Information Retrieval Models for Recovering Traceability Links between Code and Documentation

G. Antoniol*, G. Canfora*, G. Casazza**, A. De Lucia*
antoniol@ieee.org {gerardo.canfora,gec,delucia}@unisannio.it
(*)University of Sannio, Faculty of Engineering - Piazza Roma, I-82100 Benevento, Italy
(**)University of Naples Federico II, DIS - Via Claudio 21, I-80125 Naples, Italy

Abstract

The research described in this paper is concerned with the application of information retrieval to software maintenance, and in particular to the problem of recovering traceability links between the source code of a system and its free text documentation.

We introduce a method based on the general idea of vector space information retrieval and apply it in two case studies to trace C++ source code onto manual pages and Java code onto functional requirements. The case studies discussed in this paper replicate the studies presented in references [3] and [2], respectively, where a probabilistic information retrieval model was applied. We compare the results of vector space and probabilistic models and formulate hypotheses to explain the differences.

1. Introduction

Automated Information Retrieval (IR) systems are concerned with the retrieval of documents from (usually very large) document databases, based on user information needs [8]. They prepare the collection of documents for retrieval through an indexing process; user needs are captured by phrases which are themselves indexed and used to rank the documents.

IR has proven useful in many disparate areas, including the management of huge scientific and legal literature, office automation, and the support to complex engineering projects such as software engineering projects.

We believe that IR techniques can help software maintenance by providing a way to semi-automatically recovering traceability links between the documentation of a system and its source code. The underlying assumption is that programmers use meaningful names for code items; indeed, we believe that most of the application domain knowledge that programmers process when writing the code is captured by the mnemonics for identifiers.

Most of the documentation that accompany large software systems consists of free text documents expressed in a natural language. Examples include requirements and design documents, user manuals, logs of errors, maintenance journals, design decisions, reports from inspection and review sessions, and also annotations of individual programmers and teams. In addition, free text documents often capture the available knowledge of the application domain, for example in the form of laws and regulations or in technical/scientific handbooks. Therefore, techniques to recover traceability links between code and free text documents are a precious aid to software maintenance as they bridge the gap between different views of a systems, and between the system's views and its domain of application.

Reference [3] highlights several scenarios of software maintenance and evolution that would benefit from the existence of such links, including program comprehension, design recovery, requirement tracing, impact analysis, and reuse of existing software. The paper also introduces an IR method to trace source code, and in particular C++ classes, onto free text documents and discusses the results obtained in a case study where the code of the LEDA library (Library of Efficient Data Types and Algorithms — available from http://www.mpi.sb.mpg.de/LEDA) was traced back onto the manual pages. The results — measured in terms of two well known IR metrics, namely precision and recall — were encouraging and, therefore, we applied the method in other scenarios, such as impact analysis [1] and requirement tracing [2]. In all cases the results were satisfactory and this enforced our believe that IR can play a useful role in maintenance.

The case studies discussed in references [3, 2] applied a probabilistic IR model. With this model, documents D_i are ranked according to the probability $Pr(D_i|Q)$ of being relevant to a query Q extracted from a source code component. To compute this ranking we exploited the idea of a language model, i.e. a stochastic model that assigns a probability to every string of words taken from a prescribed vocabulary [7]. We estimated a language model (actually, a unigram approximation of the model) for each document or identifiable section and used a Bayesian classifier to score the sequences of mnemonics extracted from each source code class against the models. A high score indicated a high probability that a particular sequence of mnemonics be relevant to the document; therefore, we interpreted it as an indication of the existence of a semantic link between the class from which the sequence had been extracted and the document.

In the present paper we test the hypothesis that using other IR models could also give good results. In particular, we describe a method to recover traceability links that uses a vector space IR model [10, 15]. This model treats documents and queries as vectors in an n-dimensional space, where n is the number of indexing features (in our case, words in the vocabulary). Documents are ranked against queries by computing some distance functions between the corresponding vectors. In this paper, we use the cosine of the angle between the vectors to rank the documents [10].

The method has been applied in two case studies to trace C++ source code onto manual pages and Java code onto functional requirements. These case studies replicate the studies described in references [3] and [2], respectively. The results are in both cases satisfactory and confirm the hypothesis that IR, either probabilistic or vector space models, provides a practicable solution to the problem of semi-automatically recovering traceability links.

There are three main intended contribution of this paper:

- we present a method to recover traceability links between source code and free text documentation based on the general idea of vector space IR;
- we provide experimental results of applying the method in two case studies;
- we compare the performances of probabilistic and vector space IR models when applied to the particular problem of traceability link recovery and formulate hypotheses to explain the differences.

