Automated Retrieval of Artifacts Created during the Software Development Life-cycle

Hans-Peter Krüger and Pieter S. Kritzinger
Technical Report CS08-02-00

Data Network Architectures Group,
Computer Science Department University of Cape Town,
Private Bag, Rondebosch 7700, South Africa
Email: {hkruger,psk@cs.uct.ac.za}

Abstract *The number of failures of software projects not meeting the originally intended requirements are many. While often due to users and developers not sharing the same vocabulary, it is more often due to changes which are not reported or recorded somewhere along the development cycle. Software traceability (ST), is the process of tracking changes in the document corpus which are created throughout the software development life-cycle. There are known techniques, such as using traceability matrices, which attempt to solve the problem. Such mechanical methods are not only manually intensive, but they totally ignore the effects of synonymy and polysemy. Latent semantic analysis (LSA) is intended to avoid these latter effects and is largely used in the world of Information Retrieval (IR). In this report we apply LSA for the purpose of maintaining artifacts generated during the software development life-cycle and place greater emphasis than hitherto found in the literature, on term extraction in software code, something we call attribute weighting. We moreover present a software tool for the automation of the traceability process, including query refinement and show that the technique allows one to trace through the artifact corpus with the confidence that the set of artifacts affected by a change will be discovered.*

Keywords: Requirements Engineering, Software Traceability, Change Management, Latent Semantic Analysis, Relevance Feedback, Information Retrieval.

I. Introduction

The number of failures of software projects not meeting the originally intended requirements are many. When not failing, projects are often well over budget. These failings frequently arise from changed requirements, where the impact on the software already designed and coded, is not properly understood. The latter, in turn, is usually due to the fact that the interdependencies amongst the software artifacts are not known, or forgotten. Software Traceability (ST), is the term given to the process of tracking changes in the document corpus which are created throughout the software development life-cycle. There are known techniques, such as using traceability matrices, which attempt to solve the problem. Such mechanical methods are not only manually intensive, but they ignore the effects of synonymy and polysemy.

Latent semantic analysis (LSA) [4] is intended to avoid these latter effects and is largely used in the world of Information Retrieval [5]. LSA tries to reveal the latent semantics among documents that is partially obscured by variability in natural language word choice, which is often referred to as *noise* in the literature. The n words, terms or keywords in a collection of m documents created during the software development life-cycle are extracted and represented as a very sparse term-document matrix. The matrix entries, a_{ij} represent the frequency with which each term occurs in a document. That is,

$$A_{n\times m} = [a_{ij}] \quad i = 1, \ldots, n; j = 1, \ldots, m \tag{1}$$

where a_{ij} is the weight of term i in artifact j.

Clearly the matrix, say $A_{n\times m}$ can be huge and a mathematical technique, known as singular-value decomposition (SVD) is used to derive a rank-reduced matrix which, as a side effect, reduces the *noise* in the document corpus. Several techniques [9] exist to improve LSA and we will discuss these in context in Section VII.

After discussing some of the earlier work on LSA-related software traceability, we describe our prototype software developed to automate the traceability process and discuss in detail how the various software artifacts are managed. In Section V, we describe how we build the artifact corpus and then proceed with a case study where we discuss the various ways of improving LSA in the context of our own work. To the extent that we used all artifacts created in the software development life-cycle in our study, we believe this work is unique.

II. Previous Work

One of the first attempts to use LSA for traceability link discovery is presented by De Lucia *et al* [3]. An existing artifact management tool was enhanced to not only allow the discovery of traceability links between natural language text documents and source code, but also between requirement and design artifacts, namely UML use-cases and interaction diagrams, as well as test cases. Although some interesting ideas were presented, such as variable similarity thresholds and artifact categorization, their case study is not complete. Despite the claim that artifact management systems can handle natural language text artifacts, no requirements or architectural specifications were considered in their case study.

Several case studies can also be found in the literature. Marcus and Maletic [10] as well as Antoniol *et al.* [1] use two identical software projects in their respective case studies. The library of efficient data types and algorithms (LEDA), which has been developed at the Max Planck Institut für Informatik, Saarbrücken, Germany, and the Albergate project which was a final year student project for a hotel management system at the University of Verona, Italy.

Hayes *et al.* [6], [7] follow a similar path by conducting case studies on very technical artifact sets. In these studies the requirements specification of the NASA Moderate Resolution Imaging Spectroradiometer (MODIS) was used to trace between 16 high and 50 low-level requirements.

Lormans and van Deursen [8] and De Lucia *et al.* [3] conducted their studies on less technical software projects with a broader set of artifact types.

III. Traceability Performance Metrics

The two most popular metrics for evaluating IR performance are *Recall* and *Precision*[10]. Let C_i be the set of relevant artifacts and R_i the set of all retrieved artifacts for a user query i. *Recall* and *Precision* are then defined as follows:

$$Recall_i = \frac{|C_i \cap R_i|}{|C_i|} \tag{2}$$

$$Precision_i = \frac{|C_i \cap R_i|}{|R_i|} \tag{3}$$

In order to assess the overall performance on the entire system the summation over all queries is performed. I.e.,

$$Recall = \frac{\Sigma_i |C_i \cap R_i|}{\Sigma_i |C_i|} \tag{4}$$

$$Precision = \frac{\Sigma_i |C_i \cap R_i|}{\Sigma_i |R_i|} \tag{5}$$

It is intuitive that, retrieving a lower number of irrelevant artifacts for each query would result in higher precision, while a higher number of relevant artifacts would increase the recall. Both values depend on the threshold or value of the dot-product (explained in Section VII-C) between the various document vectors in the reduced space.

A metric that incorporates recall and precision into one single value is the F-measure, or balanced F-score, which computes the harmonic mean of precision and recall:

$$F(\alpha)_i = (1+\alpha) \cdot \frac{Precision_i \cdot Recall_i}{\alpha \cdot Precision_i + Recall_i} \tag{6}$$

The influence of recall or precision in the single value of $F(\alpha)$ clearly depends on the choice of α. The two commonly used α values are $\alpha = 2$ which weights recall twice as much as precision and $\alpha = 0.5$ which is the complement. Our results all report the $F(2)$ value, since we believe that recall is more important than precision for the software engineer. Failing to recall artifacts during a query of the software corpus can have serious implications for costs of introducing a requirement change while precision is easily decided by the engineer himself.

