SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

May 2018, Vol. 61 050105:1–050105:15
https://doi.org/10.1007/s11432-017-9402-3

Special Focus on Internetware: Research Progress and Current Trend

Toward accurate link between code and software documentation

Yingkui CAO[1,2,3], Yanzhen ZOU[1,2*], Yuxiang LUO[1,2],
Bing XIE[1,2] & Junfeng ZHAO[1,2,3]

[1]*Key Laboratory of High Confidence Software Technologies, Ministry of Education, Beijing* 100871*, China;*
[2]*School of Electronics Engineering and Computer Science, Peking University, Beijing* 100871*, China;*
[3]*Beida (Binhai) Information Research, Tianjing* 300450*, China*

Received 30 November 2017/Accepted 27 March 2018/Published online 20 April 2018

Abstract Recovering traceability links between source code and software documentation is an important research topic in software maintenance and software reuse. There have been a lot of research efforts in recovering traceability between documentation and code elements (class, interface, method, etc.), mostly based on program analysis. However, there are still a lot of noise links being established in existing work. In this paper, we propose a novel approach to classifying code elements, occurring in a document, into contextual code elements and salient code elements. As a result, we can filter the noise traceability links between a software document and its contextual code elements and get a higher quality link set. Our classifier is trained based on open source project Lucene's source code and 1899 StackOverflow answer documents about Lucene. We extract code elements from these documents and represent each of these code elements with a 7-dimension feature vector, then we use a decision-tree-based learning model to classify them as salient or not. In the experiments, we get a precision of 70.7% in recognizing the salient code elements of these documents and get 12% improvement compared with Rigby's work. We can filter out 56.5%∼69.3% noise traceability links with different thresholds in our classifier. It can improve the quality of traceability links between source code and their related software documents in application.

Keywords traceability link, code element, software document, contextual code element, salient code element

Citation Cao Y K, Zou Y Z, Luo Y X, et al. Toward accurate link between code and software documentation. Sci China Inf Sci, 2018, 61(5): 050105, https://doi.org/10.1007/s11432-017-9402-3

1 Introduction

Recovering traceability links between source code and software documents is an important research issue in software engineering [1, 2]. Software documentation provides helpful information for developers to understand the function or usage of the related code elements (classes, methods, etc.). Accurate traceability links between source code and software documentation are the precondition of providing smart recommendation in code search and program comprehension [3].

Existing work on recovering traceability links between source code and software documents mainly adopts the idea of program analysis [4–12]. These approaches usually include the following steps: First, they extract code words from a software document. Second, they try to map every code word to a code element in the source code, getting code word's unambiguous location in the source code with the help

* Corresponding author (email: zouyz@pku.edu.cn)

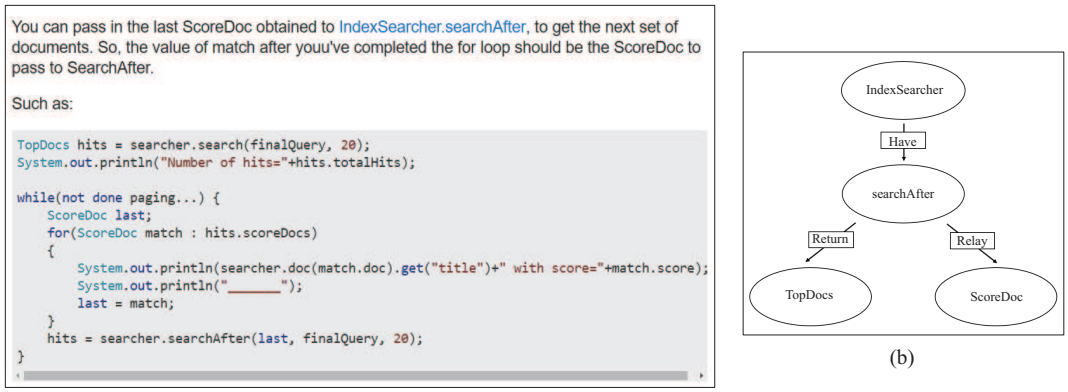

Figure 1 (Color online) (a) Code elements in a StackOverflow answer document and (b) corresponding code relation graph of code elements.

of documentation features. Third, they link a document to all its located code elements. Among these work, the RecoDoc approach [6] obtains the best 96% precision in extracting code elements from software documents in their experiments.

However, there are usually more than one or even dozens of code elements in a software document, but not each of them has been introduced sufficiently [7, 13, 14]. For example, a software document D illustrates how to reuse method m_1 and contains a code usage example and some text segments. As m_1 needs to call another method m_2 in application, m_2 also appears in document D. However, it is right to link D to m_1 rather than m_2, because the document D does not give necessary description for m_2. Hence, we need to classify the code elements existing in a software document further so as to filter out those noise links, like D and m_2.