The reminder of the paper is organized as follows. Section 2 presents the traceability link recovery process and discusses the IR model exploited. Experimental results are presented in section 3, while section 4 discusses the performances of IR models when applied to the problem of recovering traceability links. Concluding remarks and a discussion of related and future work are given in section 5.

2. An IR method to recover traceability links

Our method to recover traceability links between code and free text documentation uses the identifiers extracted from a class as a query to retrieve the documents relevant to the class. In particular, we apply a vector space IR model to rank the available documents against the class.

This section describes the overall traceability link recovery process, gives background information on the IR model applied, and discusses tool support.

2.1 The process

Figure 1 shows the overall process of traceability link recovery using IR models. The figure highlights two paths of activities, one to prepare the document for retrieval (the lower path) and the other to extract the queries from code (the upper path).

In the first path, documents are indexed based on a vocabulary that is extracted from the documents themselves. The construction of the vocabulary and the indexing of the documents are preceded by a text normalization phase performed at three levels of accuracy:

1. at the first level all capital letters are transformed into lower case letters;
2. at the second level stop-words (such as articles, punctuation, numbers, etc) are removed;
3. at the third level a morphological analysis is used to convert plurals into singulars and to reconduct all the flexed verbs to the infinity form.

The second path builds and indexes a query for each source code class. The construction of a query consists of three steps:

1. identifier extraction, that parses class source code and extracts the list of its identifiers;
2. identifier separation, that splits into separate words the identifiers composed of two or more words (i.e. AmountDue and amount_due);
3. text normalization, that applies the three steps described above for document indexing.

Finally, a classifier computes the similarity between queries and documents and returns, for each class, a ranked list of documents.

Of course, indexing the documents and the queries and ranking the documents against a query depend on the particular IR model adopted. For example, the probabilistic model applied in references [3, 2] indexes a document by

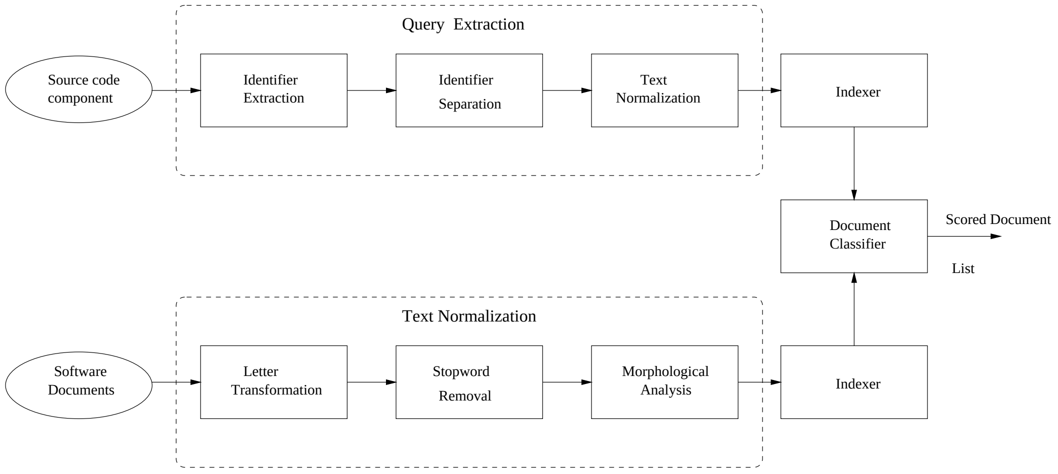

Figure 1. Traceability Recovery Process.

computing its stochastic language model [7], whereas the list of identifiers that define a query is not indexed at all. The similarity between a document and a query is computed as the product of the probabilities that each identifier in the query appears in the document too.

2.2 Vector space IR: background notions

Vector space IR models map each document and each query onto a vector [10]. In our case, each element of the vector corresponds to a word (or term) in a vocabulary extracted from the documents themselves. If V is the size of the vocabulary, then the vector $[d_{i,1}, d_{i,2}, \ldots d_{i,V}]$ represents the document D_i. The j-th element $d_{i,j}$ is a measure of the weight of the j-th term of the vocabulary in the document D_i. Different measures have been proposed for this weight. In the simplest case it is a boolean value, either 1 if the j-th term occurs in the document D_i, or 0 otherwise; in other cases more complex measures are constructed based on the frequency of the terms in the documents.

We use a well known IR metric called $tf - idf$ [15]. According to this metric, the j-th element $d_{i,j}$ is derived from the *term frequency* $tf_{i,j}$ of the j-th term in the document D_i and the *inverse document frequency* idf_j of the term over the entire set of documents. The term frequency $tf_{i,j}$ is the ratio between the number of occurrences of word j-th over the total number of words contained in the document D_i. The inverse document frequency idf_j is defined as:

$$idf_j = \frac{Total\ Number\ of\ Documents}{Number\ of\ Documents\ containing\ the\ j^{th}\ term}$$

The vector element $d_{i,j}$ is:

$$d_{i,j} = tf_{i,j} * log(idf_j)$$

The term $log(idf_j)$ acts as a weight for the frequency of a word in a document: the more the word is specific to the document, the higher the weight.

