating problem for the SVD space. No 'folding-in' is necessary, because every few months new XML dumps are published. This is not too frequently to spent the effort of calculating the SVD space from scratch.[9]
- The XML dumps are easy to parse. The structure of the XML dump is clear and always the same throughout all versions, because its verified against an XML-schema. This makes it easy to use different language versions with the search engine, for only the list of stop words and some minor changes to the list of text substitution have to adapted.
- Wikipedia has a great spectrum of articles about many topics. but they are all of the same type: encyclopedia entries. That means there is no nonsense text. Each article is about a special topic, written in a factual manner.
- The use of the Wikipedia content is free of charge (see footnote 6).

But there are also some disadvantages which should not be kept secret:

[7]usually `xmlns="http://www.mediawiki.org/xml/export-0.3/"`
[8]`http://www.mediawiki.org/xml/export-0.3.xsd`
[9]For a corpus build manually using the Wikipedia export utility a 'folding–in' option would be helpful and desirable, but this way of building a corpus was not intended at the time of planing *BosSE*.

- The content is not verified by experts therefore it might include some false information. But because Wikipedia articles are reviewed daily by many users, who (I hope) will correct any mistake and I assume that no one enters incorrect facts in bad faith, it is considered a minor drawback.
- There might be spelling mistakes and bad HTML syntax (although HTML is not necessary for a Wiki it is often used). This makes it harder to find good index terms for the search engine[10].
- The XML files have to be cleaned from pages containing listings of names, cities etc. They do not reflect normal word usage and do not contain text in the linguistically sense.
- The articles are written in a special syntax for Wiki e.g. headings are specified by surrounding them by '='. These annotations have to be changed to HTML code, so that the user can read articles easily in the standard browser when links to the articles were returned by the search engine.
- Articles are put into categories. This categorization is done by hand and is therefore not reliable and most of the categories are not very meaningful e.g. *men*, *1809 births*, *People from Baltimore* are categorizations for Edgar Allen Poe. Furthermore this additional information can obscure the actual patterns of word usage, that is why it has to be removed otherwise e.g. all men will be linked by second order co-occurrence.

Despite of these few drawbacks, which mean some more work and attention when indexing the collection and preparing it for viewing, Wikipedia is considered as good corpus for it is not domain specific and not explicitly build for research in natural language processing or IR like other (annotated) corpora. With this collection it can be examined, if LSI is suitable for everyday language written by non professionals. [Deerwester et al., 1990] uses the CISI corpus, a collection of 1460 information science abstracts and the MED document collection of 1033 medical abstracts.

[10]This really is a bigger problem than thought at the beginning of this master thesis and still is not solved. One can not think of all the absurd mistakes users can do.

4.2. The Design and Implementation of BosSE

In this section the design, structure and important algorithms of *BosSE* are described. The search engine is written in Python (version 2.4.1). The implementation was mainly done on a Macintosh PowerBook with a 550 MHz PowerPC G4 processor, 768 MB SDRAM and a 20 GB harddrive (splitted unequally on 2 partitions, that's why only 1.24 GB were available for swapping data see chapter 6).
Some modules were added to the default installation of Python. They are:

- wxPython 2.6-mac-unicode `http://www.wxpython.org/`
- numarray 1.3.3 `http://www.stsci.edu/resources/software_hardware/numarray/`
- pyxml 0.8.4 `http://pyxml.sourceforge.net/`
- cElementTree 1.0.2 `http://www.effbot.org/zone/celementtree.htm`.

The modules shown in figure 4.1 are described in the following sections.

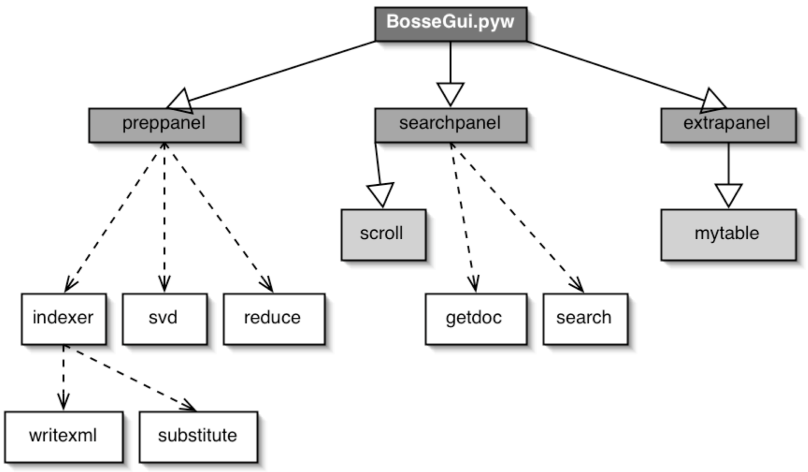
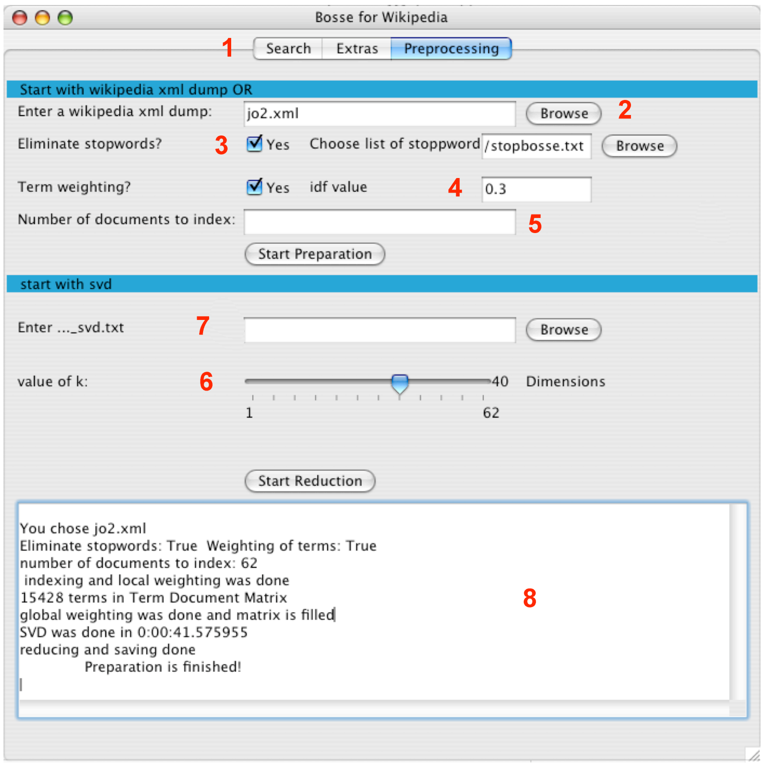


Figure 4.1.: Overview of self defined modules for *BosSE*
The first level displayed in dark grey is the starting point of the application. The boxes of the next layer (highlighted with grey) depict the modules which represent the three panels of the GUI. The boxes (light gray) of the third level are GUI modules for specific panels/windows. The boxes connected to the GUI boxes by dotted arrows stand for the modules in which the actual functions are implemented. They are called sequentially from the GUI modules.

4.2.1. The user interface

The overall look and usage of the user interface of *BosSE* is described here.

The GUI was designed using wxPython. To start the application *BosSE* the GUI has



Enter a wikipedia xml dump: jo2.xml

Eliminate stopwords? ✓ Yes Choose list of stoppword /stopbosse.txt

Term weighting? ✓ Yes idf value 0.3

Number of documents to index:

Enter ..._svd.txt

value of k: 40 Dimensions (1 – 62)

```
You chose jo2.xml
Eliminate stopwords: True  Weighting of terms: True
number of documents to index: 62
 indexing and local weighting was done
15428 terms in Term Document Matrix
global weighting was done and matrix is filled
SVD was done in 0:00:41.575955
reducing and saving done
           Preparation is finished!
```

Figure 4.2.: The user interface of *BosSE* - the preparation panel
1:three panels of *BosSE* 2:selection of search base 3:options for excluding stop words and using term weighting 4:specification of idf value for removing frequently used terms which are not listed in the stop word file 5:specification of number of documents to index 6:specification of number of remaining dimensions (k) 7:selection of svd files 8:status frame.

to be called in the shell by typing `pythonw BosseGui.pyw`. In this file (BosseGui.pyw) the application window and the top frame are defined.

As can be seen in figure 4.2 at label 1 the interface is divided into three panels: *Search*, *Extras* and *Preprocessing*.

The panels are arranged by frequency of use not chronological. This is done, because in the best case the preprocessing is done only once, whereas the search can be executed as often as needed once the SVD space is calculated.

The Preprocessing panel

For the first use of *BosSE* the user has to start with the preprocessing panel which is defined in the module `preppanel`.

The user is asked to specify which data base he wants to search (label 2 in figure 4.2). Therefore he uses the browse button, which will open a file dialog from which by default only XML files are selectable.

At the next step the user has to choose, if he wants stop words to be deleted from the index of words. Stop words are all the words that are considered common and not used for indexing. The list of stop words differ from language to language. The file containing this list must be a plain text file written in UTF-8 (or Latin-1).

If the users checks the 'yes' button behind 'Eliminate stop-words' (which is by default unchecked) a text field and another 'browse' button will turn up, with which the user can choose his list of stop words (label 3 in figure 4.2).

Another important feature is 'Term weighting'. This option is set by default. If the user leaves that choice, the frequencies of term occurrences will be weighted locally and globally (see section 4.2.2).

