A reuse repository with automated synonym support and cluster generation

Laust Rud Jensen

2004

Abstract

Having a code reuse repository available can be a great asset for a programmer. But locating components can be difficult if only static documentation is available, due to vocabulary mismatch. Identifying informal synonyms used in documentation can help alleviate this mismatch. The cost of creating a reuse support system is usually fairly high, as much manual effort goes into its construction.

This project has resulted in a fully functional reuse support system with clustering of search results. By automating the construction of a reuse support system from an existing code reuse repository, and giving the end user a familiar interface, the reuse support system constructed in this project makes the desired functionality available. The constructed system has an easy to use interface, due to a familiar browser-based front-end. An automated method called LSI is used to handle synonyms, and to some degree polysemous words in indexed components.

In the course of this project, the reuse support system has been tested using components from two sources, the retrieval performance measured, and found acceptable. Clustering usability is evaluated and clusters are found to be generally helpful, even though some fine-tuning still has to be done.

Thank you

Finishing this project has taken quite an effort, and it would not have been possible without the help of the people mentioned here. First of all, thank you to my supervisor Peter D. Mosses, and to Brian Mayoh for a steady stream of ideas and constructive criticism. Thank you to the people I have studied with who have helped me finish. Thank you to Kasper Jensen and Jonas Auken for reading the entire document when it was almost done, and to Christian Plesner Hansen for giving feedback on the early drafts. Last, but not least, thank you to my family and to my girlfriend Beth for much needed support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A high-quality code reuse repository can be a great asset in the daily work of a programmer. Modern software development is a complex affair, and instead of trying to write everything oneself, a lot of resources can be saved by reusing an existing functional and tested solution. The savings are not only in programming and testing time, but maintenance also becomes much easier. If an error is discovered in a component used in many places, fixing benefits all who use it.

1.1 Problem motivation

Reusing is not as simple as it sounds. It takes time to locate components appropriate to the current task being solved. If the programmer is not aware of the existence of a usable component, he or she is unfortunately disinclined to make the effort of looking [Ye, 2001]. This results in less reuse and possibly in inferior products as the programmer will write everything from scratch. In this case, the quality of the solution depends greatly on the experience of the programmer, and in most cases it will not be as general or easy to comprehend as a solution built with existing components. Still, having a browsable collection of components is not of much help if there is no interactive search functionality to aid the programmer in the locating process.

Furthermore, if such an interactive system is unable to locate problem relevant information within the first few attempts, the user is unlikely to return. This means that the system has to be able to handle a diverse range of user queries and take into account the many different words people use to describe the same topics.

As such a reuse support system may cover many different problem domains, some of the returned components may not be relevant to the current problem. When the results of a query have been properly categorized, the user should be able to quickly select the appropriate set of components. This leads us to the *problem description* which will guide the work in this project.

1.2 Problem description

This is the concrete description of what will be solved in this project. Later in the discussion we will refer back to this piece of text.

This is an empirical project where a reuse support system will be constructed. The report will answer the following questions: How can a reuse support system be created, which supports efficient and relevant searching of components? How can basic problems such as searching for a general subject, or locating items not matching exactly but covering the same topic, be solved? How can results of a search be presented in appropriate categories for easier human consumption, and will users benefit from such categorization?

Those are questions answered in this thesis. The project is to make a reuse support system with a good search functionality. The resulting reuse support system should be easily extensible and adaptable to specific problem domains.

1.3 Perspectives

Being able to look up solutions to concrete problems is something most programmers need on a recurring basis. Regardless of programming language and platform, much of the code programmers normally write is fairly standard and has probably been handled before. Searching the Internet does not give very focused results, and the results are not customized to the tools and application programmers interface (API) readily available to the programmer. Adding new and unknown components makes the project dependent on both the stability of the components, and of the availability of either support or the source code. Using the already available, well-known, and tested components seems like a good idea. Adding the ability to easily lookup the needed functionality in the previously located, and locally accessible, component collections, could help speed up the development process. It also gives the previously mentioned benefits of reuse, namely easier maintenance and readability [Ye, 2001, Chapter 2].

1.4 Brief system description

In order to satisfy the requirements of the problem description, a functional prototype will be created. The user will access the system with a browser, which means that the reuse support system could easily made available to many users.

The system will index documentation constructed by the Javadoc tool and make it available for search. In order to ensure better search results than simple exact text matching engines, a method from Information Retrieval called Latent Semantic Indexing will be used. This enables the identification of synonymous words and allows searches to be more topical than exact text matching offers, and gives ranked results according to relevancy. Details of the different technologies can be read in Chapter 3, which is concerned with the general technologies involved. To properly present the results, a clustering engine called Carrot will be used to group the ranked list of search results. It offers a browser-based user interface and allows the construction of various processing pipelines in order to specialize the presentation of search results. Carrot is further detailed in section 5.3.1.

1.5 Terminology

In order to facilitate the reading of the following text, a specific terminology will be used:

programmer refers to an end-user of the system – this means an actual programmer who needs some components.

user of the system is also a programmer – it is merely a different term for the same role.

term denotes a single word known by the system. This will be further detailed in Chapter 3.

component refers to a single method which is retrievable by a programmer using the system.

document is a set of terms, covering one component indexed in the system.

code reuse repository is a collection of components directly available for reuse by a programmer.

reuse support system is the system constructed in this project to help programmers find components in a code reuse repository.

1.6 Organization of the discussion

This discussion is separated into a number of chapters. Additional reasons why the described problem should be solved are given in Chapter 2. In Chapter 3, the different technologies used are introduced. Related work is examined in Chapter 4, which also contains details of the specific position taken in this project. Chapter 5 describes the implementation of the reuse support system prototype.

Chapter 6 details the chosen method of evaluation as well as the results of the evaluation. This discussion is completed with the conclusion in Chapter 7, which also contains the final thoughts on future developments of the system.

Chapter 2

Fundamental reasons and method

This chapter gives the reasons why this project could provide a helpful addition to a programmer's toolbox. It will also describe the ideas that are going to be used. First the general problem setting will be described, then the observed difficulty. Finally, the abstract idea for the solution will be presented.

2.1 Why is the problem interesting?

Commenting and documenting code is something few programmers enjoy. Using an automatic tool to index such documentation is widely used technology. The Javadoc tool by Sun and the Doxygen tool[1] are popular tools for this task – they generate static HTML-documentation based on special comments embedded within the source code. By making the written comments more readily available than the HTML generated by these standard tools, the incentive for writing good comments increases. But whether a component is going to be used to solve a given problem depends on whether such a component actually exists, and if it is possible to locate it within reasonable time [Ye, 2001, Chapter 1].

2.1.1 Writing comments

Programmers of reusable methods usually document their code in order to have a future reference, and allow other programmers to build on their work. This is especially true when the code is intended to be reused in different

[1] `http://www.doxygen.org`

contexts. We are going to be using the JAVA[2] language as the example in this discussion, but the constructed system is easy to extend for use with other programming languages. JAVA is platform independent and has a fairly simple syntax, making it easy for programmers to use. When writing programs in JAVA, a special format of comments are used to document the various parts of the code for future reference. These comments are called JAVADOC.

For an example, see the code fragments from example class `Queue` in figure 2.1 on the facing page. Comments about the entire class as such are located in the file immediately before the actual definition of the class. Within the body of the class, comments can be placed to document each method or field. These comments can be added to the member regardless of the visibility (such as `public`, `protected` or `private`) of the given member. For more details of the specific format of comments and the available commands and options, see the official documentation by SUN[3].

The JAVADOC tool is commonly used to construct a coherent low-level documentation for JAVA projects. The generated documentation is in HTML, see figure 2.2 on page 8 for a part of the JAVADOC generated for the example source. Using HTML means that the documentation is easily navigable and platform independent. The comments from the source files are collected and linked together, so that classes and interfaces used as method parameters are converted into links. This way, the documentation is easy to use for a programmer who knows what to look for. There is also a complete index which lists all methods alphabetically.

2.1.2 Locating components

When looking for a solution to a specific problem, the available documentation is often consulted. Other sources of component discovery are previous experience, other developers, or reference books. This section will discuss the JAVA API and *packages* in general. After that, the process of locating desired components will be described.

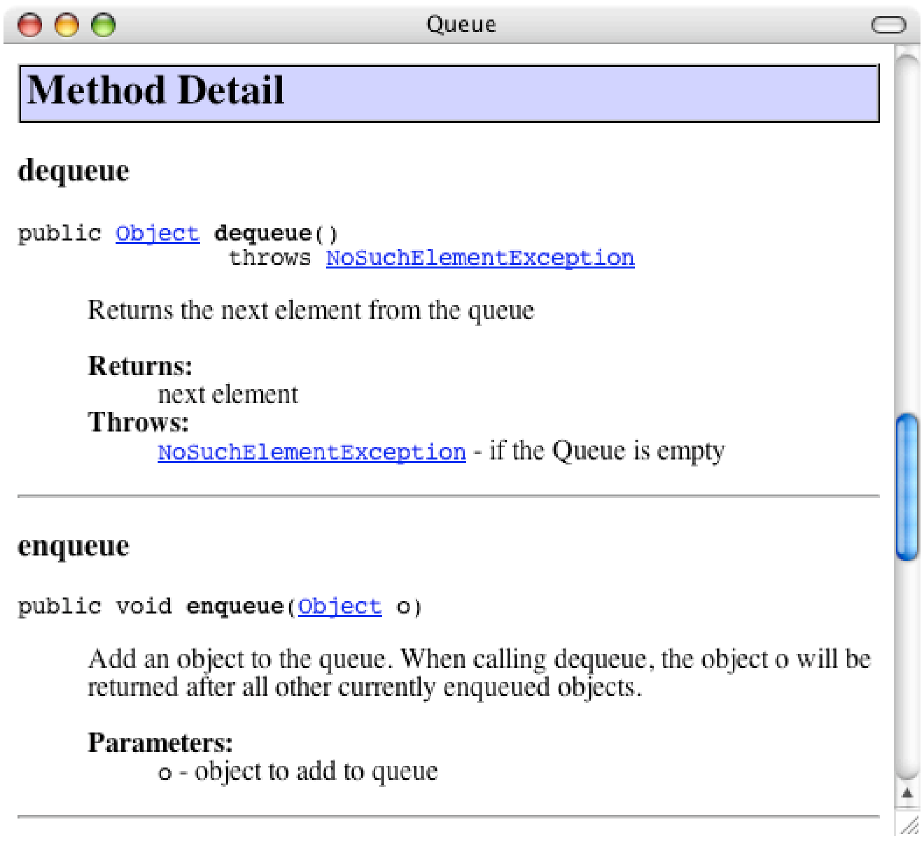
The standard JAVA API is documented using JAVADOC. It is a rather large collection of well-documented classes, which are freely available from SUN for all platforms that runs JAVA. This makes them an obvious choice for use in most projects. But locating suitable components is not as easy as it sounds. Since the collection is so large, a great number of packages are involved.

[2]`http://java.sun.com`

[3]`http://java.sun.com/j2se/javadoc/`

```
package dk.daimi.rud.util;
...
/**
* A basic FIFO queue storing Object instances. Objects are dequeued
* in the same order they were enqueued. Properly encapsulates the
* datastructure to ensure the FIFO contract is enforced. Implemented
* using a LinkedList, so enqueue() and dequeue() are O(1), and
* enqueueAll() is linear in the size of the input Collection
* @author Laust */
public class Queue {
    /** Internal representation of queue */
    private LinkedList queue = new LinkedList();
    /**
    * Add an object to the queue. When calling dequeue, the object
    * o will be returned after all other currently enqueued objects.
    * @param o object to add to queue
    */
    public void enqueue(Object o) {
        queue.addLast(o);
    }
    /**
    * Returns the next element from the queue
    * @return next element
    * @throws NoSuchElementException if the Queue is empty
    */
    public Object dequeue() throws NoSuchElementException {
        return queue.removeFirst();
    }
...
}
```

Figure 2.1: Part of a JAVA source file with JAVADOC comments.



Method Detail

dequeue

```
public Object dequeue()
               throws NoSuchElementException
```

Returns the next element from the queue

Returns:
next element
Throws:
`NoSuchElementException` - if the Queue is empty

enqueue

```
public void enqueue(Object o)
```

Add an object to the queue. When calling dequeue, the object o will be returned after all other currently enqueued objects.

Parameters:
`o` - object to add to queue

Figure 2.2: Part of generated JAVADOC displayed in a browser.

A *package* is a fundamental concept in JAVA where it is used to group a number of classes and interfaces together. See [Gosling et al., 2000, Chapter 7] for all language details on packages. In brief, the members of a given package have special privileges and only members and classes marked as `public` or `protected` are available from outside the package. This grouping also makes the *namespace* easier to manage. The idea is that in the hierarchical package structure groups classes in a logical fashion, and it should ensure that it will be easy to find appropriate components. But people rarely agree on the best name for things, so the sought components are not always where you would expect them to be, or they are called something else. This means that handling *synonyms* when searching can be of enormous help. The problem of naming things is very real, as we shall return to later.

To find a solution to a given problem, the usual process of lookup within a collection of JAVADOC files is to first find the proper package, locate the class, and finally find the appropriate method within the selected class. So it can be described as hierarchical browsing: package → class → method. Another option is to use the alphabetical method index. This process quickly gets cumbersome when you need things from multiple packages or are simply unable to find the desired functionality. Spending too much time browsing for a solution is not very interesting, nor motivating. Moreover, users often do not expect the wide range of available components and therefore resort to trying to solve the problem for themselves instead.

JAVADOC assumes prior knowledge of the contained components and familiarity with the package structure. Or put another way, you have to know where to look to find what you are looking for. This means that there is a steep learning curve in using a specific documentation generated using JAVADOC. But even though the API in the system uses the standard names for things, this does not mean the user will. The use of standard names is similar to the use of design patterns, which also helps users learn a common vocabulary, though unfortunately that does not mean all users will do so.

Since documentation generated by JAVADOC is just a collection of static HTML files, the contained information is fixed. That is to say, the user has no way of easily searching in the documentation, short of the exact match searching capabilities built into most browsers, or going via an external search engine, such as GOOGLE. But most large-scale search engines, including GOOGLE, perform exact word match searching, and the problem of finding the right search words remain. If a programming project is not publicly available, external search engines will not have had access to crawl and index the documentation and therefore be of no help for providing search functionality.

Linking documentation

When documenting a part of a class using JAVADOC, it is possible to add links to other parts of the documentation or external information. This shows up as "see also" sections in the documentation. These links can help the user finding other relevant things, but these links are added by the original programmer when documenting the class, and are limited by the programmer's knowledge of the API. This means that such informational links are not necessarily up to date, especially in situations where more than one API collection is available to the programmer. For example, let us assume that we have two different classes providing random numbers, and that they are from different API vendors. There is no reason to assume that the documentation for each package refers to the other as they come from different vendors, yet the functionality is related, and what is not found in one class may exist in the other.

One of the benefits of the simple HTML rendering of Javadoc is the fact that everything is extremely fast. There is no server-side processing overhead, so you can look at a lot of things very quickly. This implies that a search engine used to augment the Javadoc browsing should not only be good at locating components, it should also be fast, or it will not be used.

Costs of locating components

If the cost of locating a suitable component is perceived to be greater than the cost of making the solution yourself, most users will not make the effort of locating a component. But solutions created by inexperienced programmers are rarely up to the standard of the Java API, for instance, see [Ye, 2001]. This means that users should be encouraged to use existing components if at all possible.

2.2 Solving the problems

To overcome the problem of locating components, a free and searchable reuse support system is needed. As no publicly usable reuse support system seems to be available, this project will construct such a system. There does exist proprietary and expensive systems, but they are of course unavailable for research, hobbyist, and Open Source development.

This project will be used to design, implement, and test such a reuse support system. The reuse support system will be constructed using automatic methods not requiring user input, also known as unsupervised methods. This will be used in order to minimize the cost of starting to use the system. This will give easier access to components than the plain Javadoc, and a better utilization of the resources that have been spent writing and documenting components.

Existing components and subsystems will be used in the implementation where applicable in the implementation, and the reuse support system resulting from this work will be made publicly available for future development and use. The problem of synonymous words being used everywhere will be addressed by the system, and the performance of the system will have to be good in order for it to be a useful tool.

The system should make it easier to locate new and unknown components. As contents of Javadoc created with a recent Java version can be added to the index with ease, it will be possible to merge documentation from different sources into one big code reuse repository. This will make it possible to add search functionality to domain specific documentation, thus enabling programmers to more easily reuse existing proven solutions instead of starting over every time.

2.3 Conclusion

This chapter has described the setting for the project and outlined some of reasons why it is needed. The problems of locating an appropriate component in the static JAVADOC documentation, and the limited search functionality available to programmers has been described. The idea of constructing a repository to work around these problems has been put forward. Problems of imperfect documentation when using components from multiple sources has been mentioned, and solution is the proposed reuse support system.

Important points from this chapter includes the following. Information about related components manually embedded as “see also” sections in JAVADOC is insufficient when using components from multiple sources. Simple keywords matching is not useful in this context. JAVADOC assumes existing knowledge of the contained components, due to the organization. An effective reuse support system should be fast to encourage reuse.

The next chapter describes the different technologies which will be used in constructing the system.

Chapter 3

Technology

This chapter will introduce the various technologies used in this project. It will set the stage for the examination of other approaches to solve the problem described in the problem description in section 1.2. The chapter is also used as a reference when the implementation and experiments are described in later chapters. The description will start by introducing Information Retrieval in section 3.1 and then describe the chosen approach to Information Retrieval in section 3.2. Latent Semantic Indexing is the method used to extract meaningful information out of the document collection, and it is described in section 3.3. It is followed by section 3.4 about searching in the document vector matrix. Finally, section 3.5 introduces main topics in clustering usable for result presentation, and section 3.6 concludes the chapter.

3.1 Information Retrieval

This section introduces the general topic of Information Retrieval (IR). IR is an area of computer science concerned with storing and locating information. In [van Rijsbergen, 1979] the early efforts are described, and in [van Rijsbergen, 2001] more recent advances are covered. The basic idea of IR is to have a set of documents indexed electronically. What is then needed is an efficient way of finding documents relevant to a query without having to manually look through the entire collection. As [van Rijsbergen, 2001] describes, there are four main approaches to IR. These are the Vector Space Model, the Probabilistic, the Logical, and Bayesian network based approaches. Latent Semantic Indexing, which is used in this thesis, belongs to the first group. The other models will be discussed further in Chapter 4.

The Vector Space Model is discussed below and properly introduced in section 3.2.

Many people have tried to automate indexing of documents. At first it was thought computers could be programmed to "read" through an entire document collection, and then be able to extract the documents relevant to a query. However, there are huge problems with getting a computer to understand the syntax of natural language. Understanding the semantics on top of that is gigantic task.

This directed the efforts of IR to constructing a fixed and controlled vocabulary of usable keywords and then manually assigning the appropriate keywords to each document. This is a process similar to a librarian assigning appropriate keywords to an index. Needless to say, this requires a great deal of human effort, and the process is error-prone as the indexer needs to understand the given text in order to assign the appropriate keywords. By choosing the exact keywords a document is indexed with, the set of possible queries which match the document is implicitly constructed. When the end-user knows the proper keywords, finding the correct document is easy and straightforward. But according to [Furnas et al., 1987], two people choose the same term for an object with less than 20% probability. This makes it very difficult to include all the right key-words.

This difficulty directed the efforts towards simpler means. Examples include taking the most frequently occurring words in a document as its keywords. This results in an uncontrolled vocabulary which could more accurately reflect the document collection, but with lower precision since the indexing terms were not chosen by an human expert.

A very central concept in IR is relevancy. The goal is to construct a system that finds all relevant documents and as few unrelated documents as possible. Performance measurements are discussed further in chapter 6, where the experiments with the constructed system are also described.

3.2 Vector Space Model

The Vector Space Model (VSM) is a widely used representation for documents in IR. In this approach, each document is described as a vector which has an entry for each occurring term. All terms are sorted and their position in the sorted set are used as indices in the vectors. Each element in the vector is a representation of the corresponding term in the appropriate document. This means that the positions of the word in individual sentences and other fine points in the text are ignored. The document is simply seen as a *bag-of-words*, meaning that information about sentences is thrown away and only the count of how many times each word occurs is kept. A lot of the vector entries will be zero for most documents, since a collection of documents has many terms but only a relative few are normally used in each document. This means that a sparse matrix representation is often appropriate. The document vector matrix (DVM) is then simply the matrix formed by putting the column vectors of all indexed documents together. This DVM is called X in the following text, and entry x_{ij} in X refers to the relative relationship between term i and document j. The dimensionality of X is $t \times d$ where t is the total number of terms, and d is the number of documents.

One of the benefits of using the DVM is the fact that the similarity between different terms, between a term and a document, and also between different documents in the collection can easily be calculated. This is expanded upon in section 3.4.1, but first we will look at an example of the basic DVM representation.

3.2.1 Document Vector Matrix example

An example should clarify the document vector matrix (DVM) representation and motivate the following discussion – this example is inspired by [Manning and Schütze, 1999, Chapter 15]. Consider the DVM in figure 3.1. The matrix X has been constructed to give a simple representation of a set of documents. It covers a collection of six documents, using a total of five different terms, with documents represented as columns and terms as rows. For simplicity, each document is only represented by a few terms, but usually all words used in a document would be present in the document vector matrix.

The documents seem to cover two different topics, namely space and cars, but of course the choice of words within the similar documents differ. There also seems to be some overlap between documents. Compare documents d_5 and d_6. Apparently these documents have nothing in common

$$X = \left[\begin{array}{l|cccccc} & d_1 & d_2 & d_3 & d_4 & d_5 & d_6 \\ \hline \text{cosmonaut} & 1 & 0 & 1 & 0 & 0 & 0 \\ \text{astronaut} & 0 & 1 & 0 & 0 & 0 & 0 \\ \text{moon} & 1 & 1 & 0 & 0 & 0 & 0 \\ \text{car} & 1 & 0 & 0 & 1 & 1 & 0 \\ \text{truck} & 0 & 0 & 0 & 1 & 0 & 1 \end{array}\right]$$

Figure 3.1: Example document contents with simple binary term weightings.

as they share no terms directly, but if you take d_4 into account, it can be seen that the two terms `car` and `truck` are somehow related through their use, and therefore d_5 and d_6 should be similar. It is this kind of relationship Latent Semantic Indexing, which will be introduced in section 3.3, is employed to reveal. We expand upon this example in section 3.3.3. But first, it is important to remember that the system has no knowledge of the meaning of the different terms. They could be anything such as symbols, words in different languages, phrases, etc. It is the patterns of occurrences which are important.

3.2.2 Feature selection

The selection of representative features in the input data is known as *feature selection*. As huge amounts of data may be have to be represented, choosing the correct subset to represent the entire set is of immense importance.

When the source documents have been indexed, it is time to post-process the data. Usually a reduction in the total number of used words is performed, as is demonstrated in the following. This is done to lighten the load on the calculations following the selection, since the data-sets involved often are huge. Each term becomes a row and each document becomes a column in the DVM. Because some terms occurs very frequently and others very rarely, they may not be good at discriminating between important and unimportant documents. Figure 3.2 on the next page shows such a word occurrence distribution. Terms only used in one document are often removed, as they are obviously not in common use and presumably do not contain a great deal of importance in the document – this is the lower cut-off. An example could be an acronym used once in a technical manual, a simple misspelling, or an unusual word used by a single programmer when documenting a method. Those terms occurring with frequencies above the upper cut-off are too frequent to be of any use. These are handled as *stop-words*, see following sections.

There are some commonly recognized problems when choosing words to index a given document. In the literature they are called synonymy and polysemy [Berry et al., 1999, p. 336]. They are described below, and they generally decrease the performance of IR systems.