The list of identifiers extracted from a class Q — that is, a query Q — is represented in a similar way by a vector $[q_1, q_2, ...q_V]$. The similarity between a document D_i and a class/query Q is computed as the cosine of the angle between the corresponding vectors:

$$Similarity(D_i, Q) = s_{i,Q} =$$
$$= \frac{\sum_{j=1}^{V} d_{i,j} q_j}{\sqrt{\sum_{h=1}^{V} (d_{i,h})^2 * \sum_{k=1}^{V} (q_k)^2}}$$

Documents are ranked against a class by decreasing similarity.

2.3 Tool support

We have developed a toolkit that supports, and partially automates, the process shown in Figure 1.

We use top-down recursive parsers to analyze C++ and Java source code. The parse trees are traversed and each time a class is encountered the comments, if any, and the identifiers of attributes, methods, and method parameters are stored in support files. For the present study comments were disregarded: the entire traceability link recovery process relies on the mnemonics used for classes, attributes, methods and parameters.

We have integrated public domain facilities and tools developed in house to assist text processing for the English and Italian languages. Identifier separation is performed in two steps: the first step is completely automated and recognizes words separated by underscore and sequences of words starting with capital letters. The second step is semi-automatic: the tool exploits spelling facilities to prompt the software engineer with the words that might be separated. The first two steps of text normalization, namely letter transformation and stop-word removing, have also been completely automated. Finally, we have implemented a semi-automatic stemmer that uses thesaurus facilities to help users to reconduct flexed words to their roots.

The final step of cosine computation and document ranking is implemented by simple Perl scripts.

3. Case studies

In previous papers [3, 2], we applied a traceability link recovery method based on the probabilistic IR model in two case studies with different characteristics. The case studies have been replicated using the vector space IR model presented in the previous section, and the results are discussed in this section.

We assess the results using two widely accepted IR metrics, namely *recall* and *precision* [8].

Recall is the ratio of the number of relevant documents retrieved for a given query over the total number of relevant documents for that query. *Precision* is the ratio of the number of relevant documents retrieved over the total number of documents retrieved.

Recovering traceability links is a semi-automatic process. The main role of IR tools consists of restricting the document space, while recovering all the documents relevant to each source code component. Without tool support, one must analyze all the documents before discovering that a given class is not described by any document; with a restricted document space the number of documents to analyze is generally much lower. This means that high recall values (possibly 100 %) should be pursued; of course, in this case higher precision values reduce the effort required to discard false positives (documents that are retrieved but are not relevant to a given query).

It is worth noting that the recall is undefined for queries that do not have relevant documents associated. However, these queries may retrieve false positives that have to be discarded by the software engineer. To take into account such queries we used the following aggregate formulas:

$$Recall = \frac{\sum_i \#(Relevant_i \wedge Retrieved_i)}{\sum_i \#Relevant_i}\%$$

$$Precision = \frac{\sum_i \#(Relevant_i \wedge Retrieved_i)}{\sum_i \#Retrieved_i}\%$$

where i ranges over the entire query set, including the queries with no associated documents. These queries do not affect the computation of the recall ($Relevant_i$ is the empty set), while they negatively affect the computation of the precision whenever $Retrieved_i$ is not the empty set. This negative influence takes into account the effort required to discard false positives.

3.1 LEDA case study

The first case study was a freely available C++ library of foundation classes, called LEDA (Library of Efficient Data types and Algorithms), developed and distributed by Max-Planck-Institut für Informatik, Saarbrücken, Germany. We analyzed the code and the documentation of the release 3.4, consisting of 95 KLOC, 208 Classes and 88 manual pages. The aim was to map source code classes onto manual pages.

The LEDA manual pages contain a high number of identifiers that also appear in the source code. Actually, the LEDA team generated manual pages with scripts that extract comments from the source files. A markup language was used to identify the comment fragments to be extracted. Function names, parameter names, and data type names contained in these comments appear in the manual pages, thus making the traceability link recovery task easier. For this reason, and to make the results comparable with those obtained with the probabilistic model [3], we applied a simplified version of the process shown in Figure 1. The simplification concerned the identifier separation and the text normalization activities: in particular, identifier separation only consisted of splitting identifiers containing underscores, while text normalization was performed only at the first level of accuracy, i.e. the transformation of capital letters into lower case letters.

