Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 243 (2009) 121–137
www.elsevier.com/locate/entcs

Recovering Relationships between Documentation and Source Code based on the Characteristics of Software Engineering[1]

Xiaobo Wang[2] Guanhui Lai[3] Chao Liu[4]

School of Computer Science and Engineering
Beihang University
Beijing, China

Abstract

Software documentation is usually expressed in natural languages, which contains much useful information. Therefore, establishing the traceability links between documentation and source code can be very helpful for software engineering management, such as requirement traceability, impact analysis, and software reuse. Currently, the recovery of traceability links is mostly based on information retrieval techniques, for instance, probabilistic model, vector space model, and latent semantic indexing. Previous work treats both documentation and source code as plain text files, but the quality of retrieved links can be improved by imposing additional structure using that they are software engineering documents. In this paper, we present four enhanced strategies to improve traditional LSI method based on the special characteristics of documentation and source code, namely, source code clustering, identifier classifying, similarity thesaurus, and hierarchical structure enhancement. Experimental results show that the first three enhanced strategies can increase the precision of retrieved links by 5%∼16%, while the the fourth strategy is about 13%.

Keywords: Software Engineering, Information Retrieval, Traceability Recovery, Software Reuse

1 Introduction

Establishing the traceability links between software documentation and source code has been a challenging task during the whole life cycle of a software product. Usually, developers prefer updating documentation after a required feature is completed, which is prone to be forgotten easily due to limited development period. Currently, complex software systems often consist of large amount of documents, such as requirement specifications, system designs, user manuals, test reports, and maintenance logs, which contain much domain specific knowledge. If a developer

[1] This work has been supported by National Science Foundation of China (90718018).
[2] Email:bobywolf@gmail.com
[3] Email:garylai@sei.buaa.edu.cn
[4] Email:liuchao@buaa.edu.cn

doi:10.1016/j.entcs.2009.07.009

wants to maintain a legacy system, which he/she is not familiar with, a necessary task is to find the relationships between documentation and source code. Therefore, recovering and maintaining the correct traceability links can be helpful for program comprehension, software maintenance, requirement tracing, impact analysis, and software reuse.

A traceability link means that a document or a paragraph of the document is closely related with some pieces of source code. For example, there may be a link between a requirement specification and a piece of source code that implements the requirement, or an explanation document of a class is associated with the definition body of that class. Though the maintenance of traceability links is very important and useful, it is difficult for us to uncover them due to some obstacles. On one hand, we can not build a precise LL, LALR, or LR language recognizer for any kind of human languages, even for English. However, it is possible to write a complex parser for a context-sensitive programming language, such as C++. On the other hand, the association relationships between documentation and source code are rarely represented explicitly, because documentation and source code, which are both used for modeling the real world are represented at different abstract levels [10]. Therefore, extracting traceability links through natural language processing is still a big challenge.

Semiautomatic construction and maintenance of traceability links have been studied a lot in previous work [7,10,9,1,2]. Though there are some commercial integrated development tools that support the maintenance of traceability links, such as Rational Suite, DOORS, and TOORS, those tools are not satisfied by software developers because of the strong manual interventions required in constructing traceability links. Recently, most researchers try to use information retrieval approaches to extract traceability links, which are referred as IR approaches (Information Retrieval). Probabilistic model and vector space model were firstly applied to recovering traceability links by Antonial et al. [1], and the precision of the retrieved links was over 30% while the recall was about 70%. Then, Marcus et al. [10,9] employed Latent Semantic Indexing (LSI) model to improve the precision by 5%∼10%. Both software documentation and source code are considered as plain text in previous work. For instance, latent semantic indexing converts documents and source code to a series of term-document matrices, which represent the underlying meaning of those texts, and then the term-document matrices are used to compute the similarity between documentation and source code. Source code contains many special tokens, such as class names, method names, namespace names, and comments, and documentation can be categorized into different hierarchies, such as summary design documents and detailed design documents. Those special tokens in source code and the hierarchical structure of documentation can be regarded as useful characteristics of software engineering, and then we can use such characteristics to refine the traceability links extracted by IR approaches.

In this paper, we present an advanced LSI model based on the characteristics of software engineering. The major contributions of this paper include:

- Automatically extract abbreviations and domain specific vocabulary from source

code and documentation to construct similarity dictionary, which is used to deal with synonymies and abbreviations;

- Improve the precision of traceability links by applying source code clustering and identifier classifying;
- Construct an iterative refining process with automatic feedback based on the hierarchical structure of documentation.

The paper is organized as follows: Section 2 gives an overview of recovering methods based on IR techniques. The main framework of recovering process is shown in section 3. Section 4 presents the special characteristics of documentation and source code and proposes four strategies to improve current recovery model. Section 5 compares our approach with previous work through two experiments and gives the detailed analysis of parameter tuning. Finally, section 6 draws conclusion and outlines future work.