Synonymy

This concept covers the fact that multiple words can have more or less the same meaning. An example of synonymous words could be "strength" and "force" in the context "strength of mind" and "force of mind". It is a great

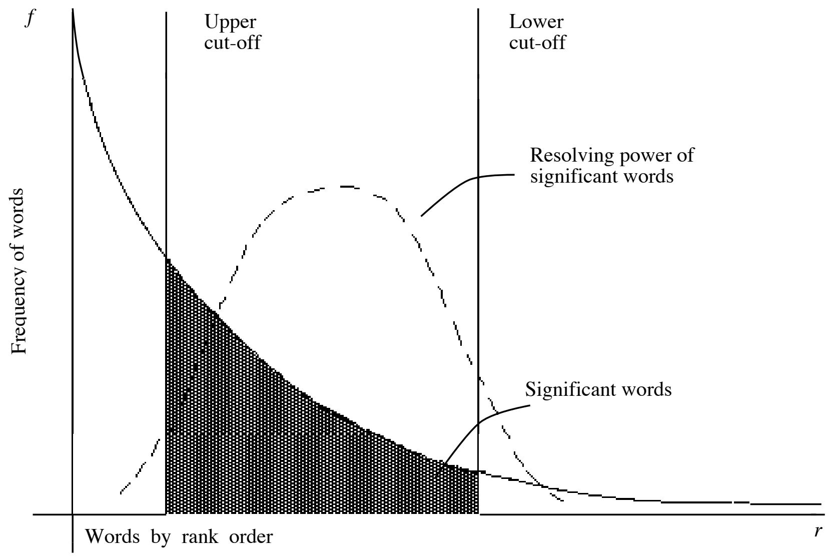


Figure 3.2: Frequency of words and assumed importance. Source [van Rijsbergen, 1979, Chapter 2].

problem for IR systems as a query from the user could contains a synonym to a term actually used in the system. Simple keyword based systems will then be unable to detect the connection and the searched word will not be found. Supporting various inflections of the same word can to some extent be solved using stemming, see below. But stemming does not solve the problem of different words not looking like each other but sharing the same meaning.

Polysemy

When the same word has more than one meaning, problems also arise. An example is the various meanings of the word "bank" – it is very dependent on the context if the financial meaning, a section of computer memory, an airplane maneuver, or even a steep slope is involved. Other common examples include "play" and "table". The multiple meaning of a single word is traditionally called polysemy in IR. Because IR systems usually have no understanding of the meaning of words, or appreciation for the different contexts that signify the meaning of a given word, they are unable to distinguish between the meanings and find the correct one in a given context. This can lead to false positives as the word being sought has a different intended meaning.

Stop-words

Whenever text is automatically indexed, some words occur very frequently. Common examples include "and", "me", "she", "that", "the", etc. These words are not good at discerning the meaning of the document, and they are commonly removed in IR systems. A more complete example of such a list of stop-words can be found at the Snowball homepage[1] – Snowball is a general language for stemming algorithms, see also the next section.

Stop-word removal is usually performed before stemming, if that is used, as undesirable removal of words would occur otherwise. However, the inclusion of stop-words in the set of terms can prove beneficial in some cases as it gives the system a bit more data to work on, especially with very short descriptions.

Stemming

Stemming is a method used to remove word inflections, in order to reduce the number of different terms for the same base word or word stem. The basic example is the removal of the plural "s" from the end of all words, which of course is too simple an algorithm to give any benefits in practice. The Porter Stemmer is an example of a more advanced and widely used algorithm [Porter, 1980]. It performs a number of passes over a word and uses tables of common inflections. The best matching inflection is then removed using a carefully constructed set of rules. Stemming is, of course, completely language dependent, so adding support for an additional language in an IR system employing stemming would also mean adding another stemmer.

Stemming is not always used in IR systems. Given a large enough text corpus, the system should be able to detect that the terms are closely related, despite the different inflections. This way, stemming does not have to be performed, and the imprecision it creates can be avoided. Latent Semantic Indexing is actually able to perform synonym identification, and therefore also associating different inflections without using the method of stemming, but more on this in section 3.3. For now it suffices to note that studies have actually shown minimal impact of stemming on Latent Semantic Indexing performance, and in some cases it has actually been found to decreased retrieval performance [Dumais, 1992, p. 5].

The fact that stemming is also an approximate approach, meaning it can never be fully able to stem all words without error, is also worth noting. An example demonstrating unfortunate behavior of a stemmer is the stemming of "business" to "busy". This is not what you would expect in a bank

[1]`http://snowball.tartarus.org/english/stop.txt`

application, for instance. Another example from the Porter stemmer is that both "console" and "consoles" stem to "consol". It would be unreasonable to expect the stemmer to know such special cases, but it underlines the subtle errors stemming can introduce.

3.2.3 Term weighting

The value of the entries in a DVM may be decided upon using any appropriate scheme. It is of course simplest to keep the occurrence count of the word in the given document. But more often some sort of term weighting is performed to reduce the differences, and add importance to the rare terms. This is generally a good way of improving retrieval performance. As [Manning and Schütze, 1999, p. 542] describes, an increase in occurrence-count from 1 to 2 should increase the importance but not actually double it. The intuition behind this is that going from 0 to 1 in occurrence count introduces a new word for a document. But going from 1 to 2 in count should not double the importance of that word.

A weighting transformation usually has two components. On component is the *global weight*, which indicates the overall importance of a given term, denoted $G(i)$ in the following. The other component is the *local weight* which used to modify the frequency of the term within a single document, denoted $L(i, j)$. This means that global weights work on a single row in the DVM representing a term, whereas the local weights is calculated for each column, representing a document. Combining local and global weighting functions is discussed in "*Combining weighting schemes*", on page 23.

According to [Dumais, 1992], usable local weights include *term frequency*, *binary weight*, and *logarithmic weight*. These are introduced below. For the global weights, the article mentions *normalization*, global frequency inverse document frequency, inverse document frequency, and the *entropy* or noise weighting. These are discussed afterwards. Finally, combining local and global weighting is discussed. Refer to table 3.1 on the following page for the symbols used in the weighting calculations.

Local weighting

In this section three different local weighting schemes are introduced. First comes term frequency, then binary, and finally the logarithmic local weighting. Remember that local weighting is applied to each document in the DVM, meaning a column in the matrix. $L(i, j)$ is used to denote the local weighting of term i in document j.

d	=	number of documents
t	=	number of terms
tf_{ij}	=	term frequency, count of term i in document $j, \geq 0$
gf_i	=	global frequency, count of term i occuring in all documents, ≥ 1
df_i	=	document frequency, number of documents the term occurs in, ≥ 1

Table 3.1: Symbols used in weighting methods.

Term frequency Term frequency is simply a count of how many times the word occurs in the document. A more precise description is:

$$L(i,j) = tf_{ij} \tag{3.1}$$

Term frequency gives an image of how related a term and a document are, but it is does not properly weight frequent and non-frequent terms against each other due to the misrepresentation when using raw occurrence counts as mentioned in section 3.2.3 on the page before.

Binary weighting Binary weighting is very simple, and it can be described by:

$$L(i,j) = \begin{cases} 1 & \text{if } tf_{ij} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

It is an extremely simple way to represent the relationship between a given term and a document. It is the opposite extreme of the raw occurrence count, but it does not record any information about how strongly related documents and terms are. Binary weighting is too simple as it retains too little information, and it is rarely used in practice.

Logarithmic weighting Finally, the logarithmic weighting is constructed by calculating:

$$L(i,j) = log_2(tf_{ij} + 1) \tag{3.3}$$

Taking the logarithm greatly reduces large differences in term frequencies. This means that documents having a great number of occurrences of a given term are not too different from documents having a smaller number. Still, a lot more information is retained than in the binary weighting but without putting too much emphasis on plain occurrence counts (tf_{ij}).

This section introduced three different local weighting schemes. The term frequency and binary weightings are too simple to be used in practice, and the logarithmic weighting is what is actually often used, and it is also used in this implementation.

Global weightings

This section introduces four different global weightings. They each give emphasis to different things, but the last one introduced, the entropy weighting, is the most successful one, according to [Dumais, 1992]. It is a common assumption in IR that if a word occurs in almost all documents, it is less suitable to distinguish between the documents and therefore it should have less weight. In the following, $G(i)$ will denote the value of the global weighting for a given term i.

Normalization If only performing the basic *normalization*, it ensures that the length of each row-vector in the DVM is 1. This is done by calculating the length of the row-vector in Euclidean space, and use the inverse of that value for the global weighting:

$$G(i) = \frac{1}{\sqrt{\sum_j (tf_{ij})^2}} \tag{3.4}$$

It gives higher values to the global weight when gf_i is low, which means that it promotes rare terms. But when using the normalization weighing scheme, only the term frequencies are used. The distribution of those frequencies is discarded and this is the reason why basic length normalization is not such a great idea. For example, a term occurring once in 100 documents is given the same global weight ($G(i) = 1/10$) as a term that occurs ten times in total, but only in a single document.

Document frequency based The global frequency inverse document frequency and the inverse document frequency global weightings are closely related. Both are shown in equation (3.5),

$$G(i) = \frac{gf_i}{df_i} \qquad \text{and} \qquad G(i) = log_2\left(\frac{d}{df_i}\right) + 1 \tag{3.5}$$

These weightings take into account the number of different documents a given term occurs in (df_i). The special thing about the global frequency inverse document frequency (GFIDF) is that it increases the weight of frequently occurring terms. However, both do not take into account the distribution of terms in documents, so only the number of documents containing a term is reflected. This means that if a term occurs ten times in two different documents (giving $gf_i = 20, df_i = 2$), the global assigned weight will be the same as a collection where a term occurs one time in one document and 19 times in one other document. This clearly shows that these weighting methods ignore the distribution of the occurrences, and only take the number of different documents a term is used in into account.

Entropy weighting Finally, we come to the most advanced weighting scheme tested in the [Dumais, 1992] article. It is an *entropy* based one, which uses information theoretic ideas. It calculates the average uncertainty of a term:

$$G(i) = 1 + \frac{\sum_{j=1}^{d} p_{ij} log_2(p_{ij})}{log_2(d)}, \qquad \text{where } p_{ij} = \frac{tf_{ij}}{gf_i} \tag{3.6}$$

This weighting function gives highest weight for terms used in few documents, and lowest weight to terms used in most documents.

Equation (3.6) is slightly changed based on the description in [Zhao and Grosky, 2002], since the original version in [Dumais, 1992] does not work properly. The original started with $1-$ instead of $1+$ as in equation (3.6). Figure 3.3 on the facing page shows that when a term occurs once in all documents out of six possible, the Dumais entropy assigns the weight of 2 to it, whereas the corrected version assigns 0. Using equation (3.6), minimum weights are given to terms equally distributed over all documents, just like it is described in [Dumais, 1992, p. 9]. The maximum weight is given to terms occurring in few documents, as in figure 3.3 on the next page, whereas a term that occurs few times in once in the entire collection should be maximally interesting, so it gets global weight nearer to 1.

This section introduced four different global weighting schemes. The goal with these weighting schemes is to weigh the occurrence counts in a way that better reflects the importance of the individual terms. Normalization gives highest weight to terms occurring few times, but it disguises the distribution of the occurrences. The GFIDF and inverse document frequency (IDF) weightings also do not take the distribution across documents into account, and this is undesirable. The entropy weighting scheme takes the distribution of a term across all documents into account, and is the only scheme discussed here which does that. Term weighing schemes are also discussed in [Salton, 1989].

gf_i	fraction	Dumais entropy	Zhao entropy
1	0.00	1.00	1.00
2	−0.37	1.37	0.63
3	−0.61	1.61	0.39
4	−0.77	1.77	0.23
5	−0.90	1.90	0.10
6	−0.10	2.00	0.00

Figure 3.3: Examples of entropy values. The assumption in these calculations is that each term has $tf_{ij} = 1$ in each place it occurs out of six possible.

Combining weighting schemes

This section discusses how local and global weighting schemes are combined to give the final weighting, and result in the processed DVM. The combination of a local weight with a global weight is simply performed by multiplying the two numbers. Let i be the index of a given term, and j the index of a given document. Then resulting DVM X of the combined weight becomes:

$$[x_{ij}] = L(i, j) \cdot G(i) \tag{3.7}$$

for each entry $[x_{ij}]$ of X.

It is common practice to use the term frequency inverse document frequency (TFIDF) normalization, which is the local term frequency (tf_{ij}) combined with the IDF. This normalization also looks at how many documents a given term is used in, and the more different documents it is used in, the less suitable it is to discriminate between documents. TFIDF is calculated by:

$$[x_{ij}] = tf_{ij} \cdot log_2 \left(\frac{d}{df_i} \right) \tag{3.8}$$

As can be seen, when a term occurs in all documents, meaning $df_{ij} = d$, the value of $[x_{ij}]$ will be 0.

In [Dumais, 1992, p. 10], they document that the combination of the logarithmic local weighting with the global entropy weighting performs best in their experiments with Latent Semantic Indexing, introduced in section 3.3 on the following page. In fact, the performance of the system is significantly higher than with any of the other combinations they tried in their experiments, and it is this combination which is used in this project.

This section described how local and global weightings are combined and cited [Dumais, 1992] for saying that the combination of local logarithmic weighting with global entropy weighting performed best in a set of experiments.

3.3 Latent Semantic Indexing

This section introduces Latent Semantic Indexing (LSI) by first giving a few examples of its use elsewhere, followed by an introduction to the Singular Value Decomposition (SVD). A discussion of choosing the appropriate richness of representation then follows, and finally, an example is given in section 3.3.3.

LSI is an approach to IR first introduced in 1990. See [Deerwester et al., 1990] for the original paper and [Landauer et al., 1998] for a more recent paper. It is a fairly simple extension to the VSM which has given remarkable results. The LSI method is covered by a patent owned by Telcordia for commercial use[2]. LSI has been used on a large number of different datasets, and the results compared to previous methods have been impressive. LSI is a specific form of Latent Semantic Analysis (LSA), and the name LSI is used when LSA is applied to IR. LSA is the general dimensionality reduction.

As an example of the many applications of LSI, it has been used to grade student papers automatically [Foltz et al., 1996]. It has also been used for cross-language retrieval of documents where a query is entered in English, and both English and French documents about that subject are returned [Dumais et al., 1996]. It has even shown performance comparable to an average person who takes a test known as the Test of English as a Foreign Language (TOEFL).

These results are achieved without using any kind of dictionary or thesaurus. By simply presenting a lot of texts to the system, it is possible to find patterns in term co-occurrences and thereby extract the “latent” semantic meanings of the words. The more text the system is exposed to, the better understanding of the use of words it shows. Given the approximate amount of text a high-school student has been exposed to, similar word understanding appears [Landauer et al., 1998].

The LSI is performed using a method known as Singular Value Decomposition, introduced in the next section. An example of the method in action can be found in section 3.3.3.

[2]U.S. patent No. 4,839,853

3.3.1 Singular Value Decomposition

Any matrix can be decomposed into three other matrices using a decomposition called SVD. It is described in some detail at the Wolfram Mathworld web-site[3]. Given a rectangular matrix X, three other matrices T_0, S_0, and D_0 are calculated, such that

$$\underset{t\times d}{X} = \underset{t\times m}{T_0}\ \underset{m\times m}{S_0}\ \underset{m\times d}{D_0^T} \tag{3.9}$$

where the matrix S_0 is a diagonal matrix of the singular values of X with entries sorted in non-increasing order. Matrices T_0 and D_0 have *orthonormal* columns. The matrix D_0 has to be transposed, written D_0^T, to allow the multiplication of the three matrices. The dimensionality of the matrices can also be seen in equation (3.9), where m is the rank of X, $m \leq min(t, d)$, t is still the number of terms, and d is the number of documents in the DVM being subjected to the decomposition.

The fact that columns of T_0 and D_0 are *orthonormal* means that they are both *unit vectors* (length 1) and *orthogonal* (perpendicular to each other). T_0 and D_0 contain the left and right singular vectors of X respectively. Since they are ortonormal, we know that

$$T_0{}^T T_0 = D_0{}^T D_0 = I \tag{3.10}$$

which will be used in section 3.4.1. The decomposition is unique except for sign-permutations in T_0 and D_0. Refer to the example in section 3.3.3 for an actual example of the SVD.

When working with LSI, the columns of the matrix T_0 is known as the term vectors. Similarly, the columns of matrix D_0 is known as the document vectors.

3.3.2 Dimensionality reduction

In order to reduce the noise in the DVM, only the $k \ll m$ largest values of S_0 are retained and the rest are set to zero. Call the resulting matrix S. The columns in T_0 corresponding to the removed entries in S_0 are removed, and the resulting matrix is called T. The similar operation is performed to get D from D_0. The reason for making T and D is that when multiplying the three matrices together, these rows and columns would not contribute anything to the result. Then

$$\underset{t\times d}{X} \approx \underset{t\times d}{\hat{X}} = \underset{t\times k}{T}\ \underset{k\times k}{S}\ \underset{k\times d}{D^T} \tag{3.11}$$

[3]`http://mathworld.wolfram.com/SingularValueDecomposition.html`

and $\hat{X}$ is the least square error approximation of the original matrix with the given number of dimensions, due to the special properties of the SVD. This means that it is the best approximation to X given that only k dimensions are allowed to represent it. This fact is used in LSI to give a simplified representation of the original DVM. Similar terms are collapsed, and the usage of words together enables the system to associate synonyms and extract the latent concepts.

This process is further described in [Deerwester et al., 1990], and some additional details are given below.

Choosing dimensionality

Selecting the optimal k for the reduction of given matrix is an open research problem [Berry et al., 1994]. Normally around $100 - 350$ dimensions are retained – this has in many contexts turned out to be a suitable number to properly represent the inherent concepts in texts in many contexts. This means that collections with tens of thousands of documents and thousands of terms can be represented using only a few hundred dimensions. By reducing the number of dimensions, information is lost, and by retaining just enough, it is possible to get rid of a lot of noise in the data and still be able to properly identify synonyms. This is where LSI is able to generalize over the data and discover the hidden or latent concepts. The more diverse the text collection is, the more dimensions are needed to properly capture the various topics.

In [Dumais, 1992], the problem of choosing a value for k appropriate to the document collection is discussed. For relatively homogeneous collections, 100 dimensions seems to be sufficient. Increasing k above this number actually decreases the precision of the same queries due to the noise it introduces. To get the best performance it is important to select a proper k for a given collection.

Since the choice of k has such a great impact on performance, the following will elaborate on the selection of k in more mathematical terms. To do this, we need to calculate the differences between matrices. These ideas come from [Berry et al., 1999]. The Frobenius norm of a matrix A with entries a_{ij} is defined in equation (3.12):

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2} \tag{3.12}$$

Using this norm, the distance between X and $\hat{X}$ can be calculated using Euclidean distance (L_2), and by looking at equation (3.14) on the next page, it can be seen that the value $X - \hat{X}$ is related to the singular values of X which are removed in $\hat{X}$. It is then easy to calculate the percentage of lost information thus:

$$\frac{\|X - \hat{X}_k\|_F}{\|X\|_F} \tag{3.13}$$

This way one has a measurement of how much of the original information is retained. Equation (3.14) is taken from [Berry et al., 1999, p. 348]. It says that the approximation of A_k to A is the best achievable with regards to the L_2 norm.

$$\|A - A_k\|_F = \min_{\text{rank}(Y) \leq k} \|A - Y\|_F = \sqrt{\sigma_{k+1}^2 + \cdots + \sigma_{r_A}^2} \tag{3.14}$$

Another approach is detailed in [Zha, 1998] where the minimal description length (MDL) is used to calculate the optimal dimensionality k for the DVM. Zha compares the values calculated by the model with the values found by performing a set of queries on the dataset, and calculating the precision. The values for optimal dimensionality correlates nicely for the example in the paper. A fact which is pointed out in the article is that a small range of k is equally suitable as far as the precision is concerned – it does not have to be the perfect k value which is used. However, this paper is not widely cited elsewhere, and it has not been possible to reproduce the results by implementing the formula described in the paper. This could be due to the fact that only the truncated SVD has been performed and all the singular values were therefore not available to perform the MDL calculation.

3.3.3 SVD example

Continuing the example from section 3.2.1, we now perform the SVD on X from figure 3.1. By performing the SVD on the matrix X, we get the three matrices T_0, S_0, and D_0: see figures 3.4, 3.5, and 3.6, respectively. As can be seen in figure 3.5 on the next page, the singular values are in fact sorted in decreasing order along the diagonal.

We now reduce the matrices in order to find the latent concepts. Choosing $k = 2$, we form the matrices T, S, and D by only keeping the gray areas of each of the depicted matrices. Only retaining two dimension is extremely harsh, but it illustrates the procedure nicely, and it does give some nice results in this small example.

Forming the matrix $\hat{X}$ is done by calculating TSD^T from the reduced representation – the result can be seen in figure 3.7 on the following page. By comparing this to the original matrix X in figure 3.1 on page 15, some

$$T_0 = \left[\begin{array}{rr|rrr} 0.44 & 0.13 & 0.48 & 0.70 & 0.26 \\ 0.30 & 0.33 & 0.51 & -0.35 & -0.65 \\ -0.57 & 0.59 & 0.37 & -0.15 & 0.41 \\ 0.58 & 0.00 & 0.00 & -0.58 & 0.58 \\ 0.25 & 0.73 & -0.61 & 0.16 & -0.09 \end{array}\right] \tag{3.15}$$

Figure 3.4: The example T_0 matrix resulting from SVD being performed on X from figure 3.1 on page 15.

$$S_0 = \left[\begin{array}{rr|rrr} 2.16 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 1.59 & 0.00 & 0.00 & 0.00 \\ \hline 0.00 & 0.00 & 1.28 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 1.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.39 \end{array}\right] \tag{3.16}$$

Figure 3.5: The example S_0 matrix. The diagonal contains the singular values of X.

$$D_0 = \left[\begin{array}{rr|rrr} 0.75 & 0.29 & -0.28 & -0.00 & -0.53 \\ 0.28 & 0.53 & 0.75 & 0.00 & 0.29 \\ 0.20 & 0.19 & -0.45 & 0.58 & 0.63 \\ 0.45 & -0.63 & 0.20 & -0.00 & 0.19 \\ 0.33 & -0.22 & -0.12 & -0.58 & 0.41 \\ 0.12 & -0.41 & 0.33 & 0.58 & -0.22 \end{array}\right] \tag{3.17}$$

Figure 3.6: The example D_0 matrix, and the shaded part is the D matrix.

$$\hat{X} = \left[\begin{array}{l|rrrrrr} & d_1 & d_2 & d_3 & d_4 & d_5 & d_6 \\ \hline \text{cosmonout} & 0.85 & 0.52 & 0.28 & 0.13 & 0.21 & -0.08 \\ \text{astronaut} & 0.36 & 0.36 & 0.16 & -0.21 & -0.03 & -0.18 \\ \text{moon} & 1.00 & 0.71 & 0.36 & -0.05 & 0.16 & -0.21 \\ \text{car} & 0.98 & 0.13 & 0.21 & 1.03 & 0.62 & 0.41 \\ \text{truck} & 0.13 & -0.39 & -0.08 & 0.90 & 0.41 & 0.49 \end{array}\right] \tag{3.18}$$

Figure 3.7: Reduced matrix $\hat{X} = TSD^T$, the result of setting $k = 2$.

interesting effects can be observed. We start the comparison by noting that the negative entries in the reduced DVM correspond to terms that have nothing to do with the representation for that document.

Comparing the documents it can be seen that the three documents d_4, d_5, and d_6 now have a lot more in common. The actual use of terms in d_5 and d_6 is now replaced with an approximation which indicates that the other term should have been represented. Taking d_3, the original document only contained the term `cosmonaut`, whereas in $\hat{X}$ the term `moon` actually carries more weight than the original one. The reduction of `astronaut` from 1.00 to 0.36 in document d_2 shows that this term was somewhat unexpected in this place.

Further calculations using this example will be shown section 3.4.2, where the document-to-document similarity is demonstrated.

3.4 Searching in the DVM

Having made a DVM and reduced the dimensions using LSI, we are now ready to perform searches.

A query is given as a string of words. It is then put through the same treatment as the original pieces of documentation – this is further detailed in the section "*Handling identifiers*" on page 64. For now it suffices to say that the terms are extracted from the string by initially splitting it at white-spaces and special characters. A term vector X_q is then created of length t, where t is the number of terms recognized by the system. The indices for the recognized terms are then found, and any terms not used in the document collection are discarded. The same normalization should be performed on the query vector as was done to the original DVM – this way the weighting of the query is comparable to the indexed documents.

In [Deerwester et al., 1990], the pseudo-document representing the query is constructed by taking the term-vector X_q and deriving a representation D_q such that it can be used as a row of D in the comparison defined in equation (3.21) on page 32. First note that the inverse of S, called S^{-1}, is easy to find. Because S is a square diagonal matrix, the inverse is found simply by creating the matrix with diagonal entries $\frac{1}{s_{ij}}$. Then it is easy to see that $SS^{-1} = I$, and it is in fact the inverse matrix.

The calculation of the query document vector is then done by calculating:

$$D_q = X_q^T T S^{-1} \tag{3.19}$$

The calculated query D_q can then be compared to the document vectors in the matrix D.

3.4.1 Matching

When it comes to actually match the query against the set of documents, a similarity measurement is used to compare the new pseudo-document and each of the existing documents in $\hat{X}$. A cutoff similarity can be chosen, say 0.8 for some measurement or the 100 best matching in order to limit the number of results, since all indexed documents will have some degree of similarity using most similarity measurements.

