Software Traceability using Latent Semantic Analysis and Relevance Feedback

Hans-Peter Krüger and Pieter S. Kritzinger
Technical Report CS08-01-00

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 *Software traceability (ST), in its broadest sense, is the process of tracking changes in the document corpus which are created throughout the software development life-cycle. However, traditional ST approaches require a lot of human effort to identify and consistently record inter-dependencies among software artifacts. In this paper we present an approach that reveals traceability links automatically using the information retrieval (IR) techniques of Latent Semantic Analysis (LSA) and Relevance Feedback and present a software tool to implement these ideas. We discuss in detail how software artifacts can be represented in a vector space model and how term extraction and weighting can be accomplished for UML artifacts, such as use-cases, interaction and state diagrams, as well as for source code and natural language text documents. We also explain how structural information which is always inherent in software artifacts can be preserved in the term extraction and weighting phase of creating traceable artifacts. Unlike other tools, we incorporate human knowledge through relevance feedback into the traceability link recovery process with the aim to improve the quality of traceability links. Finally, we illustrate the effectiveness of our tool-based approach and our proposals through a case study with a pilot software project and compare our results with those of a manual tracing process.*

Keywords: Requirements Engineering, Software Traceability, Software Development, Change Management, Latent Semantic Indexing, Relevance feedback, Information Retrieval, UML.

I. Introduction

The various types of artifacts that are produced throughout the software development life-cycle describe different levels of abstraction and perspectives of a software system. User requirements are initially formulated in natural language text during the requirements specification phase. During the design phase, a systems architecture is laid out that is capable of implementing the user and system requirements. Software components are identified and further divided into classes that implement the functionality specified in the requirements. A software architecture is usually described both in natural language text as well as in modeling languages such as UML. Many functional requirements can be mapped directly to UML use-cases, which are subsequently described in more detail through other UML diagram types like sequence, activity or state diagrams. Finally, a programming language such as Java is used to implement the user requirements according to the systems architecture.

Software traceability is concerned with tracking a change in one artifact to all the others. This is crucial to establish and maintain consistency between heterogeneous artifacts used throughout the system development life-cycle. In this paper we present a tool-assisted automated technique for traceability and to the extent that we used all artifacts normally created in the software development life-cycle, we believe this work is unique.

II. Related work

The idea of applying information retrieval methods to discover traceability links in a artifact corpus is not new. In [12] Marcus and Malectic present a traceability discovery approach that is based on LSA and supports traceability between source code and documentation. In their approach, an artifact corpus is build from text documents that are broken up into smaller sections or paragraphs, hoping that these will capture a single topic appropriately. Traceable artifacts are created from source code files while keywords are extracted from identifiers and comments. When performing a search, the user can choose the level of dimensionality reduction and a similarity threshold for the documents in the corpus. This approach seems to produce promising results. In a set of experiments, it was shown that their approach was able to discover 83% of all traceability links but unfortunately only with a precision of 53%, which means that the number of false links was actually very high.

Another approach that uses 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 (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.

Related to the same topic is the paper by Marco Lormans and Arie von Deursen [10]. In this work the authors focused on generating various requirements views through LSA between requirements-design and requirements-test cases. They applied LSA to three case study projects which differ strongly in size and provided artifacts. Unfortunately, it was not always clear

what kind of design artifacts they actually used and the results of their case studies were rather disappointing. Also in this paper, similar to the work of De Lucia [3] a discussion of how term extraction from artifacts and their weighting can be done, was missing.

In remainder of this paper we first present the functional design of LSITrace, a software tool which implements the methodology we discuss in the following three sections. In Sec. VII we describe how we build the artifact corpus from the various documents, diagrams and software code that comprise a working software system. Using LSITrace, we describe a case study and the results of experiments with different matrix reduction, threshold values and weighting techniques.

III. Tool Support

In order to test software traceability with Latent Semantic Analysis and relevance feedback we implemented the various ideas in a software tool. Fig. 1 presents an system overview of the tool. It allows one to recover traces among a variety

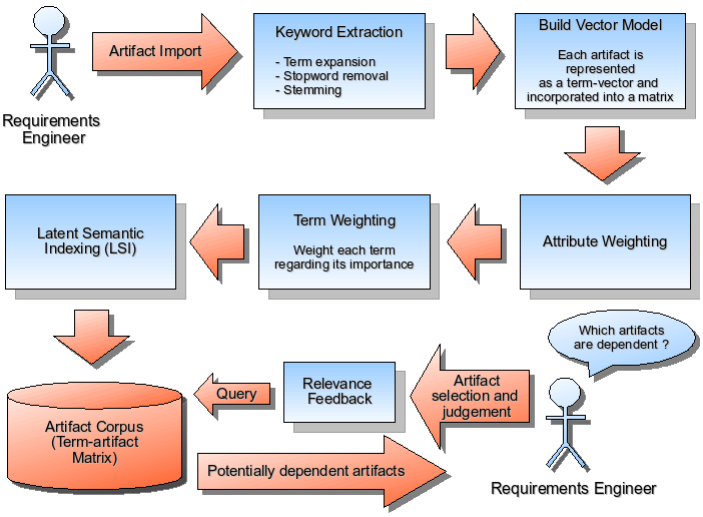

Fig. 1. The requirements engineer imports the software artifacts. The system then goes through a number of stages to create the artifact corpus. While tracing, the requirements engineer selects an artifact and issues a search request to the system. Relevance feedbacks assist the engineer to improve the search results.

of different artifacts, including PDF documents, UML use-cases, sequence and state diagrams, as well as source code classes. Firstly, the human engineer is required to import these artifacts in our as an Eclipse plugin [] designed tool. In the next phase we extract terms from the artifacts, which includes the removal of stop words and optionally the stemming of terms to their morphological root form (see Section VII-D). Software artifacts that are reduced to a set of terms will be mathematically represented as vectors (see Section IV).