If both the option for eliminating stop words and the weighting option are set to 'Yes' another text field turns up. It is called 'idf value' (see label 4 in figure 4.2). Auxiliary to the list of stop words, this field gives the opportunity to eliminate corpus independently all terms that occur in most of the articles (for details see section 4.2.2). 'idf' stands for inverse document frequency, how it is calculated will be described in equation 4.2. The smaller the idf value for a special term in more documents that term appears. If for example a word occurs in every document all words would be linked by second order co-occurrence. Mostly these words are common words like forms of *to be* which normally are listed in the file of stop words. But in some document collection some words appear in almost every article e.g. in Wikipedia the terms *category* or *image*. The list of stop words should be language specific but should not have to be revised for every corpus used with an search engine.

Finally the user can enter the number of documents which should be indexed (at position 5 in figure 4.2). If all articles of the collection are to be read in the field can be left blank. This feature was added during the phase of implementation after recognising, that it is not possible to index all 280,000 articles of the German Wikipedia with *BosSE*. The

calculation of the SVD space for such a big TDM is to CPU-intensive for the machine used.
Now the preprocessing can begin.

In the text area at the bottom of the window some information about the status of the application will be given. First it is repeated which corpus was chosen and which options were set.
The filename along with the values for the two options, the file of the list of non content words (if this option was chosen), the values of the idf text field and the document field are referenced to the `main` function of the self defined module `indexer` see section 4.2.2. When *BosSE* indexed the articles the number of documents is displayed as well as an acknowledgment that the weighting was done. After that a list of words which were eliminated from the index due to their idf value and the number of terms written to the TDM are given. When *BosSE* has finished the first part of preprocessing it will be stated that the matrix was filled.
Now this matrix will be decomposed. Therefor the TDM is given to the function `svd` in the module `svd` (see section 4.2.2). After the factorisation is finished, it will be displayed how long it took *BosSE* to calculate T, S and D.
During these first steps of indexing some files are created. The first one is a XHTML file with the name *`language`*`_D`*`number_of_documents`*`.xhtml` to which the actual text of the indexed articles is written. The other files for the search basis created here all start with the same string, which will depend on the options used: *`language`*`_D`*`number_of_documents`* `_S`*`Boolean_value_of_stop_word_option`*`_W`*`Boolean_value of_weighting_option`*`_idf` *`value_of_idf_field`*. The language string is taken from the XML file, the others are taken from the GUI. The TDM will be written to the file *`namestring`*`_tdm.txt`, the dictionary of indexed terms to *`namestring`*`_tdmdict.txt` and the result of SVD to *`namestring`*`_svd.txt` (detailed description in section 4.2.2). These files will be stored in the same directory as the article collection which should be a sub-directory of the *BosSE* source code directory. Due to this naming convention all files can easily be recognized and assigned to a search basis (see section 4.2.2).
When the SVD was done a slider (label 6 in figure 4.2) will appear in the GUI with a range from 1 to the rank of the TDM. With this slider the user can enter the number

of remaining dimensions in the SVD space, namely the value k. By default this slider is set to 150 dimensions. If the document collection contains less than 150 documents the slider is set to the half of the number of dimensions.

After choosing k, *BosSE* will proceed with the module `reduce` where the semantic space that is the matrices T, S and D will be truncated and three more files will be created one each for T, S and D (*`namestring_knumber_of_remaining_dimensions`*`_redS.txt`, `-_redT.txt`, `-_redD.txt`) (see section 4.2.2).

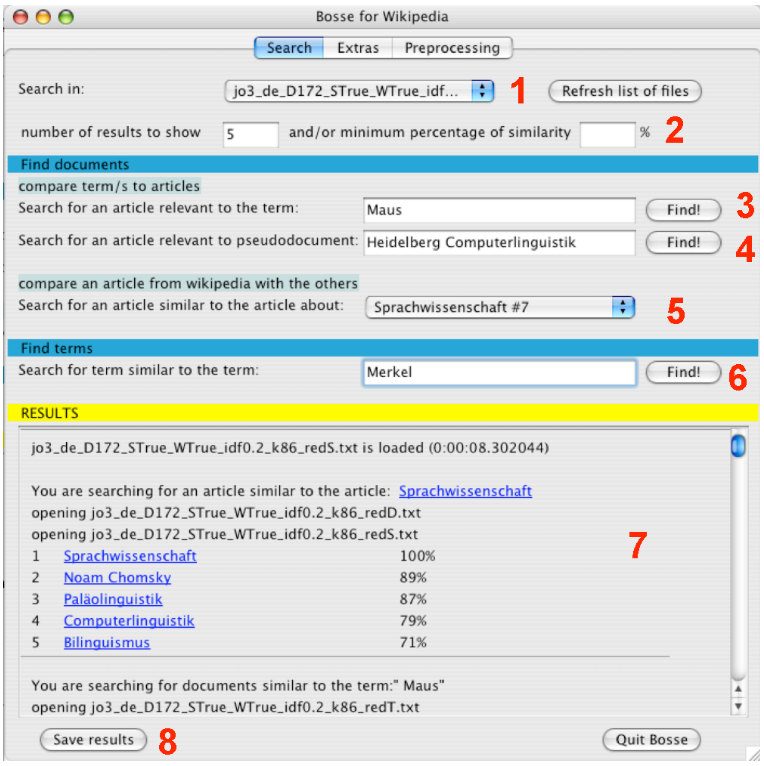
If the user wants to create different truncated SVD spaces for the same document collection he does not have to start from the beginning. He can also enter an existing `_svd.txt` file into the text field 'Enter ...svd.txt' (label 7 in figure 4.2). After a while the slider will turn up and the user can proceed.

Now the preparation is finished and the search can begin.

The search panel

The search dialog described here is implemented in the module `searchpanel`. The search dialog is designed that the user starts with the selection of a search basis. An alphabetically ordered list of all `_redS.txt` files which are found within the *BosSE* directory is displayed in a choice box. If the user just did the preprocessing he has to use the 'refresh list of files' button, to see the search basis just created. When the user chooses a basis the namestring will be saved to a variable called `fn` to be able to find all other files necessary for performing searches. The first file to be opened is the corresponding XHTML file. It will be opened using the `load` function of the cPickle module (a module that allows to save complex serialized data structures to a file and which creates these data structures when the file is read in). The XHTML file will be searched for the titles of the articles which then will displayed in a choice control next to 'search for an article similar to the article about:' (see label 4 in figure 4.3).

Now the user can enter the number of results he wants to get back from the search engine. If he does not give this value the number is set to -1. Or he can specify a minimum of similarity the results should have. Here he enters percentages. If this one is left blank the value is set to 0%. If both values are not specified all documents or terms will be returned by *BosSE* in ranked order (by percentage of similarity). When both fields are filled and for example 20 results should be shown but only with a higher or equal simi-



Search in: jo3_de_D172_STrue_WTrue_idf...

number of results to show 5 and/or minimum percentage of similarity %

compare term/s to articles

Search for an article relevant to the term: Maus

Search for an article relevant to pseudodocument: Heidelberg Computerlinguistik

compare an article from wikipedia with the others

Search for an article similar to the article about: Sprachwissenschaft #7

Search for term similar to the term: Merkel

```
jo3_de_D172_STrue_WTrue_idf0.2_k86_redS.txt is loaded (0:00:08.302044)

You are searching for an article similar to the article:  Sprachwissenschaft
opening jo3_de_D172_STrue_WTrue_idf0.2_k86_redD.txt
opening jo3_de_D172_STrue_WTrue_idf0.2_k86_redS.txt
1   Sprachwissenschaft              100%
2   Noam Chomsky                    89%
3   Paläolinguistik                 87%
4   Computerlinguistik              79%
5   Bilinguismus                    71%

You are searching for documents similar to the term:" Maus"
opening jo3_de_D172_STrue_WTrue_idf0.2_k86_redT.txt
```

Figure 4.3.: The user interface of *BosSE* - the search panel
1:selection of search basis 2:specification of number of returned results 3:term-documents search 4:query-documents search 5:document-documents search 6:term-terms search 7:status window 8:saving of results

larity of 80% and there are more than 20 results with a higher similarity than 80% only 20 will be displayed. On the other hand if there are only 10 terms or documents with 80% similarity to the query or higher only these 10 are listed.

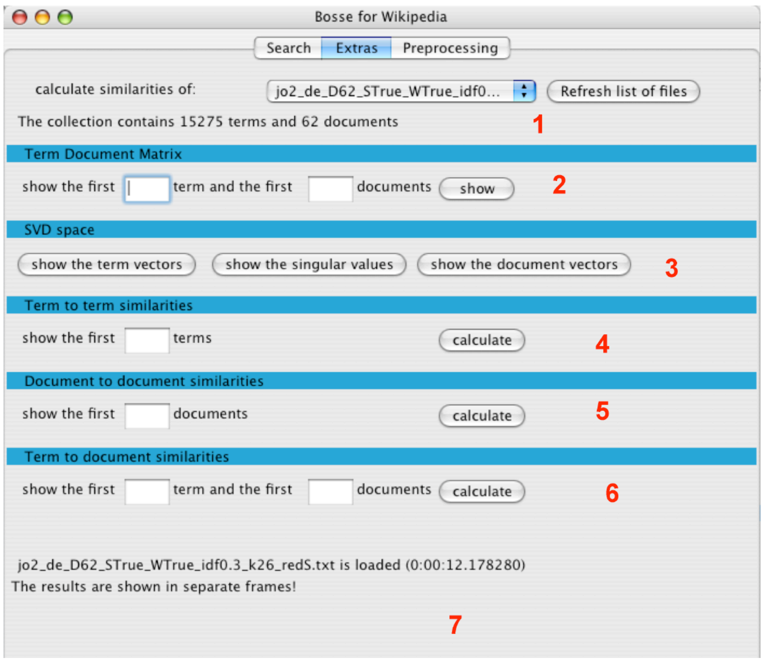
Now the user can choose between four different search types. They are:

1. search for documents similar to a term
2. search for documents similar to a query (a set of terms)
3. search for documents similar to a document
4. search for terms similar to a term

For the first search type the user can only enter one term. Here the similarities between the term vector for the entered word and all document vectors are calculated (see term to

document similarity in section 3.2.3). For this search the files `fn_redT.txt`,`fn_redS.txt` and `fn_redD.txt` are needed and opened.
After the similarities were calculated a ranked list of results is returned to the GUI and displayed in the result frame at the bottom of the window (see label 7 in figure 4.3).
This frame is a scrolled frame and implemented in the module `scroll`.
The list of results is composed of the rank, the title of the document (which is linked to the article in the XHTML file) or the term and the similarity of the result to the query in percent. By double clicking on the title, the XHTML file should be opened in the standard browser and scrolled to the appropriate article.
The results can also be saved to a text file by using the 'save results' button (label 8 figure 4.3).