There exists many different similarity measurements, and in [Jones and Furnas, 1987], they develop a method for comparing different similarity measurements geometrically. They find that many of the similarity measurements behave erratic and counterintuitively. The method for comparing similarity measurements can be used in general to characterize other measurements they did not examine.

There exists numerous variations of the few similarity measurements listed here – but often they are introduced with only loose argumentation. Some of the more well known are summarized below. See [Salton, 1989] for details on some of these methods, and [Jones and Furnas, 1987] for a thorough comparison.

Inner product is where the inner product of the query document vector and each of the document vectors in the collection is calculated. This is very easy to calculate, but the measurement is subject to a number of problems. It is overly sensitive to changes within a single vector component, and similarity values can be unbounded due to the value of a single component. This means that the strongest entry in the vectors may decide most of the similarity value.

Cosine coefficient is produced simply by calculating the angle between the query document vector and each of the documents in the collection. If both of the two vectors has length 1, then this value is the same as the inner product. As this measurement normalizes the vectors, it ignores difference in length between the query and documents. It is not victim of unbounded influence of a single entry like the inner product, but it does ignore differences in length. This means that richly indexed items are not ranked higher due to their representation.

Pseudo-cosine measure uses the L_1 or city-block distance measurement, instead of the L_2 or Euclidean used in the cosine. This measurement is dominated by axis closest to the query, and gives some strange behavior.

Other measurements tested in [Jones and Furnas, 1987] include Dice, Overlap, and Spreading activation. These will not be covered here, but based on the arguments in the article, the cosine similarity will be used in this project.

Different comparisons

The original DVM can be effectively represented as a sparse matrix because the only non-zero entries are the term frequency entries, and usually a DVM has non-zero entries in less than 1% of the entire matrix [Berry et al., 1999]. But the product of TSD^T will usually contain non-zero values most cells, and therefore take up quite a lot of memory, see figure 3.7 on page 28 for an example. The three matrices T, S, and D can be stored separately and used for calculations using much less memory than the resulting complete $\hat{X}$. This is the reason for the matrix manipulations that follow next. They are done in order to avoid having to calculate the complete matrix $\hat{X}$ and keep the result in memory.

As [Deerwester et al., 1990, p. 398] points out, three different comparisons can be calculated using the three matrices T, S, and D resulting from the SVD. The similarity between terms, between two documents, and between a single term and a document can be calculated without actually forming the $\hat{X}$ matrix. This can be used to find out which terms are relevant for a document, even though they may not actually occur in the document.

Term comparison In the original matrix X, the similarity between two terms can be calculated by comparing two rows using the similarity measurement, for instance inner product. In the reduced matrix $\hat{X}$, the inner product of two row vectors from $\hat{X}$ can also be used to denote how closely related the two terms are. Calculating the similarity for all terms can be done by forming $\hat{X}\hat{X}^T$ – the entries are then the term-to-term dot product similarity. But the same results can be found more economically by using the fact that:

$$\hat{X}\hat{X}^T = TSD^T(TSD^T)^T = TSD^TDS^TT^T = TS^2T^T \tag{3.20}$$

This matrix then contains all term-to-term similarity measurements. The calculation uses equation (3.10) on page 25 and the fact that S is a diagonal matrix.

Comparing two documents This is similar to the term-to-term comparison above, except for the fact that the comparisons are between columns, not rows. The corresponding calculation becomes, with reductions performed similarly to equation (3.20) on the page before:

$$\hat{X}^T\hat{X} = DS^2D^T \tag{3.21}$$

Comparing a term and a document As described in [Deerwester et al., 1990, p. 399], this is a bit different in that this comparison is the individual cells in $\hat{X}$. To calculate these quantities without actually forming $\hat{X}$, the following trick is used calculate cell $[\hat{x}_{ij}]$ of $\hat{X}$. Remember from equation (3.11) that

$$\hat{X} = TSD^T \tag{3.22}$$

Taking the dot product between row i of $TS^{1/2}$ and row j of $DS^{1/2}$ gives in fact cell $[\hat{x}_{ij}]$.

Comparing a query to documents Having constructed a query document as decribed in section 3.4, the chosen comparison operator is used to compare the query document to each of the documents in the collection. The output is a list of similarities – if the cosine similarity measurement is used, then it is the angle between the query document and each of the documents. This list of comparison results then has to be sorted in numerical decreasing order to find the most relevant documents. The cosine is suitable for ranking documents, but not as an absolute similarity measurement. The value of the cosine similarity depends greatly on the chosen value for k, see [Berry et al., 1994, p. 11]. This also means that choosing a cut-off value for the cosine measurement is very dependent on the chosen k.

3.4.2 Examples of comparisons

As we now know how to compare terms, documents and queries to documents, the following will demonstrate the use by expanding on the example in section 3.3.3. The original X from figure 3.1 on page 15 and the reduced matrix $\hat{X}$ seen in figure 3.7 on page 28 will be matched against a query. This will demonstrate how reducing the DVM affects retrieval.

Forming the query "`moon astronaut`" results in the query document vectors listed in figure 3.8 on the facing page. Equation (3.19) on page 29 has the elements in the calculation. The raw query document is X_q, which is the one used in the comparison with X. For the $\hat{X}$ comparison, the query vector D_q calculated is used instead. Comparing these query representations using the cosine similarity against the two matrices will result in the similarities listed in table 3.2 on the facing page.

$$\begin{aligned} X_q &= \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \end{bmatrix} \\ D_q &= \begin{bmatrix} 0.28 & 0.53 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}^T \begin{bmatrix} 0.44 & 0.13 \\ 0.30 & 0.33 \\ -0.57 & 0.59 \\ 0.58 & 0.00 \\ 0.25 & 0.73 \end{bmatrix} \begin{bmatrix} 2.16 & 0.00 \\ 0.00 & 1.59 \end{bmatrix}^{-1} \end{aligned}$$

Figure 3.8: Forming X_q and performing the calculations leading to the vector for the query document, D_q.

	d_1	d_2	d_3	d_4	d_5	d_6
X	0.41	1.00	0.00	0.00	0.00	0.00
$\hat{X}$	0.75	1.00	0.94	-0.45	-0.11	-0.71

Table 3.2: Similarity between query document and original documents.

As can be seen from this simple example, the similarity between the query and the documents better reflects the topics of the different documents in $\hat{X}$. Even though d_3 does not contain any of the search terms, it still matches the query to an impressive degree. This demonstrates the associative power of the LSI. The three last documents are a lot less related to the query in that they point in the opposite direction, which makes good sense.

3.5 Clustering

This section covers the topic of clustering. In general terms, clustering is the process of organizing a set of elements into a number of groups or clusters. These clusters should reflect a similarity between the contained elements. A cluster in this context is a set of documents which in some way are similar to each other. Clusters can be created in many ways and numerous algorithms and approximations exists to solve this problem. Below the different approaches will be roughly categorized. [Manning and Schütze, 1999] introduces many of the concepts and gives further examples.

The first choice is whether to produce a flat set of clusters or a hierarchical one. Algorithms for *flat clusters* are fairly simple and run very fast. In many contexts they give sufficiently good results to be usable. Read more details about flat clusters in section 3.5.1. *Hierarchical clusters* are more rich in their representation of the underlying structure, and they can be presented in interesting ways. They are further discussed in section 3.5.2.

Besides choosing the type of clusters, another important point is that of membership. The membership of a document to a cluster can either be hard or soft. *Hard membership* means that each document is only member of a single cluster. *Soft membership* means that the document can be a member of one or more clusters with different degrees of membership to each. As [Manning and Schütze, 1999, p. 499] notes, soft clustering is more suitable when working with natural language. This is because words can have multiple meanings and there can similarly be different aspects of a single document. Using hard clustering seems too restrictive in this context.

The reason for clustering in the first place is to enable easier data analysis. It is an intuitive way to illustrate the strengths and weaknesses of the neighboring relation. Furthermore, the ability to visualize data makes it easier to digest the information for the end-user. Note that the clusters are created from the actual data and they are not predetermined. They depend on the natural divisions in the data given by a neighboring relation.

3.5.1 Flat clustering

Usually flat clustering starts with a chosen number of clusters. All documents are then assigned to clusters randomly, and an iterative method is then used to gradually refine the clusters. At each step a document can be moved from one cluster to another in order to reduce the distance within each cluster.

Single-link clustering An example of a flat clustering algorithm is the *single-link* clustering. Here a number of clusters are initially formed in a bottom-up fashion. The similarity between two clusters is taken to be the similarity between the two closest objects from the clusters. The two closest clusters are found and merged in each iteration. Initially this sounds like a good idea, but it suffers from the so-called *chaining effect*. This is because clusters tend to get elongated, since the largest similarity is taken without taking the global context into account at each step.

Minimum Spanning Tree A different approach which does not suffer from this problem is clustering based on the minimum spanning tree (MST). Forming the MST is done by connecting all documents along the edges with the largest similarity. There are many efficient algorithms to find this tree, for instance Kruskal's Algorithm. Using such an algorithm makes it possible to create clusters taking the global cluster quality into account. This means that the similarity between two clusters is the similarity between their most dissimilar members – and this avoids the elongated clusters of the single-link method.

k-Nearest Neighbors clustering The k-Nearest Neighbors (kNN) clustering is a hard clustering method – each object only belongs to a single cluster. A *center of mass* is calculated for each cluster, which is an average of the contained objects. Several iterations follow where each object is reassigned to the cluster with the nearest center of mass. The centers are then updated again and the iteration can continue until the clustering converges.

3.5.2 Hierarchical clustering

Hierarchical clustering algorithms are usually greedy. Clusters are either formed top-down or bottom-up. Agglomerative Hierarchical Clustering (AHC) starts with a cluster for each individual object. In each iterative step, these clusters are then compared and the two most similar clusters are merged. The algorithm stops when one big top-level cluster containing all documents has been created.

Instead of forming the clusters bottom-up, the *divisive clustering* starts with a cluster containing all objects, and then splits this iteratively. The cluster to be split is the one which is the least *coherent* – a cluster with identical objects is much more coherent than one with object that are very dissimilar. The measurement of cohesiveness is one of the tuning parameters for this algorithm.

3.5.3 Online clustering

Online clustering is performed on a set of documents deemed relevant to a given query, instead of on the complete set of documents. These elements should then be clustered according to different aspects of the query. The algorithm used has to be fairly quick as the user is waiting for the response, and time has already been spent locating the results. These are the kinds of algorithms used in this project. For more details on the used algorithms, see section 5.3.1.

3.5.4 Offline clustering

When performing clustering offline, meaning that it is not relative to a given query but a general automatic clustering of all documents, there is no strict time-constraint. The idea is to perform the clustering once and for all, and then use it for recognizing the different groups when returning results.

Offline clustering can be used to find general grouping of documents and find different general topics in a collection of documents. In some IR systems it has actually been used to restrict the area of search when finding matches to a query – this way the search space becomes much smaller. This was especially important when computers had very little memory and processing power.

Self-Organizing Maps

This is a method created by Tuevo Kohonen. In his book [Kohonen, 1995] the mathematical details for Self-Organizing Maps (SOM), which is a visualization and datamapping algorithm, can be found. It also describes the Learning Vector Quantization (LVQ) method which is used for categorizing elements. Roughly speaking, SOM can be seen as an *elastic network* which approximates the high-dimensional document space. This means that the structure formed by documents in, for instance, hundreds or thousands of dimensions, are mapped into a grid of *cells* in few dimensions. This is done in a way that preserves the original spatial relationships as well as possible. By going all the way down to two dimensions it becomes possible to make a simple graphical representation of the SOM.

The process starts with a random solution which is then gradually refined to reach the final map. Several restarts have to be made in order to ensure a good solution has been found. The average quantization error is calculated and it is this measurement which is minimized during the iterations.

Within each iteration, a given document is compared to the current state of the network. The cells in the area around the document is then updated, according to parameters chosen by the user, to better reflect this document. This is repeated many times, and the documents can be presented in the same order each time or a random permutation. The order affects the final map, though not severely so [Kohonen, 1995, p. 112].

Applicability in this context The general problem with using an offline clustering method in this problem domain is the fact that the specific clustering is only interesting around the area of the query. This implies that the generated SOM has to be extremely detailed and contain an enormous amount of cells – this is not as such a deficiency in the SOM method, but because of the huge amounts of information which has to be clustered. In [Kohonen, 2000] they cluster a collection of almost seven million documents and use one million nodes in the map – the calculation took several weeks to complete. In the project they generated several levels of maps in order to facilitate navigation between different levels of detail, without recalculation. So handling large amounts of data in the generation of SOM is possible, but it costs a lot of resources.

SOM is well suited to create an overview of the different topics. Besides, the clusters will only be areas containing the artificial query document or having a border fairly close to it. It can be used to create an initial look on the different topics, which is what Self-Organizing Maps are often used for.

3.6 Conclusion

This chapter has introduced the various technologies that are going to be used in the rest of the text. We started by introducing general Information Retrieval and the Vector Space Model. Weighting schemes for transforming the original DVM was discussed, as was noise removal through dimension reduction. An example of a document vector matrix was given which was expanded when Latent Semantic Indexing was explained, and which was finally used to demonstrate searching. A section covered the different searches that are possible within the reduced DVM and finally a section discussed various approaches to clustering of data. This concludes the introduction of technologies. The next chapter will cover how others have solved the same problems as this project.

Chapter 4

Alternative approaches

In this chapter, alternative ways of solving the problem described in section 1.2 are presented. Three different areas of alternatives are discussed. These areas are other existing reuse support systems, other search methods, and finally how other user interfaces have been made.

The main focus in this chapter is on the CodeBroker system and an introduction to the Semantic Web with Description Logic. The CodeBroker system has been a major inspiration for this project, which is why it is described in such detail. As for the Semantic Web, it is a very interesting development in the way information is represented and exchanged.

The chapter is structured as follows: first comes a discussion of three reuse support systems in section 4.1 with CodeBroker as the main example. Then in section 4.2, other methods of Information Retrieval will be introduced. These are the keyword-based methods, the probabilistic methods, and at some greater length, the logic based methods. Within the discussion of the logic based methods, the Semantic Web project is described and key technologies are introduced. Finally, in section 4.3 we will take a look at different ways of making a user interface for a reuse repository.

4.1 Reuse support systems

Making and using reuse support systems is not a new idea. People have been constructing such systems based on fundamentally different methods for many years now. The following sections cover a few different systems suitable for helping with reuse when programming.

4.1.1 CodeBroker

The CodeBroker (CB) system was created by Yunwen Ye. It is documented in his Ph.D. thesis [Ye, 2001] titled "Supporting Component-Based Software Development with Active Component Repository Systems". He has also written some introductory material, such as [Ye, 2003]. Briefly, the idea is that since developers do not reuse existing code enough, it must be because they perceive the problems of reusing to be greater than the benefits. Ye then extends an existing context-aware system, to make it deliver components which may be relevant to the user's current task, without any explicit action on the user's part.

He calls this a paradigm shift from the *development-with-reuse* paradigm described in [Rada, 1995], to *reuse-within-development*. He then proceeds to integrate the delivery mechanism of components directly into the development tool. The Emacs editor with a special Java mode is used as the delivery platform, see figure 4.1 on the next page which is further described the section "*Determining current task*" on page 43. The system is set up for programming in Java, and delivers short descriptions of relevant components at the bottom of the screen. The source code is freely available[1] under the GNU Public License (GPL) license.

The problems with existing reuse support systems is the fact that the user does not think that components exist which can solve the given problem. As mentioned, users also have a tendency to perceive the cost of locating them to be greater than the cost of making a solution from scratch.

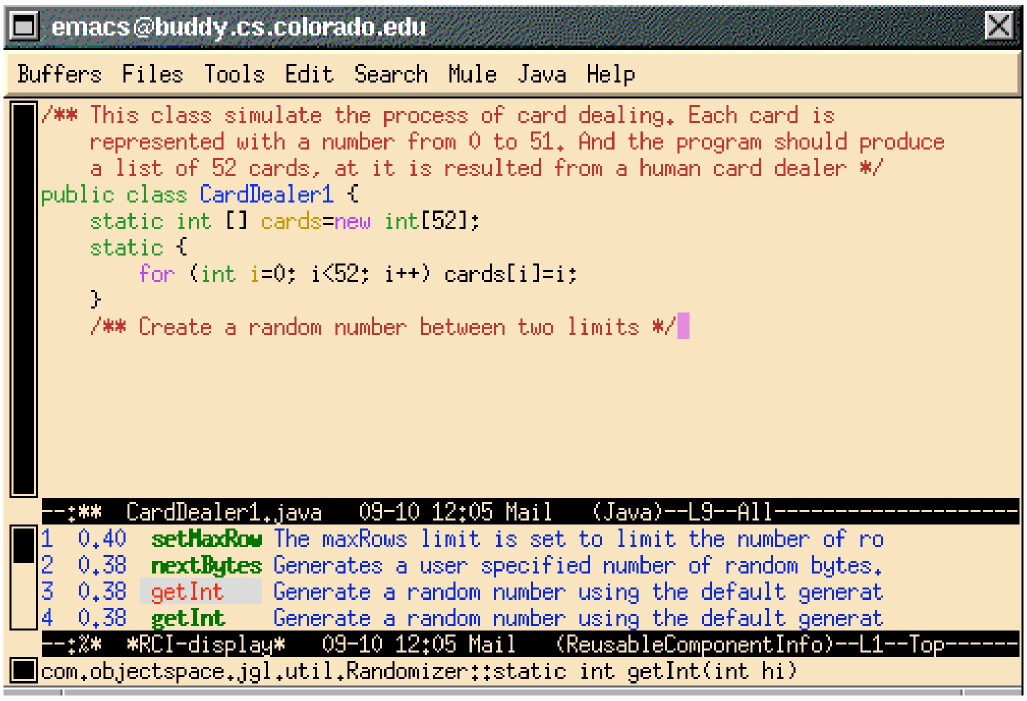
Below are described some of the different aspects of the CB system which are relevant to our discussion. The indexing performed is a bit different from the one used in this project, and it is described next. The user model is described in the section after, which also discusses the user-defined filtering of delivered components. How queries are performed is discussed at the end.

Indexing

Before the system can be used for retrievals, the raw data must be properly indexed. Here follows a description of the various parts of this procedure.

First of all, Javadoc HTML is parsed to extract the signature and description of the various methods. Two different collections of components are used. These are the Java API version 1.1.8 and Java General Library (JGL) version 3.0 from ObjectSpace Inc. These component collections contain 503 and 170 classes respectively and a total of 7.338 methods. These are chosen

[1] `http://www.cs.colorado.edu/~yunwen/codebroker`


emacs@buddy.cs.colorado.edu
Buffers Files Tools Edit Search Mule Java Help

```
/** This class simulate the process of card dealing. Each card is
    represented with a number from 0 to 51. And the program should produce
    a list of 52 cards, at it is resulted from a human card dealer */
public class CardDealer1 {
    static int [] cards=new int[52];
    static {
        for (int i=0; i<52; i++) cards[i]=i;
    }
    /** Create a random number between two limits */
--:**  CardDealer1.java   09-10 12:05 Mail    (Java)--L9--All------------------
1  0.40  setMaxRow The maxRows limit is set to limit the number of ro
2  0.38  nextBytes Generates a user specified number of random bytes.
3  0.38  getInt    Generate a random number using the default generat
4  0.38  getInt    Generate a random number using the default generat
--:%*  *RCI-display*   09-10 12:05 Mail    (ReusableComponentInfo)--L1--Top------
com.objectspace.jgl.util.Randomizer::static int getInt(int hi)
```

Figure 4.1: Example of CodeBroker in action. Source: [Ye, 2003].

because of the high level of quality and the fact that they are generally well documented. Using the system with other component collections is possible, however, is not easy to get the system running, as the source does not compile on this author's system, and no pre-compiled binary version is available.

Besides using the Javadoc as a documentation source, various other sources are used to give the system a certain level of "background knowledge". By introducing the contents of Manual (Man) pages and texts from the Java language specification, the system is better able to infer the relationships between words. This also helps the system further discovering synonymous words.

A document vector matrix (DVM) is created from the chosen documentation and two different retrieval methods are used. These are LSI and the Okapi model. For more information on the Okapi, see [Robertson et al., 1995] which has some information on the modern Okapi algorithms. These algorithms have been a major research subject since they were first introduced at Text REtrieval Conference (TREC) 1 in 1992. Ye has used the existing Okapi implementation and not reprogrammed it himself. Okapi is further discussed in section 4.2.2 on page 51.

In fact, most of the CB system is a simple extension of the Remembrance Agent[2], which is the foundation for the source of the system. The Remembrance Agent source includes the Okapi ranking, and it is a system for retrieving mails relevant to the current text. It works within Emacs, and has probably greatly influenced the choice of component delivery platform for the CB project. Other choices for freely available Okapi implementation is the Lemur project[3] or the Xapian[4] project.

The reasons for choosing to use the LSI and Okapi methods are detailed at [Ye, 2001, p. 83]. Briefly the reasons were that they required least effort to implement, they were easy to use for programmers, and that they were as effective as other more advanced methods. The fact that free-text methods are as effective as the more advanced ones is documented in [Frakes and Pole, 1994], and the fact that using a controlled vocabulary is expensive and does not give better results than the uncontrolled approach is documented in [Mili et al., 1997].

It is interesting to note that the Okapi methods consistently gives better performance than the application of the LSI method. The approach to using LSI falls outside what is usually done. The semantic space is created using various texts such as the Java Language Specification, various Man pages, and other similar sources of text. This is done to give the system a better basis for evaluating the words. But these are the only documents which are used in the initial DVM; the documents extracted from the Javadoc are then simply *folded in*, a method similar to how the query document is formed, see equation (3.19). When folding in documents, the performance may suffer greatly, as the new document are only approximately represented as an average of the constituent terms. With each new document, the representation of that document is not influenced by the other documents but only by the standard texts. This means that adding documents with few known words will result in all the unknown words to be ignored. Furthermore, Ye does nothing to recompute the weights of the different words. This could be an explanation why his LSI implementation performs consistently much worse than the Okapi one. But the fact that Okapi better handles polysemous words more accurately than LSI could also be an important factor.

Discourse model

In order to only return components the user is not already familiar with, a *discourse model* or *user model* is created. Based on previously written

[2]`http://www.remem.org`
[3]`http://www-2.cs.cmu.edu/~lemur`
[4]`http://www.xapian.org`

programs and a direct feedback mechanism, the user can filter out single methods, classes, or entire packages. The filtering can be for a single session, or it can be a permanent decision by the user. Interacting with the system to filter out unwanted results is done simply by right-clicking on the delivered component and choosing how much to exclude, and for how long.

By making this fairly simple model of the users knowledge, the system is able to hide the components which the user has already deemed irrelevant or knows about. The filtering is done by having the system retrieve everything relating to the current context, and only after the retrieval is done does the filtering take place. This means that the retrieval mechanism can be kept relatively simplistic, while still being able to personalize the presented results. This seems like a good compromise, as the user model can be expected to be much smaller than the set of all documents.

Determining current task

Instead of waiting for the user to search the repository, an agent process monitors the text the user has written in the active file. The discovery of what the task the user is currently working on is based on the comments entered in the unfinished code, and the current signature of the active expression. This means that comments have to be written before the actual method implementation, in order for the system to be truly useful. Refer again to figure 4.1 on page 41 to compare the screen-shot with the following discussion. The user writes his program in the large main area on the screen, and he or she is currently trying to shuffle a deck of cards, and has just finished the comment and some skeleton code. In the bottom area of the screen four delivered components can be seen. They each have a similarity ranking, a method name and a brief description. The user has hovered his or her mouse over the third component in the list, and can now see the full package and signature of the method at the bottom of the window.

Based on the entered comments, the agent retrieves relevant components at fixed intervals. It filters out the ones the user already knows using the process described previously. Finishing the comment delimiter in the example has triggered the agent to update the currently delivered components to reflect the updated task description. Actually, this is more or less a regular search engine embedded in the development environment. The main benefit to this approach is that the user can stay in his development tool when looking for reusable components, even though that development tool then has to be Emacs.

The CB system could get extra information about the current task by looking at the identifiers a programmer chooses for his methods and variables. But the CB system does not use the extra information contained in the identifiers to further specify the query. This means that programmers who are not writing the comments first will not get any of the benefits from the system, as it is unable to deliver components autonomously. This was a problem users noted when using the system.