Every term that is extracted from an artifact has an *artifact attribute*, i.e., the identifier of source code classes or the description of a UML use-case. The terms in an attribute are weighted differently, according to the attribute's importance to the content or meaning of the artifact itself. In an additional weighting phase (see Section V-B) we apply commonly used term weighting schemes, i.e., *Tf-Idf*, to the software artifacts and search queries.

Once the artifact corpus is built the tool allows the engineer to enter queries. The user either types in keywords that describe the artifact he is looking for, or selects existing artifacts in the corpus. In the example shown in Figure2 the engineer has decided to trace from an end-user requirement, called *Place Call*, to all system requirement artifacts. The search process starts initially and potentially dependant artifacts are ranked in a list by their similarity to the search query. The engineer either accepts the suggested artifact list or he can try to improve the results through his expert knowledge by adding relevant feedback to the discovery process. If the engineer is not satisfied with suggested artifact list, he can remove them from the candidate list and added his own suggestions the user-judged artifact list which the system then uses in a potentially improved search query. The process repeats until the engineer is satisfied with the result or no improvement happens. More details about the tool and its use can be found in the report by H-P Krüger [8].

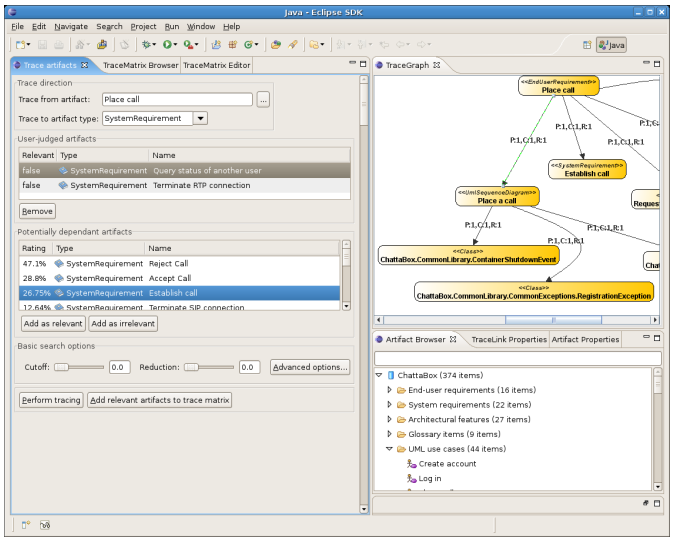

Fig. 2. The tool contains a number of different views to approach the traceability problem. The tracing and relevance feedback view is shown on the left side. The upper right side shows a graphical representation of the traceability matrix that was created among the recovery process. The bottom right view shows an artifact browser that allows to explore the corpus.

IV. Theory

In order to formalize the traceability process one derives a vector space model (VSM) [17] of the database or *corpus* of artifacts. In the VSM, 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 of m artifacts or the *corpus*. The conceptual vector space is represented mathematically by constructing a sparse matrix of terms by artifacts defined by

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

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

In order to decide whether two artifacts trace to one another, we compute the dot product, which directly corresponds to the cosine of the angle, between the column vectors $\overrightarrow{a.j}$, $j = 1,\ldots,m$. If two artifacts share common terms, their vectors will be closer to each other in n-dimensional vector space and are thus likely to share common terms.

Similarly, artifact vectors in close proximity to a query vector, which usually represents the artifact from which traceability links will be recovered, have a higher dot product than potentially unrelated artifacts, and are returned as the highest ranked artifacts. If the dot product of a retrieved artifact is below a chosen *threshold value*, implying that the vectors involved are wide apart in n-space, it is considered as not being relevant to the query.

In order to overcome the fundamental problem of *synonymy* and *polysemy* that plague many text search approaches which match words or terms of queries with words of documents, we apply a technique called Latent Semantic Analysis (LSA) [4]. There are usually many ways to express a given concept in natural language, so the literal terms in a query may not match those of a relevant document. LSA tries to reveal the underlying or latent semantics among documents that is partially obscured by variability in natural language word choice, which is often referred as *noise* in the literature. LSA applies a powerful technique in matrix computation known as Singular Value Decomposition (SVD) that reduces the dimensionality of a document matrix and as a side effect, reduces the *noise* in the document corpus. Rather than use an $n \times m$ matrix $A_{n\times m}$, one uses a factorization of the matrix, namely

$$A_{n\times m} = U_{n\times l}\Sigma_{l\times l}V^T_{l\times m} \tag{2}$$

where $U_{n\times l}$ and $V_{l\times m}$ are both orthogonal matrices, $l = min(n, m)$ and $\Sigma_{l\times l}$ is a diagonal matrix with elements $d_{ij} = 0$, whenever $i \neq j$, and $d_{ii} = d_i \geq 0$.

It can be shown that it always possible to find non-unique matrices $U_{n\times l}$ and $V_{l\times m}$ such that $d_1 \geq d_2 \geq \cdots \geq d_{r+1} = \cdots = d_n = 0$ where r is the rank of the matrix $A_{n\times m}$. Note that SVD neatly separates the original matrix $A_{n\times m}$ into the terms in matrix $U_{n\times l}$ from the artifacts in matrix $V_{l\times m}$.

When one can restricts the matrices $U_{n\times l}, V_{l\times m}$ and $\Sigma_{l\times l}$ to their first $k < n$ rows one obtains an approximation $A'_{k\times m}$ of the original matrix $A_{n\times m}$ where the number of terms have now been reduced to $k < n$.

$$A'_{k\times m} = U_{k\times l}\Sigma_{l\times l}V^T_{l\times m}. \tag{3}$$

An obvious benefit of $A'_{k\times m}$ is that it reduces the complexity of the vector space, hence decreasing both size of the corpus and the time for real time query analysis and data retrieval [9].