In a software document, some code elements are used only for context introduction, we call those code elements as contextual code elements. The links between contextual code elements and the document are called noise (weak) links. Meanwhile, there are some code elements that have been mainly discussed or introduced in the software document, we call those code elements as salient code elements. The links between salient code elements and the document are called salient (strong) links. In the above example, m_1 is the salient code element of document D.

In this paper, we propose a novel approach to recognizing the salient code elements and the contextual code elements in a software document. We build a decision-tree-based classifier and train it with 1899 answer documents from StackOverflow. Based on the results of the classifier, we can establish more accurate traceability links between code elements and software documents by filtering out those noise links. The contributions of this paper include:

(1) We quantify 4 kinds of document-related features and 3 kinds of code-related features for recognizing the salient code elements and the contextual code elements in a software document. Different from the existing work, we focus on not only the lexical feature of code snippets in document but also the semantic feature of source code.

(2) We improve the quality of traceability links between source code and software documentation further. In the experiments, we get a precision of 70.7% in recognizing the salient code elements of StackOverflow answer documents, which is more than 12% improvement compared with Rigby's work. At the same time, we can filter out about 56.5%∼69.3% noise traceability links with different thresholds.

2 Motivation example

Here we illustrate our motivation and our work by an answer post on StackOverflow firstly. As shown in Figure 1, a text paragraph and a code block compose this post. This post recommends the user can use the method searchAfter of class IndexSearcher to solve the results display problem when reuse open

source project Lucene. This post mainly describes how to use searchAfter, and it also mentions ScoreDoc and TopDocs. Furthermore, we find that this post is easier to be found with the keyword ScoreDoc than the keyword searchAfter by search engineer.

In the above example, searchAfter is the salient code element of the document, ScoreDoc and TopDocs are contextual code elements of the document. Nevertheless, all these code elements will be linked to this document by the existing work. It indicates that 2/3 links between this document and source code are not accurate, and that we need to filter out those noise links (from this document to ScoreDoc and TopDocs). For this purpose, we need to find some features to represent code elements and find approach to recognizing the salient code elements of a software document.

However, it is not easy to identify the salient code elements in a software document. This is mainly because: (1) In a software document repository, some contextual code elements, like ScoreDoc, may repeat many times in a document or occur in many documents. This fact makes it hard for TF-IDF-like approaches to distinguish salient code elements from contextual code elements. (2) Whether a code element is salient for a document cannot be judged only by its location. In the above example, the salient code element searchAfter and the contextual code element ScoreDoc both appear in document's text segment and document's code snippet. (3) It is hard to use the semantic features of documents' code snippets to recognize salient code elements. Some of these code snippets are uncompleted, some of them contain wrong grammar or inexplicable error. All these facts add uncountable difficulties to code analysis work.

In our work, we find there is other information that we can utilize. That is the relations among those code elements in source code.

For a open source project, we build up the Abstract Syntax Tree (AST) of this project. Based on this AST, we extract all the code elements and build the code relation graph. Each node of this graph represents a code elements. There will be a edge between two nodes, if the two corresponding code elements have relations (such as call, inheritance, etc.), and the edge will be labeled with the relation name.

As shown in Figure 1(a), the salient code element searchAfter has relations with all other 3 code elements. The edge labeled with have means class IndexSearcher have method searchAfter. The edge labeled as return means method searchAfter's return type is TopDocs. The edge labeled as rely means method searchAfter rely on class ScoreDoc to complete function (ScoreDoc is one parameter of searchAfter). In the code relation graph, code elements have different relations with other code elements. We use this code relation graph to map code elements into a 200-dimension vector space to assist us to recognize salient code elements.

3 Approach in detail

The basic idea of our work is to classify code elements into salient code elements and contextual code elements. In order to get a higher quality traceability link set, we just build up the traceability links between a software document and its salient code elements.

Figure 2 shows an overview of our work. Our approach mainly includes three parts: preprocessor, feature extractor, and classifier.

Preprocessor is used to identify the text segments and code snippets from software document, and to extract code elements from these text segments and code snippets. All these code elements are candidates for salient code element.

Feature extractor is responsible for calculating several document-related features and code-related features. In total, we take 4 kinds of document-related features and 3 kinds of code-related features into consideration. These data will be divided into training set and test set. Based on the features of training set, we training a decision-tree-based classifier.