Retrieval

Retrieval from the CB system can be performed in a variety of ways. The easiest way is to create proper comments for the method being written, as already described. The system uses the comment text as a query and presents relevant components based on the used words. The *signature* of the current method, meaning the return type and parameter types, is also taken into account in order to reduce the number of irrelevant components. This signature checking is rather elaborate in order to avoid returning unusable components. However, Ye's tests showed that the signature matching was more or less a futile effort, as very few components were filtered out this way. Finally, all components in the retrieved list the user is already familiar with are removed. These removed components are a part of the described user model.

The final list of retrieved components is then presented to the user at the bottom of the screen, as in figure 4.1. It consists of a primary list of one-line descriptions of the components. By holding the mouse over them, the signature of the given component is shown in the status area. Clicking it opens a browser with the JAVADOC documentation of the component, where more details can be found. It is also possible to make refinements to the retrieved list, as the user can add new components or packages to the discourse model by right-clicking on a delivered component and selecting the option from the menu.

Ye has created these multiple layers of information for a very specific reason. The process of choosing a component consists of two distinct stages. These are named *information discernment* and *detailed evaluation.* In the first, initial rough evaluation stage, the description of the different components is scanned by the user to locate components relevant to the current task. Only when a component seems promising is it evaluated more closely, in this case by opening the full documentation.

Learning the syntax and semantics of a given programming language is something all programmers have to do before being able to properly use it. An important detail is the fact that *vocabulary learning* has to take place before the programmer can properly reuse components. This means that the proper names for things have to be learned, and according to Ye, proper use of a component is best learned when it is used to solve a concrete problem. This view is a part of the reason why the delivery of components is integrated with the editor.

Evaluation results

In order to test the system Ye had a few real users try and use the system for a number of specific tasks. Generally, users like the delivery of components. However, the tasks used for testing the system were deceptively simple, such as shuffling a deck of cards or converting from one number format to another. As he describes it, the subjects are also more motivated to reuse when they are part of an experiment.

As users discovered, the system does not use the information contained in method and variable names [Ye, 2001, p. 143]. This is an obvious extension that could easily be made, enabling users who do not write many comments but use good identifiers to get a larger benefit from the system. Using identifiers is actually done in [Michail and Notkin, 1999], where they use identifiers to compare multiple software libraries to find similarities and differences, so the idea has previously been used elsewhere.

Final thoughts on CodeBroker

This section has described some aspects of the CodeBroker system. The system has been used as an inspiration for this project. The benefits seem real, and the described user feedback is mostly positive. Drawbacks include the dependency on a single development platform (Emacs), instead of being a more general system, and the fact that it is a research prototype, built with the Remembrance Agent. If the project had been started later, integration into the Eclipse[5] platform or similar IDE would have been an obvious alternative to Emacs, even though it would have demanded a larger programming effort on Ye's part.

4.1.2 Peel and CodeFinder

The Peel and CodeFinder project uses a different approach to build a searchable reuse repository [Henninger, 1997]. Instead of trying to make the best categorization and indexing initially and pay the full cost of the repository before it can be used, they make a usable version very cheaply using a semiautomated process, and then allow the users to update the descriptions as needed. The input is Lisp source code, and the tool called

[5]`http://www.eclipse.org`

PEEL is used to extract comments and strings from the source, removing a list of stop-words first. This text is then stored in a frame-based knowledge representation called KANDOR, which allows hierarchical structuring of information. The indexing is almost automatic, though some information is entered once for each file.

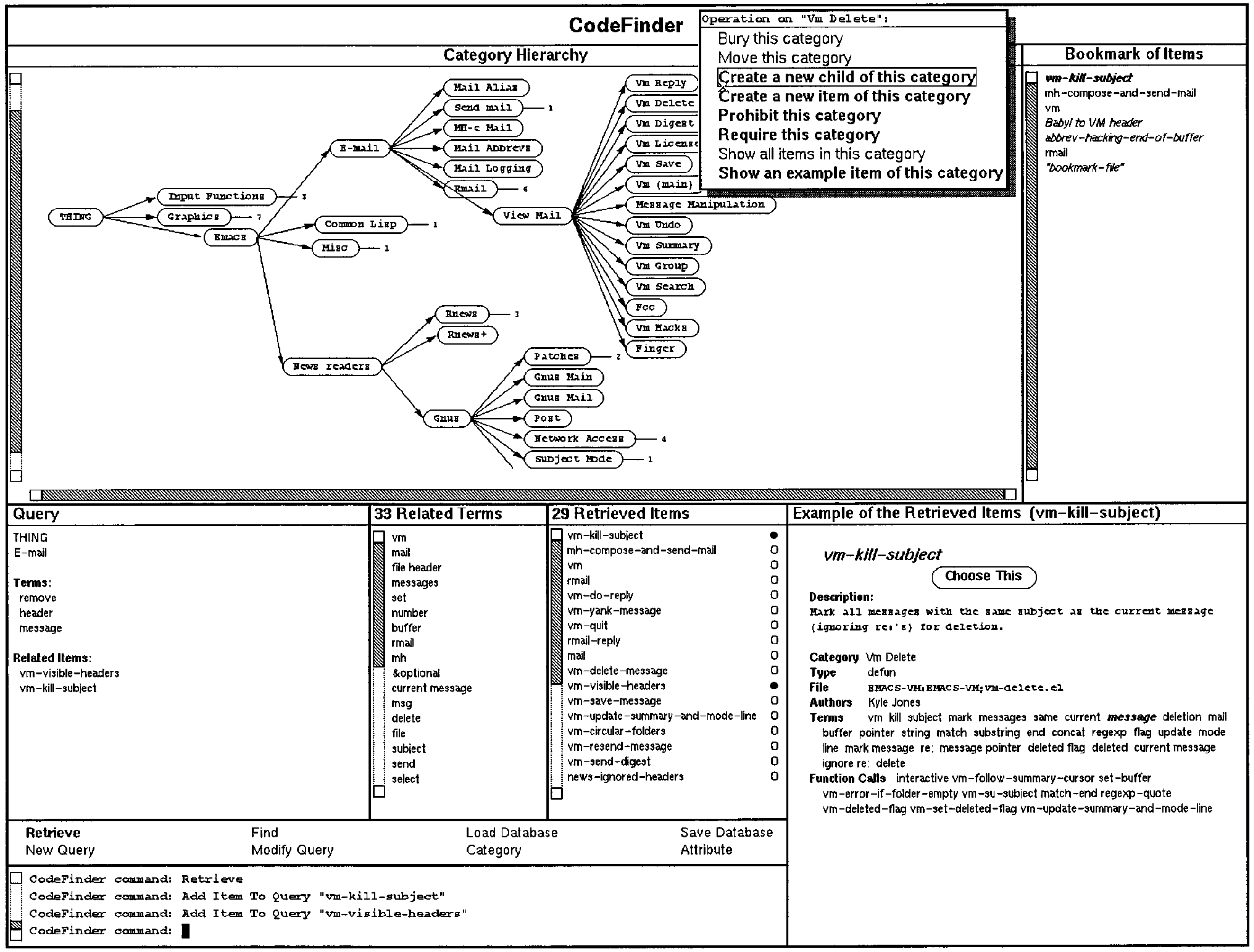
The representation of a frame for each component allows changing and updating the information as the component gets used. The initial *seeding* of the repository using PEEL does not give very detailed information, but hopefully enough to make it possible to locate the component.

Retrieval is done by a seperate program called CODEFINDER. See figure 4.2 on the facing page for an example of the interface in use. The top of the window is the current component hierarchy. Bottom left is the current query with a selection of two categories and three terms, and two related items are also selected for the query. Next is a list of terms related to the query, and then comes the retrieved items. The user can easily add new terms or mark components as relevant to the query.

Queries are matched against the repository using a technique known as *spreading activation*, which is a *soft matching* technique. A two layer network representing the components is created. One layer consists of the terms, and the other layer represents the components. In the network, components are connected to the terms they are described by. Spreading activation matching is performed by setting the activation value of the terms and components in the query to 1.0. In each iteration the activation value flows, or is distributed, to the neighborhood using a series of updating formulae. The idea is that values from different directions can meet up, and the components or terms thus activated should be relevant to the ones in the query. Dampening factors are also used to avoid spreading the activation too far, and thereby getting too unrelated information.

Spreading activation gives some of the synonym identification capabilities of the LSI method. Similar terms that occur in the same contexts will be retrieved together. However, there are quite a few parameters for the updating process which have to be properly tuned for the system to perform properly. The values for these parameters have to be determined experimentally.

The strength of the system comes from the easy updating of component representations. When a programmer finds an appropriate component, it is possible to update the representation in the repository by adding or changing indexing terms. This is done to make the component easier to find in the future, and thus evolve the repository to make retrieval of used components easier. By showing the programmer terms related to the current query, it becomes easier to reformulate the query to more precisely specify



Bookmark of Items:
vm-kill-subject
mh-compose-and-send-mail
vm
Babyl to VM header
abbrev-hacking-end-of-buffer
rmail
"bookmark-file"

Query:
THING
E-mail

Terms:
remove
header
message

Related Items:
vm-visible-headers
vm-kill-subject

33 Related Terms:
vm
mail
file header
messages
set
number
buffer
rmail
mh
&optional
current message
msg
delete
file
subject
send
select

29 Retrieved Items:

vm-kill-subject	●
mh-compose-and-send-mail	O
vm	O
rmail	O
vm-do-reply	O
vm-yank-message	O
vm-quit	O
rmail-reply	O
mail	O
vm-delete-message	O
vm-visible-headers	●
vm-save-message	O
vm-update-summary-and-mode-line	O
vm-circular-folders	O
vm-resend-message	O
vm-send-digest	O
news-ignored-headers	O

Example of the Retrieved Items (vm-kill-subject):

vm-kill-subject

Description:
Mark all messages with the same subject as the current message (ignoring re:'s) for deletion.

Category Vm Delete
Type defun
File EMACS-VM;EMACS-VM;vm-delete.el
Authors Kyle Jones
Terms vm kill subject mark messages same current ***message*** deletion mail buffer pointer string match substring end concat regexp flag update mode line mark message re: message pointer deleted flag deleted current message ignore re: delete
Function Calls interactive vm-follow-summary-cursor set-buffer vm-error-if-folder-empty vm-su-subject match-end regexp-quote vm-deleted-flag vm-set-deleted-flag vm-update-summary-and-mode-line

```
CodeFinder command: Retrieve
CodeFinder command: Add Item To Query "vm-kill-subject"
CodeFinder command: Add Item To Query "vm-visible-headers"
CodeFinder command: █
```

Figure 4.2: Example of CodeFinder interface. From [Henninger, 1997].

the wanted component.

Empirical studies of the CodeFinder system have shown that it adequately supports the locating of components. The soft matching algorithm along with the query by reformulation compensates for the vocabulary mismatch. The fact that the repository can be constructed by the semi-automatic Peel tool helps construct an initial repository from existing code modules created without reuse in mind. This gives a method for reusing the existing components, and develop the repository separately from the source files.

Comparing the CodeFinder project with the system constructed in this project, it is seen that CodeFinder is very focused on programmers adaptively modifying the repository. Updating of the model is not as straightforward in the LSI model, since using the easiest implementation, the SVD has to be recalculated after an update or the changed object will not be properly represented. However, the reuse support system created in this project should not need such an updating mechanism, as components are assumed to be well documented.

Matching queries using spreading activation is subject to the proper tuning of quite a few parameters. In LSI the crucial choice is in selecting a proper value for k for the document corpus along with proper preprocessing, as mentioned in section 3.3.2. The feedback and ranking possible in the CodeFinder program can also easily be constructed when using LSI, so there is no major difference in query expressiveness. The discovery of synonyms in CodeFinder using spreading activation seems less controlled than the semantic concepts found by co-occurrence in LSI, though no proper comparison of the two methods has been found.

4.1.3 The Guru system

The Guru system is used to index Manual (Man) pages on the IBM Aix platform [Maarek et al., 1991]. The goal is to create a searchable reuse library cheaply and still get good results. In the indexing phase they construct a profile for each component. Unfortunately, they are unable to assume any specific comment style such as the Javadoc standard, so the exact component a given comment is concerning is unknown. Instead of simply guessing, they extract information from the much higher level of information in Unix Man pages. They do this in order to ensure that they get representative text which is not simply documenting internal details, actually but usable for end-users.

They weighted constructing a system based on artificial intelligence (AI) methods against IR and chose IR because of [Maarek et al., 1991, p. 80]:

1. Lower cost because of automatic construction.
2. Transportability since no domain-specific knowledge is used when constructing the library.
3. Scalability since the library can easily be updated without using expensive human resources.

These arguments are also used when the choice is between using controlled and uncontrolled vocabularies. Using uncontrolled vocabularies gives better scalability as the set of allowed words does not have do be decided in advance of the indexing.

The indexing is based on *lexical affinities* where a set of attributes is identified for each software component. The system extracts terms from the text and identify co-occurrence patterns of terms in sentences to find these lexical affinities. The found occurrence patterns are then sorted according to their resolving power based on information measurements.

Searching

Users can search by entering a search string and get a ranked list of results, or they can choose to browse a generated hierarchy. To get good performance regarding the time it takes to search, the search and retrieval is performed using traditional *inverted index files.* The hierarchy of components is generated using an AHC algorithm, however, the search results are not clustered when retrieving.

When a plain-text query is entered, the following takes place: First a plain match is tried against the library and any matching components are returned in a ranked list. If none match, the browsable hierarchy is presented to the user.

In order to compare the query with the documents in the library, they use a similarity measurement based on the information contained in the individual terms. They mention a few different similarity measurements in the article, including the cosine. But the one they choose to use seems to be based on the inner product. By using for instance the cosine similarity instead, they could probably have achieved even more consistent results without having single terms dominate the similarity calculations. As mentioned in section 3.4.1, the inner product similarity can be victim of unbounded influence of a single component. These flaws are documented in [Jones and Furnas, 1987].

Empirical results

The performance of the system was measured by calculating precision and recall for a set of constructed queries. First they had to construct the test-sets. This was done by collecting queries from typical users. These were 3.7 words in length on average. They then selected 30 queries and had AIX experts select the appropriate response set of components in the AIX library among the 1100 indexed components. Having done this they were able to show that the GURU system performed around 15% better on average than an existing commercial system called INFOEXPLORER, which also searches MAN pages on the same platform, so the system represents an improvement over the previously best system.

Aside from the problems with using the inner product similarity for comparisons, the methods for finding relevant lexical affinities is certainly an interesting direction. And the use of domain experts for constructing the test collection is certainly desirable to be able to emulate.

4.2 Other Information Retrieval methods

As mentioned in section 3.1 when introducing IR, there are quite a few different ways of doing IR. They range from the free-text or keyword indexing methods, over the more formal probabilistic methods, to the extremely formal logic based methods. Using a more formal method usually improves retrieval precision as documents are more carefully represented. In the next sections the keywords based, probabilistic and logic based approaches to IR will be discussed. The ideas behind the Semantic Web are described at some detail within section 4.2.3 on logic based methods.

4.2.1 Keywords based

It is relatively easy to create a basic keywords based search engine using existing frameworks, as one can build on existing solutions. The JAKARTA LUCENE project[6] is a framework written in JAVA supporting the construction of specialized search engines using keyword matching. Documents can be indexed using a number of separate fields so that title, indexing date and body-text can be stored separately. Fields can be stored without being searchable, which means that you could add some data which will be available when the document is retrieved and is about to be presented, but which can not be searched. This gives maximum flexibility when choosing

[6]`http://jakarta.apache.org/lucene/`

what to store and how. The words in documents are stored in *inverted indexes* for fast query matching and efficient retrieval.

Constructing a basic search engine within the Lucene framework is easy for an experienced programmer. Standard classes exist for all the basic functionality, and if some special treatment is needed somewhere, it is easy to extend the system. However, one has to be a programmer in order to use the framework as it is not a completed application suite. It makes the constructed engine a great deal more flexible, but it does put some high demands on the programmer implementing a search engine using the Lucene framework. He or she has to connect quite a few components in order to construct a functional system, and the connection between the different parts are not at all obvious at first. However, the end-user of such an implemented search engine does not have to know anything about the underlying implementation, and merely sees a very efficient standard search engine.

Lucene supports a wide variety of query operators. Standard boolean **or** and **and** are supported, with **or** being the default operator between search terms. Queries can use wildcard searches with operators ***** and **?**, though not as the first character in a word due to performance considerations. Other implemented operators include fuzzy spelling and proximity searches where term positions are taken into account. This makes it possible to search for a exact sentence, something the VSM approaches are unable to do due to the *bag-of-words* representation of documents, see section 3.2.

Retrieval from the Lucene engine is extremely fast, but as mentioned it is unfortunately limited to keyword search. This means that such engines suffer under all the previously mentioned problems of inappropriate indexing terms.

Lucene is appropriate for applications where a fast but simple search functionality is needed. When a search task can be solved by keyword matching, the rich query language makes it a good candidate solution. Another benefit is the fact that the system also supports incremental updates while the system is online.

4.2.2 Probabilistic methods

Probabilistic methods is a branch of IR which is built on a statistical foundation. One of the interesting approaches which has given some very nice results is the Probabilistic Latent Semantic Indexing (PLSI) method, introduced below. The performance of PLSI has proven somewhat better than that of LSI in a number of experiments. The PLSI method is originally described in [Hofmann, 1999a] and further developed in [Hofmann, 1999b].

Instead of performing the SVD on the document vector matrix, which minimizes the Frobenius norm (equation (3.12) on page 26), a statistical approach is used where the approximation is optimized with regards to the predictive power of the model.

One of the benefits of this alternative representation is that polysemous or ambiguous words are represented by multiple factors for each meaning. This gives distinct advantages over LSI where polysemous words are basically represented as an average of the multiple meanings. However, as the author is not familiar with statistics and have therefore been unable to further pursue the more advanced PLSI method. This also includes methods based on Bayesian models.

The Okapi model is another example of a probabilistic method. As it is introduced, the model is based on a probabilistic model of "within-document term frequency, document length and within-query term frequency" [Robertson and Walker, 1994, p. 232]. A rough model is used to approximate a full probabilistic model of term frequencies, where occurrences of terms are categorized as relevant or irrelevant for the given document. This apparently gives considerable performance increases in comparison to more simple weighting functions. In [Robertson et al., 1995] the BM25 algorithm is introduced. This algorithm is still the benchmark used when introducing new methods at the TREC conferences.

4.2.3 Logic based methods

The logic based methods demand more of the input data. The idea is to use formalisms for describing concepts and instances in a representation only allowing one interpretation. The main drawback with using logic is the high cost of formalizing data for use in the system. First the Semantic Web will be introduced, as it is a modern application of logic based methods. Then Description Logic is described, which is an example of a logical formalism usable for the Semantic Web.

Semantic Web approaches

The Semantic Web (SW) is one of the great promises for the future. It is an expression used to denote work in many different areas. The goal is stated in an often quoted passage [Berners-Lee et al., 2001]:

> "The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation."

Where text-based IR aims to give good enough or acceptable answers for human consumption, the goal of the Semantic Web (SW) is to give specific meaning to data. This will allow agents to make informed decisions and not simply be the result of manual integration. However, the Semantic Web is surrounded with quite a lot of hype, as [Marshall and Shipman, 2003] points out, but ignoring the visions of intelligent agents planning our lives for us, there is still some interesting information integration aspects in the SW vision.

In order to give meaning to concepts in a way that can be interpreted by a computer, *ontologies* are used. An ontology is a description or specification of concepts and relationships that can exist between one or more agents. It gives a vocabulary that can be used to describe objects or *resources*. The World Wide Web Consortium (W3C) is currently working on a number of technologies in order to enable the SW to happen – it is beyond the scope of this text to properly introduce all of them, but some are briefly outlined in the following.

First an important note: Each thing described and used in the SW is named and referenced using a unique URI. This could be a reference to an existing homepage or it could simply be an address known to be unique as one controls the domain used. By using URI for naming the objects, name clashes can be avoided globally because it builds upon on the already globally unique DNS system.

A technology known as Resource Description Framework (RDF) is used for the basic data-model of resources, where each entry is a triple consisting of the parts:

$$\langle \textit{subject, predicate, object} \rangle$$

In this triple, the subject is the entry being described, the predicate is the property or characteristic being added to the knowledge graph, and object is the value of the property. Subjects and predicates are defined as URI references. Predicates could for instance come from the set of predicates defined in RDF Schema (RDFS), see below. The object in the triple can be either another URI to reference the object it refers to, or it can be a simple value such as a date or a string. RDF data is usually stored in an XML-based format known as RDF/XML[7] where the graph can be represented in an efficient way. An example of storing the creation date for a resource can be seen in figure 4.3 on the following page.

A good example of how clever it is to store data as RDF/XML instead of as plain XML, was given in a talk [Berners-Lee, 2003]: assume that data

[7]`http://www.w3.org/TR/rdf-syntax-grammar/`

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:exterms="http://www.example.org/terms/">
  <rdf:Description
    rdf:about="http://www.example.org/index.html">
    <exterms:creation-date rdf:datatype=
      "http://www.w3.org/2001/XMLSchema#date">1999-08-16
    </exterms:creation-date>
  </rdf:Description>
</rdf:RDF>
```

Figure 4.3: An example of a statement stored in RDF/XML. From `http://www.w3.org/TR/rdf-primer/#example7`.

from two different sources, both in XML, have to be integrated into a single document. First the Document Type Definition (DTD) or Schema for both have to be understood, then a combined one for the result has to be created. Then a transformation has to take place, for instance using XSL Transformations (XSLT). If the documents were in RDF/XML they could simply be concatenated together, and the graphs of their contained statements would then be merged. The problem of perhaps not having used the same ontology or vocabulary in the two documents still exists. However, there are straightforward ways to, for instance, establish equality between different parts of the graph within the standards and thus merge the information.

To make it easier to construct data for humans, an easier short-hand for writing RDF statements is the Notation 3 (N3) format. Assuming that the namespace prefixes have been defined elsewhere, the RDF statement in figure 4.3 can be written as simple as in figure 4.4 on the next page. Note that most of the representation is the definition of the used namespaces. When using N3 for writing statements, the namespace definitions will usually take up relatively little space as they are created once, and used many times within the file. This simple N3 representation can then be converted into RDF/XML using freely available tools such as Closed World Machine (CWM) [Berners-Lee, 2000].

RDFS is used for describing properties and relationships between RDF resources. Web Ontology Language (OWL) is used for adding even more vocabulary to describe relationships between classes, cardinality, and other more advanced properties. Refer to the W3C web-site[8] for more information

[8] `http://www.w3.org/2001/sw/`