The extra panel



calculate similarities of: jo2_de_D62_STrue_WTrue_idf0...

The collection contains 15275 terms and 62 documents

jo2_de_D62_STrue_WTrue_idf0.3_k26_redS.txt is loaded (0:00:12.178280)
The results are shown in separate frames!

Figure 4.4.: The user interface of *BosSE* - the search panel
1:display of dimension of TDM 2:call of the function for showing the TDM 3:buttons for displaying T, S and D 4:function displays the table of the term to term similarities 5:function for displaying the table of the document to documents similarities 6:function for displaying the table of the term to documents similarities 7:space for messages to the user

The *extra* panel is only for research purposes and is specified in the module `extrapanel`. Just as in the search panel firstly a search basis has to be chosen. Accordingly to the preprocessed search basis chosen, the files containing the TDM, the truncated matrices T, S, and D as well as the dictionary of indexed terms is opened. This will take a while depending on the size of the TDM.
After this has been done it is stated how many terms and documents are indexed in this search basis (see label 1 in figure 4.4). Now the researcher can have a look at the TDM created by the indexer (at label 2 in figure 4.4) as well as at the three matrices T, S and D - the results of the SVD (see label 3 in figure 4.4). This data is taken directly from the corresponding files.
Furthermore the user can view the different search matrices (label 4 to 6 in figure 4.4). Here the different matrix products I talked earlier about can be calculated and viewed. The actual tables displaying these data are shown in separate frames. To display them the module `mytable` is used.

After all parts of the GUI has been discussed, in the next sections the several functions of *BosSE* are described.

4.2.2. The Preprocessing

The preprocessing is the first step that has to be performed before initial use of the search features of *BosSE*. Here the bases for searching are created. The different functions which build up the step of preprocessing will be described in this section.
The chart in figure 4.5 shows the data flow within *BosSE*. Firstly the user interacts with the GUI and then starts the preprocessing.
The string from the text field for the XML dumps is read and the values of the options for excluding stop words and term weighting are saved to the Boolean variables ***`stop`*** and ***`weight`***. The string of the text field for the file of the stop word list is written to the variable ***`stoplist`*** and the ***`idf value`*** is saved to the variable ***`idf`***. If the user entered a number of documents to index this is written to the variable ***`docs`*** otherwise it is set to -1.
With this parameters read from the GUI the function `main` in the module `indexer` is called.

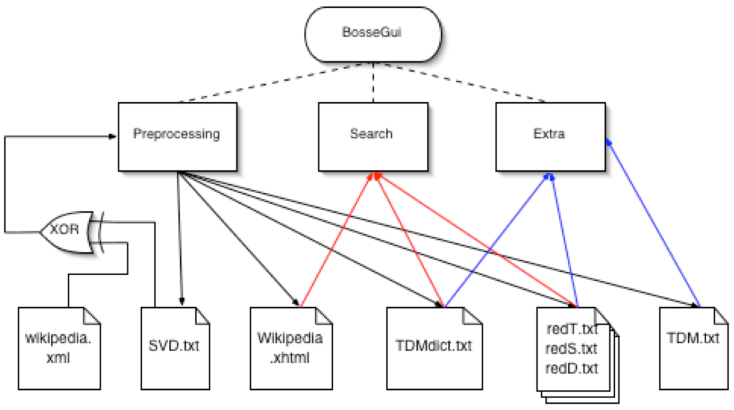


Figure 4.5.: Flow chart of *BosSE*: the text files ware written during preprocessing and are used for the search and/or for the extra features. The svd.txt is only used, when the user wants to create another SVD space for a present index.

indexer

If the option for eliminating stop words was chosen `main` firstly opens the file of stop words, reads it as a string, splits it into a list of single terms and saves it to ***`stoplist`***. The words have to be separated by spaces, for the split routine takes a space as a word boundary. The idf value, provided that the option of term weighting was set to `True` and a value was entered, is converted to a float otherwise it is set to 0. If ***`stop`*** was set to `False` the variable ***`stoplist`*** is instantiated with an empty string.

Now the actual preprocessing begins.
The XML file is opened and by using the `iterparse` function of the external module `cElementTree` a tree of the XML structure of the file is build.
The tree is searched and from the element *`<mediawiki>`* which is the first element in the file the language attribute is read. By using the function `head` of the module `writexml` the language attribute together with the usual XML declaration and the head for the XHTML structure is written to the new created XHTML file.
From each *`<page>`* element found in the XML tree the title and the text of the page (which corresponds to an article of Wikipedia) are retrieved and prepared for indexing.
For each page the following is done:

The title is searched for a colon. If the string contains a colon the article is a special

item of Wikipedia like a discussion or a user page because titles for this pages are labeled with a keyword (e.g. discussion) followed by a colon. These pages are undesirable for indexing for they might contain nonsense or no factual article and are skipped.
Afterwards the text of the page, which is found in the element *`<page><revision><text>`* is prepared for writing to the XHTML file, so that the user can read it later on. Therefore it has to be made readable. So the function `revisionsub` of the module `substitute` is called. This function deletes all tables and category links etc. from the file because they can not be displayed properly yet. Links to other Wikipedia pages are written like `[[link target|link name]]`. Only the link name shall be seen by the user when reading the articles so the link target and the squared bracket have to be deleted from the text. Headings are labeled in a Wiki by embedding the text of a heading in sequences of equal signs(=). So these sequences have to be translated to the HTML tags for headings: `<h2>` to `<h5>`. `<h1>` is already used for the heading of the whole XHTML file containing the string "Articles from Wikipedia *language* searched with *BosSE*".
`<bold>` is represented in a Wiki by enclosing the terms desired to be displayed in bold typeface in a sequence of three apostrophes ("'this should be bold"') for an italic typeface two apostrophes are used.
Furthermore some common mistakes like the usage of `<br>` in XML files instead of `<br\>` are corrected and for it is an XHTML file all ampersands are substituted by `&amp;`.
This revised text is then written together with the title to the XHTML file.
The text of the title is written to a heading element of second order (`<h2>`) to which an ID and a name attribute was given to be able to identify the article later on and anchor it with a link. The revised text follows as a new paragraph (enclosed in `<p>` and `</p>`).

Now the original is again sent to the `substitute` module but this time changed by the function `indexsub` with first converts the string to lower characters. Then everything that is not a term worth indexing will be deleted. That means all internet and E-mail addresses, tags and numbers will be removed from the text. Also all images, tables and links to categories are deleted. Otherwise (as described in section 4.1.1) the word use patterns would be obscured.
At the and all special characters are taken from the text. Here some special characters might still be lacking. First it was thought to remove all non-characters, but python

considers only ASCII characters as characters so e.g. ä, ö, ü, ß as well as all French characters would be deleted.
The hyphen was also a character to think of. It is substituted now by a blank so words that form together a new meaning like *H-Milch*, *deutsch-polnisch* or *Make-up* will be added to the index as separate words otherwise the TDM would grow bigger and bigger.

From the remaining string a list of terms called ***terms_to_index*** is build by splitting the string after each blank. The list is sorted and the occurrences of each word are counted.
For each term in this list it is checked, if it is listed in the stop word list (if the option was not chosen the ***stoplist*** is an empty string) and if the term string is longer than one character. If these requirements are fulfilled, the term is checked against a dictionary called TDMdict.
In Python a dictionary is an unsorted list of (key, value) pairs with the syntax dict={key$_1$: value$_1$, key$_2$: value$_2$... key$_n$: value$_n$}. The keys are the terms, the values are lists containing: an unique ID for the term as first element (starting at 0 for the first term), as the second element the term again and in its third element a list of frequency lists – one for each document this term appears in. For example the term *mouse* is the first term in the first document and appears in there four times the dictionary would be at first `TDMdict=`{`'mouse':[0,mouse,[[0,4]]]`}. The document numbers appoint the column to which the frequency value will be written. Index of matrices start at 0, that is why in the list of frequencies above a 0 was written although it was the first article.
If a term appears for the first time in the document collection a new entry is added to the dictionary. For example document 2 contains the term *cat* once, so the dictionary becomes `TDMdict=`{`'mouse':[0,mouse,[[0,4]]], 'cat':[1,cat,[[1,1]]]`}. If a term appears again in another document within the corpus a [document number, frequency] list element is added to the list of frequencies. That is to say if *mouse* appears again in document 5 altogether three times the TDMdict becomes {`'mouse':[0,mouse,[[0,4], [4,3]]], 'cat':[1,cat,[[1,1]]]`}.
The unique ID indicates later on the row of the TDM to which the data of the frequency list is written. During filling the dictionary it is noted which was the highest frequency of occurrence of a term in this document. We need this value for the local weighting.