To validate the results, we used a 208×88 traceability matrix linking each class to the manual page describing it. As outlined in [3], each class was described by at most one manual page, and many classes (110) were not described by any manual page. The number of links in the traceability matrix was 98. Ten manual pages did not describe LEDA classes, but basic concepts and algorithms, thus the number of relevant manual pages was 78. This means that some manual pages described more than one class: for example, very often an abstract class and its derived concrete classes were described by the same manual page.

Table 1 shows the results. The first columns (Cut) shows the number of documents retained for each query (first N documents in the ranked list); the second and the third

Cut	Retrieved	Relevant	Precision	Recall	Prob. Rel.
1	208	52	25.00 %	53.06 %	81
2	416	71	17.06 %	72.44 %	88
3	624	79	12.66 %	80.61 %	93
4	832	82	9.85 %	83.67 %	93
5	1040	85	8.17 %	86.73 %	93
6	1248	89	7.13 %	90.81 %	93
7	1456	90	6.18 %	91.83 %	94
8	1664	93	5.58 %	94.89 %	94
9	1872	95	5.07 %	96.93 %	95
10	2080	96	4.61 %	97.95 %	95
11	2288	96	4.19 %	97.95 %	95
12	2496	98	3.92 %	100.00 %	96

Table 1. LEDA results

columns show (for each cut level) the total number of retrieved documents (for all queries) and the total number of retrieved documents that are also relevant, respectively; the third and fourth columns show the aggregate precision and recall for all queries, respectively; finally, last column shows the total number of relevant documents retrieved by applying the probabilistic model [3].

The poor results of the precision are due to the fact that most of the queries (110) were derived from classes without relevant manual pages associated (these queries contribute to the total number of retrieved documents). A moderate number of retained candidates (12) was required to recover all the traceability links (100 % of recall). This provides evidence to support our hypothesis that IR models are suitable for recovering traceability links between code and documentation. A further favorable argument is the fact that the results are not very different from those achieved in [3] with a probabilistic model: indeed, the main difference is that the results of the vector space model are more smoothed, while the probabilistic model tends to retrieve very soon a high number of relevant documents (see Figure 2). However, both models converge to 100 % of recall with a moderate number of retained candidates; for the probabilistic model 100 % of recall is achieved when cutting the ranked list of retrieved documents at 17 candidates [3] (not shown in the table), thus confirming a more irregular behavior.

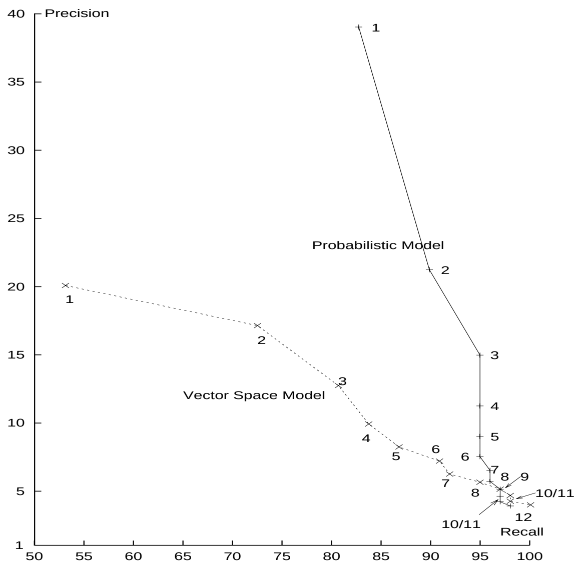

Figure 2. LEDA Precision/Recall results.

3.2 Albergate case study

The second case study was a software system, called Albergate, developed in Java according to a waterfall process. For this system all the documentation prescribed by the software development process was available (e.g., requirement documents, design documents, test cases, etc.). Albergate is a software system designed to implement all the operations required to administrate and manage a small/medium size hotel (room reservation, bill calculation, etc.). It was developed from scratch by a team of final year students at the University of Verona (Italy) on the basis of 16 functional requirements expressed (as well as all the other system documentation) in the Italian language. Albergate exploits a relational database and consists of 95 classes and about 20 KLOC. The aim of this case study was to trace source code classes onto functional requirements. We focused on the 60 classes implementing the user interface of the software system.

To validate the results, the original developers were required to provide a 16 $\times$ 60 traceability matrix linking each requirement to the classes implementing it. Most of the functional requirements were implemented by a low num-

Cut	Retrieved	Relevant	Precision	Recall	Prob. Rel.
1	60	29	48.33 %	50.00 %	29
2	120	34	28.33 %	58.62 %	41
3	180	46	25.55 %	79.31 %	45
4	240	51	21.25 %	87.93 %	51
5	300	54	18.00 %	93.10 %	57
6	360	55	15.27 %	94.82 %	58
7	420	58	13.80 %	100.00 %	58

Table 2. Albergate results with improved process

ber of classes: on the average, a requirement was implemented by about 4 classes with a maximum of 10. Most classes were associated to one requirement, only 6 classes were associated to two requirements, and 8 classes were not associated to any functional requirement. The total number of links in the traceability matrix was 58.