```
@prefix ex: <http://www.example.org/> .
@prefix exterms:  <http://www.example.org/terms/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
ex:index.html  exterms:creation-date  "1999-08-16"^^xsd:date .
```

Figure 4.4: Same RDF example as figure 4.3, in Notation 3.

and the appropriate standards.

While the Semantic Web in its current form is still far from finished, the RDF standard does make it possible to define data in a generally agreed upon language. However, when it comes to actually using the data for integration, the picture becomes more unclear. In order to properly utilize the formal knowledge stored in RDF, a powerful language is needed with existing implementations. Such a formalism is Description Logic.

Description Logic

Description Logic (DL) is a formalism for representing knowledge. It is properly described in the "Description Logic Handbook", [Baader et al., 2003]. The basic idea of DL is to make as expressive a logic with as many features as possible, while at the same time ensuring that calculations needed to make inferences can be performed efficiently. Data is represented as facts and the system is able to draw conclusions based on the given rules and facts.

The term DL is used for the representation systems, but often the word "concept" is used to refer to the expressions of a DL language, denoting sets of individuals. The word "terminology" is used to denote a (hierarchical) structure built to provide an intensional representation of the domain of interest.

DL is used in knowledge representation systems that provide a language for defining a knowledge base and tools to carry out inferences over it. The systems normally use a basic "tell and ask" interface: tell the system facts to add knowledge, and ask to get out information. As described in the section about SW above, two different levels of information are used, namely the definition of concepts and concrete individuals.

Similarly, knowledge based systems are separated into two groups – the *TBox* and the *ABox*. The TBox (terminology box) is used to define the intensional knowledge as a terminology. This gives a vocabulary to describe the individuals. The TBox is also known as a taxonomy or ontology, and this is usually a stable part of the system. The ABox (assertional box) contains the extensional knowledge, also known as assertional knowledge. This is knowledge specific to the individuals in the domain, and therefore subject to change more often. The individuals are described using the terminology defined in the TBox.

Reasoning within DL This separation into an intensional (TBox) and an extensional (ABox) part helps distinguish between the different levels of information. The TBox defines the available vocabulary, whereas the ABox defines the individuals in terms of the given vocabulary. The basic reasoning within the ABox is checking whether an individual belongs to a specified concept – this is known as *instance checking*. There are other kinds of reasoning, but they can be shown to be solvable by instance checking. In the TBox the basic reasoning is *subsumption* relation. Subsumption, denoted $C \sqsubseteq D$, can be described as checking whether the concept D is more general than concept C. That is, it checks if one concept is always contained within another.

Difficulties of use There are some difficulties with using logic based systems. It is possible, but hard, for humans to represent complex objects – it is even harder to retrieve the wanted instances. A complicated specification and query language has to be used – it has a steep learning curve, and is it worth the effort? In [Frakes and Pole, 1994] they conclude that text-based retrieval performs just as well as the more expensive methods for Information Retrieval, and the strict logic approach is not worth the effort. It is of course a different matter when integrating different tools, but for tools aimed at end-users, logic-based tools are simply too difficult to use. This is not a flaw with the DL project, but a general problem with trying to specify concepts in the world without ambiguity.

Examples of DL in use

In [Kopena and Regli, 2003] they construct a design repository for electrical components, using DL for the specification and retrieval mechanism. By creating a formal ontology for describing physical components, it is possible to use the inference system to conduct searches. These searches can then specify the specific properties a retrieved component must fulfill. To perform the inferences needed in the system they have implemented a tool called DAMLJessKB[9].

[9]`http://edge.cs.drexel.edu/assemblies/software/damljesskb/`

To give an example of the difficulties of properly specifying components using logic expressions, here is a description of a sensor which measures electrical signals:

$$ElectronicSensor \equiv Artifact \sqcap \exists function.[Measure \sqcap \exists input.Flow \sqcap \exists output.[Electrical \sqcap Signal]].$$

As can be seen, describing even the simplest physical component described in formal detail is quite verbose and difficult to grasp.

Using Description Logic for inference within the Semantic Web project is discussed in [Horrocks, 2002]. In it, the DAML+OIL combination is described as a use of Description Logic for the Semantic Web. Adding such an expressive logic to the SW project may help realize the SW vision.

4.2.4 About other Information Retrieval methods

The preceding has been a look at alternative methods for representing information for retrieval. Keywords based methods were first introduced, and Lucene was used as the main example. Probabilistic methods were briefly described, but no in-depth analysis was made, and finally we looked at logic based methods. This included a brief description of some technologies related to the Semantic Web project, and a look at Description Logic.

In the next section, alternative possibilities for output representation in the constructed system are described.

4.3 Output representation

We now turn to a different aspect of the system, namely the user interface. Flat lists and clustered results are not the only way to represent output from a search engine. As the other reuse repositories in this chapter has shown, there are very different ways to do the output representation.

The CodeBroker project tried to create a non-intrusive interface where the interaction with the repository happens as an additional benefit of commenting the code in a program. The two different passes of evaluating information was explicitly considered when the interface was constructed. *Information discernment* was supported with the short list, and *detailed evaluation* was done by opening the complete Javadoc. This layered user interface is a great idea for presenting a list of components, as it gives the user a short overview and access to the complete documentation.

A more rich interface was presented with the Peel and CodeFinder project in section 4.1.2. The user searched and refined the query by using direct relevancy feedback. Adding new words, documents, or categories to the query specification prompted an update of related terms and retrieved documents. This is a more direct method of query specification, and it gives more immediate access to the structure and information contained in the repository, than a flat list as in CodeBroker.

The final example of a user interface was given in section 4.1.3 concerning Guru. No example was shown of the output from the system, as it is a very simple presentation. The results are presented in a list with the name of the command and the similarity value next to it. This representation gives no context of the retrieved commands and no feedback as to how the query may be modified to get a more relevant result. The browsable hierarchy was an interesting idea, though.

These very diverse user interfaces gives an idea of how such a reuse interface can be constructed, and the layered approach of CodeBroker is going to be used to some extent in this project.

4.4 Position

In this section the position taken in relation to the other approaches will be described. Specifically, it will be describe why the problem has been attacked this way.

4.4.1 Discussion of possibilities

As has just been described, there are quite a few other ways this problem could have been solved. This section covers the reasons why this specific approach was chosen.

Each of the described existing reuse systems have one or more major drawback. The CodeBroker system is locked intimately with the Emacs editor, and porting it to a more modern Java development environment would constitute a significant programming effort. The Peel and CodeFinder package provided a fairly rich user interface, but was tied to Lisp code and used spreading activation for searches. And finally the Guru system was limited to handling Man pages.

Building on an open framework such as Lucene would be easy to do, but it would suffer under the problems of keyword matching. But doing things the completely formal way with Description Logic or other logic-based methods would require a large investment of human effort in order to create the initial repository.

The goal of this project is to create an efficient and usable reuse support system. By decoupling the system from a specific programming environment and system platform and placing it on a web-server, it becomes as easily portable as other programs written the JAVA language. Because the repository is centralized, maintenance becomes easy, and only a single machine has to be updated when the documentation is changed.

Performing searches in the system should be as easy as possible. As it can safely be assumed that all programmers using the system are familiar with the standard interface to search engines, using the system is very easy. The Vector Space Model approach used here ensures that queries are matched more appropriately than simple keyword matching would. Using an uncontrolled vocabulary ensures that a wide variety of terms are usable when searching, and this further lowers the difficulties of using the system. When reviewing results in the simple unclustered form, they are simply presented as a ranked list of methods. But using the clustering facilities enables a quicker *information discernment* phase.

Since the input data from the system is generated from plain JAVADOC generated HTML files, no additional cost is incurred to generate the repository. If a collection of classes are meant for reuse, it is reasonable to assume that they are documented to a certain degree. Seen in this light, an investment of cheap computer processing time could potentially make existing resources much more accessible to users.

Regarding changes of the repository, there are a few approaches of accommodating them. When methods are added or changed, the easiest solution is to throw the entire index away, and index the entire collection from scratch. This takes about an hour of computer-time on a modern machine, and could easily be done as a nightly job, for instance. But striving for a more elegant and evolution friendly solution, there are methods to update the result from the SVD without performing the full calculation. These methods of SVD updating will be discussed briefly in section 7.2.4, but see [Berry et al., 1994] for more information.

4.4.2 Conclusion on position

In order to make a successful system which fulfills the problem description in section 1.2, the above aspects have all been taken into account. The automatic handling of synonyms that LSI provides is a great asset and it is one of the primary reasons why the method was chosen.

The fact that users can use free text to search the repository and not have to learn yet another query language, greatly lowers the bar for new users.

4.5 Conclusion

This chapter has discussed alternative approaches to the problem being solved in this project. First a discussion of three different existing reuse support system were presented. Even though the retrieval technologies underlying each of them were quite different, the underlying idea of promoting reuse was a common factor. Following that, different approaches to Information Retrieval were discussed and especially the logic based methods were introduced at some length. The ideas and a few of the basic technologies of the Semantic Web were introduced.

The reuse support systems discussed earlier were also used to represent different possible user interfaces, ranging from simply presenting an extremely simple ranked list, to a rather elaborate graphical user interface with direct relevancy feedback.

Having looked at ways of solving the problem, the details of the implementation is discussed in the next chapter.

Chapter 5

Implementation

This chapter describes the system that has been created. It details the implementation choices and the exact algorithms chosen to solve the problems. The methods of data processing which have been used are also described. Finally, the different available user interfaces will be described.

5.1 Outline of implementation

The basic idea underlying the implementation of the system has been to create a number of separate subsystems and components. This was done in order to make future development as easy as possible. Whenever possible, design patterns have been used to decouple different parts of the system, and general ideas from framework programming have been used to make experiment configurations easy to change. This is why *factory method* and *template method* patterns have been used throughout the system [Gamma et al., 1994]. This use of patterns has resulted in some elegant solutions, and made it fairly easy to parameterize solutions by inheritance and overriding factory methods. An example of this is the ease with which one can change vector normalization.

There are two distinct phases in the system. The first is the indexing phase, in which the index of the code reuse repository is created. This has to be done once, and takes about an hour on a modern computer for the entire JAVA API. The other phase is the interactive use, where a programmer uses the system to find relevant components. As will be shown later, there are different ways to access the search front-end of the system. There is a user interface for exploring, where one can navigate between the different components and see the related terms and components for each. This is done in order to ensure that the components are properly represented, and

$$\boxed{\text{Javadoc} \overset{\texttt{indexer}}{\longrightarrow} \text{XML file} \overset{\texttt{parser}}{\longrightarrow} \text{matrix} \overset{SVD}{\longrightarrow} \text{reduced matrix}}$$

Figure 5.1: Flow of data when indexing.

```
<typedecl name="dk.daimi.rud.util.Queue" type="class" id="rud581">
  ...
  <method signature="public void enqueue (java.lang.Object o)"
    id="rud583" anchor="enqueue(java.lang.Object)">
  Add an object to the queue. When calling dequeue, the object o
  will be returned after all other currently enqueued objects.
  </method>
  ...
</typedecl>
```

Figure 5.2: XML representation of a single component extracted from Javadoc, here `dk.daimi.rud.util.Queue.enqueue()`.

it is mainly for tuning the system performance. The other user interface is more like a normal search engine, where a user enters a query and gets a list of results. It is not entirely like a normal search engine though, as this interface does have some powerful options, such as clustering of the search results. More on the user interface in section 5.3 on page 67.

Now follows a description of the indexing phase where the index is constructed.

5.1.1 Indexing phase

The phase of indexing Javadoc files is subdivided into four steps. See figure 5.1 for an overview of the flow of data when indexing. The first step is the initial indexing where a collection of Javadoc files are parsed and output to XML, see figure 5.2. The figure gives a continuation of the example in figure 2.2 on page 8, and shows what information is extracted from the HTML files. The reason for using XML for output is, that it is both easy to produce and very easy to parse. It also gives a logical way of representing the nested structure of classes with methods without much redundancy.

Instead of extracting information from HTML generated by the Javadoc tool, an alternative would be to specialize the Javadoc tool with a custom Doclet that generates XML instead of the standard HTML output. Doclets are plugins written for the Javadoc tool, and the interface allows extensions to be made which can build greatly on the existing implementation. In fact, SUN has made such an XML producing Doclet[1]. The benefit of extracting comments from the generated HTML files instead of using such a Doclet is that a searchable index can be constructed even if the source code is not available, as is often the case.

For each method, the full `signature` with all parameter types is stored in an attribute. The `anchor` within the HTML file is also stored to allow the creation of links directly back to the JAVADOC documentation. This makes it possible for the user to browse the documentation further after having found an interesting method. Notice how each `typedecl` (type declaration) and `method` gets a unique identifier. This makes it easier to refer to specific methods. It will also be used in the experimental interface, as shall be seen later.

Some information is left out when indexing. Only methods are indexed in the system, so class level comments are ignored. This is done for simplicity as only one type of retrievable component has to be handled when doing it this way. Adding comments regarding entire classes is a part of the future work, see section 7.2.4.

As can be seen in figure 5.2 on the facing page, some other parts of the documentation are left out as well. In this example, it is the comment for the parameter which is removed. In general, all additional fields beyond the main comment is removed. This is done in order to avoid "see also" sections, information on thrown exceptions, and other miscellaneous information. The current prototype is made as simple as possible, but to change the behavior of what fields to include is simply to change the script which parses the JAVADOC files.

The script then takes all text used to describe the details of a method and strips the HTML-tags, leaving only the contained text. The generated XML file is now ready for the next step.

5.1.2 Forming document vector matrix

Next step in the process is reading the XML file and collecting all the words it contains. The following steps are implemented in JAVA, except for the truncated SVD calculation. A Simple API for XML (SAX) parser is used to process the rather large XML file. SAX is built for batch processing which is exactly what is needed here. Choosing SAX over Document Object Model (DOM) comes naturally, as the nesting in the generated XML document is not complex and searches or selections are not going to be performed directly on the structure – everything is to be read and stored internally for

[1]`http://wwws.sun.com/software/xml/developers/doclet/`

use in the later stages. All words used in all methods are collected and the methods described in the next section are used to split identifiers.

A document vector matrix (DVM) is created by constructing a document vector for each indexed method in the XML file, which is then stored in a sparse matrix representation. The DVM is stored as a huge sparse matrix which is created using the CERN Colt[2] collection of components. These components have been indispensable in this regard, as the matrix library is very well designed and efficiently implemented. Besides supporting sparse and dense matrices with the same interface, it is also easy to create a view of part of a matrix as a new matrix without copying the underlying data. This makes calculations on, for instance, column vectors very easy to do as a view of the column can simply be created. One can then perform calculations with vectors without being concerned with where they came from. This convenient functionality simplifies some parts of the code a great deal.

Handling identifiers

In order to separate identifiers into multiple terms, a form of identifier-splitting is employed. When this is done, `anIdentifier` becomes the two words `an` and `Identifier`. Acronyms within identifiers are also supported, so for instance `checkCVSStatus` becomes `check`, `CVS`, and `Status` as would be expected. The splitting of identifier is similar to the approach in [Michail and Notkin, 1999], except for fact that the more elaborate procedures, such as common prefix removal, dictionary lookup of connected words, and abbreviation expansion, are all left out. These extensions can be added if needed, but since Java allows extremely long identifiers and the use of packages for namespace control, programmers should have no need to use identifier prefixes to create globally unique identifiers. This means that the problem of skipping these advanced heuristics should not be too great. However, the real need for identifier splitting can be seen by looking at the enormous amount of methods in the Java API beginning with `set` or `get`. These prefixes are the standard convention for methods giving controlled access to data within an object and taking these apart gives more relevant words to index.

After the identifier splitting, all words are converted to lowercase in order to allow words which only differ in case to be treated as the same term.

[2]`http://hoschek.home.cern.ch/hoschek/colt/`

Normalizing the DVM

In order to avoid too much garbage in the list of indexed terms, the terms that are only used once globally are removed, as are numbers. Stemming is available in the system using the Porter Stemmer, and it is easy to enable when starting the indexing process. The fact that stemming should be available is based on the discussion in section 3.2.2. To get the best performance from the system, the local normalization used is the logarithm of occurrence count, and the global normalization is the modified entropy measurement. Refer back to section 3.2.3 for the discussion of weighting alternatives.

The resulting sparse matrix is then output in *Harwell-Boeing* sparse matrix format, which only stores the non-zero entries and their indices [Duff et al., 1992, appendix A]. The class which constructs this output is good example of a component which could easily be used in a different project, as it is self-contained.

Before the SVD is calculated, the entire collected set of data is stored as a file using standard JAVA serialization of an object. This ensures that the DVM, lists of terms and documents and general configuration is stored. A more advanced representation could be used, but this format works well for this prototype.

5.1.3 Calculating the truncated SVD

The software package which is used to perform the truncated SVD is able to handle huge sparse matrices without excessive memory usage. It operates directly with the sparse matrix representation and the performance is good. A lot of the calculation speedup is due to the fact that matrix multiplication is very fast when using sparse matrices. The software used is the SVDLIBC[3] program. It is a more user friendly interface of the original SVDPAKC[4] program package, described in [Berry et al., 1993]. Both packages reads matrices written in Harwell-Boeing sparse matrix format, and output the three matrices resulting from the truncated SVD calculation.

5.1.4 Finishing the indexing phase

The result of the truncated SVD is stored in three files. These each contain a dense matrix, one for each of the matrices T, S, and D. From these files the corresponding matrices are created in the JAVA program and joined with the

[3]http://tedlab.mit.edu/~dr/SVDLIBC/

[4]http://www.netlib.org/svdpack/

serialized stored information from the matrix forming step. This updated object is then stored back to a file, again using simple serialization. Much of the matrix computation needed when searching has been performed and is also stored in the file. The complete data has now been constructed and the serialized file is used as input for the search interface.

5.1.5 Changing configuration for indexing

The entire indexing phase is automated. All the user needs to supply is the location of the JAVADOC HTML files, at what URL the same documentation can be reached using a browser, and in what directory to store the data. The JAVADOC files must reside on the local file-system in the current version for the indexing, as simple file-system traversal is used to find the HTML files. The rest is a scripted process. Allowing indexing of online documentation is a planned future extension.

Changing the setup of the system, for instance, to use a different kind of normalization is done by subclassing the relevant part of the system and provide an alternative implementation for the *factory method.* This underlines the fact that the implementation is not monolithic, as such research prototypes can have a tendency to become. For more details on which command-line options are available when indexing, see Appendix B.

5.2 Searching

Even though there are different ways the user can give his or her query to the system, the back-end is always the same. A query is taken as a string and split using the same process as when indexing methods. This means that identifier splitting is performed and numbers are removed. The same local normalization is performed, currently the logarithm of occurrence count for nonzero entries. Global normalization is done by multiplying each nonzero entry with the corresponding global normalization value from the indexing phase. In order to reduce the query vector to the same dimensionality as the truncated DVM, it is transformed using equation (3.19) on page 29.

To calculate the similarity between the constructed query document and all other documents, the cosine similarity measurement is used. Refer to section 3.4.1 for other possible similarity measurements.

In order to rank the results, all documents are compared to the pseudo-document of the query using the similarity measurement. The resulting list is sorted by decreasing similarity, and the top n documents are returned – the user usually chooses the value for n on the front-end.

5.3 User interface

As mentioned earlier, there are two different user interfaces. One is similar to ordinary search engines in that users enter a query, and a list of results is presented. It is also possible to get clustered results, in order to easier get an overview of the returned components. The other interface gives an unpolished look into the engine with very much information available on each indexed method, making it possible to explore the structure. The user can also enter a query, and the results are presented as a ranked list. Clicking on a component shows details and the associated terms with weights and the methods in the immediate neighborhood. This interface is further described in section 5.3.2. Both interfaces are browser-based, and need a web-server to run. This allows multiple users to access the system concurrently on a single central machine.

According to [Belkin et al., 2003], giving a larger input area and a general hint to the user of entering more text actually encourages the user to enter a longer query specification. This is currently only used in the internal interface, but it is an obvious extension to the regular front-end. As the system current selection of components does not encompass huge amounts of text, it is probable that a user at some point will enter one or more query terms not used in the system. It is not very gracefully handled in the graphical interface at the moment, as a blank page is simply returned. The more terms the user adds when search, the higher the probability of getting something relevant out of a query.

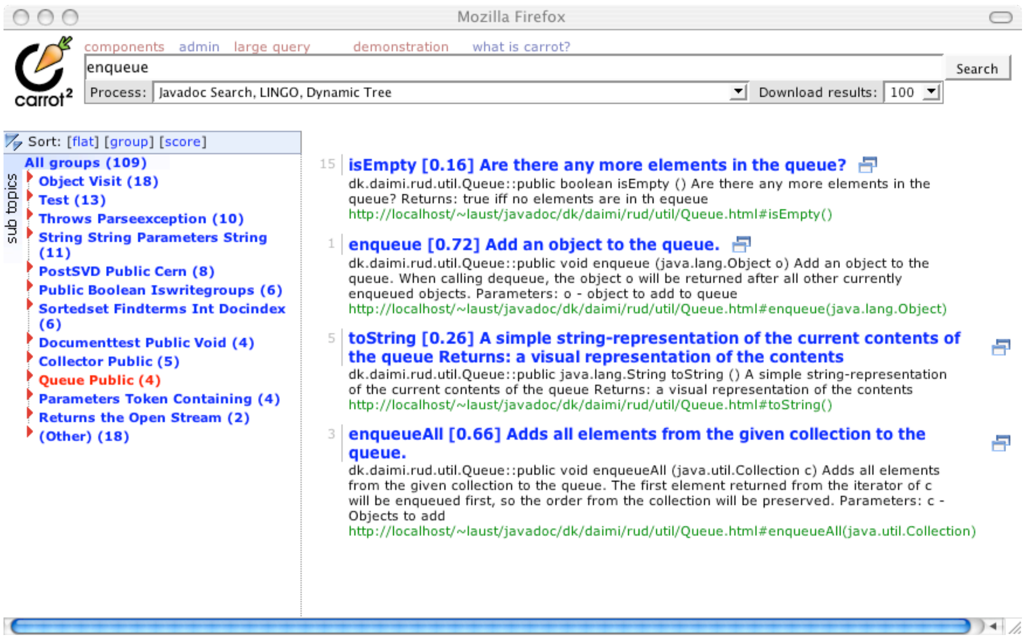
In order to cluster search results as outlined in the problem description, the Carrot framework is used. The next section is a short introduction to Carrot, followed by the specific use in this project.

5.3.1 Carrot framework

The first version of the Carrot[5] framework was created by Dawid Weiss as a part of his Masters Thesis [Weiss, 2001]. It is a framework written in Java to analyze output of search engines and perform various kinds of clustering. Since then the framework has evolved and it is released as Open Source.

The framework is now in its second generation named Carrot 2. Several people have contributed their own work, and because the framework consists of independent components which communicate using agreed upon XML formats, extensions are easy to make. Other people have successfully used the framework to conduct experiments using the existing clustering

[5]`http://www.cs.put.poznan.pl/dweiss/carrot/`



enqueue

Process: Javadoc Search, LINGO, Dynamic Tree

Download results: 100

Sort: [flat] [group] [score]

All groups (109)
- Object Visit (18)
- Test (13)
- Throws Parseexception (10)
- String String Parameters String (11)
- PostSVD Public Cern (8)
- Public Boolean Iswritegroups (6)
- Sortedset Findterms Int Docindex (6)
- Documenttest Public Void (4)
- Collector Public (5)
- Queue Public (4)
- Parameters Token Containing (4)
- Returns the Open Stream (2)
- (Other) (18)

15 **isEmpty [0.16] Are there any more elements in the queue?**
dk.daimi.rud.util.Queue::public boolean isEmpty () Are there any more elements in the queue? Returns: true iff no elements are in th equeue
http://localhost/~laust/javadoc/dk/daimi/rud/util/Queue.html#isEmpty()

1 **enqueue [0.72] Add an object to the queue.**
dk.daimi.rud.util.Queue::public void enqueue (java.lang.Object o) Add an object to the queue. When calling dequeue, the object o will be returned after all other currently enqueued objects. Parameters: o - object to add to queue
http://localhost/~laust/javadoc/dk/daimi/rud/util/Queue.html#enqueue(java.lang.Object)

5 **toString [0.26] A simple string-representation of the current contents of the queue Returns: a visual representation of the contents**
dk.daimi.rud.util.Queue::public java.lang.String toString () A simple string-representation of the current contents of the queue Returns: a visual representation of the contents
http://localhost/~laust/javadoc/dk/daimi/rud/util/Queue.html#toString()

3 **enqueueAll [0.66] Adds all elements from the given collection to the queue.**
dk.daimi.rud.util.Queue::public void enqueueAll (java.util.Collection c) Adds all elements from the given collection to the queue. The first element returned from the iterator of c will be enqueued first, so the order from the collection will be preserved. Parameters: c - Objects to add
http://localhost/~laust/javadoc/dk/daimi/rud/util/Queue.html#enqueueAll(java.util.Collection)

Figure 5.3: Example of a clustered search result, using the LINGO algorithm.

techniques or constructing new ones. See figure 5.3 for an example of CARROT in action. The interface is properly described in "*Using* CARROT *for clustering results*" on page 70.

Components and processes

The two basic building blocks in CARROT are *components* and *processes*. A component can be of one of the three types *input*, *filter*, or *output*. See figure 5.4 on the next page for a simple diagram showing the interaction between components. Each component is implemented and deployed separately as an application to a webserver. Components do not need to be on the same webserver in order to function together as all communication is done using HTTP, however, running all components on one computer does give a significant speed advantage. In order for the central CARROT controller to access the component, a description of the component is created in XML. Components are used in processes, which are also stored in XML files.

A process description consists of an input, a possibly empty list of filtering components and finally an output component. The input component is given the query in XML and generates the data by querying a search engine. In the original CARROT experiments this was output from a web search engine such as GOOGLE. After the input has generated the resulting XML in

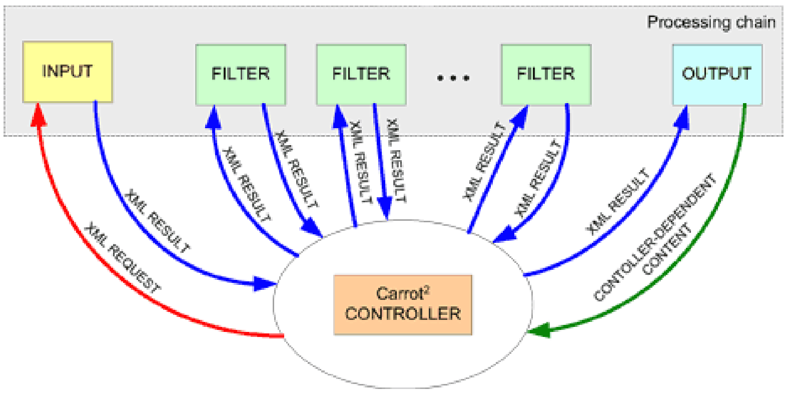


Figure 5.4: Data flow when processing in Carrot from input to output. From the manual, available from the Carrot homepage.

a standard format, the data is sent to filters for processing. The set of currently available filters include stemming in English and Polish, clustering algorithms, and some debug filters. As for output components, only one is available at the moment, but it does allow for the display of clustered results with sorting and subset viewing directly in the user's browser, without consulting the server again.

Available clustering components

Three algorithms are currently available within the Carrot framework for clustering of search results. These are briefly described below, where references to more information are also listed. First the AHC algorithm will be sketched, then the Suffix Tree Clustering (STC), and finally the very interesting Lingo clustering filter will be described.

Agglomerative Hierarchical Clustering This component was implemented by Michał Wróblewski as part of his master's thesis [Wróblewski, 2003]. The algorithm is optimized for handling the short texts of search results, and the idea behind it has already been briefly described in section 3.5.2.

The Suffix Tree Clustering algorithm The STC component was a part of the original Carrot project documented in Dawid Weiss's master's thesis [Weiss, 2001]. Besides the STC algorithm, he also created a Polish stemmer, and collected a large text corpus of Polish documents for testing. This corpus is now also freely available.

Suffix trees are structures containing all suffixes of a set of strings. It can be constructed in linear time proportional to the size of the input, and enables fast lookup of substrings. In STC, clusters are formed according to identical phrases found in the documents.

Lingo Currently this is the most advanced Carrot clustering component. It has been constructed by Stanisław Osiński as a part of his master's thesis [Osiński, 2003]. It uses LSI to conceptually cluster the results. The quality of the generated clusters is quite impressive when used on regular search results, and hopefully equally impressive when it is used to cluster Javadoc search results.

Using Carrot for clustering results

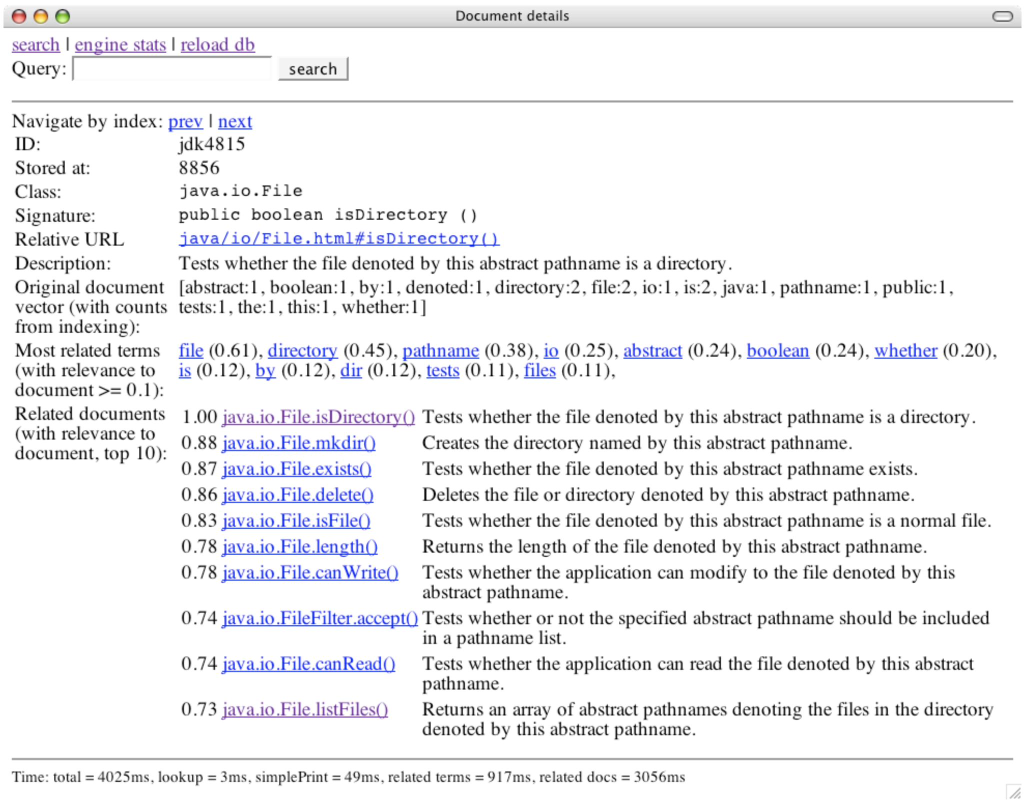
The Carrot framework has been used to apply various clustering techniques on the returned results. First of all, an input filter formats the output of the search engine in a manner which can be parsed by the other Carrot components. This is done by building an input component which is an interface to the search system. It takes the given query and performs a normal query matching as described in section 3.4. In order to have a valid Carrot result set, results are formatted for this special format. In other words, the ranked list is retrieved, and the texts belonging to the individual components are formatted and output as a list of results to Carrot *input* component. This list can then be taken apart again by the algorithms in Carrot, clustered and presented in the browser.

The components are then connected in a pipeline manner, starting with the constructed input component, and which ends in a output module. After the input formatting step, the standard stemming and clustering components can be used. See figure 5.3 on page 68 for an example of both Javadoc search and Lingo in action.

5.3.2 Exploratory interface

In order to give a better understanding of the various parts of the VSM data, this interface gives a more raw look what is going on under the hood. This interface can show two kinds of results: normal list of search results and a detailed view of a single document with closest related terms and documents. The related items are calculated using the matrix computations described in section 3.4.1.

One important detail of this interface is the possibility of changing the reduced dimensionality representation – the value of k can be changed on



Navigate by index: prev | next

ID:	jdk4815
Stored at:	8856
Class:	`java.io.File`
Signature:	`public boolean isDirectory ()`
Relative URL	`java/io/File.html#isDirectory()`
Description:	Tests whether the file denoted by this abstract pathname is a directory.
Original document vector (with counts from indexing):	[abstract:1, boolean:1, by:1, denoted:1, directory:2, file:2, io:1, is:2, java:1, pathname:1, public:1, tests:1, the:1, this:1, whether:1]
Most related terms (with relevance to document >= 0.1):	file (0.61), directory (0.45), pathname (0.38), io (0.25), abstract (0.24), boolean (0.24), whether (0.20), is (0.12), by (0.12), dir (0.12), tests (0.11), files (0.11),

Related documents (with relevance to document, top 10):

1.00	java.io.File.isDirectory()	Tests whether the file denoted by this abstract pathname is a directory.
0.88	java.io.File.mkdir()	Creates the directory named by this abstract pathname.
0.87	java.io.File.exists()	Tests whether the file denoted by this abstract pathname exists.
0.86	java.io.File.delete()	Deletes the file or directory denoted by this abstract pathname.
0.83	java.io.File.isFile()	Tests whether the file denoted by this abstract pathname is a normal file.
0.78	java.io.File.length()	Returns the length of the file denoted by this abstract pathname.
0.78	java.io.File.canWrite()	Tests whether the application can modify to the file denoted by this abstract pathname.
0.74	java.io.FileFilter.accept()	Tests whether or not the specified abstract pathname should be included in a pathname list.
0.74	java.io.File.canRead()	Tests whether the application can read the file denoted by this abstract pathname.
0.73	java.io.File.listFiles()	Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.

Time: total = 4025ms, lookup = 3ms, simplePrint = 49ms, related terms = 917ms, related docs = 3056ms

Figure 5.5: Exploratory interface with detailed document view.

the fly. Since there are some rather heavy matrix computations involved in calculating the similarities, these matrices are cached once calculated, which is a part of the indexing process. Having this knob available for tweaking online makes it very easy to test different settings. Note that the selected value for k is global, so all users on the web-server using the service will share the same resulting representation.

An interesting feature is the detailed view, shown in figure 5.5. In this view all details for a single document are shown. This includes the unique identifier from the XML representation, the column index in the DVM, the class, the signature, a link to the Javadoc entry, the full description, the closest related terms, and the top 10 related documents. Both terms and documents are shown with their similarity and with links to more details. Clicking on a term performs a query with only that term. Clicking on a document shows the detailed view of it. At the bottom of the screen is timing measurements of the current query in milliseconds, using the wall-clock time. Notice that changing the value of k takes effect immediately, without any long recalculation. This is because the matrices used when performing searches are cached in a way that facilitates this change.

5.4 Conclusion

This chapter has described the two different phases of using the system. First the construction of the desired index, and then how the index can be accessed. The general ideas behind the implementation have been covered, but most of the details have deliberately been left out. Refer to Appendix A for details of accessing the source code, and Appendix B for information on how to use the indexer and access the search systems.

The two different user interfaces were described. One is ready for end-users, but it does not yet have quite the amount of information desired, and the other giving all details, but without any nice packaging or clustering.

Having described the constructed system, we are now ready to perform a series of experiments in order to assess the usability and performance of the system.

Chapter 6

Experiments

Now that the details of the implementation have been described, it is time for a proper test of the system. This chapter contains details on the method of evaluation, the selected dataset and the results gained from the evaluation. In order to measure the retrieval performance, a selection of queries and expected responses were created. This enabled precise measurements of how good the system was at returning the expected results. The goal of this chapter will be to evaluate the performance of the system and find out if the problem in section 1.2 has been solved satisfactory. Retrieval performance experiments were performed both with and without stemming in order to compare the results of the two methods.

Section 6.1 describes how performance is measured in IR systems. The details of the data selected for experiments can be found in section 6.2. Section 6.3 details the experiments performed on the data, and the results. Finally, section 6.4 concludes the chapter.

6.1 Measuring performance in IR

Given a large enough collection of documents, finding the relevant ones becomes a problem. To properly evaluate the performance of an IR system, the collection of relevant documents given a query must be known. But if one can easily find all the relevant documents for a query, there really does not seem to be any reason to use the system. Hence, when the collection of documents becomes large enough, it becomes difficult to be certain that all relevant documents have been found. The problem of testing systematically is one of the reasons for the Text REtrieval Conference (TREC) conferences[1]. Here some huge datasets are given to the participants along with a set of

[1]http://trec.nist.gov

queries and the appropriate responses. The participants then compete to get the best performance using automated methods. To evaluate the different systems, the measurements in [Voorhees and Buckland, 2002, Appendix A] are used. In that chapter, they introduce a program called TREC_EVAL[2] used to calculate the performance measurements. TREC_EVAL performs the tedious calculations needed to find the precision and recall, which are both defined and described in section 6.1.1. See [Robertson, 2000] for a detailed account of evaluation in IR.

6.1.1 Precision and recall

Precision and recall are the traditional measurements for gauging the retrieval performance of an IR system. Precision is the proportion of retrieved material which is actually relevant to a given query, and recall is the proportion of relevant material actually retrieved:

$$\text{precision} = \frac{\#\text{relevant retrieved}}{\#\text{total retrieved}} \tag{6.1}$$

$$\text{recall} = \frac{\#\text{relevant retrieved}}{\#\text{total relevant in collection}} \tag{6.2}$$

These two measurements are defined in terms of each other, what is reported is the interpolated precision at recall levels of 10%, 20%, ..., 100%. These numbers are then plotted as a graph. Another measurement is to report the precision at a different number of documents retrieved, for instance at 5, 10, 20, and 100 retrieved documents. Finally, calculating the average precision over different levels of recall is also performed. The calculations are based on sets of queries and corresponding results. Using TREC_EVAL, the calculations are simple to perform as the output from the system and the expected results merely have to presented in the format appropriate for the utility.

6.2 Selected dataset

In order to facilitate the location of important components, only component collections of high quality and consistency have been included in the dataset. The JAVA API is one such collection and version 1.4.1 J2SE which has been used contains 2,723 classes. The CODEBROKER system, by comparison, used JAVA API version 1.1.8 which only had 503 classes. Besides the JAVA

[2]`ftp://ftp.cs.cornell.edu/pub/smart/`

System	Data	Docs	Terms	Methods	k
CodeBroker	Java 1.1.8, JGL 3.0	78,475	10,988	7,338	278
This, unstemmed	Java 1.4.1, Colt 1.0.3	27,609	6,735	27,609	148
This, stemmed	Java 1.4.1, Colt 1.0.3	27,609	4,171	27,609	225

Table 6.1: Size of the repository in comparison with CodeBroker.

API, version 1.0.3 of the the Colt collection of components was added. As was described in chapter 5, these components were successfully used when implementing the prototype. This component collection adds another 476 classes. Table 6.1 compares the sizes of the repository of CodeBroker with this project. As can be seen, much more text was indexed in CodeBroker, but with only a quarter as many retrievable components. The much higher number of terms in CodeBroker is also due to the larger amount of text indexed. The values for k in the last column will be described in section 6.3.1 on the following page.

6.3 Experiments performed

As there has not been enough time to conduct a proper series of real-world experiments involving end-users, the following will be a documentation of the lab-tests performed with the system. As [Robertson, 2000] puts it, there are two very distinct kinds of tests: lab-tests and real-world tests. In lab-tests, a fixed set of queries with corresponding suitable responses are used to evaluate the performance of the system. Examples of such sets are the TREC collections, or the automated tests used in this project. But in real-world tests, the user gives his or her subjective judgment of whether a given query was suitably answered. The results in this kind of test are much more “fuzzy”, by comparison to the clean lab-tests.

An inherent problem in IR is that the stated request is not the same as the actual information needs of the user, and therefore the relevance of a given document should be judged in relation to the information needs and not the request. This is in contrast to the lab-test environment where the relevant documents have already been determined and performance can be measured exactly. The queries constructed for this test each have a set of associated relevant methods, which should be highly ranked in a proper response.