IV. Automating the Traceability Process

We developed an automation of the LSA corpus building and query feedback process in order to implement and verify our proposals. Figure 1 presents a process overview of the prototype tool. The implementation was done as an Eclipse

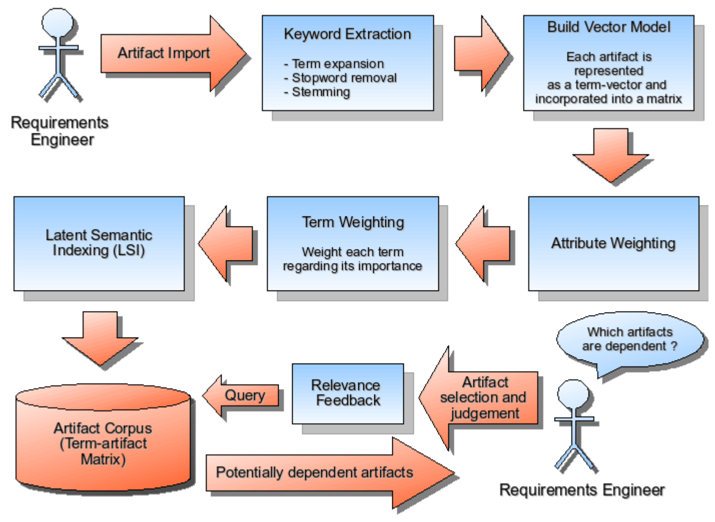

Requirements Engineer
Artifact Import
Keyword Extraction
- Term expansion
- Stopword removal
- Stemming
Build Vector Model
Each artifact is represented as a term-vector and incorporated into a matrix
Latent Semantic Indexing (LSI)
Term Weighting
Weight each term regarding its importance
Attribute Weighting
Which artifacts are dependent ?
Query
Relevance Feedback
Artifact selection and judgement
Artifact Corpus (Term-artifact Matrix)
Potentially dependent artifacts
Requirements Engineer

Fig. 1. The stages of the process are first to import the software documents followed by a number of processing stages until the artifact corpus has been created. The software engineer selects an artifacts for which he wants obtain all dependent artifacts and he submits a search request to the system.

(`http://www.eclipse.org/`) plug in, so as to take advantage of a widely used software development platform.

The way that the prototype manages each of the various artifact types created during the software development life-cycle is described in the following sections.

A. Natural language text documents

In principle the prototype accepts any kind of natural language text in PDF format. This seemed fair since natural language text documentation, such as requirements specifications or design documentation, is usually available in PDF format or can easily be exported to such. The engineer can view PDF documents in the build-in viewer and is able to mark text passages as necessary. The selected text is subsequently extracted from the document and added as traceable artifacts to the corpus, as shown in Figure 2.

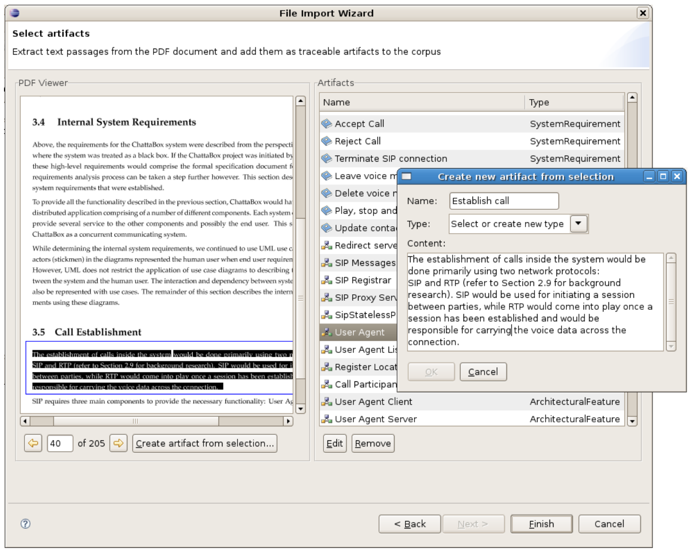
We present an text artifact vector as the weighted sum of the two *attribute vectors* $\overrightarrow{\text{Name}}$ and $\overrightarrow{\text{Content}}$, see Eq. 7. The $\overrightarrow{\text{Content}}$ vector contains terms that were extracted from the description of an text artifact, i.e. from a user requirement. $\overrightarrow{\text{Name}}$ contains the terms extracted from the name of the text artifact. The variables w_1 and w_2 are called *attribute weights* as they weight the importance of terms extracted from the artifact attributes.

$$\overrightarrow{\textbf{Text}} = w_1 \cdot \overrightarrow{\text{Name}} + w_2 \cdot \overrightarrow{\text{Content}} \tag{7}$$

B. UML diagrams

UML diagrams, or similar graphic descriptions, are widely used in software development and have to be considered part of the software development life-cycle document corpus. Our prototype allows the importation of UML diagrams, such as use cases, interaction and state diagrams, provided these artifacts are available in the Metadata Interchange (XMI) format, which they normally are.

A use case vector, see Eq. 8, is composed of the user-case name, the name of the associated subject or system and involved



File Import Wizard

Select artifacts

Extract text passages from the PDF document and add them as traceable artifacts to the corpus

PDF Viewer

3.4 Internal System Requirements

Above, the requirements for the ChattaBox system were described from the perspecti
where the system was treated as a black box. If the ChattaBox project was initiated by
these high-level requirements would comprise the formal specification document f
requirements analysis process can be taken a step further however. This section des
system requirements that were established.

To provide all the functionality described in the previous section, ChattaBox would ha
distributed application comprising of a number of different components. Each system
provide several service to the other components and possibly the end user. This s
ChattaBox as a concurrent communicating system.

While determining the internal system requirements, we continued to use UML use c
actors (stickmen) in the diagrams represented the human user when end user requiren
However, UML does not restrict the application of use case diagrams to describing
tween the system and the human user. The interaction and dependency between syste
also be represented with use cases. The remainder of this section describes the intern
ments using these diagrams.

3.5 Call Establishment

The establishment of calls inside the system would be done primarily using two n
SIP and RTP (refer to Section 2.9 for background research). SIP would be used for i
between parties, while RTP would come into play once a session has been establish
responsible for carrying the voice data across the connection.