Choosing the number of dimensions k for $A'_{k\times m}$ is an interesting problem. While a reduction in k can remove much of the noise, keeping too few dimensions may lose important information. As discussed Deerwester *et al* [4] using a test database of medical abstracts, LSA performance can improve considerably after 10 or 20 dimensions, peaks between 70 and 100 dimensions, and then begins to diminish slowly.

In Section VIII we describe the results of a case study. The results prove that reducing the dimensionality of matrix $A_{n\times m}$ by 90%, that is, $k = 0.1$ not only reduces the storage and retrieval complexity, but improves the accuracy of a search significantly. This has to do with that fact that LSA spatially separates relevant from irrelevant documents. LSA is a global clustering technique that exploit inter-term relationships or the co-occurrence of terms, among all documents in the corpus to cluster similar documents in rank reduced space. In the case of software traceability, this means that software artifacts with a similar term usage will be closer to each other in rank reduced space. The clustering process groups similar artifacts that are related to a common concept and separates artifacts that are not related to the concept. This process of clustering is illustrated in Figures 3 and 4. The latter figure illustrates the rank reduced corpus in which 3 clusters have been identified. Clearly these are only illustrations since more than 3 dimensions, such as artifacts will have, cannot be visualized.

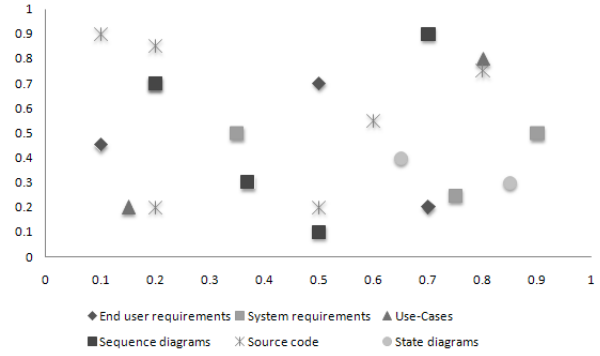

Fig. 3. Artifact space before clustering through LSA

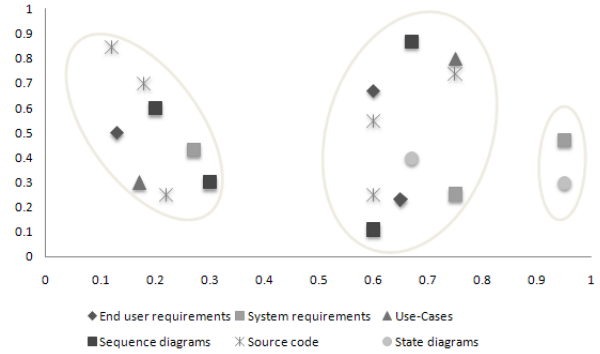

Fig. 4. Artifact space after clustering through LSA

V. Improving Latent Semantic Analysis (LSA)

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

A. 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. Natural language text artifacts, such as requirements or architectural specification, are a major part of the software development life-cycle corpus, stemming is useful to improve the traceability results as we shall see in Sec. IX.

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.

B. Term Weighting

Before we describe the various weighting schemes found in the literature [11], [4] 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, gf_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

$$\overline{tf}_{i,j} = \frac{tf_{i,j}}{\max_l\{tf_{l,j}\}} \tag{4}$$

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

$$w_{ij} = l_{ij} \times g_i \tag{5}$$

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

$$idf_i = \log\left(\frac{N}{af_i}\right) + 1 \tag{6}$$

The best known weighting scheme for natural language text documents is described by Salton *et al.* [15] and balances local- and global-weights and is known as the *Tf-Idf* scheme defined as follows:

$$\begin{aligned} tfidf_{i,j} &= \overline{tf}_{i,j} \cdot idf_i \\ &= \frac{f_{i,j}}{\max_l\{f_{l,j}\}} \cdot \log\left(\frac{N}{af_i}\right) \end{aligned} \tag{7}$$

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

$$\begin{aligned} \overline{tf'}_{i,j} &= \frac{tf_{i,j}}{af_i} \\ tfi_i &= 1 - \frac{1}{logN}\sum_{j=1}^{N} tf_{i,j} \times \log \overline{tf'}_{i,j} \end{aligned} \tag{8}$$

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

C. 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 Sec. I, is weighted by w_i where

$$w_i = \left(0.5 + 0.5 \times \frac{f_i}{\max f_i}\right) \times \log\left(\frac{N}{n_i}\right) \tag{9}$$

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. 5, considers the entropy of term i in the global context.

D. Relevance Feedback (RF)

Another way of improving the results of LSA is to use a local clustering technique or relevance feedback. Relevance feedback guides searches toward relevant documents by giving the system feedback as to which documents returned by a previous search are relevant to the initial query. The system can then use the feedback to perform a subsequent search that will result in a list of documents with a higher precision and recall.

In Dumais, et. al [5], Salton [16] 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 just view 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 $\overrightarrow{a_{\cdot j}}$, $j = 1, \ldots, m$ as before. The constants $\alpha, \beta, \gamma \leq 1$ are multipliers.

$$q_{i+1} = \alpha\, q_i + \beta \sum_{\text{relevant}} \frac{a_{\cdot j}}{|a_{\cdot j}|} - \gamma \sum_{\text{non-relevant}} \frac{a_{\cdot j}}{|a_{\cdot j}|} \tag{10}$$

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 show, in Sec. IX, where we use $\alpha = 1, \beta = \gamma = 0.5$ [14], that feedback can improve certain query searches by as much as 20 percent.