```
topic = 4
query = Random a sequence
relevant = jdk9541 jdk9540 colt522
```

Figure 6.1: Example input query specification for performance test.

6.3.1 Queries with expected results

In order to measure the performance of the system, 19 queries with relevant components were constructed. The input queries were taken from [Ye, 2001, Appendix A], but with updated components related from the chosen component collections. In figure 6.1, a query usable for measuring system performance can be seen. Because the experiments were first carried out without stemming, and the original query from [Ye, 2001] was "Randomize a sequence", manual stemming was performed, resulting in the query "Random a sequence". The fact that the term `randomize` does not exist in the system merely reflects that not enough text has been indexed in order to give the system a good vocabulary. As is noted in [Deerwester et al., 1990, p. 404]:

> In theory, latent semantic analyses can extract at least some of the commonalities [in] usage of stemmed forms. In practice, we may often have insufficient data to do so.

An example of this is given in the section "*Testing the effect of stemming*" on page 80.

Each test query is stored in a separate standard JAVA property file as seen in figure 6.1. The query is given a unique topic number, a query string specifying the needed solution, though often in vague terms, and a set of document identifiers which have been deemed relevant to the query, listed in no particular order. Document identifiers can be found either by looking them up in the XML file generated by the indexing phase, or by using the actual system to search for the component. Since document identifiers are impractical to use, the benchmark program constructs a query report for each of the performance queries. An output corresponding to the query in figure 6.1 can be seen in figure 6.2 on the facing page. Here it can be seen that reasonable components have been specified as relevant to the query in that they solve the stated problem. The figure also shows the extracted query terms from the input string. Note that no stop-words have been used in the experiments, but this is an obvious set of experiments for future work.