2 Overview of recovering methods based on IR techniques

Extracting the relationships between documentation and source code can be regarded as an IR query process, in which source code is converted to query terms and software documentation constitutes literature library. First, documentation and source code need to be preprocessed for information retrieval, such as retrieval items extraction, conversion between lowercase and uppercase, stop words removal, and etyma generation. Second, similarity is computed by applying some IR models, and then a list of documents sorted descendingly by correlation degree is generated. Finally, the records with the correlation degrees above the threshold value are selected from document list. When IR approaches are applied to the recovery of traceability links, there must be an important premise that most of the identifiers in source code should be named with meaningful words, which also exist in software documentation. To validate the hypothesis, we manually checked 50 source code files from Linux software repository and 5 commercial software products, respectively. Almost 97% of identifiers in source code are meaningful except some local variables.

2.1 Probabilistic Model (PM)

The first probabilistic model applied to IR was presented by Maron and Kuhns in 1960 [11]. The similarity between documentation and source code depends on the probability that whether a document is relevant to a piece of source code, that is to say, all the documents are sorted by their correlation probabilities statistically.

When we use PM to recover traceability links, query terms consist of the characteristic items extracted from source code. Here, the characteristics mean the meta data of free text, which contains words, phrases, or sentences. According to PM approach, the similarity between documentation and source code can be computed as a conditional probability. Suppose that there are n documents, let D denotes

ith document and Q represents a piece of source code. Then, similarity calculation formula is

$$Similarity(Q, D) = P(D|Q) = \frac{P(Q|D)P(D)}{P(Q)} \tag{1}$$

The similarity between Q and D is equal to the probability that both Q and D appear at the same time. The similarity between a piece of source code and a document can be computed by applying equation 1, and then all the values will be sorted descendingly and filtered by a predetermined threshold. Probabilistic model is easy to implement and has definite physical meanings, but it can not deal with synonymies and abbreviations, moreover, etymas must be generated when documents are preprocessed.

2.2 Vector Space Model (VSM)

Vector space model was proposed by Salton for SMART information retrieval system at Cornell [8]. Either a query or a document is viewed as a vector of terms (or words), and the lingual similarity of free text is transformed to spatial similarity, that is to say, the similarity among text vectors is applied to document retrieval. Theoretically, the smaller the vector angle is, the higher similarity between a query and a document is. When VSM is applied to recovering the relationships between documentation and source code, every characteristic of documentation or source code can be viewed as one dimension of text space, and then the vector space is the set of those characteristics. Any retrievable document is expressed as a vector in the text space, called document vector. Let $D = (w_1, w_2, \cdots, w_m)$ denotes a document, and $(d_1, d_2, \cdots, d_m)^T$ represents the corresponding vector, where d_i is the weight of characteristic w_i in document D, and m is the dimension of vector space. Similarly, a piece of source code is expressed as $Q = (q_1, q_2, \cdots, q_m)^T$. Therefore, the similarity between Q and D can be computed by cosine formula (see equation 2) which is widely used in IR approaches.

$$Similarity(Q, D) = \frac{\sum_{i=1}^{m} d_i q_i}{\sqrt{\sum_{i=1}^{m} d_i^2 \sum_{i=1}^{m} q_i^2}} \tag{2}$$

If there are N documents, those document vectors can be represented as a $M \times N$ matrix called term-document matrix, whose rows are M words (or terms) and N is the number of documents.

The high dimension of term-document matrix due to the large size of documents leads to the waste of memory, low efficiency, and big noises. Because all the terms (or words) are independent in VSM, it is still difficult to cope with synonymies and polysemies. Furthermore, programmers often use abbreviations to name identifiers in source code, for example, "arg" stands for argument and "class" is usually abbreviated to "cls", but VSM can not give correct results in that case. However, VSM is actually a general representation method for text documents, and term-document matrix can be used by any retrieval model as well as a powerful tool for information retrieval.

2.3 Latent Semantic Indexing (LSI)

LSI is a new algebraic model for information retrieval based on VSM. The basic assumption of LSI model is that there exists some implicit relationships among the words of documents, that is to say, there are some latent semantic structures in free text. Semantic structure means an abstract semantic format which consists of semantic category and semantic relationship in natural languages. First, documents are represented as a large term-document matrix introduced in 2.2. Second, the VSM space is truncated and transformed to LSI subspace by applying singular value decomposition method (SVD) to the term-document matrix. Finally, we can compute the similarity by equation 2 in LSI subspace and filter result list according to a predetermined threshold, and then the traceability links between documentation and source code are retrieved.

The low rank approximation of term-document matrix provided by LSI can filter a lot of noises and has a better spatial and temporal efficiency. LSI takes the relationships among words into account, so synonymies and polysemies can be processed correctly. Moreover, etyma generation is unnecessary, which simplifies the preprocessing step.