VI. Measuring LSA Traceability Performance

The two most popular metrics for evaluating IR performance are *Recall* and *Precision*[12]. Let C_i be the set of relevant artifacts for a user query i and R_i the set of all retrieved artifacts. *Recall* and *Precision* is then defined as follows:

$$Recall_i = \frac{|C_i \cap R_i|}{|C_i|} \tag{11}$$

$$Precision_i = \frac{|C_i \cap R_i|}{|R_i|} \tag{12}$$

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

$$Recall = \frac{\Sigma_i |C_i \cap R_i|}{\Sigma_i |C_i|} \quad (13)$$

$$Precision = \frac{\Sigma_i |C_i \cap R_i|}{\Sigma_i |R_i|} \quad (14)$$

In general, retrieving a lower number of artifacts for each query would result in higher precision, while a higher number of retrieved artifacts would increase the recall. Both values depend on the threshold used to cut the ranked list: in general, the higher the threshold the lower the recall and the higher the precision; and vice versa.

Another measure that is often used incorporates recall and precision into one single value. The most popular single value measure is the F-measure or balanced F-score as it computes the harmonic mean of precision and recall. It is also known as the $F(1)$ measure and given by:

$$F(\alpha)_i = (1+\alpha) \cdot \frac{Precision_i \cdot Recall_i}{\alpha \cdot Precision_i + Recall_i} \quad (15)$$

The influence of recall or precision in the single value of F can be changed by changing the α value. The two commonly used F measures are $F(2)$ which weights the recall twice as much as precision and $F(0.5)$ which is the other way around. In our experiments we decided to use $F(2)$ as the metric since we believe that recall is more important than precision for the software engineer. Failing to find important artifacts in the traceability link recovery process has serious implications for the correctness and costs of introducing a requirement change; hence completeness is preferred to precision.

VII. Building the Artifact Corpus

In section I we introduced the vector space model to present all software development life-cycle artifacts as n-dimensional vectors. However, this representation raises several questions. Which attributes of the software artifacts have to be incorporated in the vectors, i.e., are method names and comments required in the vector representation to reflect the meaning of a source code artifact? Which terms have to be extracted from these artifact attributes? Do only nouns and identifiers have to be considered? And finally, how must these terms be weighted to reflect their importance?

In the case study presented in Section VIII, we try to preserve existing dependency information inherent in software artifacts, such as use-case associations, by emphasizing the importance of terms that occur in these associations. We call this technique *attribute weighting*.

A. Natural language text artifacts

Natural language text artifacts, such as requirements or architectural specification, are a fundamental part of every software development life-cycle. We present an text artifact vector as the weighted sum of the two *attribute vectors* $\overrightarrow{\text{Name}}$ and $\overrightarrow{\text{Content}}$, see Eq. 16. The $\overrightarrow{\text{Content}}$ vector contains terms that were extracted from the description of an text artifact, i.e. from a user requirement. The $\overrightarrow{\text{Name}}$ 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.

$$\overrightarrow{\textbf{Text}} = w_1 \cdot \overrightarrow{\text{Name}} + w_2 \cdot \overrightarrow{\text{Description}} \quad (16)$$

B. UML diagrams

The UML notation comprises a large number of different diagram types. We concentrate on the representation of a subset of the most important diagrams, namely use-case, sequence as well as state diagrams. A use-case vector, see Eq. 17, is composed of the user-case name, the name of the associated subject or system and involved actors. Additionally, we also incorporate the comments that are either associated with the subject or the user-case itself. Use-cases can have a *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.

$$\begin{aligned}\overrightarrow{\textbf{Usecase}} &= w_1 \cdot \overrightarrow{\text{SubjectName}} + w_2 \cdot \overrightarrow{\text{UsecaseName}} \\ &+ \sum_{i=1}^{|\text{Actors}|} \left(w_3 \cdot \overrightarrow{\text{ActorName}_i} \right) + \sum_{i=1}^{|\text{Comments}|} \left(w_4 \cdot \overrightarrow{\text{Comment}_i} \right) \\ &+ \sum_{i=1}^{|\text{Associated usecases}|} \left(w_5 \cdot \overrightarrow{\text{UsecaseName}_i} \right) \end{aligned} \quad (17)$$

The representation of a UML sequence and state diagrams is slightly more complex than for a use-case. A sequence diagram vector, see Eq. 18, 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.

$$\begin{aligned}\overrightarrow{\textbf{SequenceDiagram}} &= w_1 \cdot \overrightarrow{\text{InteractionName}} + \sum_{i=1}^{|\text{Lifelines}|} \left(w_2 \cdot \overrightarrow{\text{LifelineName}} \right) \\ &+ \sum_{i=1}^{|\text{Messages}|} \left(w_4 \cdot \overrightarrow{\text{MessageName}_i} \right) \\ &+ \sum_{i=1}^{|\text{Comments}|} \left(w_3 \cdot \overrightarrow{\text{Comment}_i} \right) \end{aligned} \quad (18)$$

The representation of a UML state diagram is very similar to the sequence diagram. A state diagram vector, see Eq. 19, is composed of the state machine name, all diagram comments, all state names as well as the state actions (entry, exit and do). Additionally, also the labels of the state transition are incorporated in the diagram representation.

$$\begin{aligned}\overrightarrow{\textbf{StateDiagram}} &= w_1 \cdot \overrightarrow{\text{StateMachineName}} + \sum_{i=1}^{|\text{Comments}|} \left(w_3 \cdot \overrightarrow{\text{Comment}_i} \right) \\ &+ \sum_{i=1}^{|\text{States}|} w_2 \cdot \overrightarrow{\text{StateName}} + w_4 \cdot \left(\overrightarrow{\text{StateEntry}} + \overrightarrow{\text{StateExit}} \right) \\ &+ w_5 \cdot \overrightarrow{\text{StateDo}} + \sum_{i=1}^{|\text{Transitions}|} w_6 \cdot \overrightarrow{\text{TransitionLinkName}_i} \end{aligned} \quad (19)$$

C. Source code

Changes to user or system requirements invariably result in modifications to the source code and vice versa. These kind of changes are costly as source code artifacts are complex. Since

source code artifacts describe a software system on the lowest abstraction level, the number of artifacts the human engineer has to examine for the purpose of validating and maintaining traceability links is higher than for other artifacts types.