```
Relevant to 4: "random a sequence"
Query terms: [sequence, random], skipped terms: [a]
Query terms (stemmed): [sequenc, random], skipped terms: [a]
jdk9541: java.util.Collections.shuffle(java.util.List,
                                        java.util.Random)
jdk9540: java.util.Collections.shuffle(java.util.List)
colt522: cern.colt.list.AbstractList.shuffle()
```

Figure 6.2: Expanded query specification for verification.

The displayed readable output is not used by the benchmark program, but is merely generated to make it easier to ensure that proper components have been listed as relevant. As this examples shows, it is easy to specify queries for measuring retrieval performance. Appendix C contains the complete list of 19 queries with expected results formatted similarly to figure 6.2.

In order to measure performance over several choices of k, each of the input queries are run against the system for a series of different values of k, with 1,000 results recorded for each query configuration. Refer back to section 3.3.2 for a discussion of the significance of the k value. In figure 6.3 on the next page, the magnitude of the singular values from the truncated SVD of X without stemming can be seen. The y-axis has a logarithmic scale in order to better show the large magnitude differences. The lower choices of k have large singular values which corresponds to strong concepts.

In figure 6.4 on the following page the average precision of all the queries can be seen in relation to the value of k. The optimal value of k for this particular collection of documents with the defined queries is around 148 when not using stemming. With stemming the optimal performance can be found around $k = 225$, see table 6.1 on page 75.

The interpolated precision at the 11 standard cutoff values are shown as a graph for both the unstemmed and the stemmed version of the experiment in figure 6.5 on page 79. Table 6.2 on page 79 shows the values which the graph depicts. The "non-interpolated" values at the bottom of the table reflects the performance over all relevant documents. A high value means that the system returns the relevant documents with high rank. As can be seen, the stemmed version performs slightly better that the unstemmed version on average. The stemmed version is superior especially on the low levels of recall. Unfortunately, both configurations seem to have rather poor recall with regard to the expected results, as can be seen in table 6.4 on page 82.

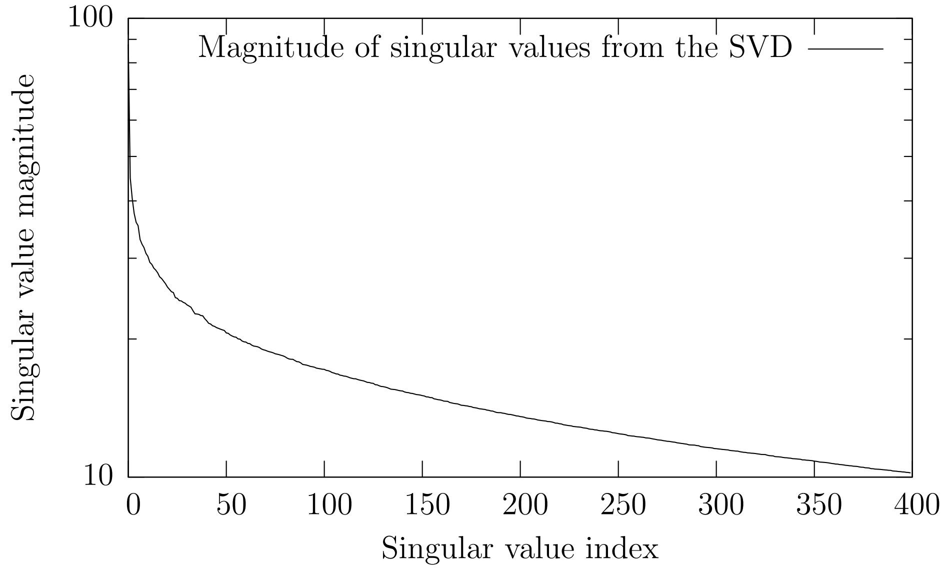


Figure 6.3: Relative magnitude of singular values of the DVM.

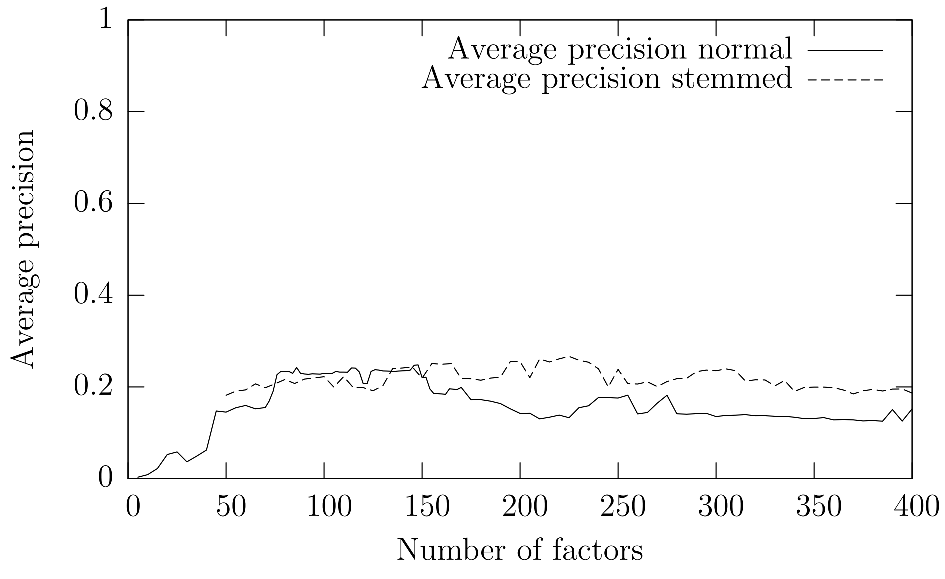


Figure 6.4: Precision as related to dimension of representation.

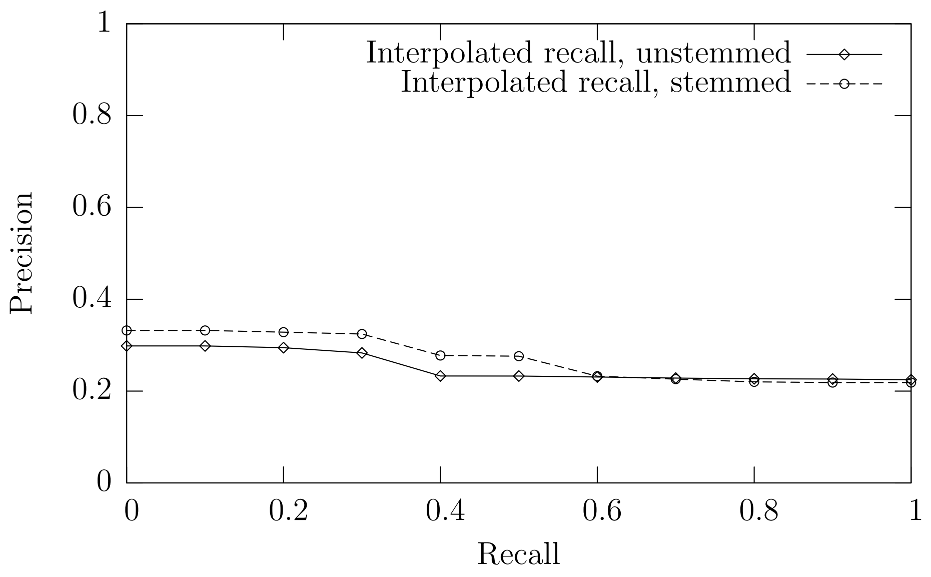


Figure 6.5: Precision and recall at optimal dimensionality for both configurations.

Recall Level Precision Averages

	Unstemmed	Stemmed
Recall	Precision	
0.00	0.2984	0.3321
0.10	0.2984	0.3321
0.20	0.2945	0.3284
0.30	0.2831	0.3243
0.40	0.2329	0.2775
0.50	0.2328	0.2760
0.60	0.2307	0.2324
0.70	0.2284	0.2261
0.80	0.2267	0.2201
0.90	0.2263	0.2186
1.00	0.2247	0.2186
Average precision over all relevant documents		
non-interpolated	0.2480	0.2666

Table 6.2: Precision and recall averages.

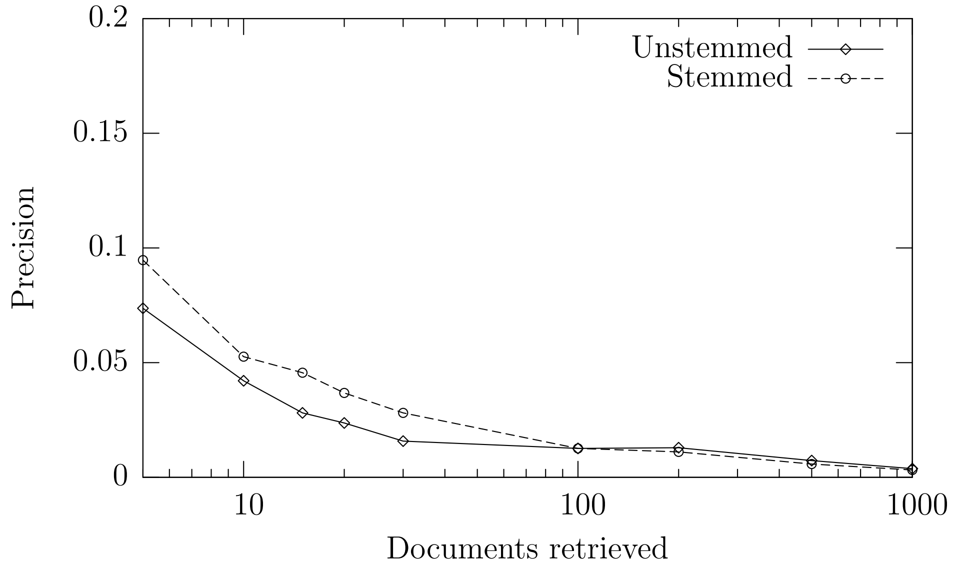


Figure 6.6: Document level average precision.

Testing the effect of stemming

Since the performance of the system has not been very impressive, judging by the numbers, stemming was attempted in order to get higher recall. Usually the use of stemming leads to an increase in recall at the expense of a decrease in precision, as reported in section 3.2.2. However, the following experiments were performed in order to see if stemming would have a positive effect on the performance of the system, as the amount of indexed data seems to be insufficient to properly have identified different inflections of terms. An example which was discovered when performing the benchmarks is the two terms `string` and `strings`. Searching for one does not result in retrieval of documents using the other without stemming, as was otherwise expected.

In figure 6.6 the performance of the system is measured after a number of documents have been retrieved. Notice the range of the y-axis does not go all the way up to one, as the values are fairly small and would be indistinguishable otherwise. Table 6.3 on the facing page has the data used to make the graph. It also contains two values labeled R-precision. These are the precisions after the number of relevant documents have been retrieved. As it detracts from the importance of the exact ranking, it is more useful when there are many relevant documents to each query – this is not the case in these experiments, and it is included only for completeness.

To better understand the performance of the two configurations, the

Document Level Averages

	Unstemmed	Stemmed
	Precision	
At 5 docs	0.0737	0.0947
At 10 docs	0.0421	0.0526
At 15 docs	0.0281	0.0456
At 20 docs	0.0237	0.0368
At 30 docs	0.0158	0.0281
At 100 docs	0.0126	0.0126
At 200 docs	0.0129	0.0111
At 500 docs	0.0073	0.0058
At 1000 docs	0.0038	0.0032
	R-Precision	
Exact	0.2478	0.2018

Table 6.3: Document level average precision.

performance for each of the 19 queries is shown in table 6.4 on the next page. The first column is the query number, followed by the number of relevant components chosen for the query. Then follows columns with the number of found relevant components within the first 1,000 results returned and the average precision resulting from the query. These measurements are from the optimal representation dimensions documented earlier, see table 6.1. The last row of the table contains the total relevant components and the averages of the precisions.

Notable differences in performance with and without stemming can be seen in queries 5 and 10. All relevant components are found for query 5 within the first 1,000 results returned when stemming is used. On the other hand, none of the relevant components are found for query 10 when stemming is enabled. This demonstrates nicely the fact that stemming helps to a certain degree, but in some cases it decreases the performance.

In table 6.5 on page 83 the first 20 results returned for query 11 matched with stemming can be seen. The component ID is shown for each, and the similarity to the query is also shown. The full path to the method is created by putting together the package name, class name and the method name, as seen in the column “Method”. The first two results are the relevant ones, marked with •. There are no missing relevant methods. All methods are shown without parameters, so the methods at indices 15 and 19 are not the same, as they have different signature when looking at the full method specification.

		Unstemmed		Stemmed	
ID	Relevant	Found	Avg. precision	Found	Avg. precision
1	1	1	1.0000	1	0.5000
2	8	6	0.0102	8	0.0271
3	1	1	1.0000	1	1.0000
4	3	3	0.0060	3	0.0040
5	2	1	0.0147	2	0.0346
6	3	3	0.0203	3	0.0153
7	5	5	0.0131	5	0.0094
8	5	1	0.0002	5	0.0632
9	4	3	0.0075	3	0.0286
10	20	20	0.0544	0	0.0000
11	2	2	1.0000	2	1.0000
12	3	3	0.3333	3	0.3746
13	8	8	0.0587	8	0.0279
14	1	1	1.0000	1	0.3333
15	2	2	0.0077	2	0.0105
16	4	4	0.0850	4	0.6136
17	6	6	0.0174	6	0.0171
18	1	1	0.0526	1	1.0000
19	3	2	0.0022	2	0.0072
All	82	73	0.2480	60	0.2666

Table 6.4: Details of the individual queries. See also Appendix C.

6.3.2 Comparing clustering methods

The three clustering methods available in Carrot at the moment are AHC, STC, and LINGO. They have very different performance and output, even though results are presented using the same visualization. To compare the methods, the set of top 20 documents from query 11 listed in table 6.5, was put through each of the three clustering algorithms. See Appendix C for additional details on the queries. The following sections will look at the generated labels for the clusters and the general grouping of the documents. The fact that stemming was used to produce the results in table 6.5 makes no difference in the following, as clustering is a completely separate process.

The tables created to document these clustering algorithms can be hard to take in, as a lot of information is presented and the tables have to compared in order to be understood. They are presented for completeness, and to enable easier comparison between the algorithms.

#	ID	Rel.	Sim.	Method
1	jdk4828	●	0.63	`java.io.File.mkdir()`
2	jdk4829	●	0.60	`java.io.File.mkdirs()`
3	jdk11881	○	0.48	`javax.naming.directory.DirContext.` `createSubcontext()`
4	jdk4815	○	0.47	`java.io.File.isDirectory()`
5	jdk11026	○	0.47	`javax.imageio.ImageIO.setCacheDirectory()`
6	jdk4816	○	0.47	`java.io.File.isFile()`
7	jdk4808	○	0.46	`java.io.File.getCanonicalPath()`
8	jdk4821	○	0.45	`java.io.File.delete()`
9	jdk11907	○	0.44	`javax.naming.directory.InitialDirContext.` `createSubcontext()`
10	jdk14702	○	0.44	`javax.swing.filechooser.FileSystemView.` `createNewFolder()`
11	jdk23539	○	0.43	`org.omg.CORBA.ORB.create_environment()`
12	jdk4825	○	0.42	`java.io.File.listFiles()`
13	jdk14714	○	0.42	`javax.swing.filechooser.FileSystemView.` `getParentDirectory()`
14	jdk4823	○	0.48	`java.io.File.list()`
15	jdk4835	○	0.40	`java.io.File.createTempFile()`
16	jdk15211	○	0.40	`javax.swing.JFileChooser.` `setCurrentDirectory()`
17	jdk4819	○	0.40	`java.io.File.length()`
18	jdk14715	○	0.39	`javax.swing.filechooser.FileSystemView.` `createFileSystemRoot()`
19	jdk4834	○	0.39	`java.io.File.createTempFile()`
20	jdk4833	○	0.39	`java.io.File.listRoots()`

Table 6.5: Document descriptions, top 20 for query 11, stemmed version. Original query was “create a directory on a floppy disk”.

Cluster labels

The clusters labeled as seen in table 6.6 on the next page are generated by running the list of documents in table 6.5 through each of the three clustering algorithms. As seen in the table, the types of cluster labels are quite distinct for the three methods. LINGO uses a degree of soft clustering, which can be seen in the total count of clustered documents which is 23, and only 20 documents were given as input. The generated labels are all short, but somewhat descriptive.

For the STC algorithm, the labels are lists of substrings relevant to the contained documents. Notice how the 20 input documents now are represented with 37 instances in clusters. This means that there is a lot of redundancy in the clusters, which we will see in "*Cluster contents*", next. Finally, the AHC algorithm creates a deep hierarchy, even for only 20 documents. The labels for the clusters are used to reflect the hierarchy, so A1.1.2 is a child cluster to A1.1, and sibling to A1.1.3. AHC uses hard clustering, which can be seen in that each document only occurs once in the tree. Moving down through the categories, the contents becomes subsets of the contents of the parent cluster.

Cluster contents

To get a better feeling for how the documents are clustered, take a look at table 6.7 on page 86. This table shows the placement of documents in clusters. Compare with table 6.5 to see what the documents are, and table 6.6 for the associated cluster labels. For instance, the cluster A1.1.2 contains only the two documents relevant to the query (JDK4828 and JDK4829), however, this cluster is three levels deep, so navigating to exactly that cluster in a first attempt is unlikely. The AHC hierarchy is deep, and which makes it difficult to gauge which path is relevant at any given moment in the navigation. Using STC results in some long labels. When clustering more than 20 documents it becomes even worse.

The LINGO algorithm gives few clusters and the contents is generally coherent in that the documents are usually similar, objectively.

Results of clustering

Of the three available methods, the LINGO algorithm clearly produced the most coherent and usable clusters with the standard configuration, especially because there was no deep hierarchy. Unfortunately, the names of clusters are unfortunately not very intuitive with the current configuration. It has been difficult to document the output of clustering, but hopefully the

Cluster	Elements	Cluster title
		Lingo
L1	6	Denoted by this Abstract Pathname
L2	3	Parent Directory
L3	2	File Objects
L4	2	Attributes
L5	2	Value
L6	5	Array of Strings
L7	5	(Other)
7	23	Total listed
		STC
S1	16	files, directories, pathname
S2	7	default, using, null
S3	10	creating
S4	4	file denoted by this abstract pathname, denoted by this abstract pathname
4	37	Total listed
		AHC
A1	16	Abstract Pathname
A1.1	9	Denoted by this Abstract Pathname
A1.1.1	5	File or Directory, Returns, String
A1.1.1.1	2	Pathname Denotes a Directory
A1.1.1.2	2	Included, Parent, Pathname Does not Denote a Directory , then this Method
A1.1.2	2	Creates the Directory Named by this Abstract Pathname
A1.1.3	2	Tests whether the File Denoted by this Abstract Pathname
A1.2	4	Method
A1.2.1	2	Invoking, Null, Generate
A1.2.2	2	File Objects, File system Root Directory
A1.3	3	Parent, Sets, Null
A2	4	New
A2.1	2	Creates and Binds a New Context , along with Associated, Createsubcontext name , Attributes for Details
13	20	Total listed

Table 6.6: Cluster labels for the 20 documents listed in table 6.5.

#	Method	Rel.	L1	L2	L3	L4	L5	L6	L7	S1	S2	S3	S4	A1	A1.1	A1.1.1	A1.1.1.1	A1.1.1.2	A1.1.2	A1.1.3	A1.2	A1.2.1	A1.2.2	A1.3	A2	A2.1
1	mkdir()	●	●	○	○	○	○	○	○	●	○	●	○	●	●	○	○	○	●	○	○	○	○	○	○	○
2	mkdirs()	●	○	●	○	○	○	○	○	●	○	●	○	●	●	○	○	○	●	○	○	○	○	○	○	○
3	createSubcontext()	○	○	○	○	●	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	●	●
4	isDirectory()	○	●	○	○	○	○	○	○	●	○	○	●	●	●	○	○	○	○	●	○	○	○	○	○	○
5	setCacheDirectory()	○	○	○	○	○	●	○	○	●	●	●	○	●	○	○	○	○	○	○	○	○	○	●	○	○
6	isFile()	○	○	○	○	○	○	○	●	●	○	●	●	●	●	○	○	○	○	●	○	○	○	○	○	○
7	getCanonicalPath()	○	○	○	○	○	○	○	●	●	○	○	○	●	●	●	○	○	○	○	○	○	○	○	○	○
8	delete()	○	●	○	○	○	○	○	○	●	○	○	●	●	●	●	●	○	○	○	○	○	○	○	○	○
9	createSubcontext()	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	●
10	createNewFolder()	○	○	○	○	●	○	○	●	○	●	●	○	○	○	○	○	○	○	○	○	○	○	○	●	○
11	create_environment()	○	○	○	●	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	●	○
12	listFiles()	○	●	○	○	○	○	●	○	●	●	○	○	●	●	●	○	●	○	○	○	○	○	○	○	○
13	getParentDirectory()	○	○	●	●	○	○	○	○	●	○	○	○	●	○	○	○	○	○	○	○	○	○	●	○	○
14	list()	○	●	○	○	○	○	●	○	●	●	○	○	●	●	●	○	●	○	○	○	○	○	○	○	○
15	createTempFile()	○	○	○	○	○	○	○	●	●	●	●	○	●	○	○	○	○	○	○	●	●	○	○	○	○
16	setCurrentDirectory()	○	○	●	○	○	○	○	○	●	●	○	○	●	○	○	○	○	○	○	○	○	○	●	○	○
17	length()	○	●	○	○	○	●	○	○	○	○	○	●	●	●	●	●	○	○	○	○	○	○	○	○	○
18	createFileSystemRoot()	○	○	○	○	○	○	○	○	●	○	●	○	●	○	○	○	○	○	○	●	○	●	○	○	○
19	createTempFile()	○	○	○	○	○	○	○	●	●	●	●	○	●	○	○	○	○	○	○	●	●	○	○	○	○
20	listRoots()	○	○	○	●	○	○	○	○	●	○	○	○	●	○	○	○	○	○	○	●	○	●	○	○	○

Table 6.7: Clustering of the list of documents in table 6.5.

above discussion has given an idea of the strengths and weaknesses of the various methods.

When testing the system, the clustering has proven helpful. Still, the parameters of the algorithms have to be tuned a bit more carefully to get the best possible performance. It is difficult to tell what a cluster contains by simply looking at the label. However, using the clustering interface does enable one to skip large groups of components at a time. Even though the labels are not very good, when opening a category it is easy to gauge if anything it contains is going to be interesting.

6.4 Conclusion

This chapter introduced the standard methods for evaluating Information Retrieval systems. In order to assess the performance of the system, a collection of queries with relevant responses were created. Looking at the precision recall graph shows a system which unfortunately has low average precision. A more detailed analysis of the individual queries showed how some are handled extremely well, while others are not properly answered at all, with regards to the expected results. Using stemming resulted in an overall increase in precision, but also a slightly lower recall. The precision was higher within the first 100 retrieved documents, which means that the results a user is most likely to see are of slightly higher quality.

Evaluation of the clustered results is less easy to conclude upon. Some difficulties were discovered when evaluating the output, most importantly the quality of the cluster labels. In order to be truly useful, these have to improve.

Generally it seems like the system needs more text to work on in order to do a proper synonym association. This would probably negate the currently positive effects of stemming and generally give better recall rates.

Having concluded upon the experiments, it is now time to conclude upon the results of the entire project.

Chapter 7

Conclusion

Here the entire project will be concluded, specifically on what was said in the introduction, with focus on the problem description.

7.1 Conclusion

The goal in this project was to create an easy to use reuse support system with automatic support for synonyms, and grouping of search results. This goal has been achieved with the constructed system. The documented experiments shows that the retrieval performance is acceptable, and clustering does provide some help in filtering contents.

As an added bonus of having solved the stated problem, a Vector Space Model-based search engine using LSI has been constructed. This engine is usable in other contexts with some minor changes, and for instance using it with components from a different programming language is easy to do.

7.1.1 Important points

Important points from this project include:

- The reuse support system has been built on existing technology where available. The author has constructed the entire pre-processing stage, interfaced with existing solutions where applicable and created multiple views of the repository for experiments and exploration. The constructed system is modular in design, and it is easily extensible. Work is far from completed – many ideas still not implemented, and other algorithms such as BM25 should be implemented for comparing and possibly improving performance. Section 7.2 on the next page contains many ideas for possible future extensions.

- Reasons for design decisions have been given, and relevant literature has been cited throughout the text. Details and results from many sources have been brought together in order to build on top of previous results instead of starting from scratch.
- The clustering of search results works, but label generation is not properly tuned to this application yet. However, the contents of the clusters actually makes a lot of sense most of the time, so they are far from useless. Clustering algorithms are used in a *black-box* fashion, and better tuning could probably be done by looking more closely at the algorithms.
- With chosen set of data, stemming helped on retrieval performance, contrary to the expectation. It is speculated that this benefit will disappear when larger sets of documents are indexed.
- Strengths of the reuse support system include that building the index is unsupervised and therefore cheap and easy to do. The system is written in Java and the supporting utilities are all platform independent, or can easily be ported. The fact that the system is built on the well-documented LSI method is also worth mentioning.
- It is a weakness of the created system that it is currently not possible to update the contained documents, and that the entire component collection has to be kept in memory. Having everything in main memory does give great performance, but at the cost of high memory demands.
- A few reusable components have resulted from the project, Harwell-Boeing sparse matrix output being the most obvious.
- Since the interface to Carrot is only one of the interfaces, other result presentations are easy to make.
- The source code will be released under the GPL.

To access an online demonstration version of the system, open the address `http://www.daimi.au.dk/~rud/speciale/demo/`. This is not a permanent online systemURL, and it will only work until the examination date.

7.2 Future work

This section contains ideas for future work which would be interesting directions for the project. Some of the ideas are still fairly rough, but striving toward better integration and usability are some of the key ideas expressed here.

7.2.1 Relevancy feedback

Add the possibility of relevancy feedback so that the user can easily reformulate the query using the relevant documents discovered. This is easy to implement, and gives good results, see [Letsche and Berry, 1997]. It can be implemented by giving the user a choice of rating each returned result as relevant or irrelevant – and then submitting this information to the system. It is then possible to form a new pseudo-document consisting of information from the relevant documents but excluding the irrelevant ones. This new query usually gives a much more precise picture of the solution the user is looking for than the initial usually very limited list of search-terms.

One way of implementing this marking of relevant documents would be by allowing the user to keep a “document basket”. The basket can then have a “find related” link which performs the search using the collection of all the documents in the basket. This is a good example of an already known metaphor reused in an interesting context.

Find similar on search results Add a “find similar” link to each search-result. This is a simple form of feedback mechanism, and simply taking the selected document as the query makes searching the neighborhood very easy. Adding a similar “find random” button could encourage the user to explore the system in order to locate unknown components. This is, however, a step in the direction of “Today’s hint” in various Office packages which are usually disabled by users as quickly as they figure out how to do it. The reason why is that because the fragment of information has no connection with the current problem situation of the user, it is not relevant and only seen as a distraction.

7.2.2 Additional experiments

Aside from the experiments described in Chapter 6, other configurations could be tried to see how it would affect the performance.

Stop-words Due to an oversight when performing the documented experiments, stop-words were included in the index. By eliminating these from the indexing phase, the system could potentially get a little better performance.

Weighting methods In the experiments performed here, the logarithm and entropy weighting method was used after the recommendations in [Dumais, 1992]. Using the TFIDF or other combinations of weighting would be interesting to see how it affects performance with this set of data.

Clustering labels The current use of the Carrot framework is simple black-box reuse. By properly tuning the parameters for the Lingo clustering algorithm, better cluster labels should be possible to generate. Judging from the labels created when clustering web-results, it is simply a matter of finding the right parameter values [Osiński, 2003].

Experiment configuration In order to facilitate future experiments, it should be made easier to change the configuration. The current framework-based model is great while developing the system, but it is cumbersome to recompile the system just to try out a different weighting method. Allowing configurations defined in simple property-files would make the configuration of experiments much easier.

7.2.3 Expanding Javadoc

Find related in Javadoc An idea would be to augment the original Javadoc HTML documents with "find related" links for each method and class. The links could either be to static HTML pages or directly into the search system. This way old and new information can be integrated, and if the static solution is chosen, the resulting set of documents can be used without running any kind of special server for it. This gives even greater portability, and requires only disk-space for the search results. Augmenting a given set of Javadoc files will be easy to do. If the dynamic version is chosen instead, it will be possible to jump back and forth between the traditional Javadoc and the searching system. This allows each system to be developed independently and still gets the best of both worlds.

Additional navigation Another extension along similar lines is creating a web-page with a new frameset which is used to contain the Javadoc display. This new frame will then contain related information based on the currently shown object or function. Looking up and mapping the keys back and forth should pose no major problem as the naming in Javadoc of elements is predictable. Creating this interactivity using a bit of JavaScript which calls a server-side script should be easy. Keeping a list of favorite packages and methods in this window would be an obvious feature – this way the user can quickly find the most often used items while still having the search functionality. The "find related" could also be placed in this window for maximum portability. The active Javadoc window of course has to be indexed by the system for this to work.

Support filtering of components Add some filtering to the system and allow the user to make, for instance, package restrictions. This could be an advanced user model as in the CodeBroker system. This means that each user has a profile, stored in a cookie between sessions for instance. This profile can be used to remove unwanted components by defining a filter which either says what to include or what to exclude, for instance "`search only java.util.*`" would be a filter limiting searches to a single package, and "`exclude *.CORBA.*`" would leave out all components in CORBA packages.

7.2.4 Expanding data

Add other documentation Add relevant text resources not directly retrievable to the index – this is referred to as *dead text* in the following. Large amounts of indexed text is known to cause the LSI to better distinguish and compare the different words. As the system indexes more text, the definitions of the different concepts improves. By, for instance, adding the text of a few Java books, the system will get better at associating terms programmers are familiar with. Documents could be created for each paragraph, for instance. These document vectors are then added to the document vector matrix (DVM), and included when performing the SVD but excluded when matching queries. An easy way to implement this is to use the feature in Colt to make a view of a part of a matrix. This will allow the system to have more known indexed terms, and should also be tested both with and without stemming. It would be a good idea to look at the term-term similarity to determine how this would affect the general associativity of terms.

Add class documentation Along similar lines, add the general description of each class from Javadoc to the index, at least as *dead text* using the principle described above. It could also be used to make classes retrievable and not just the methods within the classes as it is now. It should give a better definition of the terms and more context for the methods. This can be done using a few lines of regular expressions in the initial HTML parsing phase, and by expanding the XML parsing phase.

Increase modularity Use more than one XML file for input to the parser. This will enable modularization of the input files and allow easy customization of the system, as an XML file can be created for a subset of some Javadoc documented set of packages. A good example could be to skip the CORBA section from the standard API, but include a specific user-created library. Along similar lines, it could be an obvious extension to allow the user to choose between different indexed libraries. Normally one would have it all in one big repository, but projects could need specific documentation separate from the main documentation. Allowing the user to switch more easily than now should be easy to add. Similarly, it could be an idea to add better support for Javadoc from multiple sources, so that links to multiple sources of components can more easily be created.

Indexing convenience To make the system easier to use, the possibility of indexing online documentation automatically should be added. This way, by given a single URL, the system could index and create proper links without any further configuration. Currently both a local path and an URL has to be given for the documentation.

Updating matrix online Instead of having to recompute the entire SVD, it is possible to perform updates of the matrices. [Berry et al., 1994] gives details of such a method.

7.2.5 Performance enhancements

Computation parallelization Using multiple threads to solve queries in parallel could probably speed up the returning of results on systems with multiple CPU. Even though the entire list of results has to be sorted before the results can be returned in globally ordered ranking, some optimization is still possible. The comparison with all document vectors is currently linear, comparing with one column at a time. This is easily converted into a parallel calculation where each computer can work on a fixed subrange of the matrix. This would give a great speedup, but as the processing time for the entire comparison is currently around 1-2 seconds, it would only be necessary for much larger repositories.

Inverted indexes for lookup As the entire model is currently held in main memory, the memory requirements are large, and the approach does not scale to much larger collections. By keeping the terms and retrievable documents in an inverted index, it would be possible to reduce this memory consumption since only the needed parts would be fetched from disk. The inherent scaling problem in the Vector Space Model is that all document vectors have to be compared to the query, so keeping them in memory does seem like a good idea.

7.3 Final thoughts

This chapter has concluded on the entire project. A lot of ideas for future work has been presented, and there are many useful extensions which could be made to the system.

Appendix A

Directory layout

This appendix covers the general layout of the enclosed CD.

`/report/`	`complete report as pdf`
`/src/`	`root source folder`
`    /script/`	`index building scripts`
`    /data/`	`example data, ready for use`
`/src/carrot/`	`complete source for the Carrot project, with the constructed Javadoc input component added`
`/test/input/`	`test cases, stored as property files`
`/test/output/`	`output produced when running the test cases`
`/webapps/`	`prepackaged .war files for deployment on the Tomcat 4 webserver`

Appendix B

Running the system

This appendix describes how the system can be used to index JAVADOC documentation and how to configure the system for access using a web-browser.

As mentioned in the conclusion, there is an online demonstration version of the system at `http://www.daimi.au.dk/~rud/speciale/demo/`. It will only be online for a limited time.

The following assumes some form of UNIX as the platform. This project was developed under MAC OS X 10.0.3.

B.1 Prerequisites

Ensure the following are installed:

- BASH shell (2.05b.0(1) used),
- Tool Command Language (TCL) shell (tclsh),
- JAVA 1.4 (1.4.2 used),
- APACHE ANT build tool, recent version (1.6.1 used),
- TOMCAT 4 webserver (4.1.30 used), and
- SVDLIBC[1] program is assumed to be in the path.

Extract source code archive. Call the base directory `$BASE`. An example could be `$BASE = ~/rud-thesis/src/`.

[1]`http://tedlab.mit.edu/~dr/SVDLIBC/`

B.2 Indexing

Generate JAVADOC if not already existing. Notice that all methods listed in the JAVADOC is added to the index. This makes it easy to index `private` parts of classes by simply generating JAVADOC including private elements. Locate root directory of documentation tree, call this `$DOCROOT`. Ensure that `$DOCROOT` is accessible from the web. Let us store the public URL to the documentation in `$URL`. Now choose a name for the index, call this `$INDEX`. An example of this could be "`textindex`". This is created as a directory, as the various programs will store intermediate files in the same directory as the generated XML file. Now execute the following:

```
cd $BASE
mkdir data/$INDEX
script/crawler.sh $DOCROOT > data/$INDEX/$INDEX.xml
```

This generates the XML representation of the JAVADOC files. The available options for the crawler can be seen in figure figure B.1 on page 102.

The second part of the indexing is done as follows, where 100 is the maximum value for k, singular values calculated:

```
script/parser.sh -t 100 -u $URL data/$INDEX/$INDEX.xml
```

If more dimensions are needed at a later point, the process can be restarted from the SVD calculation thus:

```
script/parser.sh -t 300 -s "svd post-svd" -u $URL \
  data/$INDEX/$INDEX.xml
```

This way only the `svd` and `post-svd` stages of the processing is performed. Please note at this point that it is not necessary to recalculate the SVD in order to use fewer dimensions in the representation. This is available in the web-interface.

Now the final index has been created. It is stored in a file named:

```
data/$INDEX/$INDEX.xml.out-post.searchvector
```

This is a serialized instance of a `SearchVector` object, usable from the search engine. We are now ready to start the web-interface.

B.3 Accessing the search engine

First of all, ensure that the Tomcat engine is able to start. Then copy all the `*.war` files from `$BASE/webapps` to `/webapps/` under the Tomcat installation. If a different path to the generated datafile is used than the standard one, edit the file at

```
$BASE/src/carrot/components/inputs/javadoc-input/web/WEB-INF/web.xml
```

to point to the correct path. It is the parameter called `searchVectorDataFile`. After this file has been changed, the component has to be built again using Ant. Type

```
cd $BASE/src/carrot/
ant build
```

The generated `*.war` files in `$BASE/src/carrot/tmp/` has to be copied as above to the Tomcat deployment directory.

Ensure that Tomcat has enough memory to run the engine by typing (assuming Bash shell):

```
export JAVA_OPTS=-Xmx800m
```

This sets the upper bound of the memory consumption to 800 MB, which is usually more than enough.

Assuming that Tomcat runs on localhost:8080, the search service should now be available at:

```
http://localhost:8080/carrot2-web-controller/
```

and the back-end exploratory interface is available at:

```
http://localhost:8080/carrot2-javadoc-search/
```

```
file crawler
usage: script/crawler.sh [-h] [-d] [-p parser] [-f prefix]
  [-s start] startdir

Options are:
-h          for this help
-d          enable debugging information
-p          set parser to the given command
-f          set the id prefix string to the given, default jdk
-s          set the id start int to the given value, default 1
startdir    is where the crawling should start
            Note: the generated XML will be output to stdout
```

Figure B.1: CRAWLER.SH commandline options.