In this case study we applied the full version of the text processing steps in Figure 1 (these steps are described in section 2.1). The motivation was that the relative distance between source code and documents was higher than in the `LEDA` case study. Common words between requirements and classes were quite infrequent in the Albergate system: in fact unlike `LEDA` manual pages, Albergate functional requirements were produced in the early phases of the software development life cycle. Moreover, the Italian language has a complex grammar: verbs have much more forms than English verbs, plurals are almost always irregular, and adverbs and adjectives have irregular forms too.

Table 2 shows the results of this case study (the meaning of the columns is the same as in Table 1). Unlike the `LEDA` case study, the results of the vector space model are not very different than those produced by the probabilistic model [2] (see Figure 3). All the traceability links were recovered by considering the first seven documents for each class. However, the probabilistic model tends to retrieve sooner most of the relevant documents (100 % of recall was obtained by considering the first six documents for each class).

4. Discussion

This section analyzes and discusses the results achieved in the two case studies and draw some lessons learned. Although the limitations of the two case studies (in particular, the small size of Albergate and the fact that the LEDA documentation is extracted from comments contained in the source files) do not allow to draw definitive conclusions, some basic considerations can be already outlined. We address three points:

- how helpful is an IR model in a traceability link recovery process ?
- why the performances of the probabilistic and the vector space models are different ?
- how precision could be improved ?

The following subsections attempt to provide an answer to the questions above.

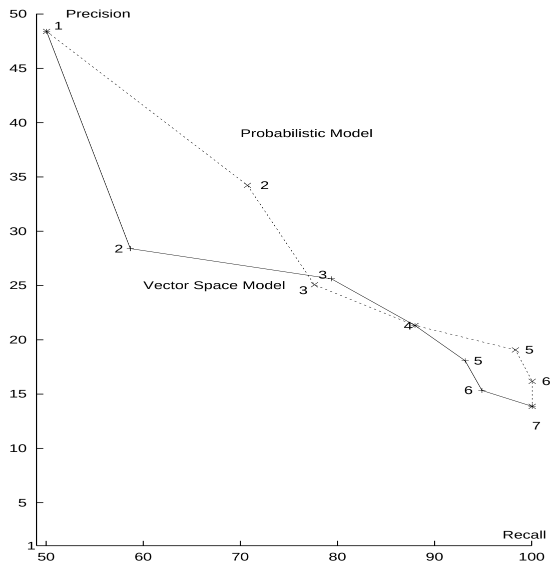

Figure 3. Albergate Precision/Recall results with improved process.

4.1 Benefits of IR

To demonstrate the benefits of using an IR model for recovering traceability links between code and documentation, we compared the results achieved in the two case studies with the probabilistic and vector space IR models with the results obtained by using the `grep` UNIX command, as proposed by Maarek *et al.* [12]. In fact, `grep` provides the simplest way to trace source code components

	Single Code Item				Code Items *or* Combined			
	#Queries	#Empty Set	Mean Size	Max Size	#Queries	#Empty Set	Mean Size	Max Size
Albergate	4834	4575	5	14	60	0	11	13
LEDA	4670	451	20	88	208	1	75	88

Table 3. `grep` **results.**

(e.g., classes) onto high level documentation (e.g., manual pages and/or requirements). The search can be done at least in two ways: in the first approach each class identifier is used as the string to be searched into the files of high level artifacts while the second approach considers the *or* of the class identifiers.

Table 3 shows the results of the `grep` approach: it is worth noting that for the Albergate system 94% of the single item queries gave empty results while if items are *or* combined 94% of classed were traced onto 10 or more requirements. Empty sets are less frequent for `LEDA`; however, the average number of traced manual pages is quite high (20 and 75, respectively). Even worse the `grep` approach did not offer any way to rank the retrieved requirements. From a practical point of view this means that the maintainer has to examine a large number of candidates with the same priority. `grep` baseline results were thus judged quite unsatisfactory compared with IR results.

In the previous sections we have evaluated the results using the IR metrics *recall* and *precision*. To achieve an indication of the benefits of using an IR approach in a traceability link recovery process, we have also introduced a *Recovery Effort Index* (*REI*), defined as the ratio between the number of documents retrieved and the total number of documents available:

$$REI = \frac{\#Retrieved}{\#Available}\%$$

This metric can be used to estimate the percentage of the effort required to manually analyze the results achieved by an IR tool (and discard false positive), when the recall is 100 %, with respect to a completely manual analysis[1]. For a given software system, the quantity $1 - REI$ estimates the effort saving deriving from the use of an IR method to recover traceability links, with respect a completely manual analysis. The lower the REI the higher the benefits of the IR approach.