In order to represent source code artifacts adequately in vector space, we propose the model shown in Eqs. 20 - 25. 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. Additionally, we also incorporate comments and string literals terms found in the body of methods.

$$\overrightarrow{\textbf{PackageDeclaration}} = w_1 \cdot \overrightarrow{\text{Comment}} + \sum_{i=1}^{|\text{Subpackages}|} \left(w_2 \cdot \overrightarrow{\text{SubpackageName}_i} \right) + w_3 \cdot \overrightarrow{\text{Name}} \quad (20)$$

$$\overrightarrow{\textbf{ClassDeclaration}} = w_1 \cdot \overrightarrow{\text{Comment}} + w_2 \cdot \overrightarrow{\text{Name}} + \sum_{i=1}^{|\text{Super classes}|} w_3 \cdot \overrightarrow{\text{SuperClassName}_i} \quad (21)$$

$$\overrightarrow{\textbf{FieldDeclaration}} = w_1 \cdot \overrightarrow{\text{Comment}} + w_2 \cdot \overrightarrow{\text{Type}} + w_3 \cdot \overrightarrow{\text{Name}} \quad (22)$$

$$\overrightarrow{\textbf{MethodDeclaration}} = w_1 \cdot \overrightarrow{\text{Comment}} + w_2 \cdot \overrightarrow{\text{ReturnType}} + w_3 \cdot \overrightarrow{\text{Name}} + \sum_{i=1}^{|\text{Parameters}|} \left(w_4 \cdot \overrightarrow{\text{ParamType}_i} + w_5 \cdot \overrightarrow{\text{Name}_i} \right) \quad (23)$$

$$\overrightarrow{\textbf{MethodBody}} = \sum_{i=1}^{|\text{Comments}|} \left(w_1 \cdot \overrightarrow{\text{Comment}_i} \right) + \sum_{i=1}^{|\text{String literals}|} \left(w_2 \cdot \overrightarrow{\text{Literal}_i} \right) + \sum_{i=1}^{|\text{Remaining identifiers}|} w_3 \cdot \overrightarrow{\text{Ident}_i} \quad (24)$$

$$\overrightarrow{\textbf{Class}} = \sum_{i=1}^{|\text{Package declarations}|} \left(\overrightarrow{\text{Declaration}_i} \right) + \overrightarrow{\text{ClassDeclaration}} + \sum_{i=1}^{|\text{Method declarations}|} + \left(\overrightarrow{\text{Declaration}_i} + \overrightarrow{\text{MethodBody}} \right) + \sum_{i=1}^{|\text{Field declarations}|} + \overrightarrow{\text{Declaration}_i} \quad (25)$$

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 [12], we produced an abstract syntax tree (AST) of every source code artifact. This allowed 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. Term extraction

After we identified the artifact attributes to incorporate in the vector representation of the various software artifacts, we next extract terms from these attributes. In the first step of the extraction process every term is separated from the initial string and saved 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 subsequently found.

As an example, the tree in Figure VII-D illustrates the term extraction process for the label of a lifeline message of an UML sequence diagram which consists of a condition and method call. In the extraction process we separate the original string shown in the root node into its constituent terms. Compound terms such as `onReceiveData` and `SIPMsg` are further divided and added among the compound term to 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 Sec. V-A). Although LSA itself provides a mean to overcome *synonymy* we found in our case study (see Sec.VIII) that additional stemming of terms improves search results.

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 VII-D, 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 [13]. We mainly enhanced this list with keywords usually used in programming languages, such as `int`, `boolean` or `object`.

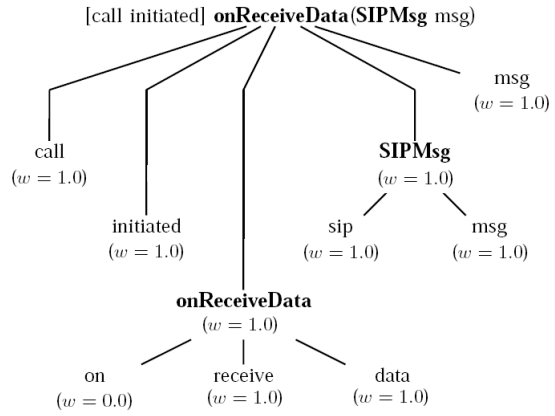

Fig. 5. Example expanded term tree

VIII. 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. Confidential requirements mostly require software companies not to make artifacts such as required, available. Open software systems are largely bereft of documentation other than the source code. Hence like other studies, for instance that by Marcus [12], 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), and which was proscribed to follow best programming practises in describing the software in a detailed and complete fashion using requirements analysis, design, implementation and testing phase.

Several case studies can also be found in the literature. Marcus and Maletic [12] 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 [10] and De Lucia et al. [3] conducted their studies on less technical software projects with a broader set of artifact types.

A. Tracing Links manually

In order to assess the performance of the automatic traceability link recovery method, one obviously needs to manually identify the best set of traceability links among the chosen software artifacts. 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 LSATrace tool removes the individual human interpretation which is inevitably present in any manual traceability technique. However, in order to say something about the performance of our proposals, we needed the manually discovered links, however biased.

In the case study 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 get a better understanding of the dependencies between the artifacts. In the end we 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.

IX. 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. We applied the most common term weights found in the literature [16] 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 as S_n and defined as follows:

$$\begin{aligned} S_n = (&L_{Corpus}, G_{Corpus}, L_{Query}, G_{Query}) \text{ where} \\ &L_{Corpus} \in \{Log, MaxTf, Tf\} \wedge \\ &G_{Corpus} \in \{Entrophy, Idf, None\} \wedge \\ &L_{Query} \in \{AugTf, Log, MaxTf, Tf\} \wedge \\ &G_{Query} \in \{Entrophy, Idf, None\} \end{aligned} \tag{26}$$

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 (see amount to 108 for our choice of weights in Sec. V-B); 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.