3 The Main Framework of Recovering Process

Recall and precision are commonly used to measure the quality of IR approaches. Recall is the measurement of the ability to retrieve correct results (see equation 3), while precision is the measurement of the accuracy of retrieved results (see equation 4).

$$Recall = \frac{\text{correct results retrieved}}{\text{total correct results}} \tag{3}$$

$$Precision = \frac{\text{correct results retrieved}}{\text{total results retrieved}} \tag{4}$$

The experiments conducted by Antonial et al. showed that probabilistic model was better than vector space model with 30% precision and 70% recall [1]. Then, Marcus et al. applied LSI model to the recovery of traceability links, and precision can even reach 70% while recall is about 60% [10]. When the recall keeps invariant, the precision can be improved by 5%∼10%, so LSI is more powerful than either PM or VSM to a certain extent. Though all of PM, VSM, and LSI can be applied to recover traceability links, the results are not as good as applied them to information retrieval. Documentation and source code are different from plain text, and they have many special characteristics with respect to software engineering, for instance, documentation might contain data dictionary, UML diagrams, and class explanations; structural entities in source code also have call relationships, inheritance relationships, and dependence relationships. Previous recovering methods based on IR techniques treat documentation and source code as plain text, but those special characteristics can be used to make improvement on the precision and the recall of traceability recovery. Our recovering process of traceability links is divided into

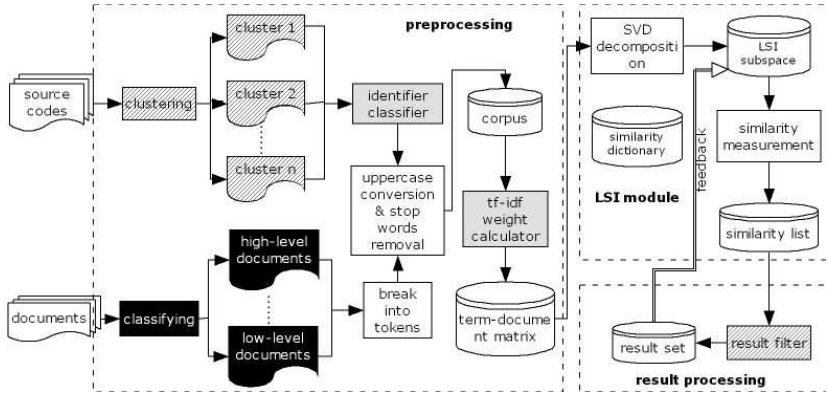

Fig. 1. Recovering process of traceability links based on the characteristics of software engineering

three modules, preprocessing module, LSI module, and result processing module (see figure 1), which will be described in detail in the following sections.

3.1 Preprocessing Module

The core idea of LSI approach is that semantic similarity can be transformed to spatial similarity, so documentation and source code must be converted to a series of term vectors.

Firstly, the identifiers, such as class names, method names, field names, and comments, are extracted from source code. Then, the identifiers that contains more than one word, for example, list_polygon_unite, height_of_window, are broken into a list of words, here, two lists are $L1 =$ (list polygon unite) and $L2 =$ (height of window). In addition, the keywords corresponding to specific programming language and stop words must be removed in the preprocessing step, while capital characters are converted to lowercase.

Secondly, all the documents with multiple formats, such as WORD, HTML, RTF, and PDF, will be converted to plain text files. Moreover, in order to achieve better results, the text files are partitioned into small documents with approximately same size. Documents can also be partitioned according to chapters, sections, or paragraphs, and then stop words removal and uppercase conversion are operated on the small documents as same as on source code.

Thirdly, the identifiers and characteristics, which are extracted from documentation and source code, are stored in a corpus (denoted by S). Then, we build a large vector space V based on S, where $V = \{w_i, w_i \in S, i \in [1, \cdots, m]\}$ and m is the dimension of V, in other words, the number of the words in corpus S. Both of documentation and source code are represented as text vectors, and the weight of every word in corpus S is computed by $tf\text{-}idf$ method [8].

Finally, all the text vectors constitute a $m \times n$ term-document matrix D (see equation 5), where n is the total number of documents and source code, and matrix

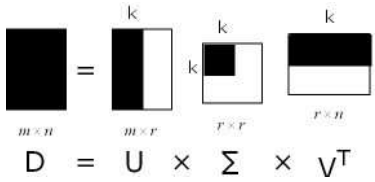

Fig. 2. Decomposition process of SVD

D is the output of preprocessing step that will be delivered to LSI module.

$$D = (D_1 D_2 \cdots D_n) = \begin{pmatrix} d_{1,1} & d_{1,2} & \cdots & d_{1,n} \\ d_{2,1} & d_{2,2} & \cdots & d_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ d_{m,1} & d_{m,2} & \cdots & d_{m,n} \end{pmatrix} \tag{5}$$