Classifier is used to classify the code elements. The output of the classifier is a float value in the range of [0,1], which stands for weighted salience of a code element. Links between software document and its

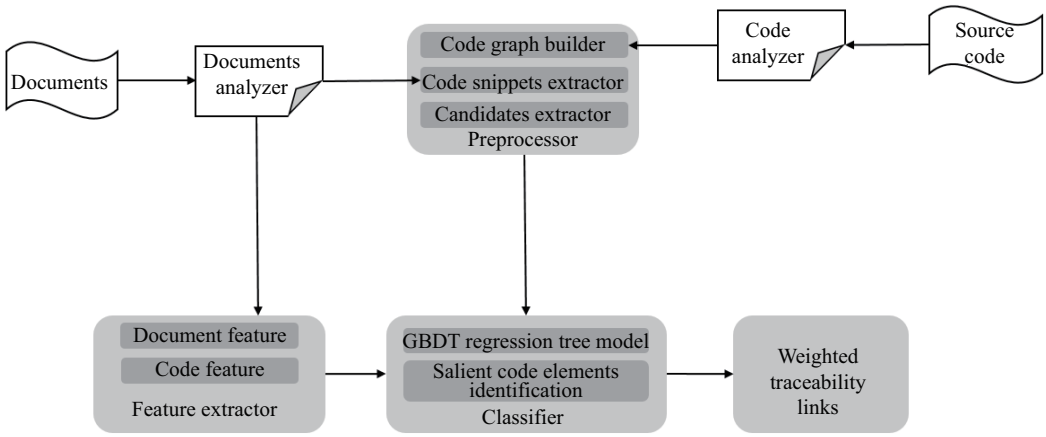

Figure 2 Overview of our approach.

salient code elements are kept, and noise links will be filtered out.

3.1 Preprocessing source code and software documents

As mentioned in Section 2, there are 2 kinds of data involved in our approach: project source code and software documents. We preprocess the source code and documents as follows.

When preprocess the source code, we construct the Abstract Syntax Tree (AST) of the source code firstly. Next, we traverse all the nodes of the AST and get the list of code elements in the source code. We mainly focus on 3 kinds of code elements: class, method, and interface. Then, we mine the relations among different code elements based on the source code's AST. These relations include inheritance (between class and class), implement (between method and class), call (between method and method) and rely (between method and method), etc. Finally, we build a code relation graph for this software project based on these code elements and relations.

The preprocessing of software documents mainly includes 3 steps. First, We parse and divide the document content into code snippets and natural language text. Second, we tokenize the natural language text and filter out useless stop words and HTML tags. Third, we extract code elements from software documents. For this purpose, we adopt some idea of RecoDoc [6] in our approach. RecoDoc [6] tries to identify code-like terms in documents using patterns. These patterns include parentheses for functions, camel cases for types, etc. In order to filter out the mismatch terms, we use the code elements in the relation graph to match these code-like terms. After that, we get all the candidate code elements of a software document. We extract 4 code elements from the posts in Figure 1, "searchAfter", "ScoreDoc", "TopDocs" and "totalHits".

3.2 Feature selection

In our work, we totally select 7 kinds of features to identify the salient code elements. All this features are divided into 2 categories: Document-Related Feature and Code-Related Feature. We will list all these features' definitions and describe how to quantify them in this part.

Document-related feature refers to features that we could identify from the software document directly, including TF-IDF, Location, Expwords, and DocCos.

(1) TF-IDF. The term frequency (TF) and inverse document frequency (IDF) of each candidate code element in the documents. Here TF is the occurrence frequency of the code element in a document and IDF is the inverse of the code element's occurrence frequency in a document set. TF-IDF follows the formula (1), where $\mathrm{tf}_{t,r}$ is the number of occurrences of code element t in document r, df_t is the number of document (contained in the document set) in which the code element t occurs, N is the number of documents in the document set.

$$\mathrm{tf}-\mathrm{idf}_{t,r}=(1+\log(\mathrm{tf}_{t,r}))\times\log\left(\frac{N}{\mathrm{df}_t}\right). \tag{1}$$

(2) Location. The code element's position in which a code element occurs in a document. A document mostly is composed of natural language text and code snippet. As the example in Figure 1 shows, the code elements appear in the natural language text or in the code snippet of the document. Thus, the value of feature Location takes one of the following three values: Text (only appears in the document's text), Code (only appears in the document's code snippet), Both (appears in document's text and code snippet).

(3) Expwords. The number of special words appearing around a code element in a software document. Based on our observation, we find there are usually some special words around the import code elements. These special words include special verbs (such as use, choose, etc.) and positive words (such as valuable, preferred, etc.).

(4) DocCos. The cosine similarity between a code element's comments in source code and the software document. Some software project's source code contains natural language comments or natural language signature content (e.g., the variable names). It may be helpful to calculate this similarity for identifying the salient code elements in a document.

Code-Related Feature refers to features that we could identify from the source code, including Type, Relation, and Distance.