In our report, because of the relevance feedback process, we placed more emphasis on recall than precision in our experiments and therefore used $F(2)$, which weights the recall twice as much as precision. In all experiments we traced from the user requirements to each of the other artifact classes. Moreover, we can obviously only report on results for which we had manually traced the links (see Sec. VIII-A) for comparison.

In summary, the statistics of our artifact corpus is given in Table IX.

A. Results

The fundamental question, naturally, is how well does LSA work when tracing amongst software artifacts? Before deciding that question, we investigated the best combination of corpus and query weighting to use. It turned out that there was no single answer to this question. Figure 6 plots the F(2) value when tracing from End-user requirements to Use cases for a threshold of 0.1 and various matrix A' reductions and all possible combinations of the weightings described in Sec. V-B. Not only does the best combination depend on the percentage of the matrix reduction, but it also depends on what is being traced to and the threshold value.

The results of all these experiments are simply to numerous to display and the reader is referred to [8] for all the data. We determined that the weighting scheme which gave the best overall results uses the corpus weighting Tf-Idf and Tf-Entrophy for the query weighting. We therefore used this weighting scheme in all the results to follow.

For the results in Fig. 7 we plotted the F(2) value while tracing from End-user requirements to all remaining artifacts

	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
CORPUS ARTIFACT STATISTICS.

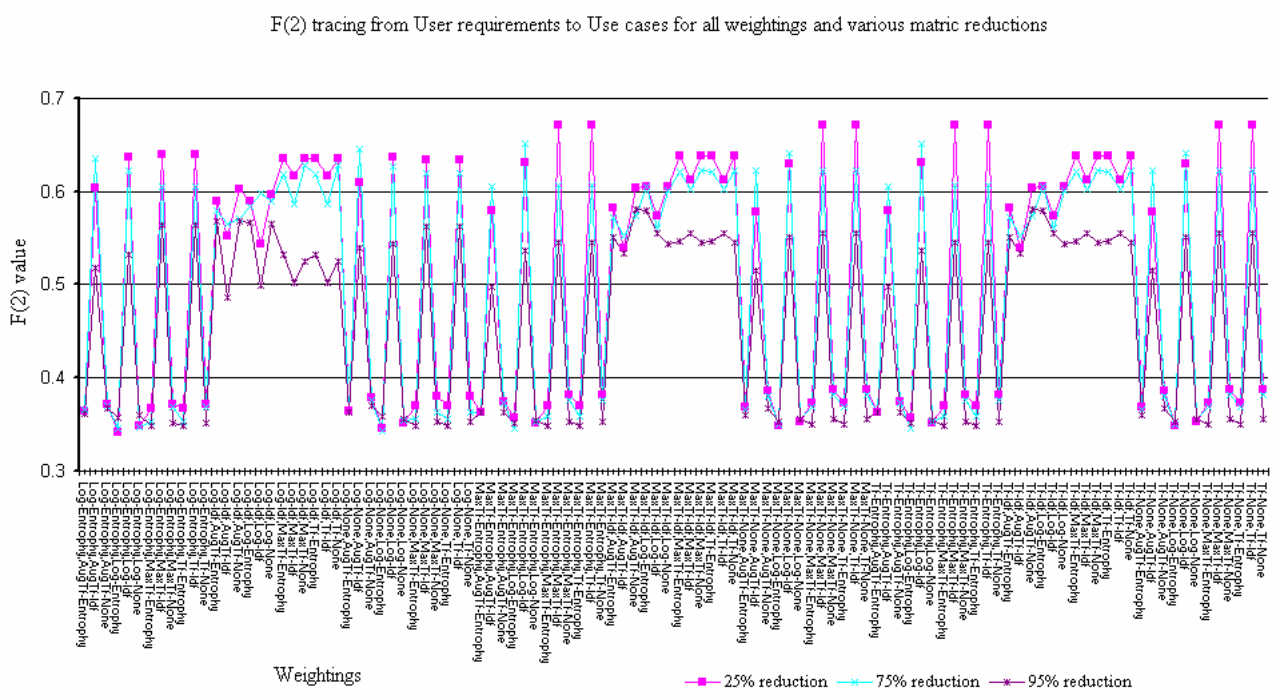

Fig. 6. Tracing from End-user requirements to Use cases for a threshold of 0.1 and various matrix A' reductions and all weighting combinations.

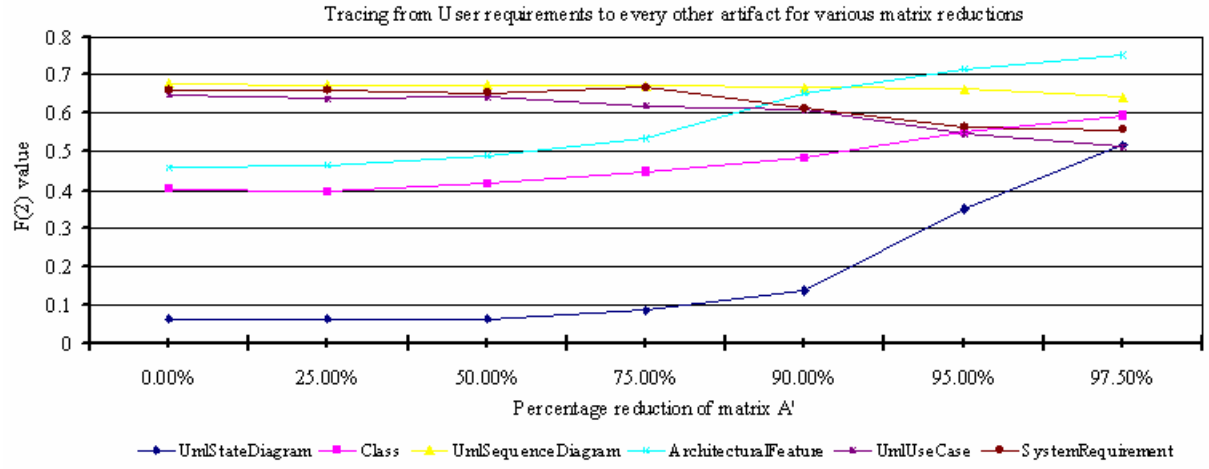

Fig. 7. Tracing from End-user requirements to remaining artifacts for threshold value 0.1 and various matrix A' reductions with stemming.