3.2 LSI Module

The main difference between LSI and VSM is performing dimension reduction operation on term-document matrix, and an approximate low-rank matrix is used to represent the vector space of the original corpus. Here, term-document matrix D is decomposed to matrix D_k whose rank is at most k ($k \ll \min(m, n)$). Matrix D can be rewritten to a product of three matrices (see equation 6), where U ($m \times r$ order matrix) and V^T ($r \times n$ order matrix) are left and right singular value matrices, respectively. r order diagonal matrix Σ consists of all the singular values of D, whose diagonal entries are sorted descendingly, and r is the rank of D.

$$D = U \Sigma V^T \tag{6}$$

According to SVD approach, the first k largest singular values and their corresponding singular vectors constitute a new k order matrix D_k (called LSI subspace, see equation 7), which is nearly equal to original term-document matrix D.

$$D_k = U_k \Sigma_k V_k^T \tag{7}$$

The decomposition process of SVD is shown in figure 2. The retrieval ability of SVD decomposition is the same as VSM, while the dimension of new LSI subspace is greatly reduced [4]. Then, the similarities between documentation and source code can be computed by applying the equation 2.

3.3 Result Processing Module

The similarity list produced by LSI module will be filtered by a predetermined threshold. The size of the list is equal to the production of the number of pieces of source code and the number of documents. For instance, if a project contains 400 pieces of source code and 100 documents, the result list should have 40000 entries.

The higher the similarity is, the more the semantics of documentation is close to the characteristic set of source code. However, the threshold that is up to result set can not be a fixed value. Two filtering policies are employed here to refine similarity list in this work:

Cut-point method Set a constant number C, only the top C records in similarity list are selected as actual results. C is positive correlated with recall, while it is negative correlated with precision. Therefore, C should be selected appropriately.

Threshold value method Set a threshold value S, only the values greater than or equal to S will be saved as results. Mostly, S is chosen between 50% and 70%.

4 Enhanced Strategies

The main steps of recovering process are described in previous section, but traditional LSI model does not take the special features of documentation and source code into account. There are mainly three defects that previous LSI method have.

- Treat software documentation and source code as plain text files, and the weights of all the words in corpus is equal. The difference among various identifiers and the hierarchical structure of documentation are not considered;
- Can not cope with all the synonymies, especially when abbreviations exist in source code;
- The quality of results can not be improved by feedback, such as user feedback and the results of previous iteration.

In our study, four enhanced strategies are proposed in following sections based on the characteristics of documentation and source code.

4.1 Source Code Clustering

The entities extracted from source code may contain many kinds of relationships, for example, call relationships, inheritance relationships, and implementation relationships. In this work, inheritance relationship among classes are employed to perform cluster analysis, that is to say, if one class is the ancestor of another class, they are assigned to same cluster. Source code clustering algorithm is shown in algorithm 1, and figure 3 gives an example of source code clustering. There is a relationship between a document d and a piece of source code q_i, and then the similarity between d and all the other items that belong to the cluster of q_i should be set to a higher value. The values which are greater and equal than $threshold_{high}$ are selected from the result set directly, while the ones below $threshold_{low}$ are thrown away. $\bar{S}$ in step 6 is the average similarity of all the entities of the cluster which q belongs to. All the entries in similarity list that belong to set $[threshold_{low}, threshold_{high}]$ will be processed by clustering algorithm, while $threshold_{enhanced}$ is the threshold that is related to average similarity $\bar{S}$. Furthermore, those three thresholds must be selected according to the range of similarity list.

Source code clustering step is added to preprocessing module, and filtering policy

Algorithm 1

Input: *source code clusters C, similarity list L*
Output: *similarity list after applying source code clustering*
Step 1: Fetch a similarity $Sim(d_i, q) \in L$
Step 2: If $Sim(d_i, q) > threshold_{high}$ then goto step 3 else goto step 4
Step 3: retrieve the traceability link between d_i and q, goto ***end***
Step 4: If $Sim(d_i, q) > threshold_{low}$ then goto step 5 else goto step 9
Step 5: Compute average similarity, $\bar{S} = \sum_{q_j \in C[q]} Sim(d_i, q_j)/|C[q]|$
Step 6: If $\bar{S} > threshold_{enhanced}$ then goto step 7 else goto step ***end***
Step 7: $Sim(d_i, q) = \bar{S}$
Step 8: retrieve the link between d_i and q, goto ***end***
Step 9: reject the link between d_i and q
End: Return similarity list

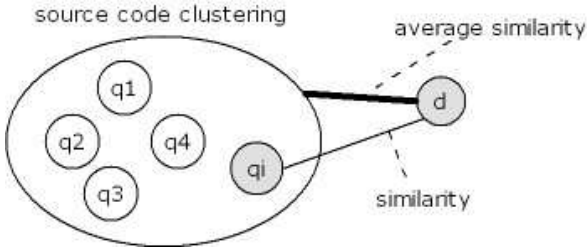

Fig. 3. Source code clustering example

must be modified to refine similarity list (see the hatched parts in figure 1).