If the user chooses to use term weighting the local weighting is applied after the loop over the ***terms_to_index*** list is finished, that is to say when all terms of the document were indexed and the corresponding data was written to the TDMDdict.

local term weighting

The local weighting scheme measures the importance of a term within the document and it is called Normalized Term Frequency (NTF).

$$f_{i,m} = \frac{freq_{i,m}}{max_{j,m}} \tag{4.1}$$

The normalized term frequency $f_{i,m}$ is calculated by dividing the frequency of occurrence of a term t_i in document D_m ($freq_{i,m}$) by the highest frequency of a term t_n in document D_m ($max_{i,m}$) as shown in equation 4.1. Which means that words that occur often in a document get a higher importance for the text because it is thought to be characteristic for this document. This normalized term frequency substitutes the raw frequency in the list of term frequencies.

When all this was done for every `<page>` element in the XML file or for as many documents as stated by the user the closing tags will be added to the XHTML file and it will be closed.

global term weighting

Then the global weighting scheme is applied on the gathered data. It calculates the Inverse Document Frequency (IDF) (idf_i in equation 4.2). This measures the importance of a term within the document collection. It is calculated by dividing the number of all documents in the collection N by the number of documents the term occurred in (n_i) and takes the logarithm of this quotient to the basis of 2.

$$idf_i = \log \frac{N}{n_i} \tag{4.2}$$

If this value is smaller than the idf value specified by the user, the entry for this term is deleted from the TDMdict.

A term that occurs in every document of the collection gets a lower value and is thereby considered as not as significant for distinction between documents or as expressive for the meaning of an article as terms that occur rarely throughout the document collection. Together with the NTF the IDF gives a good weighting scheme $w_{i,m}$:

$$w_{i,m} = f_{i,m} \times idf_i \tag{4.3}$$

Of course any other weighting scheme could be used with LSI, but this is the most popular one, thus it was decided to use this.
The $w_{i,m}$ is written to the TDMdict and replaces the $f_{i,m}$ The TDMdict is then written to a text file (with the ending `_tdmdict.txt`).
It might be said, that weighting is not needed for LSI for it should decrease the importance of words that are often together with many other words by itself. This could be true and is a matter of evaluation in chapter 5, so until now it is not obliged but a recommended option.

After the weighting of the terms was done, the TDM is build.
First it is counted how many terms were indexed, therefor the length of the TDMdict has to be calculated.
The number of documents were counted during the indexing process. Now a matrix of size $t \times d$ filled with zeros is created.
For each entry in the TDMdict, that is for each term, frequencies (or the weighted frequencies) are written to the matrix: the unique ID gives the row to write to. The first element of each list in the list of frequencies gives the number of the column and the value is given by the second element of each list.
For the small TDMdict example from above the TDM is (without term weighting):

Table 4.1.: Example of a TDMfrom a TDMdict

t\D	0	1	2	3	4
0	4	0	0	0	3
1	0	1	0	0	0

This matrix is saved to a file by using the `dump` function of the external cPickle module (which allows to save complex data structures to a file and create them again, when the file is read using the `load` function).

The step of indexing is finished and the TDM is returned to `preppanel` from where the function `svd` from the module `svd` is called.

svd

To perform the SVD the external module *numarray* [11] is loaded. From this package the function `la.singular_value_decomposition(TDM)` is used. It takes a matrix as input and returns a tuple of the three matrices T, S and D. How SVD works and how it is calculated was described in detail in chapter 3. The resulting tuple is stored to a file ending in `_svd.txt`.
Using this file the user can produce different truncated SVD spaces (that is to say the value of k is different) for the same index without doing the indexing and calculation of T, S and D again.
The resulting tuple along with the rank of the TDM , which is equal to the dimension of the square matrix S, is returned to the GUI and the slider (label 6 in figure 4.2) turns up.

The user specifies a value for k that is to say how many dimensions he wants to keep. Then the `main` function of the module `reduce` is called.

reduce

This functions takes the first k columns of T and writes them to a matrix T_{new} of size $t \times k$, the first k rows of D^T and writes them to the matrix $Dnew^T$ of size $k \times d$. From S the first k values are taken and written to a square diagonal matrix $Snew$ of size $k \times k$. To save this data for searching a file for each truncated matrix is created.
The preprocessing phase is completed.

4.2.3. The Search

After the user had chosen the search basis he likes to search in from the choice dialogue depicted in figure 4.3 at label 1, the function `getdoc` is called.

[11] `http://www.stsci.edu/resources/software_hardware/numarray/`

getdoc

The XHTML file for the chosen search basis is opened and read in. From the external module `pyxml`[12] the XML parser named PyExpat is called. With this parser a XPath expression is evaluated with which the titles of all articles are found. The titles found are returned as a list to the GUI where they are displayed in a choice box (label 5 in figure 4.3).
Now the user can start the searching.

search

Depending on what the user entered the `search` function of the module `search` is called with different arguments.
If the user entered a maximum value of documents to return or a lower limit of similarity, these values are also send to the search function.
Supposing that the user forgot to choose a search basis but entered a query (a term or a document), an error message will be shown in the scroll frame at the bottom of the window.
The calculation of the similarities is basically the same for each search type. The similarities between a query vector called `quvector` and each vector of a matrix called `compareTo` have to be calculated (see below). But the calculation of the `quvector` and the matrix `compareTo` is different from search type to search type. This preliminary work will be described first.

Term to document search

If the user wants to find documents similar to one term, he enters one word in the field next to label 3 in figure 4.3.
The query string (here containing one word) will be converted to lower characters, for all terms in the TDMdict are stored in lower case. If the query contains more than one word the search is stopped and an error message is displayed for the user, that he should only enter one term. Otherwise the search continues.
For this search the matrices TD_T and TD_D have to be calculated (equations 3.11 and equation 3.12).

[12] `http://pyxml.sourceforge.net/`

First it is checked which files needed for performing the search were already read in and which have to be opened and loaded by the `cPickle` module. That is to say if the user performs more than one search the files do not have to be opened each time. For this search type the three matrices are needed so the data of the files
`searchbasis_redT.txt`, `searchbasis_redD.txt` and `searchbasis_redS.txt` has to be read in.
The ID of the term is looked up in the TDMdict. If no corresponding key to the entered search term was found, a message saying that this word is not in the list of searchable terms is displayed. The ID gives the number of the row in which the data for the vector representing this term in the SVD space is written. This vector is written to the matrix named $T_{k,qu}$ (T of the query term) which this time only contains one row and is used instead of the whole matrix T_k. One can also use the whole matrix T_k and multiply it first by a query vector, which notes which term occurs how often in the query. This query vector would be of size $t \times 1$ and would contain zeros except for one component which is set to 1, for the query term. The result of this multiplication is the same matrix $T_{k,qu}$ as if using just this one row out of T_k.
$T_{k,qu}$ is multiplied by $S_k^{\frac{1}{2}}$ accordingly to the function for scaling T_k into SVD space for calculating similarities between terms and documents as shown in equation 3.11. $T_{k,qu}$ is of size $1 \times k$ and $S_k^{\frac{1}{2}}$ of size $k \times k$ so TD_T becomes $1 \times k$.
The similarities between this query vector and all document vectors has to be calculated. Therefor the document vectors have to be scaled as described in equation 3.12. D_k^T of size $d \times k$ has to be multiplied by $S_k^{\frac{1}{2}}$ $(k \times k)$. The matrix TD_D is then of size $d \times k$.
The `quvector` for this search is TD_T and the matrix `compareTo` is TD_D.

Query to document search

If the user types one or more words in the text field for the second search type (label 4 in figure 4.3) the `search` function will be called for searching documents similar to a query. For this search the pseudo-document $\vec{d_q}$ (equation 3.13) and the matrix DD_T (equation 3.9) have to be calculated.
So the files `searchbasis_redT.txt`, `searchbasis_redD.txt` and `searchbasis_redS.txt` are needed and again it is checked if some of them were already opened. The query is used like a document, but therefor it has to be folded into the SVD space to become a pseudo-document.

First it is converted to lower case characters and a vector is build from it containing the raw frequency of terms in the query. If a user wants to emphasize one term of the query, that is to say he wants to give it more weight than the other(s) he can repeat that word. This is taken into account when the pseudo-document is build. For example the query vector for the query "mouse computer computer" would be $\begin{pmatrix} 1 \\ 2 \end{pmatrix}$.
The words are looked up in the TDMdict. The IDs give the number of the rows of T_k in which the data for these terms were stored. These rows are taken from the matrix T_k and form the matrix T_{qu} (of size $length\,of\,query \times k$). By multiplying the transposed query vector $\vec{qu}^T$ (size $1 \times length\,of\,query$) with T_{qu} the pseudo-document $\vec{d_q}$ of size $1 \times k$ is created as written in equation 3.13.
The matrix D_k contains in its rows the document vectors to which the pseudo-document is compared. Therefor it has to be scaled by S_k which creates the matrix DD of size $d \times k$ as stated in equation 3.9. The `quvector` for this search is $\vec{d_q}$ and the matrix `compareTo` is DD.

Document to Document Search

Here the user chose one document and wants to get similar documents.
For this search type the matrix DD_T (equation 3.9) has to be calculated. Here only the files `searchbasis_redD.txt` and `searchbasis_redS.txt` have to be read in, if they were not already opened.
First the D_k matrix is multiplied with S_k as described in equation 3.9 to get DD. From the GUI the ID of the document to which the user wants to find similar articles was returned. To get the data for the chosen document out of DD the $(ID - 1)$th column has to be extracted. This row is the `quvector` and DD is the matrix `compareTo`.

Term to Term Search

For this search, TT has to be calculated (equation 3.7) and only two files would have to be opened: `searchbasis_redT.txt` and `searchbasis_redS.txt`.
The user entered one word in the text field for the term to term search. If he entered more terms, an error message will be displayed and the search is stopped.
First of all the T_k matrix is scaled by S_k which leads to the matrix TT which is the `compareTo` matrix of this search type.