SIP requires three main components to provide the necessary functionality: User Ag

40 of 205

Artifacts

Name	Type
Accept Call	SystemRequirement
Reject Call	SystemRequirement
Terminate SIP connection	SystemRequirement
Leave voice m	
Delete voice r	
Play, stop and	
Update conta	
Redirect serve	
SIP Messages	
SIP Registrar	
SIP Proxy Serv	
SipStatelessP	
User Agent	
User Agent Lis	
Register Locat	
Call Participan	
User Agent Client	ArchitecturalFeature
User Agent Server	ArchitecturalFeature

Create new artifact from selection

Name: Establish call

Type: Select or create new type

Content:

The establishment of calls inside the system would be done primarily using two network protocols:
SIP and RTP (refer to Section 2.9 for background research). SIP would be used for initiating a session between parties, while RTP would come into play once a session has been established and would be responsible for carrying the voice data across the connection.

Fig. 2. Import of natural language text artifacts: The engineer marks a text section in a PDF document which will be subsequently imported into LSITrace as a new traceable artifact.

actors. Additionally, we also incorporate the comments that are either associated with the subject or the user-case itself. Use cases can have an *include* and *extends* relationship to other use cases. In order to preserve these relationships we also incorporate the names of related use cases into the use case vector.

$$\begin{aligned}\overrightarrow{\textbf{Usecase}} &= w_1 \cdot \overrightarrow{\text{SubjectName}} + w_2 \cdot \overrightarrow{\text{UsecaseName}} \\ &+ \sum_{i=1}^{|\text{Actors}|} \left(w_3 \cdot \overrightarrow{\text{ActorName}_i} \right) + \sum_{i=1}^{|\text{Comments}|} \left(w_4 \cdot \overrightarrow{\text{Comment}_i} \right) \\ &+ \sum_{i=1}^{|\text{Associated usecases}|} \left(w_5 \cdot \overrightarrow{\text{UsecaseName}_i} \right) \qquad (8)\end{aligned}$$

The representation of a UML sequence and state diagrams is slightly more complex than for a use case. A sequence diagram vector, see Eq. 9, is composed of the interaction or diagram name, the names of all classes involved (lifelines), the labels of all messages exchanged among the classes as well as the diagram comments.

$$\begin{aligned}\overrightarrow{\textbf{SequenceDiagram}} &= w_1 \cdot \overrightarrow{\text{InteractionName}} + \sum_{i=1}^{|\text{Lifelines}|} \left(w_2 \cdot \overrightarrow{\text{LifelineName}} \right) \\ &+ \sum_{i=1}^{|\text{Messages}|} \left(w_4 \cdot \overrightarrow{\text{MessageName}_i} \right) \\ &+ \sum_{i=1}^{|\text{Comments}|} \left(w_3 \cdot \overrightarrow{\text{Comment}_i} \right) \qquad (9)\end{aligned}$$

The representation of a UML state diagram is very similar to the sequence diagram. A state diagram vector, see Eq. 10, is composed of the state machine name, all diagram comments, all state names as well as the state actions (entry, exit and do). The labels of the state transition are also incorporated in the diagram representation.

$$\begin{aligned}\overrightarrow{\textbf{StateDiagram}} &= w_1 \cdot \overrightarrow{\text{StateMachineName}} + \sum_{i=1}^{|\text{Comments}|} \left(w_3 \cdot \overrightarrow{\text{Comment}_i} \right) \\ &+ \sum_{i=1}^{|\text{States}|} w_2 \cdot \overrightarrow{\text{StateName}} + w_4 \cdot \left(\overrightarrow{\text{StateEntry}} + \overrightarrow{\text{StateExit}} \right) \\ &+ w_5 \cdot \overrightarrow{\text{StateDo}} + \sum_{i=1}^{|\text{Transitions}|} w_6 \cdot \overrightarrow{\text{TransitionLinkName}_i} \qquad (10)\end{aligned}$$

C. Source code

An estimate of the impact a requirement change will have upon the existing source code is crucial to any decision about such changes. Clearly without the ability to search through, or query, the source code, traceability is not much use. Our prototype allows the engineer to import class artifacts that were developed in the Microsoft C# programming language. The reason for his is that we used a software project in our case study (Section VI) that was developed with this language. Clearly the language used could have been different without affecting our discussion of the principles.

In order to represent source code artifacts adequately in vector space, we propose the model shown in Eqs. 11 - 16. In this model, the smallest traceable source code artifact is a class. A class vector is composed of package, class, field and method declarations, which contain associated comments and identifier names. We moreover incorporate comments and string literals terms found in the body of methods.

$$\begin{aligned}\overrightarrow{\textbf{PackageDeclaration}} &= w_1 \cdot \overrightarrow{\text{Comment}} + \sum_{i=1}^{|\text{Subpackages}|} \left(w_2 \cdot \overrightarrow{\text{SubpackageName}_i} \right) \\ &+ w_3 \cdot \overrightarrow{\text{Name}} \qquad (11)\end{aligned}$$

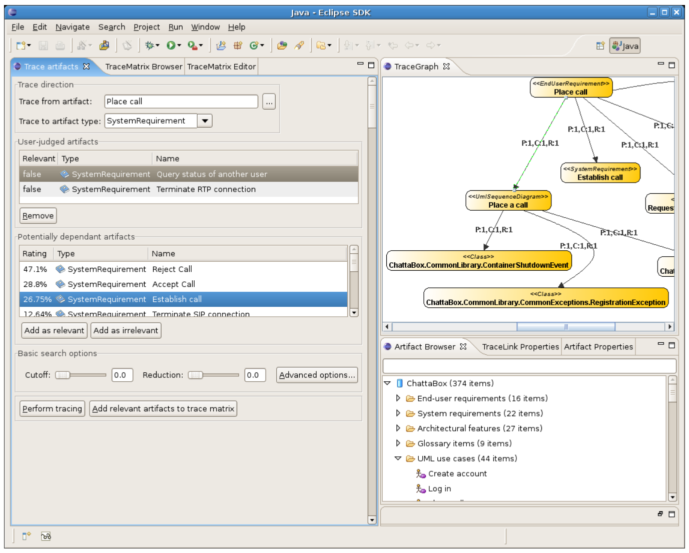
$$\begin{aligned}\overrightarrow{\textbf{ClassDeclaration}} &= w_1 \cdot \overrightarrow{\text{Comment}} + w_2 \cdot \overrightarrow{\text{Name}} \\ &+ \sum_{i=1}^{|\text{Super classes}|} w_3 \cdot \overrightarrow{\text{SuperClassName}_i} \qquad (12)\end{aligned}$$