for a threshold value 0.1 and various matrix A' reductions using stemming. The F(2) values range from a low 51% for tracing to Use cases to a high of 75% for architectural features. Although an exact comparison is not possible since the experiments differ, these results are considerably better than those found by Settini *et al* [18]. We subsequently repeated this experiment by tracing from End-user requirements to remaining artifacts averaged over all matrix reductions for various threshold values. Threshold

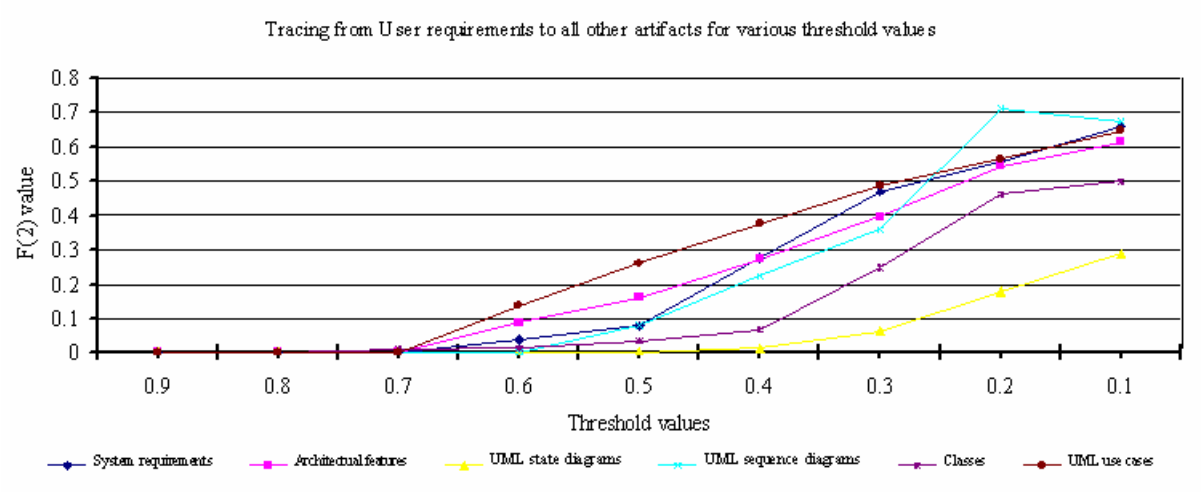

Fig. 8. F(2) value tracing from End-user requirements to remaining artifacts averaged over all matrix reductions for various threshold values.

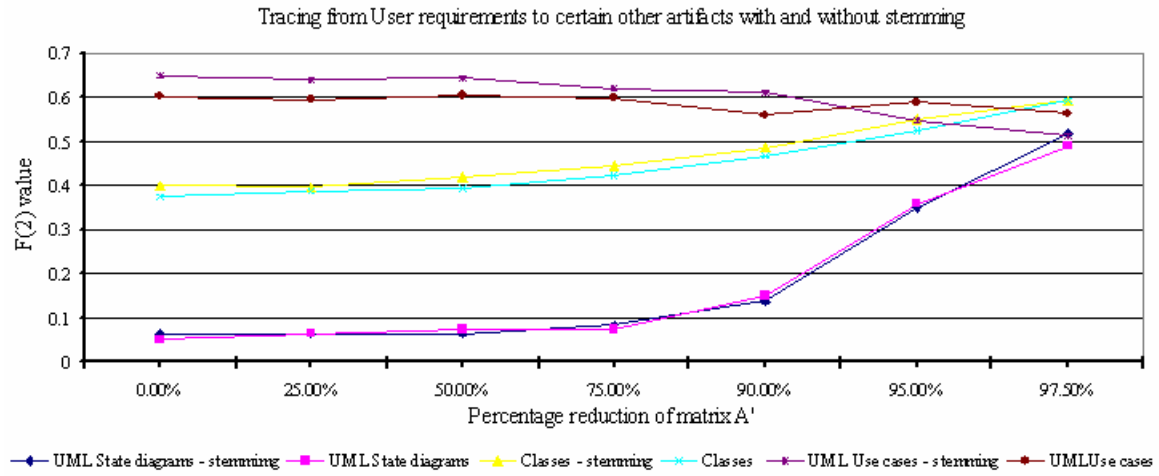

Fig. 9. F(2) value tracing from End-user requirements to remaining artifacts for threshold value 0.1 and various matrix A' reductions with and without stemming.

values of less than 0.3 are clearly so low that all significance is lost. Even for a threshold value of 0.1 the F(2) value, averaged over matrix reductions from 0% to 97.5%, is a low 30%. It is obvious from these last two set of results that a matrix reduction of 97.5% and a threshold value of 0.1 give the best results for the value F(2) indicating that there must be a great deal of sematic noise in our artifact corpus. Since a large matrix reduction also improves the efficiency of the technique, this result is particularly significant. Next we wanted to know the effect of stemming on the efficiency of LSA and our methodology. Figure 9 is therefore a plot of the F(2) value tracing from End-user requirements to remaining artifacts for threshold value 0.1 and various matrix A' reductions with and without stemming. Although the improvement in the results is marginal, stemming has clearly improves the F(2) value at all matrix reduction and in almost all cases.

B. Relevance Feedback

In the final experiment we used relevance feedback to iteratively refine a search query. As described by Lundquist *et al.* [11], relevance feedback is a process through which a query is selectively modified to retrieve more relevant documents from a collection than the unmodified original version. The query can be modified by either adjusting the term weights, i.e., increasing or decreasing the weight of a term, by adding new terms or by using the combination of these two approaches. The process of judging and performing searches can be repeated until the desired quality of traceability links is reached.