The ID of the term is looked up in the TDMdict and the corresponding row is read from TT. This forms the `quvector`.

Similarity calculation

When this work was done the similarity calculation is done independently from the search type. Accordingly to equation 3.2 the cosine between the `quvector` and each row of the `compareTo` matrix is calculated.
Now this value has to be converted to percentage of similarity. This is done by equation 3.3. If this value is greater than or is equal to the threshold value for similarity (if the user did not enter a value it was set to 0) the number of the row of the `compareTo` matrix along with the cosine value and the percentage of similarity is written to the result list.

Ranking

The list of results are sorted by the cosine value, because it is a float and therefore more accurate than the percentage which is an integer.
When the user specified the peak value of documents to return (x), only the first x entries of the list are kept.

Representation of Results

If the list of results is empty a message will appear in the text area at the bottom, saying that no similar documents or terms compatible to the query and/or similarity requirements were found.
Otherwise the query is displayed in the text area and it is stated which file had to be opened.
There are two different representations for the results.
If the user was searching for terms, the TDMdict is used again. This time only the values of the dictionary are needed that is to say the list of {ID, term, list of frequencies}. The row number saved to the result list is compared to the IDs in the TDMdict. If it matches the term string is extracted and written to the result text area led by the rank and followed by its percentage of similarity.

If the user searched for documents, the title of the documents have to be found, for in the result list only the number of the row of the D_k matrix is listed. Therefore the XHTML file has to be searched for a heading element `<h2>` with an ID equal to the row number. For this the PyExpat parser and an XPath expression is used again to find the title of the article, which is stored in a name attribute of the `<h2>` element. If it was found a link object with a hyper reference to the article in the XHTML file is created. Then the results can be displayed in the GUI, starting with the rank, followed by the link object and the similarity percentage.

The results can be stored to a text file by using the save button. During the search process all important data was written to a string which can be written to a chosen text file.

4.2.4. The extra features

If the user wants to see the values and matrices used within *BosSE*, he can use this panel. First he has to choose a search basis.
Then the files `*searchbasis*_redT.txt`, `*searchbasis*_redD.txt`, `*searchbasis*_redS.txt`, `*searchbasis*_tdm.txt` and `*searchbasis*_TDMdict.txt` are opened and loaded. The dimension of the TDM gives the number of terms and documents in the search basis. This information is send to the GUI (see label 1 in figure 4.4).
If the user wants to see the entries from the TDM he has to enter how many terms (x) and documents (y) he wants to get displayed. These values are taken and a new grid window specified in the `mytable` module is created, showing the xth first rows and the yth first columns of the TDM. The terms are shown in the first column of the tables for it is easy to get the corresponding terms for the rows, whereas the document titles are not displayed, because then the XHTML file had to be searched again for the titles whose IDs correspond to the column numbers.
If the user likes to see the entries of the T, S, or D matrices he has to click on the appropriate button (shown in figure 4.4 at label 3). Then the whole data is taken from the corresponding files and shown in separated frames. The rows of T_k are named by the terms.
To display the term to term similarities, the user has to enter the number of terms (x) he wants see the similarities for.
The matrix T_k will be truncated to hold only the information of the first x terms. This $x \times k$ matrix $T_{x,k}$ is used instead of T_k in equation 3.6. This leads to a matrix containing

the dot products between all term vectors. This matrix is shown in a separate window. For the display of the similarities between terms and documents or between documents the procedure is the same only by using $T_k \, and \, D_k$ or only D_k and use them with equations 3.11, 3.12 and 3.8.

After the implementation of *BosSE* has been described it is of interest how good its performance is.
The retrieval performance is discussed in the following chapter.

Chapter 5

The Retrieval Performance of BosSE

The retrieval performance of *BosSE* was tested with a corpus of 169 articles from the German Wikipedia. The articles were extracted by using the export feature of Wikipedia mentioned in section 4.1.1. The articles in the test corpus are listed in Appendix A.
This document collection was indexed in four different ways:

- stop words were excluded (option S=True) and term weighting was applied (W=True) (called TT)
- only term weighting was used (S=False W=True) (called FT)
- only the option for excluding of stop words was chosen (S=True W=False) (called TF)
- neither elimination of stop words (S=False) nor term weighting was used (W=False) (called FF).

The attributes of these different index variants are shown in table 5.1.

	TT	FT	TF	FF
number of documents	169	169	169	169
number of unique terms	23360	24165	23360	24165
mean number of terms per documents	138.2	142.9	138.2	142.9

Table 5.1.: Attributes of the search bases

The number of unique terms is 24,165 or 23,360 (when only content words are indexed). That leads to a TDM of 4,083,885 or 3,947,840 cells respectively.
BosSE even kept those terms which occur in only one document resulting in an average number of 138.2 unique terms in each article, whereas [Deerwester et al., 1990] removed these terms from the corpora used (CISI and MED). For the MED, which consists of 1033 medical abstracts, an average of 50.1 terms per documents was calculated. For the CISI, a corpus of information science abstracts, this number was 54.5. Adding this feature to *BosSE* has to be considered (see section 7.2.1).
For each of the four different index variants (TT, FT, TF, FF) search bases with different numbers of remaining dimension (k={25, 50, 75, 100, 125, 150, 169}) are calculated, which leads to 28 search bases.
The value of k is a crucial value for the performance of a search engine using LSI. If too many dimensions are kept, the latent semantic structure cannot be revealed since the documents and words are not projected near enough to each other and too much noise remains. If k is too small then too many words and/or documents will be projected into a dimension, destroying the latent semantic structure. I want therefore to evaluate which value of k is best for this test corpus and whether there is a difference between the different search types and the different index types.
For each of the search bases 12 searches were performed: three word to documents searches, three query to documents searches, three document to documents searches and three word to words searches, resulting in 336 single searches.
For this performance test all items receiving a similarity value of 75% or higher are considered relevant to the query. The queries and the relevant items are shown in Appendix B.

The performance is measured using the Recall–Precision scheme.
The precision (P) states how many of the returned results are relevant to the query. The formula for calculation of the precision is shown in equation 5.1.

$$P = \frac{number\,of\,returned\,relevant\,items}{number\,of\,returned\,items} \tag{5.1}$$

The value ranges from 0 and 1, where 1 represents the best precision, i.e. that all returned items are relevant to the query.

The recall of a search (R) indicates completeness of the search results. It states how many of all the relevant results were found by the search engine. It is calculated using equation 5.2.

$$R = \frac{number\,of\,returned\,relevant\,items}{number\,of\,relevant\,items} \tag{5.2}$$

The value of R lies between 0 and 1, where 0 shows that no relevant item was returned, whereas 1 illustrates that all relevant items were returned by the search engine.

The F-measure (F) combines the precision and recall values into one value. It is calculated by using the equation 5.3.

$$F = \frac{2PR}{P+R} \tag{5.3}$$

The value of the F-measure also ranges from 0 to 1. An explanation of the F-measure is given in [Rennie, 2004].

In the next sections the four different search types and the performance of the different indexed search bases with varying values of k will be discussed.
Some charts are shown, depicting the course of precision, recall and F-measure over varying values of k. The curves in these charts are approximations, since it is only possible to test some values of k and no formula is available for calculation the precision and recall for any desired k. In other words, the conjunctions between the points of actual tested k only depict trends.

5.1. The document to documents search

As can be seen in figure 5.1 the recall for all index types (depicted by small dotted lines) decreases when the number of remaining dimensions grow.
It is interesting that the recall for the index type FF (grey small dotted line) stays at 1 and only declines when k is greater than 100. That means that *BosSE* is not able to find all relevant documents when $k \geq 100$ because too much "noise" remains in the corpus obscuring the latent semantic structure.
Stated simply, on the other hand, the precision increases with a growing k. But it has

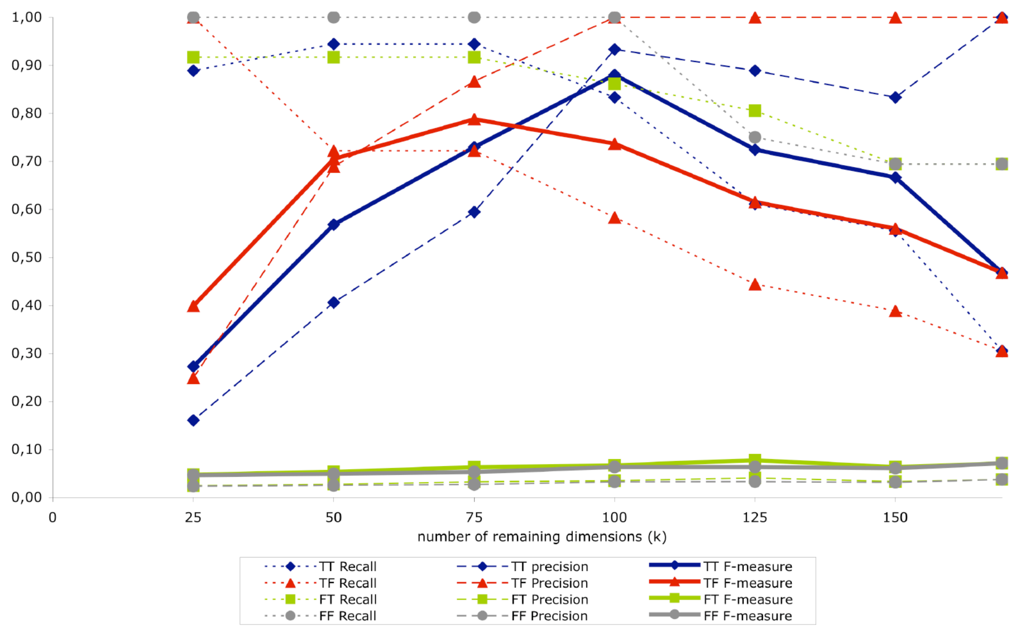


Figure 5.1.: Precision, recall and F-measure curves for document to documents searches with different values for k

to be noted that the precision for search bases where the stop words were not excluded (FT and FF (green and grey dashed lines)) always lies below a value of 0.05. That means that not even 1 out of 20 returned documents is relevant to the document entered. The precision for the TF search bases stays at 1 when $k \geq 100$.