$$\overrightarrow{\textbf{FieldDeclaration}} = w_1 \cdot \overrightarrow{\text{Comment}} + w_2 \cdot \overrightarrow{\text{Type}} + w_3 \cdot \overrightarrow{\text{Name}} \qquad (13)$$

$$\begin{aligned}\overrightarrow{\textbf{MethodDeclaration}} &= w_1 \cdot \overrightarrow{\text{Comment}} + w_2 \cdot \overrightarrow{\text{ReturnType}} + w_3 \cdot \overrightarrow{\text{Name}} \\ &+ \sum_{i=1}^{|\text{Parameters}|} \left(w_4 \cdot \overrightarrow{\text{ParamType}_i} + w_5 \cdot \overrightarrow{\text{Name}_i} \right) \qquad (14)\end{aligned}$$

$$\begin{aligned}\overrightarrow{\textbf{MethodBody}} &= \sum_{i=1}^{|\text{Comments}|} \left(w_1 \cdot \overrightarrow{\text{Comment}_i} \right) + \sum_{i=1}^{|\text{String literals}|} \left(w_2 \cdot \overrightarrow{\text{Literal}_i} \right) \\ &+ \sum_{i=1}^{|\text{Remaining identifiers}|} w_3 \cdot \overrightarrow{\text{Identifier}_i} \qquad (15)\end{aligned}$$

$$\begin{aligned}\overrightarrow{\textbf{Class}} &= \sum_{i=1}^{|\text{Package declarations}|} \left(\overrightarrow{\text{Declaration}_i} \right) + \overrightarrow{\text{ClassDeclaration}} \\ &+ \sum_{i=1}^{|\text{Method declarations}|} + \left(\overrightarrow{\text{Declaration}_i} + \overrightarrow{\text{MethodBody}} \right) \\ &+ \sum_{i=1}^{|\text{Field declarations}|} + \overrightarrow{\text{Declaration}_i} \qquad (16)\end{aligned}$$


Java - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Trace artifacts
TraceMatrix Browser
TraceMatrix Editor
TraceGraph
<<EndUserRequirement>> Place call
<<SystemRequirement>> Establish call
<<UmlSequenceDiagram>> Place a call
<<Class>> ChattaBox.CommonLibrary.ContainerShutdownEvent
<<Class>> ChattaBox.CommonLibrary.CommonExceptions.RegistrationException
P:1,C:1,R:1
Remove
Add as relevant
Add as irrelevant
Advanced options...
Perform tracing
Add relevant artifacts to trace matrix
Artifact Browser
TraceLink Properties
Artifact Properties
Java

Trace direction

Trace from artifact: Place call

Trace to artifact type: SystemRequirement

User-judged artifacts

Relevant	Type	Name
false	SystemRequirement	Query status of another user
false	SystemRequirement	Terminate RTP connection

Potentially dependant artifacts

Rating	Type	Name
47.1%	SystemRequirement	Reject Call
28.8%	SystemRequirement	Accept Call
26.75%	SystemRequirement	Establish call
12.64%	SystemRequirement	Terminate SIP connection

Basic search options

Cutoff: 0.0 Reduction: 0.0

- ChattaBox (374 items)
  - End-user requirements (16 items)
  - System requirements (22 items)
  - Architectural features (27 items)
  - Glossary items (9 items)
  - UML use cases (44 items)
    - Create account
    - Log in

Fig. 3. Tracing and relevance feedback view is shown on the left hand side. The upper right hand side shows a graphical representation of the matrix.

In order to recognize and weight the various class attributes, we implemented a source code parser. Rather than simply extracting terms by applying regular expressions like Marcus and Malectic [10], we produced an abstract syntax tree (AST) of every source code artifact. This allows us to weight attributes according to the number of times they occur in the hierarchy. Implementing a fully featured parser for a modern programming language is cumbersome and complex task so we support only a subset of the grammar that is required to extract the earlier described artifact attributes and skipped the rest.

D. *Automated recovery of traceability links*

The main feature of the prototype is the ability to recover traceability links for a given artifact automatically. Figure 3 shows the relevant user interface on the left hand side. In order to start the traceability link recovery process, the engineer needs to select an artifact type to search from. Furthermore, the engineer has to specify to what artifact type traceability links should be found. For example, in the example in Figure 3, the engineer has decided to trace from an end-user requirement, called *Place Call*, to all system requirement artifacts. After the engineer has pressed the *Perform tracing* button, the search operation starts and potentially relevant artifacts are ranked in a list according to their similarity to the search query.

E. *Managing traceability links*

With the prototype system, the engineer is also able to establish traceability links manually among artifacts. He can either modify the trace matrix in a flat table representation or through connecting artifacts in a graph representation, as shown in Figure 3.

V. Building the Artifact Corpus

In section I we presented a comprehensive model that shows how to represent software artifacts in the vector space model. We explained which attributes, i.e. method names or comments, have to be incorporated in the vectors to reflect the meaning of the original software artifacts. However, two questions remain open for discussion. First of all, which terms have to be extracted from the original artifacts. Do all terms have to be incorporated in the vector representation of the artifact or is it sufficient to concentrate on just a few. The other question is how to weight these terms appropriately so that they reflect the meaning of the artifact best.

We address the latter question experimentally in our case study in Section VI in which we study the performance of several weighting schemes. The following section gives detail information on how to extract terms from the artifacts.

A. *Term extraction*

With the artifact attributes to incorporate in the vector representation of the various software artifacts identified, we next extract terms from these attributes. The first step of the extraction process is to separate every term from the initial string and save it in a hash table. Every hash table entry consists of a key represented by the term itself and the weight of the term, initially set to zero and increased by one every time the term is found thereafter.

By example, the tree in Figure V-A illustrates the term extraction process for the label of a lifeline message of a UML sequence diagram consisting of a condition and method call. In the extraction process we separate the original string shown at the root into its constituent terms. Compound terms, such as onReceiveData and SIPMsg, are further divided into their constituent terms that are, in turn, incorporated in the hash table. The rationale behind this is that compound terms often represent identifiers, i.e., classes or names. Assuming that most software developers follow best programming practises by giving identifiers meaningful names, constituent terms can be useful in determining the importance of the artifact itself.

After all attribute terms have been extracted, terms are stemmed to their morphological root (see Section VII-D). Although LSA itself provides a means to overcome *synonymy*, we found that in our case study, that additional stemming of terms improves search results, except where source code is involved.