4.2 Identifier Classifying

There are many kinds of identifiers in source code, and they should play different roles in similarity computation. For example, if a class name "A" occurs in one document, the definition body of class "A" should be related to that document with higher probability. In our study, all the identifiers are classified to three categories:

(i) Class names, which often appear in documentation, e.g., API usage and UML class diagram;

(ii) Various comments, including class comments, method comments, and other comments, should be assigned with different weights;

(iii) General identifiers, all the identifiers except class name and comments.

If a class definition body and the document which contains the name of that class, the similarity between them will be increased by 20%. According to the importance of different types of comments, the weights of them are set to 2.0, 1.5, and 1.0, respectively. The relationship between those weights and the quality of results will be discussed in detail in 5.3. Here, we add an identifier classifier to preprocessing module and change the weight computation method of identifiers (see the parts with gray background in figure 1).

4.3 Similarity Thesaurus

LSI module can deal with synonymies to some extent, but only the word whose occurrence frequency is higher will be detected. Moreover, abbreviations in source code are widely used by programmers, which can not be processed by traditional LSI model. In our approach, a similarity thesaurus is employed to cope with synonymies and abbreviations. Let tuple $< k_i, k_j, a_{ij} >$ denotes an entry of thesaurus, where k_i and k_j are the terms in vector spaces, and $a_{ij} \in [0, 1]$ is the similarity coefficient of k_i and k_j. Similarity thesaurus can be constructed by two ways:

- Generate from data dictionary that might exist in documentation;
- Extract abbreviations from source code automatically.

After thorough study on various abbreviation extraction methods, we selected the scoped approach that was proposed by Emily et al [6].

When similarity thesaurus (denoted by T) is used to adjust the similarity computation of terms, a new formula (see equation 8) is presented based on traditional cosine similarity measure (see equation 2) [5].

$$Sim_T(Q, D) = \frac{\sum_{i=1}^{m} d_i q_i + \sum_{<k_i, k_j, a_{ij}> \in T} a_{ij}(d_i q_j + d_j q_i)}{\sqrt{\sum_{i=1}^{m} d_i^2 \sum_{i=1}^{m} q_i^2}} \tag{8}$$

The corresponding semantic relationships among characteristics are processed by similarity thesaurus, so the resulting precision is more higher than previous LSI method. The enhanced strategy is easy to implement, in other words, only a similarity thesaurus that will be used by similarity calculator is added to LSI module.

4.4 Hierarchical Structure of Documentation

In this work, all the documents are divided into into two categories, high-level conceptual documents and low-level implementation documents. High-level documents contain requirement specifications, summary designs and user manuals, while low-level documents include detailed designs, API guides, data format specifications, etc. However, the documents can also be divided into more than two categories, our approach does not restrict the number of levels.

Low-level documents are usually closely related to source code. Class names, method names, or comments are often included by low-level documents. Moreover, low-level documents can be considered as further detailed versions of high-level documents, such as, algorithm designs, concept explanations, and implementation details. Therefore, low-level documents can be regarded as a bridge between source code and high-level documents, and an iterative refining process can be constructed according to the hierarchical structure of documentation. First of all, the traceability links of low-level documents and source code are extracted, and then they

are used as feedback to revise current IR model. New query vectors are generated through learning and accumulating the concepts of low-level documents, and then those concepts will be used to retrieve relationships from higher-level documents. That iterative process will be terminated when the highest-level documents are processed.

Refining process can be considered as an automatic procedure with user feedback. User feedback could greatly improve the retrieval ability of IR models, but it requires a lot of manual interventions [12]. In this study, we propose a new refining process, where the documents are processed level by level, and the result of lower-level documents are used to retrieve higher-level documents. Salton presented three approaches which took advantages of user feedback to revise IR model [12], and Ide dec-hi method is employed to improve our LSI retrieval model. The basic idea of Ide dec-hi is that current query vector is revised by all the relevant document vectors and one non relevant vector, which can be represented as equation 9.

$$Q_{new} = Q_{old} + \sum_{\text{all relevant}} D_i - \sum_{\text{one non relevant}} D_j \tag{9}$$

However, relevant vectors are provided not by user in our automatic refining process, so using all the relevant vectors to revise current query vector will be inaccurate. The experiments conducted by Cleland-Huang et al. showed that almost all the traceability links with high similarity were correct links, while the lower ones were error links [3]. Therefore, we decide to pick the most similar vector in result list when new query vector is generated, and equation 9 is revised to equation 10.

$$Q_{new} = Q_{old} + \sum_{\text{one relevant}} D_i - \sum_{\text{one non relevant}} D_j \tag{10}$$

If the hierarchical structure of documentation exists, it is easy for us to assign them to different levels. Requirement specifications and summary design documents belong to high-level documents, while detailed designs, API usages, and test reports belong to low-level documents (see the parts with black background in figure 1).