For the TT (blue lines) and TF (red lines) search bases the precision and recall curves have contrasting slopes resulting in F-measure curves describing arches where the vertex for the TT F-measure curve (blue line) lies with $k = 100$ and for TF at $k = 75$.

In conclusion, for the document to documents search excluding stop words is crucial.

The optimal search base for this type of search is the TT with $k = 100$. Here the results are very good. For example a search for similar documents to an article about "Die Sendung mit der Maus" (a TV show for children with an animated mouse) returned all relevant articles plus the article "Maus (EDV)" (about the input device in computing).

If term weighting is not desired a value for $k = 75$ should be used.

It has to be noted, that the TT index scheme only performs better with $k \geq 100$. For smaller k the index variant TF returns better results.

5.2. The word to documents search

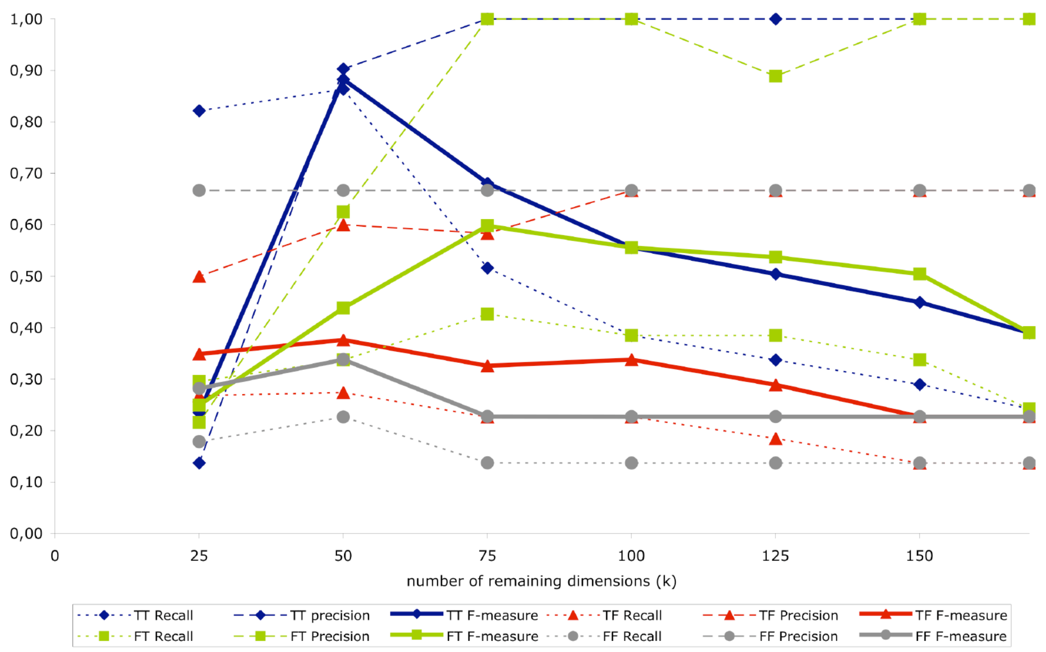


Figure 5.2.: Precision, recall and F-measure curves for word to documents searches with different values for k

The recall, precision and F-measure curves for this search are shown in figure 5.2.
The recall for search bases where stop words were not excluded (TF– red dotted line and FF– grey dotted line) is inefficient. The values lie under 0.3 which means that less than 3 out of 10 relevant documents are returned.
The precision (except for $k = 25$) always lies over 0.5 which means that most of the returned documents are relevant to the entered term.
Another striking feature is that the precision for FF search bases (grey dashed line) is constantly 0.65, and thus is likely to be a coincident.
Similar to the document to documents search the performance of word to documents search depends particularly on one option. This time it is the term weighting. For the search bases where this option is not set to True (TF and FF) the F-measure always stays under 0.4.
The curves for the precision and recall of the index types TT and FT again run counter to each other.

The best performance was observed for the search base TT_{50}.
Here reliance on the correct chosen value for k is especially relevant. The curve for the TT index type shows an explicit peak at $k = 50$.

5.3. The query to documents search

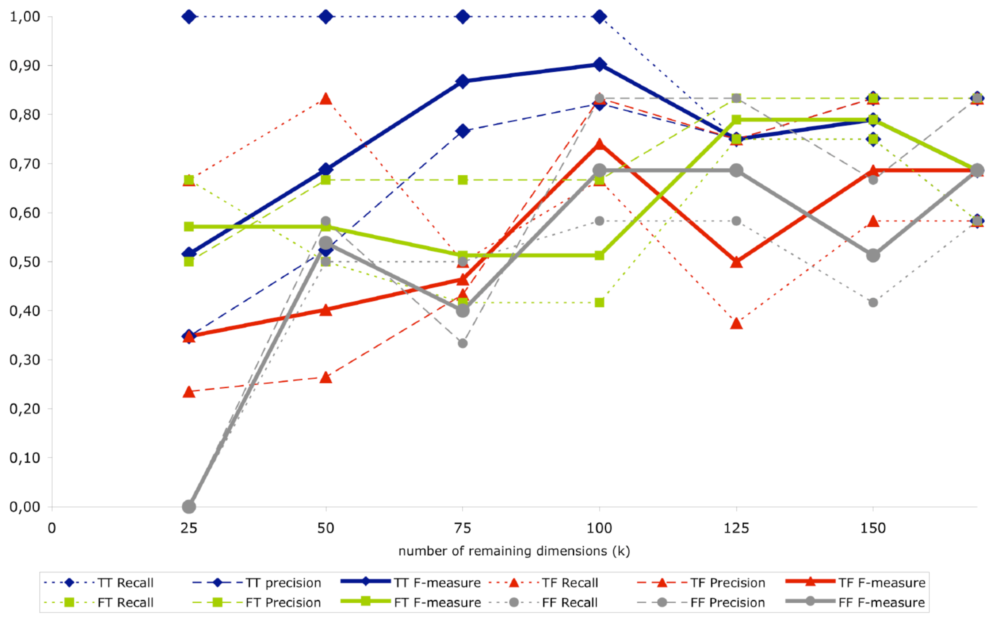


Figure 5.3.: Precision, recall and F-measure curves for query to documents searches with different values for k

As can be seen in figure 5.3 this search type is different to the previous ones.
This search does not depend on one indexing option.
The best retrieval performance is found again when using TT search bases and k lies between 50 and 100. For $k > 125$ the search bases indexed with the FT index variant performs better. For $75 < k < 125$ the TF and FF schemes are the second best indexing schemes.
In contrast to the word to documents and document to documents searches, here no index scheme performs extremely badly.
Once again the precision and recall curves usually run oppositionally in their slope.
In conclusion this search is the one which depends least on the indexing scheme but still heavily on the number of remaining dimensions.

5.4. The word to words search

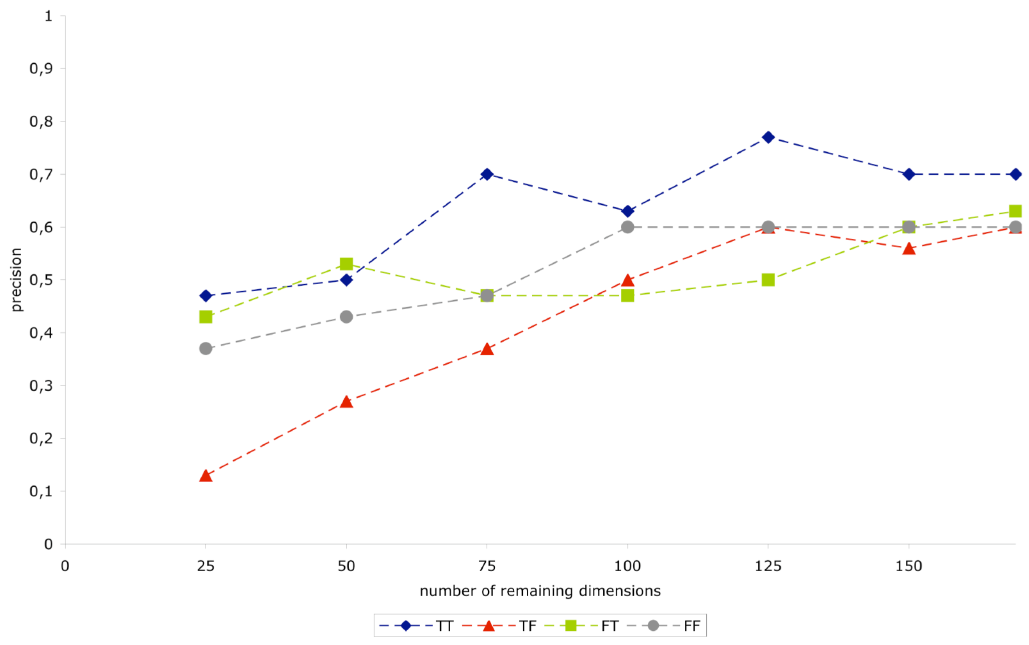


Figure 5.4.: Precision for word to words searches with different values of k

This search is different to all the others. Here words instead of documents are returned. In contrast to the other searches, not all words that have a minimum similarity of 75% are returned, only the first 10 results.
It is hard to say which word has to be considered relevant to the query, for synonyms might be relevant as well as hyperonyms, hyponyms or even antonyms. Thus the recall was not calculated only and the precision is shown in figure 5.4.
The best precision is achieved by using the TT_{125} search base. Unexpectedly the TF precision has the worst performance, even that of the search bases, which do not exclude stop words and term weighting is better. That means that LSI on its own works efficiently, but when changing the indexing process both given index options should be used.