Relevance feedback, as the name implies, dependents upon user input to determine relevant documents. Alternatively one can simply assume that only the top ranked documents are relevant without any user intervention. In our experiments we used neither of these techniques. As explained in Sec. VIII-A, we traced the links between some artifacts manually in order to compare our automated technique with what may be considered the correct results. In our query refinement we simulate the user feedback by finding the top ranked artifact in our manually traced matrix. If the artifact can be found in the traceability matrix we mark it as being relevant, otherwise it is marked as irrelevant. Afterwards the artifact will be incorporated into a new, potentially improved search query and another search operation performed. We arbitrarily performed these query improvements five times computing the F(2) value at all threshold levels at every query iteration. In order not to distort the performance

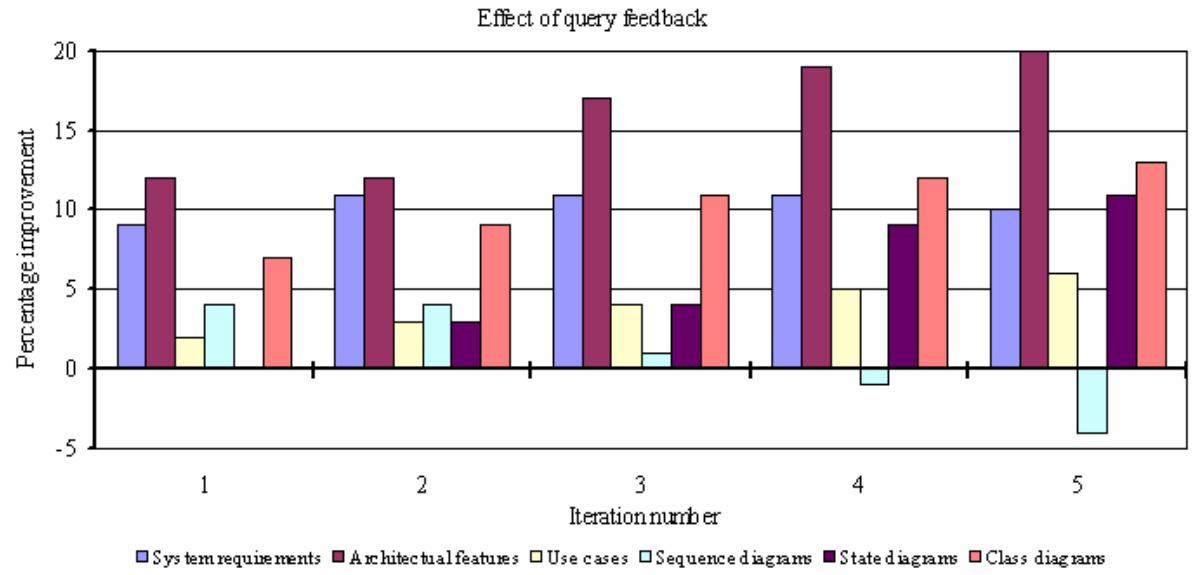

Fig. 10. F(2) percentage performance increase at every query refinement iteration averaged over all threshold levels and 95 percent matrix reduction.

results we do not consider artifacts previously considered in the result set. Each time we only considered the residual collection in which all previously used artifacts have been removed.

Table 10 shows the F(2) values averaged over all threshold levels for each query refinement iteration. Iteration 1 denotes the original performance without applying relevance feedback. The first query refinement in which only a single artifact was rated improved the results for almost all artifacts types. Results improved in average over all threshold levels by 9% for system requirements, 12% for architectural features, 2% for use-cases, 4% for sequence diagrams, none for state diagrams and 7% for class diagrams.

Further iterations continued to improve the results for all artifact types except for sequence diagrams. After 5 iterations results improved by 10% for system requirements, a staggering 20% for architectural features, 6% for use-cases, -4% for sequence diagrams, 11% for state diagrams and 13% for class diagrams.

X. Conclusion

The objective of software traceability is to find the links between user requirements and artifacts produced during the software development life-cycle. Although techniques for generating and validating traceability are available, in practice it often suffers from the huge effort and complexity of creating and maintaining traces or from incomplete trace information that cannot assist software engineers in real-world problems.

As an alternative we present a tool-supported approach to automatically trace between various software artifacts normally created in the software development life-cycle using LSA and relevance feedback. We also describe some enhancements to the basic LSA method, including various term weighting and in particular, what we call attribute weights. To the extent that we used all artifacts normally created in the software development life-cycle, we believe this work presented is unique.

With a huge database of results there are still many aspects of the methodology that can be explored. Why are the particular weightings we used generally the best, but not the best for all artifacts? No doubt the length of the various artifacts have an impact on this. The one conclusion that is evident is that a threshold value of 0.1 and a matrix reduction of 95% deliver the best F(2) result. It is important to emphasize again that our results in all cases reflect both the validity and the precision in the F(2) value.

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] Hans-Peter Krüger. Software traceability using latent semantic analysis and relevance feedbac. Master's thesis, University of Cape Town, September 2008.

[9] T. A. Letsche and M. W. Berry. Large-scale information retrieval with latent semantic indexing. *Information Sciences*, 110(1–4):105 – 137, 1997.

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

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

[12] 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.

[13] 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.

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

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

[16] 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.

[17] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[18] Raffaella Settimi, Jane Cleland-Huang, Oussama Ben Khadra, Jigar Mody, Wiktor Lukasik, and Chris DePalma. Supporting software evolution through dynamically retrieving traces to uml artifacts. In *IWPSE '04: Proceedings of the Principles of Software Evolution, 7th International Workshop*, pages 49–54, Washington, DC, USA, 2004. IEEE Computer Society.