5 Experiments and Discussion

In this work, we set up two experiments to validate the effectiveness of our revised recovery model and compare the results with previous work. The first experiment employs the data set used in [1,10], release 3.4 of LEDA (Library of Efficient Data Types and Algorithms). LEDA is a free software, which contains 97000 lines of code, 219 classes, and 238 pages of documents. Because there is no explicit hierarchical structure in its documents, only first three enhanced policies can be applied to that experiment. The second experiment was performed to check the fourth enhanced policy, and IBS (Ice Breaker System, see literature [3]) was selected as data set, which contains 72 classes, 18 packages, and more than 180 functional requirements.

LEDA3.4	Original	After Cleaning (piece)
Source code	487	487
Documents	238 pages	110
Total		597

Table 1
LEDA data set after data cleaning

Cut point	Correct links retrieved	Incorrect links retrieved	Missed links	Total links retrieved	Precision	Recall
1	91	6	59	97	93.81%	60.67%
2	122	72	28	194	62.89%	81.33%
3	131	160	19	291	45.02%	87.33%
4	137	251	13	388	35.31%	91.33%
5	144	341	6	485	29.69%	96.00%
6	145	437	5	582	24.91%	96.67%

Table 2
Recovered links, recall, and precision using cut point method for LEDA

5.1 First Experiment

The whole LEDA library, including valid source code, demo programs, test cases, and all the documents, is analyzed by IR system. We have observed that the number of documents is much less than the number of source code files, so documents are converted to query vectors to retrieve relevant pieces of source code, which is the same as a piece of source code is considered as query vectors. Table 1 shows the data set after data cleaning. Because of different partition policy of documents, the number of documents may be different. Here, the number of documents is 110, the size of similarity list is 110×487 accordingly. Through sampling analysis, there are 150 correct traceability links in LEDA, including inheritance relationship. The number of links is larger than the result in [9], where the number is 114, because Marcus et al did not take inheritance relationships into account. Table 2 is the result applying cut-point filtering policy, while figure 4 and figure 5 compare the precision and the recall of probabilistic model (PM) in [1], traditional LSI model (LSI) in [9], and improved LSI module of our method (ALSI).

Looking into the precision and recall in figures, we can see:

(i) In the perspective of recall, probabilistic model is the best, while our ALSI model is the worst. [1] assumed that one piece of source code could not be related to more than one document, and [9] did not consider the inheritance