5.5. Conclusion

The F-measures for the three search types which return documents are shown in figure 5.5. As can be seen, the search bases indexed with the TT scheme perform best. Only for the lowest value of k tested (25) and the original dimension of the TDM (169) do other index schemes outperform TT.

The worst performance is achieved by the FF scheme, but it still depends on the value of k.

The F-measure curves for the TF and FT schemes are always in the region of 0.5 where for TF the best value for $k = 100$ and for FT $k = 125$. This would suggest that the term weighting is more crucial to the performance of the search engine than the exclusion of non index terms.

It seems that the best performance for all four index schemes is about $k = 100$ which is about 60% of the original dimension.

But this value is not assignable to all corpora, in particular not to larger ones. In these document collections the number of new terms, which will be added to the index when another document is preprocessed, decreases during the indexing process. Thus for twice the number of documents there will not be twice as many terms. The dimensions of the TDM therefore do not grow linearly with the growing number of documents and the k value cannot be generally set.

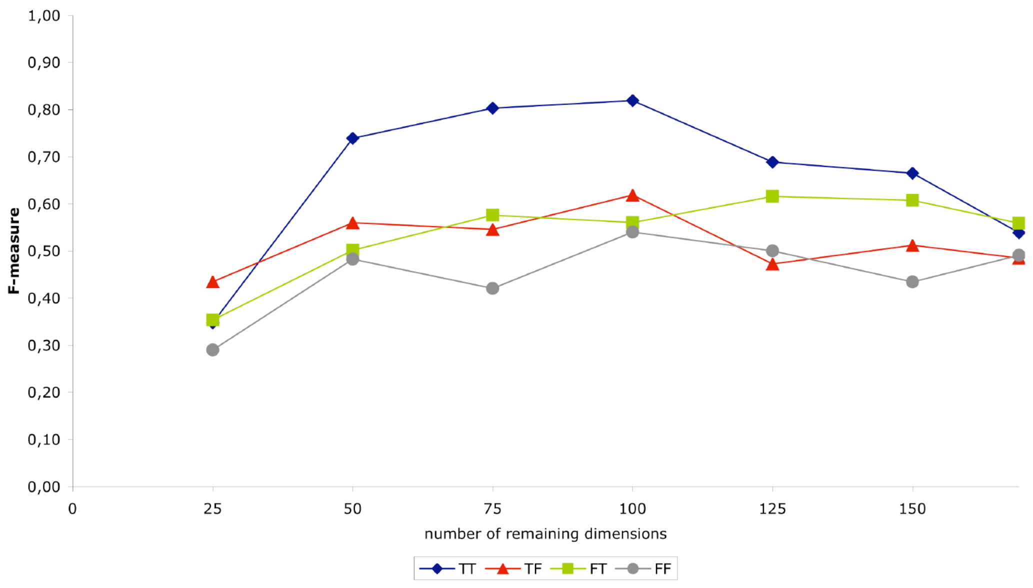


Figure 5.5.: The averaged F-measures of the four index schemes for the searches returning documents.

Chapter 6

Unresolved problems

6.1. Preprocessing

During the phase of preprocessing the XML file is read in and parsed. The Wiki syntax is removed, replaced by HTML equivalents and written to the XHTML file. Not all Wiki instructions can already be translated to HTML.
If there are HTML errors within the XML file, if for example the closing tag for an element is missing, then this error is transferred to the XHTML file. When this file is opened during search, it cannot be parsed and the titles of the articles cannot be displayed. An HTML validator might assist.

The SVD calculation requires too much time and uses too much memory. For example for a 24,164$\times$179 TDM it takes 6:25 minutes. Indexing a corpus of about 500 documents and performing the SVD took a matter of hours for the computer used. The working memory was too small, so swap files had to be created. But 800 MB was soon insufficient. Thus corpora of 700 and more documents could not be indexed.
The run–time on faster machines has to be examined. It may be possible to decrease memory load and run–time by using the svdpackC[1] instead of the numarray module. An attempt has already been made to reduce the degree of accuracy by decreasing the number of post decimal positions. This was not successful since significant information for some entries e.g. the algebraic sign were thereby lost.

[1]`www.netlib.org/svdpack/` last visited 12.1.2006

6.2. Search

The text files needed for searching are quite extensive. For example, for the test corpus used above, the file containing T of a search base where stop words were not excluded but term weighting was applied, has a size of 84.9 MB.
Depending on the search type, up to three of these files have to be loaded by the `pickle` module (see section 4.2.1), which can take some time and is disturbing for the user. Other possibilities saving and loading complex data types to/from files have to be considered. Depending on the number of results to be returned, once the necessary files are loaded the search itself is quite fast and user friendly.

On Windows there is a problem with opening the link to a document directly from the GUI of *BosSE*. Although all participating modules were examined and the preferences for the standard browser were set, this problem has not yet been solved. But it is possible to copy the link and paste it into the address field of a browser.

6.3. Extra features

The features available on the extra panel are time consuming, once again connected to the problem of loading the text files containing T, S, D and the term-document dictionary. But these features are only for research purposes, the time factor can be disregarded since they are very rarely used and are not necessary for the actual search.

Chapter 7

Future work

Some improvements are listed in this chapter, which could possibly be added to *BosSE*. The implementation and testing of these features is beyond the scope of this master thesis.

7.1. The Folding–in

Folding-in is a mathematical method to project new documents or terms into the SVD space without recalculating the three matrices T, S and D (see [Berry et al., 1995]).
To fold-in a new document, represented by an $m \times 1$ vector $\vec{d}$, the projection $\hat{d}$ of $\vec{d}$ into an existing SVD space is be calculated using equation 7.1.

$$\hat{d} = \vec{d}^T T_k S_k^{-1} \tag{7.1}$$

To fold-in a new term, represented by an $1 \times n$ term vector $\vec{t}$, the projection $\hat{t}$ of $\vec{t}$ into an existing SVD space is calculated using equation 7.2

$$\hat{t} = \vec{t} D_k S_k^{-1} \tag{7.2}$$

At the start of this master thesis the aim was to index the whole German Wikipedia. Therefore, folding-in was considered unnecessary.
However since the discovery, that only a few hundred documents can be indexed, folding-in has become more important for updating and extending the document collection without having to re–do the time consuming SVD calculation.

7.2. Changes to indexing

7.2.1. Exclusion of terms

In *BosSE* even words that occur in only one document are indexed whereas the authors of [Deerwester et al., 1990] propose to exclude these words: "Since few terms which appear in only one document (and were thus excluded by LSI) are used in the queries, the omission of these words is unlikely to be a major determinant of performance.". This exclusion feature could be added to *BosSE*. In my opinion, these words would be good candidates for adding to the SVD space, using folding-in.

7.2.2. Stemmer

When using a stemming algorithm, only the stems of words are written into the TDM. In that case the number of index terms would be reduced and even more importantly a more exact latent semantic structure might be revealed resulting in better retrieval performance. "Stemming, however, seems to capture some structure LSI was unable to capture [...]" [Berry et al., 1995, p. 404].
Stemming could be added to *BosSE* as an option during preprocessing.
In [Klein-Berning, forthcoming 2006] *BosSE* is already used for research in multilingual Information Retrieval using LSI, and among other things the usefulness of stemming to LSI is about to be tested.

7.3. More search types

For some applications it might be useful to find the most significant word or sentence for a paragraph or for an entire document e.g. in automatic text summarization.
Thus search types and the corresponding preprocessing steps have to be added in order to perform a document to term/sentence, paragraph to term/sentence and term to sentence search.

Chapter 8

Conclusion

The implementation of *BosSE* was extremely successful, despite exceeding the estimated time frame. It is currently being used for another master thesis about multilingual Information Retrieval using LSI ([Klein-Berning, forthcoming 2006]).

Unfortunately it was impossible to index and search a corpus of significant size, such as the complete of the German Wikipedia, because of the limited capacities of the computer used. But for smaller corpora *BosSE* works extremely well.
The results are promising. The retrieval performances for all four search types are unexpectedly good, although it was difficult to evaluate the term to terms search.
The performance of LSI improves when the options of term weighting and exclusion of stop words are used. Results show that term weighting is actually slightly more important than the exclusion of non content words.
The number of remaining dimensions for truncating the TDM matrix, is still crucial to the performance of LSI.

BosSE is the first search engine to use LSI to search for terms and documents in different ways.

A tool is now available with *BosSE*, which makes it possible to test and evaluate crucial values within LSI.
BosSE is applicable to many fields of Natural Language Processing. As previously stated the possibilities of multilingual IR are currently being tested using *BosSE*. It could also be extended to become a text summarizing tool, or to build thesauri.