This metric also measures the ratio between the precision of the results achieved on the same software system by a completely manual process, namely P_m, and a semi-automatic process, namely P_t, that exploits IR, when the recall is 100 %:

[1]At the moment we have not statistically validated the relation between this metric and the traceability link recovery effort; this will be part of future work.

$$\frac{Precision_m}{Precision_t} = \frac{\#(Relevant \wedge Retrieved_m)}{\#(Relevant \wedge Retrieved_t)} \frac{\#Retrieved_t}{\#Retrieved_m}$$

Note that the number of relevant documents retrieved is the same in both processes (all relevant documents) and that the documents retrieved with a manual analysis are just all documents available, then:

$$\frac{Precision_m}{Precision_t}\% = \frac{\#Retrieved_t}{\#Available}\%$$

that is the REI for the semi-automatic process.

The values of REI registered in the two case studies for the vector space IR model are rather different: Albergate requires 43.75 % REI to achieve 100 % recall, whereas `LEDA` only requires 13.63 % REI. A possible explanation is that the set of available documents in the Albergate case study is smaller (16 functional requirements versus the 88 manual pages of `LEDA`); to get the same REI as in the `LEDA` case study the maximum recall would have to be achieved with about 2 documents retrieved (that also means about 50 % of precision). However, this is very unlikely to be achieved with IR methods, that generally aim to retrieve a small percentage of a huge document space. Therefore, it is likely to hypothesize that greater benefits (and lower values of REI) are achieved for document spaces of greater size.

Alternatively, the REI could be computed with respect to a manual analysis supported by `grep` (queries with or combined items). In this case, the REI is computed as the ration between the number of relevant documents retrieved with an IR approach and the number of documents retrieved by `grep`[2]. For the vector space model the values for REI are 54.54 % in the Albergate case study and 16 % in `LEDA`.

4.2 Probabilistic versus vector space model

The two case studies suggest that both IR models (vector space and probabilistic) are suitable for the problem of recovering traceability links between code and documentation. The results are very similar, in particular with respect to the number of documents a software engineer needs to analyze to get high values of recall. This also means that

[2]Of course, this requires that grep based approach achieves 100 % recall, as in our case.

Cut	Retrieved	Relevant	Precision	Recall	Prob. Rel.
1	60	23	38.33 %	39.65 %	15
2	120	33	27.50 %	56.89 %	17
3	180	38	21.11 %	65.51 %	20
4	240	46	19.16 %	78.86 %	23
5	300	48	16.00 %	82.75 %	28
6	360	52	14.44 %	89.65 %	30
7	420	54	12.85 %	93.10 %	32

Table 4. Albergate results with simplified process

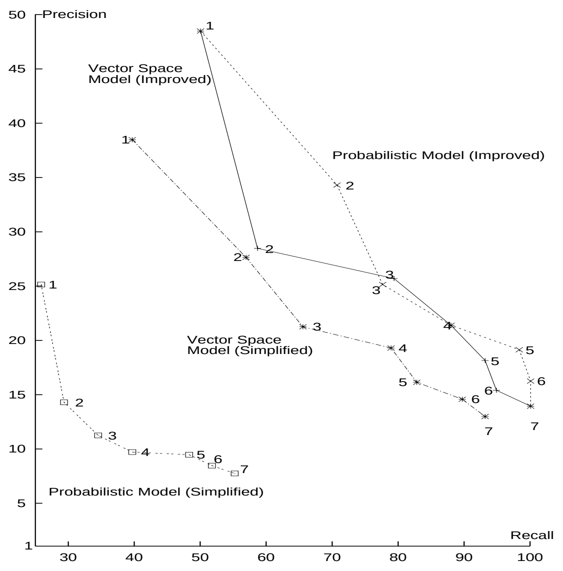

Figure 4. Albergate Precision/Recall results.

the two models achieve similar values of REI. However, Figures 2 and 3 show that the results of the vector space model are more smoothed than the corresponding results achieved with a probabilistic model. Moreover, the probabilistic model tends to retrieve sooner most of the relevant documents.

A possible explanation is in the nature of the two models. The probabilistic model associates a source code component (in our case studies a class) to a document based on the product of the probabilities that each code component identifier appears in the software document [3, 2]:

$$Similarity(D_i, Q) = \prod_{k=1}^{m} Pr(w_k \mid D_i) \simeq$$
$$\simeq Pr(Q \mid D_i) \simeq Pr(D_i \mid Q)$$

These probabilities are computed on a statistical basis and code component identifiers that do not appear in the document are assigned a very low probability. Conversely, the similarity measure of a vector space model only takes into account the code identifiers that also appear in the document and weight the frequencies of the occurrences of such words in the document (code component) with respect to a measure of the diffusion of such words in other documents (code components, respectively).