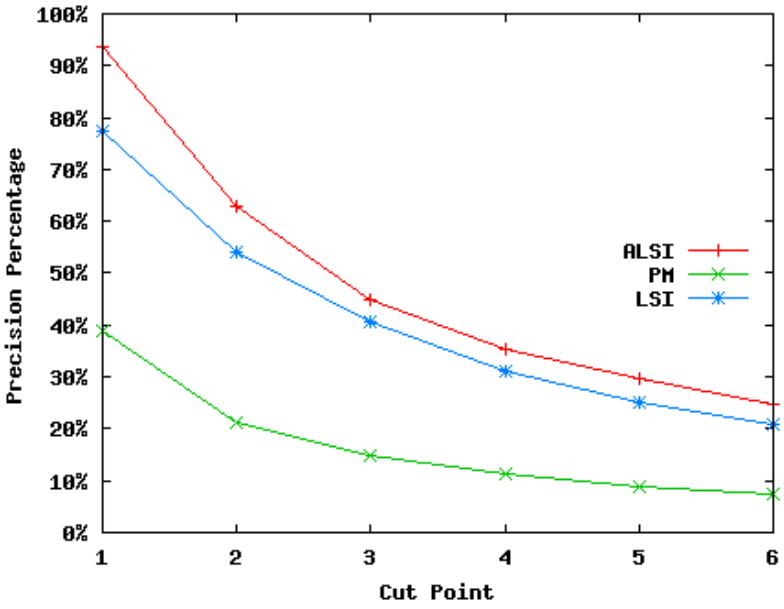

Fig. 4. Precision comparison

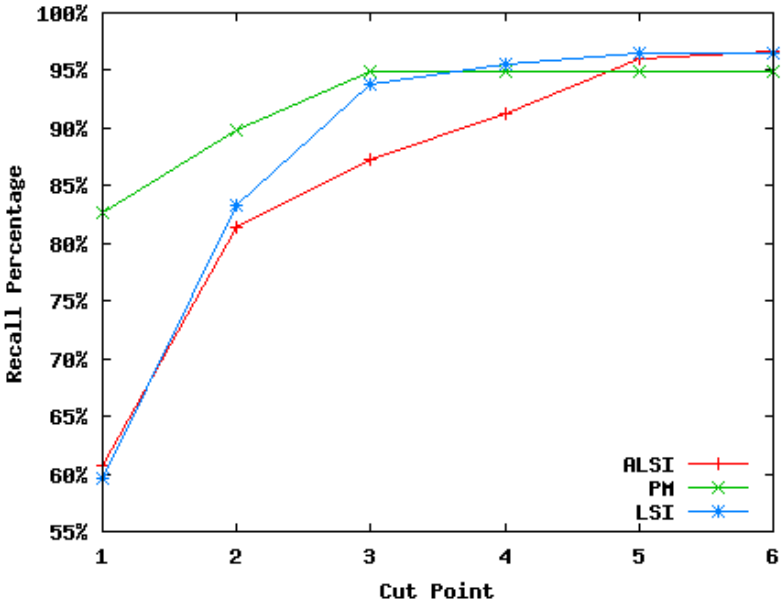

Fig. 5. Recall comparison

relationships in source code. As a result, there are only 88 and 144 correct traceability links, respectively. However, our approach does not have those restrictions, and there are totally 150 correct links retrieved. The decrease of recall is due to the different constraints of experiments accordingly.

(ii) In the perspective of precision, our improved LSI model is the best obviously. The high recall of probabilistic model is related to specific constraints, but

its low precision reflects insufficient ability of recovery. The precision of our approach is improved by 5%∼16% in comparison to traditional LSI model.

(iii) Almost all the traceability links with high similarity is correct. When cut point is set to 1, the precision of ALSI model can even reach 93.81%, and there are only 6 incorrect links.

When threshold value method is applied to filter result list, we used 0.5 as initial threshold. There are 143 correct links out of 247 links retrieved, and 57.89% precision and 95.33% recall are reached. In [9], when threshold was set to 0.6, LSI module can get 71.01% recall and 42.98% precision. Therefore, compared to traditional LSI method, ALSI can improve precision and recall greatly.

Threshold value method is better than cut point method in our experiments, which could improve precision by 5%∼16%. There are three reasons contributing to the improvement on results:

(i) Some implicit associations are extracted from source code by clustering, which will increase the base of correct links;

(ii) The are a lot of class names, class comments, and method comments in the documents of LEDA, so the relationships between source code and documentation can be made more obvious by setting different weights to those identifiers;

(iii) Similarity thesaurus can deal with synonymy and abbreviation, which can improve precision to a certain extent.

5.2 Second Experiment

The second experiment was performed to check the effectiveness of the fourth enhanced strategy, iterative refining process based on the hierarchical structure of documentation. Here, we employ IBS system as data set, and all the documents are categorized to three levels. The precision of ALSI model is compared with original probabilistic model (PM) and improved probabilistic model (APM) [3] when recall is approximately equal. Table 3 shows that ALSI can increase the precision of recovery by 13%∼17% and 5% in comparison with original probabilistic model and improved probabilistic model, respectively.

While recall is approximately equal, the fourth enhanced policy can improve precision according to the second experiment. In each iteration, previous results are learned to correct current query vectors, that is to say, the concepts learned from lower-level documents are used to improve the retrieval of higher-level documents.

5.3 Parameter Tuning

Identifier classifying enhanced policy assigns different weights to the identifiers that are divided into three categories. The similarity of documents and a piece of class definition code is increased by t, when the name of class exists in that document. To determine the multiple t of similarity, we study the relationships between t and the quality of results in the first experiment. Figure 6 shows that the precision of results decreases sharply when $t > 1.2$, and when t is set to 1.2, precision reaches

Retrieval Model	Recall	Precision
PM	90.47%	20.43%
	95.01%	16.81%
APM	90.48%	31.72%
	95.69%	25.65%
ALSI	90.47%	37.36%
	95.53%	30.69%

Table 3
Comparison result of IBS

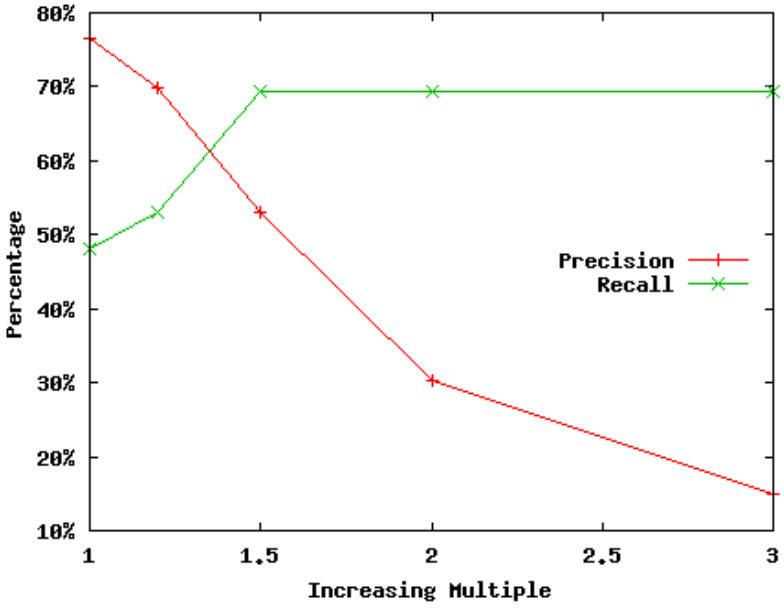

Fig. 6. The relationships between increasing multiple of similarity and the quality of results