In the final step, we traverse the hash table and remove what are known as *stop words*. Stop words are auxiliary words like conjunctions and prepositions that do not contribute directly to the semantics. As shown in Figure V-A, the weight of term `on` was identified as a stop word and set to 0.0 and is thus no longer part of the vector representation of the attribute. In order to recognize stop words we used a slightly modified list of terms created at the University of Tennessee [11]. We mainly enhanced this list with keywords usually used in programming languages, such as `int`, `boolean` or `object`.

The profile of our final artifact corpus is given in Table I.

	Number of vectors	Mean vector length (Number of terms)	Mean frequency of terms in vectors	% of terms in vectors with frequency 1
End-user requirements	16	22.5	1.48	77%
System requirements	22	26.5	1.66	71%
Architectural features	27	67	1.7	71%
Use-Cases	44	8.8	1.01	99%
Sequence diagrams	14	65.6	1.7	66%
State diagrams	12	27.5	1.59	70%
Classes	230	85.8	1.49	82%

TABLE I

Profile of the corpus artifacts used in the case study

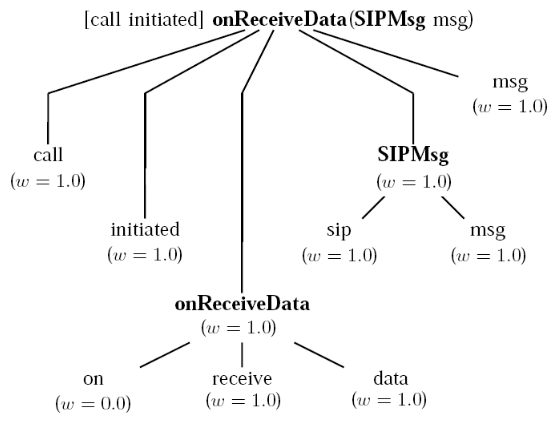


Fig. 4. Example expanded term tree

VI. Case Study

As most authors will know, one of the hard parts about studying traceability in the software development life-cycle is to find properly maintained, relevant artifacts for a working software system. The need for confidentiality requires professional software development companies not to make their software documentation, if there is any, public. On the other hand, open software systems mostly lack documentation other than the source code. Hence, like other studies, for instance that by Marcus [10], we had to resort to an internal software project. The most suitable project was the implementation of a voice over IP (VoIP) system, based on the Session Initiation Protocol (SIP). The main object of the project was to use best programming practises in describing the software in a detailed and complete fashion using requirements analysis, design, implementation and testing.

A. Tracing Links manually

One obviously needs to manually identify the best set of traceability links among the chosen software artifacts in order to determine how accurate the automatic traceability link recovery method is. Finding the best such set of traceability links is clearly subjective, since the decision whether or not software artifacts are dependent upon one another is to a certain degree a matter of interpretation. This is particularly true for artifacts like requirements, that are usually described in natural language text on an abstract level. On the other hand, deciding whether two source code artifacts, like classes, are dependent is much easier. A simple rule could be: "Class A is considered to be dependent upon class B if class A accesses fields, methods of class B or is derived from it". The advantage of the prototype tool described in Section IV, is that it indeed removes the human interpretation which is inevitably present in any manual traceability technique.

However, in order to say something about the performance of our proposal, we needed the manually discovered links, however biased. We recovered traceability links as best as possible and put a great deal of effort in validating them. We had to become acquainted with the project and examined the provided source code and documentation in detail. We also compiled and ran the source code to gain a better understanding of the dependencies between the artifacts. The process of going through each artifact to trace the links manually is a huge task with potentially m^2 documents to inspect. In the end we manually recovered traceability links from each user requirement to

– other user and system requirements, including
– UML use cases, collaboration and state diagrams and
– C# Classes

We also asked another experienced software developer to check the manually found traceability links, discussed changes and added or removed traceability links where necessary. In the end we determined 593 links from 16 end-user requirements to the artifact types mentioned above.

B. Experiments and Results

In all experiments, the original matrix A was transformed using to the applicable weighting schemes, its SVD computed and the resultant matrix A' used for the analyses. The choice of parameters for any one experiment is huge however: one can apply stemming or not, the number of combinations of global, local and query weightings amount to 108 for our choice of weights in Section VII-A; the matrix reduction can range from 0 to 95% and the choice of threshold value can be anywhere between 0 and 1. Consequently, only a fraction of all results can be reported here.

VII. Improving Latent Semantic Analysis

The open literature contains several references [12], [5] to various techniques to improve the performance of LSA. We discuss our interpretation and use of these in the following sections.

A. Term Weighting

We applied the most common term weights found in the literature [9], [4], [14] to our corpus of software artifacts and queries. A configuration of term weights that consists of a local and global weight, is denoted in our case study (see Figure 5) as S_n defined as follows:

$$\begin{aligned} S_n = (L_{Corpus}, G_{Corpus}, L_{Query}, G_{Query}) \text{ where} \\ L_{Corpus} \in \{Log, MaxTf, Tf\} \wedge \\ G_{Corpus} \in \{Entrophy, Idf, None\} \wedge \\ L_{Query} \in \{AugTf, Log, MaxTf, Tf\} \wedge \\ G_{Query} \in \{Entrophy, Idf, None\} \end{aligned} \quad (17)$$

In order to describe the various weighting schemes found in the literature, we first define tf_{ij}, the frequency of the term i, in the artifact j; af_i the artifact frequency or the number of artifacts of the total N in which term i occurs and finally, af_i the absolute frequency with which term i occurs in the entire corpus.

A normalized term-frequency $\overline{tf}_{i,j}$ of term i in document j is given by

$$\overline{tf}_{i,j} = \frac{tf_{i,j}}{\max_l\{tf_{l,j}\}} \quad (18)$$

where clearly the maximum term-frequency is computed over all terms in document j. In the event that there are large differences in the term frequencies, $\log(\overline{tf}_{ij} + 1)$ takes the log of the raw term frequency, thus dampening effects of large differences in frequencies.

In order to improve the performance of LSA, a term i can be given a *global weight* g_i to stress its information content across the document corpus, and a *local weight* l_{ij} to stress the content of the term i in document j.