Therefore, the probabilistic model is more suitable for cases where the presence of code component identifiers that do not belong to the software document is low: this is also the reason why, with respect to the best match, the probabilistic model performs better in the LEDA case study (82.65 % of recall) than in the Albergate case study (50 % of recall). It is worth noting that the probabilistic model is also used in speech recognition [7] and information theory [6] fields, where the aim is to associate a received sentence to a possible transmitted sentence, with a very low error probability. Conversely, the vector space model fits cases where each group of words is common to a relatively small number of software documents. This means that the vector space model does not aim to the best match, but rather to regularly achieve the maximum recall with a moderate number of retained documents.

This hypothesis is supported by the results obtained by applying the simplified version of the text processing steps in Figure 1 to the Albergate case study, with both the probabilistic and the vector space models. The simplified versions of the identifier separation and text normalization steps produce code components and software documents with a higher number of different words. Table 4 shows the results achieved, while Figure 4 depicts the Precision/Recall curves of the two IR models, in both simplified and improved processes. For the vector space model the results of the simplified and improved processes are not very different. Conversely, the differences are evident when applying the two versions of the text processing steps with the probabilistic model [2]. This means that unlike the probabilistic model, the vector space model is able to achieve higher recall values based on a smaller number of relevant words in a source code component.

Percentage	Retrieved	Relevant	Precision	Recall
90 %	59	29	49.15 %	50.00 %
70 %	101	38	37.62 %	65.51 %
50 %	158	50	31.64 %	86.20 %
30 %	265	55	20.75 %	94.82 %
10 %	484	58	11.98 %	100.00 %
min(10 %, best 7)	329	58	17.62 %	100.00 %

Table 5. Albergate results using a threshold

4.3 Retrieving a variable number of documents per query

In our case study we have retained for each query a fixed number of documents. The results achieved for the recall can be considered good, as in both case studies we were able to achieve 100 % of recall with a moderate number of retained candidates per query.

We wondered if with a variable number of retained candidates per query we could improve precision and REI, while maintaining a maximum recall. The approach adopted to test this hypothesis consisted of using a threshold on the similarity values to prune the ranked list of documents retrieved by a query. In particular, for each query Q we computed the value of such a threshold t_Q as a percentage of the similarity measure of the best match:

$$t_Q = c * [\max_i s_{i,Q}]$$

where $0 \leq c \leq 1$. A query Q returns all and only the documents D_k such that $s_{k,Q} \geq t_Q$. Of course, the higher the value of the parameter c the smaller the set of documents returned by a query.

Table 5 shows the results achieved with the Albergate case study using different values of the parameter c (and then different thresholds). The results are not very encouraging, as the maximum recall is achieved when setting the threshold to only 10 % of the highest similarity measure. Using this percentage, the average number of retrieved documents per query is 9, while 3 documents are retrieved in the best case, and 15 documents in the worst case.

Although the results are worse than the results achieved with a fixed cut (first 7 documents in table 2), they still demonstrate the benefits of using an IR approach: indeed, when the recall is 100 % (c = 10 %) the REI is 50.41 %; this means that presumably about 50 % of the effort can be saved by only discarding the documents whose similarity measure is below 10 % of the best match.

Of course, the results can be improved by mixing a variable and fixed cut: each query retrieves only the documents with a similarity measure greater than a given threshold, but no more than a fixed number. As an example, last row in table 5 shows the results achieved by considering as the number of documents retrieved by each query the minimum between 7 and the number of documents whose similarity value is higher than 10 % of the best match. In this case the results are much better than the results achieved with a fixed cut (the first 7 documents in table 2): the average number of retrieved documents is 6 and the REI is 34.27 %, that means that the percentage of effort saved might be more than 65 %.

The issue of retrieving a variable number of documents per query needs further statistical investigations and this will be part of our future work.

5. Concluding remarks

We have presented an IR method to recover traceability links between code and free text documentation and have applied it to trace C++ and Java source classes onto manual pages and functional requirements, respectively. The method relies on a vector space IR model and ranks documents against a query (a list of identifiers extracted from a classe) by computing a distance between the corresponding vector representations.

A goal of this paper was to demonstrate that vector space IR performs as well as probabilistic IR. We have replicated the case studies presented in references [3] and [2] (which applied probabilistic IR), using a vector space model and the results support our hypothesis that IR, either probabilistic or vector space models, provides a practicable solution to the problem of semi-automatically recovering traceability links between code and documentation.

The paper has discussed the differences between the two IR models. In particular, the vector space model exhibits a behavior more regular than the probabilistic model and requires less effort in the preparation of the query and document representations. On the other hand, the probabilistic model performs better when the constraint of manually analyzing a reduced number of documents is stronger than achieving 100 % recall.