69.85% and recall is 63.33%. The main reason is that LEDA is a software library for numerical computation, and mathematical names are widely used to name classes, e.g., "matrix", "vector", and "point", which occur frequently in the documents of LEDA. However, those words in documents are actually not real class names.

Different weights are assigned to comments according to their types, such as, class comments, method comments, and general comments. When the weight of class comments is increased, precision and recall will fall after rising, which is the same as method comments and general comments. When the weight of class comments is set to 2 and all the other weights keep invariant, peak value of precision and recall is reached. Moreover, the weights of peak values of method comments

and general comments are 1.5 and 1.5, respectively. The weight of comments can not be increased arbitrarily. tf-idf weight calculator computes the weight of a word not only in current document but in whole corpus, so any weight that is relatively too high will make tf-idf algorithm fail.

6 Conclusion

In this paper, we present four enhanced strategies based on the characteristics of software engineering to improve existing LSI approaches. We perform two experiments and the results are analyzed in comparison with previous related work by Antoniol et al [1] and Marcus et al [9]. Compared with existing IR approaches, the first three enhanced strategies increased the precision by 5%∼16%, while the fourth is over 13%.

Though similarity thesaurus can process synonymies and abbreviations, it still needs some manual efforts. If a developer is not familiar with the legacy system that he/she is maintaining, it is very difficult for him/her to input correct entries to the thesaurus. Moreover, the preprocessing step of documentation and source code is still a tricky and challenging job, and the classifying of documents according to the hierarchical structure of documentation also needs manual interventions. Therefore, more attention should be payed on the preprocessing step, and the automatic classification of documents is necessary. In the future, we will exploit more characteristics of documentation and source code with respect to software engineering to improve the precision of retrieving traceability links. In addition, more experiments on large open source projects, for instance, Apache, Eclipse, and GCC, will be performed with our improved IR approach.

References

[1] Antoniol, G., G. Canfora, G. Casazza, A. D. Lucia and E. Merlo, *Recovering traceability links between code and documentation*, IEEE Transaction on Software Engineering **28** (2002), pp. 970–983.

[2] Antoniol, G., G. Canfora, A. de Lucia and E. Merlo, *Recovering code to documentation links in oo system*, in: *WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering* (1999), p. 136.

[3] Cleland-Huang, J., R. Settimi, C. Duan and X. Zou, *Utilizing supporting evidence to improve dynamic requirements traceability*, in: *Proceedings of the 13th IEEE International Conference on Requirements Engineering*, 2005, pp. 135–144.

[4] Frakes, W. and R. Baeza-Yates, "Information Retrieval: Data structures and Algorithms," Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[5] Hayes, J., A. Dekhtyar and J. Osborne, *Improving requirements tracing via information retrieval*, in: *Proceedings of 11th IEEE International Requirements Engineering Conference*, 2003, pp. 138–147.

[6] Hill, E., Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. Pollock and K. Vijay-Shanker, *Amap: automatically mining abbreviation expansions in programs to enhance software maintenance tools*, in: *MSR '08: Proceedings of the 2008 international workshop on Mining software repositories* (2008), pp. 79–88.

[7] Liu, D., A. Marcus, D. Poshyvanyk and V. Rajlich, *Feature location via information retrieval based filtering of a single scenario execution trace*, in: *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering* (2007), pp. 234–243.

[8] Manning, C. D., P. Raghavan and H. Schütze, "An Introduction to Information Retrieval," Cambridge University Press, Cambridge, England, 2008.

[9] Marcus, A. and J. Maletic, *Recovering documentation-to-source-code traceability links using latent semantic indexing*, in: *Proceedings of 25th International Conference on Software Engineering*, 2003, pp. 125–135.

[10] Marcus, A., J. I. Maletic and A. Sergeyev, *Recovery of traceability links between software documentation and source code*, International Journal of Software Engineering and Knowledge Engineering **15** (2005), pp. 811–836.

[11] Maron, M. E. and J. L. Kuhns, *On relevance, probabilistic indexing and information retrieval*, Journal of ACM **7** (1960), pp. 216–244.

[12] Salton, G. and C. Buckley, *Improving retrieval performance by relevance feedback*, Journal of the American Society for Information Science **41** (1990), pp. 288–297.