Appendix A

The articles of the test corpus

42 (Antwort)
ABAP
Alan Smithee
Albert Einstein
Albert Fraenkel
Altburg (Burg)
Altherrenverein
Altweibersommer
Ameisenigel
Amos Oz
Andreas Eschbach
Andy Warhol
Angela Merkel
Angelina Jolie
Anluven und Abfallen
Archäologie
Armin Maiwald
Atonale Musik
August Mayer
Austen Henry Layard
Autonomieprinzip
Barockmusik
Bela Lugosi
Benoȋt Mandelbrot
Bianca - Wege zum Glück
Bigos
Bilinguismus
Bill Gates
Blackout
Blaue Hörner
Bosse
Britisches Museum
Bundesmonopolverwaltung für Branntwein
Burgwedel
Carl Hagenbeck
Christoph Biemann
Clone Wars (Cartoon)
Colin Firth
Computerlinguistik
Cordula Stratmann
Der Wolf und die sieben jungen Geißlein
Die Sendung mit der Maus
Dirk Bach
Distinktives Merkmal
Double Density
Edgar Allan Poe

Eforie Nord
Elefant (Jagdpanzer)
Elefant, Tiger und Co.
Elefanten
Elefantenschule
Elefantenvögel
Elefantiasis
Ernst Toller
FBI
Feldberg im Schwarzwald
Fettes Brot
Frank Elstner
Fruchtobst
Fräuleinwunder
Fury in the Slaughterhouse
Gene Roddenberry
Genfer Konventionen
Geocaching
Gesunder Menschenverstand
Gesundheitsvorsorge
Global Positioning System
Grabenkrieg
Grisu
Hansekogge
Hauptverkehrszeit
Hausziege
Heidelberg
Henning Kagermann
Hockey
IBM
Irisdruck
Ispell
Israeliten
Iteso
Janosch
John Langshaw Austin
Johnnie Walker
Johnnie Walker (Radiomoderator)
Jugendpresse
Julia - Wege zum Glück
KZ Bergen-Belsen
Kai Wingenfelder
Kaiserrecht
Karl August Möbius
Karl Lohmann
Karl Markus
Kashima (Ibaraki)
Kinder-Schokolade
King’s College (Cambridge)
Klaus Tschira
Kloster Grönenbach
Knut Hickethier
Königreich Bayern
Landesrundfunkgesetz
Laredo (Spanien)
MLP AG
Magisterarbeit
Mammut
Mannesmann Mobilfunk
Martinsdom (Bratislava)
Maus (EDV)
Mausschwanzartige
Max Mallowan
Metal Matrix Composite

Middlesex University
Mirage (Hotel)
Mittelstand
Muckefuck
Mufti
Märchen
Mäuse
Neoliberale Einheitspartei
Nimrud
Noam Chomsky
Norbert Blüm
Optimist (Bootsklasse)
Paläolinguistik
Pandia
Parlamentarismus
Partei der Alternativen Bürgerbewegung 2000 Deutschlands
Pergamonmuseum
Phönizier
Pinne (Segeln)
Plan 9 from Outer Space
Quint Buchholz
Radio ffn
Ralph Caspers
Raumschiff Enterprise- Das nächste Jahrhundert
Reismalz
Rentenmark
Rossmann
SAP AG
Sachtext
Schillerstraße
Ska Jazz
Skywarn
Sprachwissenschaft
Stephen Hawking
Studium
Suprematismus
Swanfassung
Telenovela
Thomas Gottschalk
Tierpark Hagenbeck
Tigerenten Club
Tonke Dragt
TorDACH
Trekkie
Umberto Eco
Universitätsplatz (Heidelberg)
Vorderasiatische Archäologie
Vorsorgeprogramm
Warengruppe
Watergate-Affäre
Wental
Whoopi Goldberg
Wir Kinder aus Bullerbü
Wirtschaftskontrolldienst
Wochenfluss
Zoo Hannover
Zoo Heidelberg
Zoo Leipzig
Zwiebelfisch (Buchdruck)

Appendix B

Queries

B.1. word to documents search

query: “Linguistik”

relevant documents: “Sprachwissenschaft”, “Computerlinguistik”, “Noam Chomsky”, “Bilinguismus”, “Paläolinguistik”, “Distinktives Merkmal”, “John Langshaw Austin”

query: “Archäologie”

relevant documents: “Archäologie”, “Vorderasiatische Archäologie”, “Max Mallowan”, “Austen Henry Layard”, “Nimrud”, “Britisches Museum”, “Phönizier”, “Pergamonmuseum”

query: “Politik”

relevant documents: “Neoliberale Einheitspartei”, “Norbert Blüm”, “Angela Merkel”

B.2. Query to documents search

Qu1= query: “Fernsehsendung Improvisation Fernsehpreis Comedy”

relevant documents: “Schillerstraße”, “Cordula Stratmann”

Qu2= query: “grüner Pullover Sachgeschichten Sonntag”

relevant documents: “Christoph Biemann”, “Die Sendung mit der Maus”, “Armin Maiwald”, “Ralph Caspers”

Qu3= query: “schlechtester Film aller Zeiten Wood”

relevant documents: “Plan 9 from outer space”, “Bela Lugosi”

B.3. Document to documents search

query: “Austen Henry Layard”

relevant documents: “Austen Henry Layard”, “Archäologie”, “Vorderasiatische Archäologie”, “Max Mallowan”, “Nimrud”, “Britisches Museum”

query : ”Die Sendung mit der Maus”

relevant documents:“Christoph Biemann”, “Die Sendung mit der Maus”, “Armin Maiwald”, “Ralph Caspers”

query: “Geocaching”

relevant documents: “Geocaching”, “Global Positioning System”

B.4. Word to words search

query: Watergate

query: TV

query: Ausgrabung

References

[Penguin, 2001] *The new Penguin dictionary.* Penguin Group, 2001.

[Wikipedia, 2005] Wikipedia- Die freie Enzyklopädie, 2005. URL `http://de.wikipedia.org`.

[Berry et al., 1995] M. W. Berry, S. Dumais, and G. W. O'Brien. Using Linear Algebra for Intelligent Information Retrieval. *SIAM Review*, 37(4):573–595, 1995.

[Deerwester et al., 1990] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by Latent Semantic Analysis. *Journal of the American Society For Information Science*, 41:391–407, 1990.

[Golub and Van Loan, 1996] G. H. Golub and C. F. Van Loan. *Matrix Computations.* The Johns Hopkins Press, Baltimore and London, third edition, 1996.

[Klein-Berning, forthcoming 2006] J. Klein-Berning. Multilinguales Information Retrieval mit Latent Semantic Indexing (working title). Master's thesis, University of Heidelberg, Heidelberg, forthcoming 2006.

[Kontostathis and Pottenger, 2004] A. Kontostathis and W. M. Pottenger. A framework for understanding Latent Semantic Indexing (LSI) performance. *Preprint submitted to Elsevier Science*, 2004.

[Lang and Pucker, 1998] C. B. Lang and N. Pucker. *Mathematische Methoden in der Physik.* Spektrum Akademischer Verlag, Heidelberg, Berlin, 1998.

[Letsche, 1996] T. A. Letsche. Toward Large-Scale Information Retrieval Using Latent Semantic Indexing. Master's thesis, University of Tennessee, Knoxville, Tenessee, 1996.

[Moler, 2004] C. Moler. Numerical Computing with MATLAB chapter 10, 2004 URL `http://www.mathworks.com/moler/eigs.pdf`.

[Persson, 2005] P.-O. Persson. Script to MIT 18.335: Introduction to Numerical Methods (fall 2005) lecture 3, 2005. URL `http://www-math.mit.edu/~persson/18.335/lec3.pdf`.

[Rennie, 2004] J. D. M. Rennie. Derivation of the F-Measure, 2004. URL `http://people.csail.mit.edu/jrennie/writing/fmeasure.pdf`.

[Strang, 2003] G. Strang. *Lineare Algebra.* Springer, Berlin, Heidelberg, 2003.

Glossary

corpus

A corpus is a collection of writings of a special kind (here encyclopedia entries) or sometimes on a particular subject. In this thesis the corpus is considered to be subject independent.

diagonal matrix

In a diagonal matrix only the cells in the main diagonal (from top left corner to bottom right corner) are non-zero.

IDF:**Inverse Document Frequency**

IDF is a global weighting scheme that measures the importance of a term within a document collection.

IR:**Information Retrieval**

LSI:**Latent Semantic Indexing**

NTF:**Normalized Term Frequency**

NTF is a local weighting scheme that measures the importance of a term within a document.

orthogonal

Vectors are orthogonal, if their dot product is equal to 0 that is to say if they form a right angle.

orthonormal

An orthonormal matrix is a matrix whose rows (or column) vectors are orthogonal to each other, which means that their dot product is equal to 0 (geometrically spoken, the vectors form a right angle) and have the length 1.

rank

The rank (r) of a matrix is the smaller of the number of linearly independent rows and the number of linearly independent columns in a matrix: $k = min(t, d)$. Typically in IR it is equal to the number of documents.

sparse matrix

Many of the cells of a sparse matrix are set to 0.

SVD:**Singular Value Decomposition**

SVD is a method for factorization of any rectangular matrix

TDM:**Term Document Matrix**

TDMdict

A TDMdict is a Python dictionary (a list of `<key:value>` pairs), which holds all necessary information about an indexed term in a document collection. The key is the term, the value is a list of: unique ID, term, list of frequency list containing list of `<document number, frequency>` pairs. Example: {`'mouse':[0,mouse,[[0,4], [4,3]], 'cat':[1,cat,[[1,1]]]`}

transposed matrix

To transpose a matrix means to interchange its rows with its corresponding columns. Those matrices are marked by ^{T}.

Acknowledgements

I want to thank Dr. Karin Haenelt, my major advisor, for her support and motivation.
Prof. Dr. Hellwig for introducing me to computational linguistics and for his guidance throughout my studying.
Many thanks belong to Benny who supported me throughout the work and was a great help during all ups and downs :-X. Now we can go geocaching, to the movies, sleep late on weekends .... -yes, I have got my life back!
Thanks to Jette for testing *BosSE* and making suggestions for improvement.
I am also thankful to Suzanne and Thorsten who showed consideration for me and like Jette and Benny did the proofreading.
Special thanks belong to Isobel Ryder-Grabolle for her proofreading.
For supporting me during my whole education and for much more, I want to thank my parents
TNX 1.0E6 to my beloved Titanium PowerBook - thank you for not letting me down. I promise I will never sell you!