```
XML parser - turns a proper XML-file into a reduced matrix
  representation
usage: script/parser.sh [-h] [-d] [-t svdtriplets]
  [-p precision] [-s stages] [-m magnitude] [-u urldir]
  [-o outputfile] xmlfile

Options are:
-d       enable debug output
-h       for this help
-t       set the wanted number of SVD triples (default: 5)
-m       set maximum magnitude for calculated eigenvalues
         (default: 1e-10)
-u       url dir for javadoc links (default: "")
-o       output file base-name (default: xmlfile.out)
-s       stages to perform (default: complete-svd, other
         values are: pre-svd, svd, post-svd)
xmlfile  the input file to parse
```

Figure B.2: PARSER.SH commandline options.

Appendix C

Example queries

This appendix contains the complete list of 19 queries used to benchmark the performance of the system. For each the query string and extracted terms is shown. The list of relevant components is also given with the key and associated component.

C.1 Queries with formatted results

```
Relevant to 1: "judge if a file is a directory"
Query terms: [directory, file, if, is], skipped terms: [judge, a]
Query terms (stemmed): [file, directori, if, is], skipped terms: [a, %
      judg]
jdk4815: java.io.File.isDirectory()

Relevant to 2: "split a string into substrings"
Query terms: [substrings, string, into, split], skipped terms: [a]
Query terms (stemmed): [string, into, substr, split], skipped terms: [a]
jdk5938: java.lang.String.substring(int)
jdk5939: java.lang.String.substring(int, int)
jdk5946: java.lang.String.split(java.lang.String, int)
jdk5947: java.lang.String.split(java.lang.String)
jdk10278: java.util.StringTokenizer.hasMoreTokens()
jdk10279: java.util.StringTokenizer.nextToken()
colt4891: ViolinStrings.Strings.split(java.lang.String)
colt4892: ViolinStrings.Strings.split(java.lang.String, int)

Relevant to 3: "determine if it is a leap year"
Query terms: [year, it, determine, if, is, leap], skipped terms: [a]
Query terms (stemmed): [year, it, if, is, leap, determin], skipped %
      terms: [a]
jdk9634: java.util.GregorianCalendar.isLeapYear(int)
```

```
Relevant to 4: "random a sequence"
Query terms: [sequence, random], skipped terms: [a]
Query terms (stemmed): [sequenc, random], skipped terms: [a]
jdk9541: java.util.Collections.shuffle(java.util.List, java.util.Random)
jdk9540: java.util.Collections.shuffle(java.util.List)
colt522: cern.colt.list.AbstractList.shuffle()

Relevant to 5: "return true if the string is capital"
Query terms: [the, string, true, if, return, capital, is], skipped %
      terms: []
Query terms (stemmed): [the, capit, string, true, if, return, is], %
      skipped terms: []
jdk5340: java.lang.Character.isTitleCase(char)
colt4860: ViolinStrings.Strings.isTitleCase(java.lang.String)

Relevant to 6: "Change the file name"
Query terms: [the, file, name, change], skipped terms: []
Query terms (stemmed): [the, file, name, chang], skipped terms: []
jdk4812: java.io.File.canRead()
jdk4813: java.io.File.canWrite()
jdk4830: java.io.File.renameTo(java.io.File)

Relevant to 7: "create a random sequence from the original one"
Query terms: [the, one, create, original, from, sequence, random], %
      skipped terms: [a]
Query terms (stemmed): [the, creat, from, origin, sequenc, random, %
      on], skipped terms: [a]
colt4514: jal.Object.Modification.random_shuffle(java.lang.Object[], %
      int, int, java.util.Random)
colt4515: jal.Object.Modification.random_shuffle(java.lang.Object[], %
      int, int)
jdk9541: java.util.Collections.shuffle(java.util.List, java.util.Random)
jdk9540: java.util.Collections.shuffle(java.util.List)
colt522: cern.colt.list.AbstractList.shuffle()

Relevant to 8: "append two strings"
Query terms: [append, strings, two], skipped terms: []
Query terms (stemmed): [append, string, two], skipped terms: []
jdk5975: java.lang.StringBuffer.append(java.lang.Object)
jdk5977: java.lang.StringBuffer.append(java.lang.StringBuffer)
jdk5978: java.lang.StringBuffer.append(char[])
jdk5979: java.lang.StringBuffer.append(char[], int, int)
jdk5941: java.lang.String.concat(java.lang.String)

Relevant to 9: "get the intersection of two sets"
Query terms: [the, of, sets, two, get, intersection], skipped terms: []
Query terms (stemmed): [the, of, two, set, get, intersect], skipped %
      terms: []
jdk9530: java.util.Collection.retainAll(java.util.Collection)
```

```
jdk10231: java.util.Set.retainAll(java.util.Collection)
colt4542: jal.Object.Sorting.set_intersection(java.lang.Object[], %
       java.lang.Object[], java.lang.Object[], int, int, int, int, int, %
       jal.Object.BinaryPredicate)
colt4760: jal.String.Sorting.set_intersection(java.lang.String[], %
       java.lang.String[], java.lang.String[], int, int, int, int, int, %
       jal.String.BinaryPredicate)

Relevant to 10: "delete common elements from a sequence"
Query terms: [from, delete, sequence, common, elements], skipped %
       terms: [a]
Query terms (stemmed): [element, from, delet, sequenc, common], %
       skipped terms: [a]
colt4505: jal.Object.Modification.unique(java.lang.Object[], int, int)
colt4506: jal.Object.Modification.unique(java.lang.Object[], int, int, %
       jal.Object.BinaryPredicate)
colt4599: jal.SHORT.Modification.unique(short[], int, int)
colt4600: jal.SHORT.Modification.unique(short[], int, int, %
       jal.SHORT.BinaryPredicate)
colt3761: jal.BYTE.Modification.unique(byte[], int, int)
colt3762: jal.BYTE.Modification.unique(byte[], int, int, %
       jal.BYTE.BinaryPredicate)
colt3885: jal.CHAR.Modification.unique(char[], int, int)
colt3886: jal.CHAR.Modification.unique(char[], int, int, %
       jal.CHAR.BinaryPredicate)
colt4009: jal.DOUBLE.Modification.unique(double[], int, int)
colt4010: jal.DOUBLE.Modification.unique(double[], int, int, %
       jal.DOUBLE.BinaryPredicate)
colt4133: jal.FLOAT.Modification.unique(float[], int, int)
colt4134: jal.FLOAT.Modification.unique(float[], int, int, %
       jal.FLOAT.BinaryPredicate)
colt4257: jal.INT.Modification.unique(int[], int, int)
colt4258: jal.INT.Modification.unique(int[], int, int, %
       jal.INT.BinaryPredicate)
colt4381: jal.LONG.Modification.unique(long[], int, int)
colt4382: jal.LONG.Modification.unique(long[], int, int, %
       jal.LONG.BinaryPredicate)
colt4723: jal.String.Modification.unique(java.lang.String[], int, int)
colt4724: jal.String.Modification.unique(java.lang.String[], int, int, %
       jal.String.BinaryPredicate)
colt4507: jal.Object.Modification.unique_copy(java.lang.Object[], %
       java.lang.Object[], int, int, int)
colt4508: jal.Object.Modification.unique_copy(java.lang.Object[], %
       java.lang.Object[], int, int, int, jal.Object.BinaryPredicate)

Relevant to 11: "create a directory on a floppy disk"
Query terms: [directory, create, disk, floppy, on], skipped terms: [a]
Query terms (stemmed): [floppi, creat, directori, disk, on], skipped %
       terms: [a]
```

```
jdk4828: java.io.File.mkdir()
jdk4829: java.io.File.mkdirs()

Relevant to 12: "add days to a Date object"
Query terms: [days, to, object, add, date], skipped terms: [a]
Query terms (stemmed): [to, object, add, dai, date], skipped terms: [a]
jdk9499: java.util.Calendar.add(int, int)
jdk9500: java.util.Calendar.roll(int, boolean)
jdk9501: java.util.Calendar.roll(int, int)

Relevant to 13: "create a random number between two numbers"
Query terms: [numbers, between, create, two, random, number], skipped %
       terms: [a]
Query terms (stemmed): [creat, between, two, random, number], skipped %
       terms: [a]
jdk8099: java.security.SecureRandom.setSeed(long)
jdk8101: java.security.SecureRandom.next(int)
jdk10165: java.util.Random.setSeed(long)
jdk10166: java.util.Random.next(int)
jdk10168: java.util.Random.nextInt()
jdk10169: java.util.Random.nextInt(int)
colt2114: cern.jet.math.IntFunctions.random()
colt2351: cern.jet.random.Uniform.nextIntFromTo(int, int)

Relevant to 14: "get the modification time of a file"
Query terms: [the, of, time, get, file, modification], skipped terms: %
       [a]
Query terms (stemmed): [the, of, modif, time, get, file], skipped %
       terms: [a]
jdk4818: java.io.File.lastModified()

Relevant to 15: "given a file and a directory copy the file into the %
       directory"
Query terms: [the, directory, and, into, file, copy, given], skipped %
       terms: [a]
Query terms (stemmed): [the, and, copi, into, file, directori, given], %
       skipped terms: [a]
jdk4806: java.io.File.getAbsolutePath()
jdk4801: java.io.File.getName()

Relevant to 16: "check if a directory exists, if not then create it"
Query terms: [directory, create, it, if, exists, not, then, check], %
       skipped terms: [a]
Query terms (stemmed): [creat, it, directori, if, exist, not, then, %
       check], skipped terms: [a]
jdk4814: java.io.File.exists()
jdk4828: java.io.File.mkdir()
jdk4829: java.io.File.mkdirs()
jdk4815: java.io.File.isDirectory()
```

```
Relevant to 17: "print out 0 - 52 in a random order"
Query terms: [out, print, order, random, in], skipped terms: [a]
Query terms (stemmed): [out, print, order, random, in], skipped terms: %
       [a]
colt2351: cern.jet.random.Uniform.nextIntFromTo(int, int)
colt2114: cern.jet.math.IntFunctions.random()
jdk10169: java.util.Random.nextInt(int)
jdk10165: java.util.Random.setSeed(long)
jdk8099: java.security.SecureRandom.setSeed(long)
jdk8101: java.security.SecureRandom.next(int)

Relevant to 18: "check if the character is a digit, and if it is, %
       throw it into the StringBuffer. Else ignore it"
Query terms: [the, and, ignore, throw, digit, into, character, check, %
       string, buffer, else, it, if, is], skipped terms: [a]
Query terms (stemmed): [the, and, throw, digit, into, check, string, %
       buffer, it, if, els, is, charact, ignor], skipped terms: [a]
jdk5341: java.lang.Character.isDigit(char)

Relevant to 19: "copy the elements of the array into a vector"
Query terms: [the, of, into, vector, copy, array, elements], skipped %
       terms: [a]
Query terms (stemmed): [the, element, of, copi, into, vector, arrai], %
       skipped terms: [a]
jdk9445: java.util.Arrays.asList(java.lang.Object[])
jdk10373: java.util.Vector.addElement(java.lang.Object)
jdk10387: java.util.Vector.addAll(java.util.Collection)
```

Appendix D

Legal information

These copyright notices are from the Colt homepage[1].

[1]http://hoschek.home.cern.ch/hoschek/colt/V1.0.3/doc/cern/colt/doc-files/license.html

Index

Bibliography

[Agnosti et al., 2001] Agnosti, M., Crestani, F., and Pasi, G., editors (2001). *Lectures on Information Retrieval*, number 1980 in LNCS. Springer. (cited on pp 118, 119)

[Baader et al., 2003] Baader, F., McGuinness, D. L., Nardi, D., and Patel-Schneider, P. F., editors (2003). *The Description Logic Handbook*. Cambridge University Press. (cited on p 55)

[Belkin et al., 2003] Belkin, N., Cool, C., Kelly, D., Kim, G., Kim, J.-Y., Lee, H.-J., Muresan, G., Tang, M.-C., and Yuan, X.-J. (2003). Query length in interactive information retrieval. In *SIGIR' 03*. ACM Press. (cited on p 67)

[Berners-Lee, 2000] Berners-Lee, T. (2000). *CWM – Closed World Machine*. `http://www.w3.org/2000/10/swap/doc/cwm.html`. (cited on p 54)

[Berners-Lee, 2003] Berners-Lee, T. (2003). The semantic web: Build the metadata, and it will come. `http://www.edventure.com/pcforum/transcript/the%20semantic%20web.pdf`. (cited on p 53)

[Berners-Lee et al., 2001] Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*. (cited on p 52)

[Berry et al., 1993] Berry, M., Do, T., Krishna, G. O. V., and Varadhan, S. (1993). *SVDPACKC (Version 1.0) User's Guide*. (cited on p 65)

[Berry et al., 1999] Berry, M. W., Drmač, Z., and Jessup, E. R. (1999). Matrices, vector spaces, and information retrieval. *SIAM Review*, 42(2):335–362. (cited on pp 16, 26, 27, 31)

[Berry et al., 1994] Berry, M. W., Dumais, S. T., and O'Brien, G. W. (1994). Using linear algebra for intelligent information retrieval. In *SIAM Review*, volume 4, pages 573–595. Society for Industrial and Applied Mathematics. (cited on pp 26, 32, 59, 94)

[Deerwester et al., 1990] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R. (1990). Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407. `http://download.interscience.wiley.com/cgi-bin/fulltext?ID=10049585&PLACEBO=IE.pdf&mode=pdf`. (cited on pp 24, 26, 29, 31, 32, 76)

[Duff et al., 1992] Duff, I. S., Grimes, R. G., and Lewis, J. G. (1992). *Users' Guide for the Harwell-Boeing Sparse Matrix Collection (Release I)*. (cited on p 65)

[Dumais, 1992] Dumais, S. T. (1992). Enhancing performance in latent semantic indexing (lsi) retrieval. Technical Report 21236, Bellcore. `http://citeseer.ist.psu.edu/dumais92enhancing.html`. (cited on pp 18, 19, 21, 22, 23, 24, 26, 92)

[Dumais et al., 1996] Dumais, S. T., Landauer, T. K., and Littman, M. L. (1996). Automatic cross-linguistic information retrieval using latent semantic indexing. In *Workshop on Cross-Linguistic Information Retrieval*, pages 16–23. SIGIR. (cited on p 24)

[Foltz et al., 1996] Foltz, P. W., Laham, D., and Landauer, T. K. (1996). Automated essay scoring: Applications to educational technology. Technical report, University of Colorado. (cited on p 24)

[Frakes and Pole, 1994] Frakes, W. B. and Pole, T. H. (1994). An empirical study of representation methods for reusable software components. In *IEEE Transactions on Software Engineering*, volume 20, pages 617–630. IEEE Press. (cited on pp 42, 56)

[Furnas et al., 1987] Furnas, G. W., Landauer, T. K., Gomez, L. M., and Dumais, S. T. (1987). The vocabulary problem in human-system communication. *Commun. ACM*, 30(11):964–971. (cited on p 14)

[Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns – elements of reusable object-oriented software*. Addison-Wesley Professional. (cited on p 61)

[Gosling et al., 2000] Gosling, J., Joy, B., Steele, G., and Bracha, G. (2000). *The Java(TM) Language Specification*. Addison-Wesley Professional, 2nd edition. (cited on p 8)

[Henninger, 1997] Henninger, S. (1997). An evolutionary approach to constructing effective software reuse repositories. *ACM Transactions on Software Engineering and Methodology*, 6(2):111–140. (cited on pp 45, 47)

[Hofmann, 1999a] Hofmann, T. (1999a). Probabilistic latent semantic analysis. In *Proceedings of Uncertanity in Artificial Intelligence, UAI 99*, Stockholm. (cited on p 51)

[Hofmann, 1999b] Hofmann, T. (1999b). Probabilistic latent semantic indexing. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 50–57. ACM Press. (cited on p 51)

[Horrocks, 2002] Horrocks, I. (2002). DAML+OIL: a description logic for the semantic web. *Bull. of the IEEE Computer Society Technical Committee on Data Engineering*, 25(1):4–9. (cited on p 57)

[Jones and Furnas, 1987] Jones, W. P. and Furnas, G. W. (1987). Pictures of relevance: A geometric analysis of similarity measures. *Journal of the American Society for Information Science*, 38(6):420–442. (cited on pp 30, 31, 49)

[Kohonen, 1995] Kohonen, T. (1995). *Self-Organizing Maps.* Number 30 in Spring Series in Information Sciences. Springer. (cited on p 36)

[Kohonen, 2000] Kohonen, T. (2000). Self-organizing maps of massive document collections. *IEEE Transactions on Software Engineering.* (cited on p 37)

[Kopena and Regli, 2003] Kopena, J. B. and Regli, W. C. (2003). Design repositories for the semantic web with description-logic enabled services. In Decker, S., editor, *Semantic Web and Databases*, pages 349–356. (cited on p 56)

[Landauer et al., 1998] Landauer, T. K., Foltz, P. W., and Laham, D. (1998). An introduction to latent semantic analysis. In *Discourse Processes*, number 25, pages 259–284. (cited on p 24)

[Letsche and Berry, 1997] Letsche, T. A. and Berry, M. W. (1997). *Large-Scale Information Retrieval with Latent Semantic Indexing*, volume 100. Information Sciences. `http://dx.doi.org/10.1016/S0020-0255(97)00044-3`. (cited on p 91)

[Maarek et al., 1991] Maarek, Y. S., Berry, D. M., and Kaiser, G. E. (1991). An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813. (cited on p 48)

[Manning and Schütze, 1999] Manning, C. D. and Schütze, H. (1999). *Foundations of Statistical Natural Language Processing.* MIT Press, Cambridge, Massachusetts. (cited on pp 15, 19, 33, 34)

[Marshall and Shipman, 2003] Marshall, C. C. and Shipman, F. M. (2003). Which semantic web? In *Proceedings of the fourteenth ACM conference on Hypertext and hypermedia*, pages 57–66. (cited on p 53)

[Michail and Notkin, 1999] Michail, A. and Notkin, D. (1999). Assessing software libraries by browsing similar classes, functions and relationships. In *Proceedings of the 21st international conference on Software engineering*, pages 463–472. IEEE Computer Society Press. (cited on pp 45, 64)

[Mili et al., 1997] Mili, H., Ah-Ki, E., Godin, R., and Mcheick, H. (1997). Another nail to the coffin of faceted controlled-vocabulary component classification and retrieval. In *Proceedings of the 1997 symposium on Software reusability*, pages 89–98. ACM Press. (cited on p 42)

[Osiński, 2003] Osiński, S. (2003). An algorighm for clustering of web search results. Master's thesis, Poznań University of Technology, Poland. (cited on pp 70, 92)

[Porter, 1980] Porter, M. F. (1980). An algorithm for suffix stripping. *Program*, 14(3):130–137. (cited on p 18)

[Rada, 1995] Rada, R. (1995). *Software Reuse*. Intellect. (cited on p 40)

[Robertson, 2000] Robertson, S. E. (2000). Evaluation in information retrieval. In [Agnosti et al., 2001], pages 81–92. (cited on pp 74, 75)

[Robertson and Walker, 1994] Robertson, S. E. and Walker, S. (1994). Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In Croft, W. B. and van Rijsbergen, C. J., editors, *Proceedings of the 17th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval. Dublin, Ireland, 3-6 July 1994 (Special Issue of the SIGIR Forum)*, pages 232–241. ACM/Springer. (cited on p 52)

[Robertson et al., 1995] Robertson, S. E., Walker, S., Jones, S., Hancock-Beaulieu, M. M., and Gatford, M. (1995). Okapi at TREC 3. In Harman, D. K., editor, *Overview of the Third Text REtrieval Conference (TREC-3)*, pages 109–126, Gaithersburg, Maryland. National Institute of Standards and Technology (NIST), National Institute of Standards and Technology. (cited on pp 41, 52)

[Salton, 1989] Salton, G. (1989). *Automatic text processing: the transformation, analysis and retrieval of information by computer*. Addison-Wesley series in computer science. Addison-Wesley. (cited on pp 22, 30)

[van Rijsbergen, 1979] van Rijsbergen, C. J. (1979). Information retrieval. (cited on pp 13, 17) [van Rijsbergen, 2001] van Rijsbergen, C. J. (2001). Getting into information retrieval. In [Agnosti et al., 2001], pages 1–20. (cited on p 13)

[Voorhees and Buckland, 2002] Voorhees, E. M. and Buckland, L. P., editors (2002). *The Eleventh Text REtrieval Conference (TREC 2002).* Department of Commerce, National Institute of Standards and Technology. (cited on p 74)

[Weiss, 2001] Weiss, D. (2001). A clustering interface for web search results in english and polish. Master's thesis, Poznań University of Technology, Poland. (cited on pp 67, 69)

[Wróblewski, 2003] Wróblewski, M. (2003). A hierarchical www pages clustering algorithm based on the vector space model. Master's thesis, Poznań University of Technology, Poland. (cited on p 69)

[Ye, 2001] Ye, Y. (2001). *Supporting Component-Based Software Development with Active Component Repository Systems.* PhD thesis, University of Colorado. `http://www.cs.colorado.edu/~yunwen/codebroker/`. (cited on pp 1, 3, 5, 10, 40, 42, 45, 76)

[Ye, 2003] Ye, Y. (2003). Programming with an intelligent agent. *IEEE Intelligent Systems*, 18(3):43–47. (cited on pp 40, 41)

[Zha, 1998] Zha, H. (1998). A subspace-based model for information retrieval with applications in latent semantic indexing. Technical Report CSE-98-002, Department of Computer Science and Engineering, Pennsylvania State University. (cited on p 27)

[Zhao and Grosky, 2002] Zhao, R. and Grosky, W. (2002). Negotiating the semantic gap: from feature maps to semantic landscaps. *Pattern Recognition*, 35(35):593–600. (cited on p 22)