Global weightings are meant to diminish the influence of words that occur frequently or in many of the documents. The weight w_{ij} for a term i in document j is defined to be

$$w_{ij} = l_{ij} \times g_i \tag{19}$$

As opposed to local weighting schemes, global weighting schemes take the distribution of terms in the whole document corpus into account in order to weight terms within a document appropriately. The inverse document frequency or *Idf-factor* is a well known global weighting scheme [5] which is based on the premise that terms which occur in many documents are not very useful in distinguishing a relevant document from non-relevant ones. Terms that occur in many documents are, therefore, assigned a smaller weight. The Idf-factor of term i is given by

$$idf_i = \log\left(\frac{N}{af_i}\right) + 1 \tag{20}$$

The best known weighting scheme for natural language text documents is described by Salton *et al* [13] and balances local- and global-weights. It is known as the term frequency-inverse, document-frequency (Tf-Idf) scheme, defined as follows:

$$\begin{aligned} tfidf_{i,j} &= \overline{tf}_{i,j} \cdot idf_i \\ &= \frac{f_{i,j}}{\max_l\{f_{l,j}\}} \cdot \log\left(\frac{N}{af_i}\right) \end{aligned} \tag{21}$$

Another global weighting scheme, or entropy scheme, first proposed by Dumais [5], normalizes the term frequency $tf_{i,j}$ by dividing by af_i as follows

$$\begin{aligned} \overline{tf'}_{i,j} &= \frac{tf_{i,j}}{af_i} \\ tfi_i &= 1 - \frac{1}{logN}\sum_{j=1}^{N} tf_{i,j} \times \log \overline{tf'}_{i,j} \end{aligned} \tag{22}$$

The term $\frac{1}{logN}\sum_{j=1}^{N} ft_{i,j} \times \log \overline{tf'}_{i,j}$ is called the *entropy*.

B. Query Weightings

Apart from weighting of terms in the document collection, we also need to consider an appropriate weighting scheme of the terms in the user query. Every term in the query vector $\vec{q} = (q_1, \ldots, q_k)$ for k the threshold value discussed in Sectio VII-C, is weighted by w_i where

$$w_i = \left(0.5 + 0.5 \times \frac{f_i}{\max_i\{f_i\}}\right) \times \log\left(\frac{N}{n_i}\right) \tag{23}$$

where f_i is the frequency of term i in $\vec{q}$ and $\max f_i$ the highest frequency of a term in the query vector. Apart from the term frequency, this weighting scheme also involves a global weighting which, as already seen in Eq. 19, considers the entropy of term i in the global context.

In order to find the best weighting scheme S_n (Eq. 17), we refer to Figure 5, a plot of the resultant F(2) value while tracing from User Requirements to classes for a threshold value of 0.1, matrix A' reductions, and all corpus and query weighting combinations.

Although not very clear from that figure, the corpus weighting Tf-Idf and Tf-Entrophy for the query weighting gave the best result for tracing to Classes.

In another experiment, we selected the weighting which gave us the best F(2) value at a 97.5% matrix reduction and a threshold value of 0.1. The results are shown in Figure 6 and at 97.5% matrix reduction, the F(2) value lies between 0.5 and 0.75 which is very good.

C. Effect of Threshold Value

In the vector space model, a software artifact is represented as an n-dimensional vector, where n is the number of unique terms, or words, that appear across the database or *corpus* of artifacts. The threshold is defined as the dot product between the vectors in the n-dimensional space.

As $\cos(\frac{\pi}{2}) = 0$, the dot product $\overrightarrow{a} \cdot \overrightarrow{b}$ of two perpendicular vectors $\overrightarrow{a}$ and $\overrightarrow{b}$ is always zero. Thus, given that thew two vectors have length a and b, respectively, the angle between them is given by:

$$\theta = \arccos\left(\frac{\overrightarrow{a} \cdot \overrightarrow{b}}{ab}\right).$$

For more than two dimensions, this formula can be used to define the concept of angle and it is the cosine value of this angle which defines the threshold. The larger the bundle of vectors covered by a particular angle from the query vector, the higher the probability that the artifacts recovered will contain similar terms or the higher the recall. On the other hand the smaller the angle the higher the precision, since spurious vectors or noise will not be included. The results of tracing from End-user requirements to remaining artifacts, averaged over all matrix reductions, for various threshold values with corpus weighting Tf-Idf and Tf-Entrophy for the query weighting, is shown in Figure 7. At low threshold values, that is, wider angles, the mean F(2) value is, as expected, better because of a higher recall value.

D. Effect of stemming

Stemming [5], or conflation, attempts to reduce all morphological variants of word to its stem or root form. Thus the terms of a query or document are represented by stems rather than by the original words such as the word ”call” rather than “calls” or “calling”. We applied stemming by employing a lookup table which contains relations between root forms and inflected forms. To stem a word, the table is queried to find a matching inflection. If a matching inflection is found, the associated root form is returned.

Natural language text artifacts, such as requirements or architectural specification, are a major part of the software development life-cycle corpus. Stemming has a huge influence as we can see in Figure 8, particularly at higher threshold values. Not surprisingly, stemming has no effect in the case of Classes.

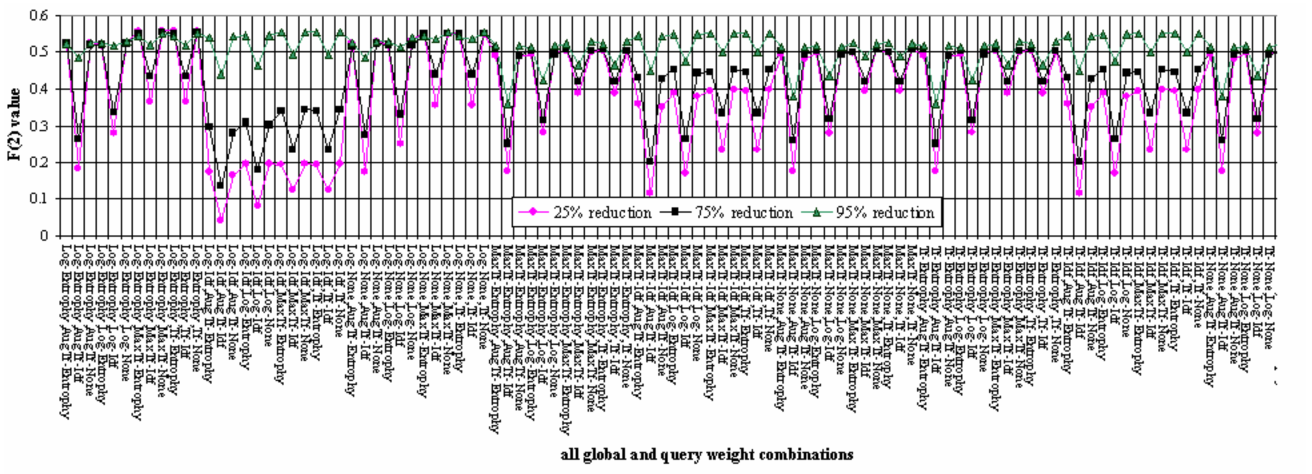


Fig. 5. Tracing from User Requirements to Classes for a threshold of 0.1, various matrix A' reductions and all corpus and query weighting combinations