In our knowledge, the issue of recovering traceability links between code and free text documentation is not largely investigated and very few contributions appear in the literature. A number of related papers are in the area of impact analysis. They assume the existence of some forms of ripple propagation graph describing relations between software artifacts, including code and documentation, and focus on the prediction of the effects of a maintenance change request on both the source code and the specification and design documents [16].

TOOR [13], IBIS [11], and REMAP [14] are a few examples of CASE tools that maintain traceability links among various software artifacts. However, these tools are focused on the development phase and either force naming conventions or require human interventions to define the links.

Reference [12] introduces an IR method to automatically assemble software libraries based on a free text indexing scheme. The method uses attributes automatically extracted from natural language IBM RISC System/6000 AIX 3 documentation to build a browsing hierarchy which accepts queries expressed in natural language.

Several software reuse environments use IR to index and retrieve the reusable assets. The RSL [5] system extracts free-text single-term indices from comments in source code files looking for keywords like “author”, “date created”, etc. REUSE [4] is an information retrieval system which stores software objects as textual documents in view of retrieval for reuse. Similarly, CATALOG [9] stores and retrieves C components each of which is individually characterized by a set of single-term indexing features automatically extracted from natural language headers of C programs.

Future work will be devoted to further investigate the factors that produce differences between the results achieved by the vector space and the probabilistic models. We are also working on the definition of an improved method to prune the ranked list of documents by analyzing the distribution of the similarity measures. Finally, we aim to a major improvement of the traceability link recovery process by removing the constraints that documents and code insist on the same vocabulary. In particular, our hypothesis of investigation is that for software systems available in multiple releases, a probabilistic mapping between the vocabularies of documents and source code can be statistically established.

6. Acknowledgements

We would like to express our gratitude to Ettore Merlo for the stimulating discussions we had during the work presented here. He also parsed the Albergate source code.

This research is supported by the project ”Virtual Software Factory”, funded by Ministero della Ricerca Scientifica e Tecnologica (MURST) and jointly carried out by EDS Italia Software, University of Sannio, University of Naples ”Federico II”, and University of Bari.

References

[1] G. Antoniol, G. Canfora, G. Casazza, and A. DeLucia. Identifying the starting impact set of a maintenance request. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 227–230. Zurich, Switzerland, March 2000.

[2] G. Antoniol, G. Canfora, G. Casazza, A. DeLucia, and E. Merlo. Tracing object-oriented code into functional requirements. In *Proceedings of the 8th International Workshop on Program Comprehension*, pages 227–230. Limerick, Ireland, June 2000.

[3] G. Antoniol, G. Canfora, A. DeLucia, and E. Merlo. Recovering code to documentation links in object-oriented systems. In *Proceedings of the IEEE Working Conference on Reverse Engineering*, pages 136–144. Atlanta, Georgia, IEEE Comp. Soc. Press, October 1999.

[4] S. P. Arnold and S. L. Stepowey. The reuse system: Cataloging and retrieval of reusable software. In W. Tracz, editor, *Software Reuse: Emerging Technology*. IEEE Comp. Soc. Press, 1987.

[5] B. A. Burton, R. W. Aragon, S. A. Bailey, K. Koelher, and L. A. Mayes. The reusable software library. In W. Tracz, editor, *Software Reuse: Emerging Technology*, pages 129–137. IEEE Comp. Soc. Press, 1987.

[6] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley Series in Telecommunications, John Wiley & Sons., New York, NY 10158-0012, 1992.

[7] R. DeMori. *Spoken dialogues with computers*. Academic Press, Inc., Orlando, Florida 32887, 1998.

[8] W. B. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[9] W. B. Frakes and B. A. Nejmeh. Software reuse through information retrieval. In *Proceedings of 20-th Ann. HICSS, Kona (HI)*, pages 530–535, January 1987.

[10] D. Harman. Ranking algorithms. In *Information Retrival: Data Structures and Algorithms*, pages 363–392. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[11] J. Konclin and M. Bergen. gibis: a hypertext tool for exploratory policy discussion. *ACM Transaction on Office Information Systems*, 6(4):303–331, October 1988.

[12] Y. Maarek, D. Berry, and G. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, 1991.

[13] F. A. C. Pinhero and A. G. J. An object-oriented tool for tracing requirements. *IEEE Software*, 13(2):52–64, March 1996.

[14] B. Ramesh and V. Dhar. Supporting systems development using knowledge captured during requirements engineering. *IEEE Transactions on Software Engineering*, 9(2):498–510, June 1992.

[15] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, 24(5):513–523, 1988.

[16] R. J. Turver and M. Munro. An early impact analysis technique for software maintenance. *Journal of Software Maintenance - Research and Practice*, 6(1):35–52, 1994.