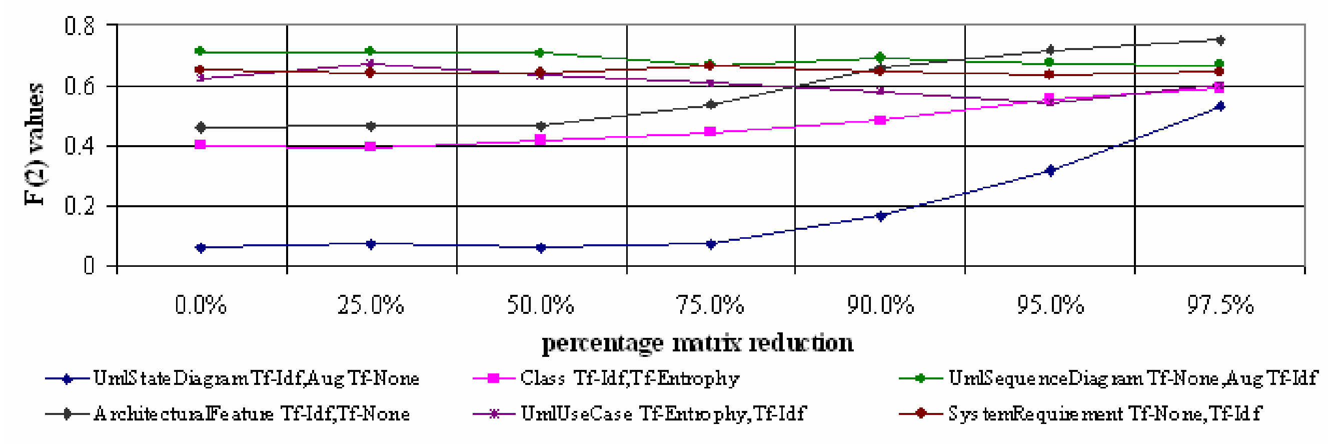


Fig. 6. Tracing from User Requirements to Classes for a threshold of 0.1, various matrix reductions and the best corpus and query weighting in each particular case at a 97.5% matrix reduction

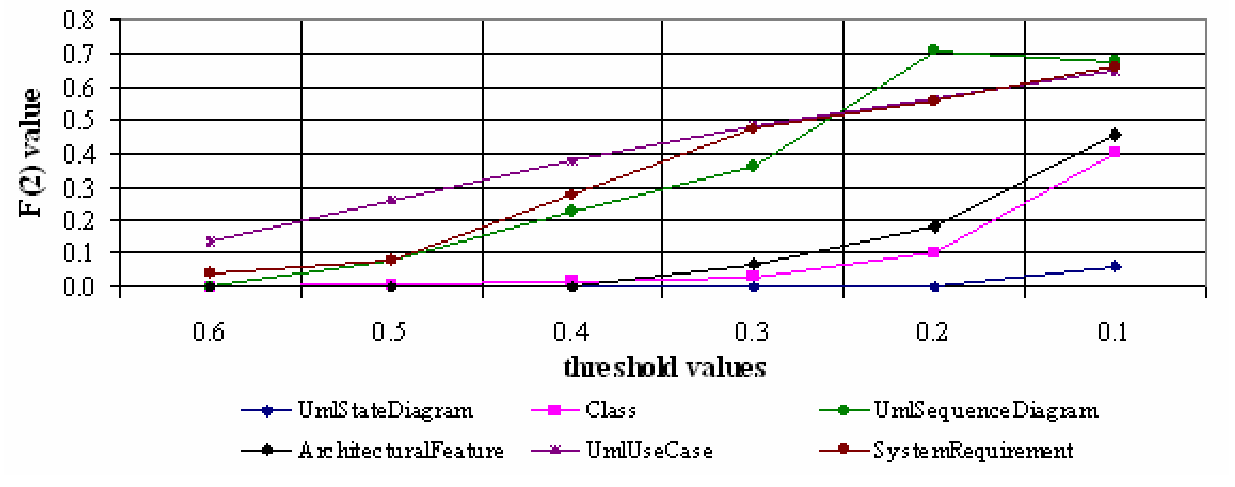


Fig. 7. Tracing from End-user requirements to remaining artifacts, averaged over all matrix reductions, for various threshold values with corpus weighting Tf-Idf and Tf-Entrophy for the query weighting

E. Relevance Feedback (RF)

In our prototype and as advocated by others such as Dumais, *et al.* [5], Salton [14] and Lee [2], the software engineer can either accept the artifacts which are returned by the system, or he can attempt to improve the results based on his own expert knowledge through the process of relevance feedback.

For this, he judges some of the suggested artifacts as relevant or irrelevant by selecting them from the candidate list and pressing the *Add as relevant* or *Add as irrelevant* button. The

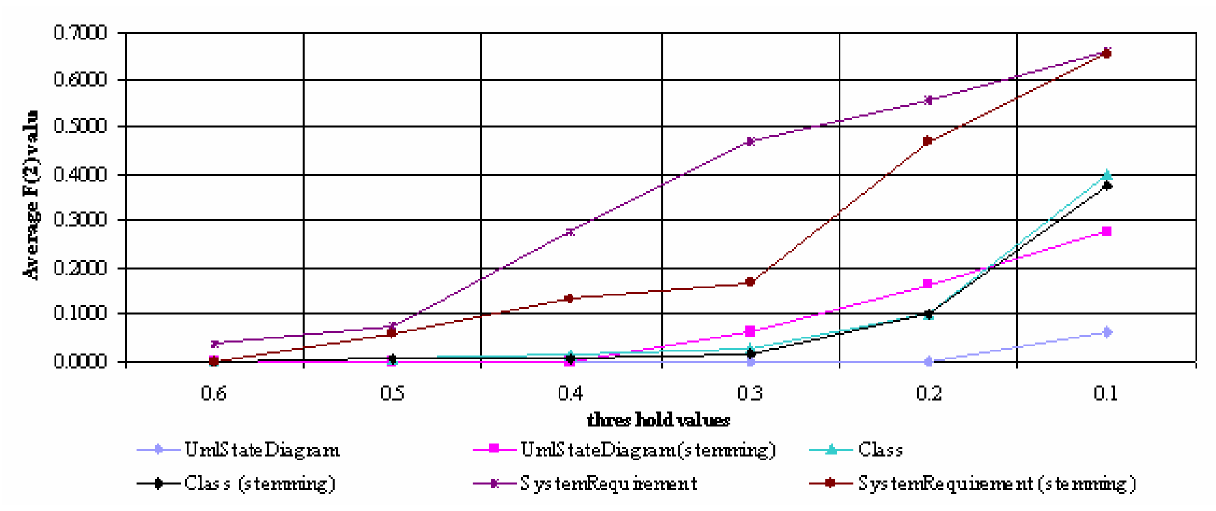


Fig. 8. Tracing from End-user requirements to some other artifacts for various threshold values with and without stemming with corpus weighting Tf-Idf and Tf-Entrophy for the query weighting.

artifacts will be removed from the candidate list and added to the user artifact list above. The system is subsequently advised to incorporate these artifacts into a potentially improved search query. At the same time, the engineer is allowed to modify a number of retrieval parameters, such as cut-off thresholds, dimensionality reduction through LSI, query- and corpus weightings and stemming for the next iteration if the artifact is considered relevant.

In Dumais, *et al.* [5], Salton [14] and Lee [2], usage of relevance feedback was found to greatly improve overall search performance in text documents. They discovered that queries composed from the highest ranked relevant document, returned by the initial query, gave an average overall improvement of 33% and queries composed of the three highest ranked relevant documents gave an average overall improvement of 67%. Their studies also found that the user typically views only a small number of the documents returned by the initial search in order to locate a few relevant documents. On the average, the most relevant document was the top ranked document and the three most relevant documents were within the top seven ranked documents. In their definition, q_i represents the i-th query, the document vectors are designated $\overrightarrow{a_{\cdot j}}, \quad j = 1, \ldots, m$ as before. The constants $\alpha, \beta, \gamma \leq 1$ are multipliers such that

$$q_{i+1} = \alpha\, q_i + \beta \sum_{\text{relevant}} \frac{a_{\cdot j}}{|a_{\cdot j}|} - \gamma \sum_{\text{non-relevant}} \frac{a_{\cdot j}}{|a_{\cdot j}|} \qquad (24)$$

In contrast to the work of the authors mentioned above, the software development life-cycle document corpus does not contain only text documents. Nevertheless we found that, as shown in Table II, with $\alpha = 1, \beta = \gamma = 0.5$ [12], feedback can improve certain query searches by as much as 20 percent.

While not a spectacular improvement, as one would expect with such an inhomogeneous corpus of artifacts, the improvement is nevertheless significant.

	Artifact Type					
Iteration	**System requirements**	**Architectural features**	**Use-Cases**	**Sequence diagrams**	**State diagrams**	**Classes Classes**
1	9	12	2	4	0	7
2	11	15	3	4	3	9
3	11	17	4	1	4	11
4	11	19	5	-1	9	12
5	10	20	6	-4	11	13

TABLE II
AVERAGE PERCENTAGE INCREASE IN F(2) AT EVERY QUERY REFINEMENT ITERATION OVER ALL THRESHOLD LEVELS AND 95% MATRIX REDUCTION WITH CORPUS WEIGHTING TF-IDF AND TF-ENTROPHY FOR THE QUERY WEIGHTING.

VIII. CONCLUSION

Improving the quality of professionally developed software systems has been an objective of research for software engineers and computer scientists since the early years of computing. Despite the progression through structured programming, object oriented programming, model driven architectures, UML and similar CASE tools, and so, on the progress, in comparison to progress in say, telecommunications, has been meagre, to say the least. The authors believe that a large factor contributing to this limited success in software engineering, as compared to other engineering disciplines, is the important human element involved. Human subjectivity and egoism play a role in writing computer code and always will.

In this paper we present a technique to reduce the impact of human subjectivity and the limited human capability to recognise synonyms and polysemy, introduced by the various developers in the document corpus of the development life-cycle. While other researchers have done similar work, we believe that the work reported here is more comprehensive in terms of the types of artifacts used, as well as the exposition of LSA improvement techniques. Our results show that software traceability, based on LSA, is viable in that, depending on the type of artifact, the reduction of the word-document matrix, the weightings used and the threshold value, an accuracy of 75% can be achieved in a metric which favours recall over precision.

IX. ACKNOWLEDGEMENT

We would like to thank Paolo Pileggi for his careful proofreading of the original draft.

References

[1] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[2] H. Chuang D. Lee and K. Seamons. Effectiveness of document ranking and relevance feedback techniques. *IEEE Software*, 14(2):67–75, March/April 1997.

[3] A. de Lucia *et al.* Enhancing an artefact management system with traceability recovery features. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenanc*, pages 306–315, 2004.

[4] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. L, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41:391–407, 1990.

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

[6] Jane Huffman Hayes, Alex Dekhtyar, and James Osborne. Improving requirements tracing via information retrieval. In *RE*, pages 138–, 2003.

[7] Jane Huffman Hayes, Inies C. M. Raphael, Elizabeth Ashlee Holbrook, and David M. Pruett. A case history of international space station requirement faul. In *ICECCS*, pages 17–26, 2006.

[8] Marco Lormans and Arie van Deursen. Can LSI help Reconstructing Requirements Traceability in Design and Test? In *CSMR*, pages 47–56, 2006.

[9] Carol Lundquist, David A. Grossman, and Ophir Frieder. Improving relevance feedback in the vector space model. pages 16–23. CIKM, 1997.

[10] A. Marcus and J.I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proc. of 25th International Conf. on Software Engineering*, pages 125–135, Portland, Oregon, 2003.

[11] Christos H. Papadimitriou, Hisao Tamaki, Prabhakar Raghavan, and Santosh Vempala. Latent semantic indexing: a probabilistic analysis. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 159–168, New York, NY, USA, 1998. ACM.

[12] G. Salton and C. Buckley. Improving retrieval performance by relevance feedback. Technical Report 87-881, Department of Computer Science, Cornell University, November 1987.

[13] G. Salton and C. Buckley. Parallel text search methods. *Communications of the ACM*, 31(2):202–215, February 1988.

[14] G. Salton and C. Buckley. Term weighing approaches in automatic text retrieval. *Journal of the American Society for Information Science*, 41(4):288–297, 1990.