(5) Type. The type of code element in source code. We mainly focus on three types: class, interface and method. We regard the field of a class as a special kind of method. When a class and one of the class's method occur in a document at the same time, the method is likely to be more important. The class may only serve as the background.

(6) Relation. The number of relation of a code element with other code elements in the same document. Among the code elements, there are many kinds of relations in the source code, such as inheritance relation between class and class, call relation between method and method and rely relation between method and method, etc. In all the code elements occurring in a document, we believe that more relations a code element wiht others, more important the code element is.

(7) Distance. The distance between a code element and a document's semantic center. We use Code Embedding approach [14] to map all code elements to a 200-dimension vector space. We get the vectors for all the code elements occurring in a document and average these vectors as the semantic center of this document. The Euler distance between a code element's vector and semantic center will be the value of this feature. When we map the code elements into a vector space, we find the salient code elements tend to be the ones which are the nearest to the document center.

Word2vec [15] compute continuous vector representations of words, i.e., mapping words into a vector space. Similarly, Code Embedding try to represent a code elements with a vector. All the code elements compose words set and all project source code compose document set. In order to train the model, we use the code elements relations in the relation graph to form the the loss function. For example, there are two code elements in the relation graph: h and t. These two code elements are related by relation l which starts from h to t. We annotate this relation as $\langle h,l,t\rangle$. We will give it a mathematical constraint expressed in the potential space as follows:

$$\mathrm{loss}(\langle h,l,t\rangle)=|M_l h+l-M_l t|. \tag{2}$$

Here h and t represent the vectors of the code element h and t in the representation space, M_l and l is a linear transformation and a translation related to the relation type. If code relation graph contains relation $\langle h,l,t\rangle$, we prefer to make $\mathrm{loss}(\langle h,l,t\rangle)$ as small as possible; otherwise, make $\mathrm{loss}(\langle h,l,t\rangle)$ as large as possible. In this way, code elements with similar code contexts can be constrained in the same flow pattern. The more similar is the context, the tighter is the constraint, the closer is the distance between two code elements in the representation space. In general, we use the batch gradient descent method to train the learning model, the loss function is shown in formula (3).

$$\mathrm{Loss}(G) = \sum_{\langle h,l,t\rangle \in G, \langle h',l',t'\rangle \notin G} \frac{1}{1+\mathrm{e}^{-(\mathrm{loss}(\langle h,l,t\rangle)-\mathrm{loss}(h',l',t'))}}. \tag{3}$$

3.3 Classifier construction

In the above description, we extract and quantify features to represent code elements in a software document. Combining these features to get the best classification result is a time-consuming work. Therefore, we adopt a supervised machine-learning algorithm in our work.

We choose Gradient Boosting Decision Tree (GBDT) [16, 17] as our learning model. GBDT is an ensemble of multiple regression trees, each of them is used to learn the residuals of the previous trees. The advantage of this model is that it can use all kinds of data as input without normalization preprocessing. Not only the features with continuous value (like DocCos, Distance), but also the features with discrete value (like Type, Location) can be directly used in the model. At the same time, the loss function of GBDT is very robust to outliers, and it is easy to adjust model.

There are three classifiers are involved in our experiments.

Normal classifier. First, we extract code elements from each document, calculate the 7 features' value for all the code elements as the way described in Subsection 3.2, and represent each code element with a 7-dimension feature vector.

Next, we put the feature vectors into the classifier, and the classifier will output a score (we call this as original score) for each code element. This score indicates how important the corresponding code element is. The output of classifier is continuous value from 0 to 1 ([0, 1]).

Finally, we select a threshold t to decide the code elements is salient or not. A code element will be classified as salient for the document when the code element's score is larger than or equal to t. In the experiments, we get the highest F-value when the threshold is 0.4, with a recall of 84.6% and a precision of 62.3%.

When we look back on the results, we find some very interesting phenomenon: First, the output score of salient code elements from different documents are very different, some documents' salient code elements get high classifier scores (which are more than 0.8), but some documents' salient code elements get very low scores (which are less than 0.4). Therefore, it is hard to improve the classifier's performance if we just set a global threshold; Second, the feature values of code elements from different documents are very different, too. For example, there are three candidate code elements in document D_1, whose Distance values are {0.0833, 0.0500, 0.0042}, there are another document D_2, which has only one candidate code element, whose Distance feature value is 0. Therefore, this kind of difference may also affect the performance of our classifier. So we modified our classifier further:

Modified classifier I. This classifier normalize some features' value of normal classifier. These features include TF-IDF, Expword and Distance. For example, in document D_1, the feature Distance of these three code elements are {0.0833, 0.0500, 0.0042}. We sum up these three values, getting 0.1375 (0.0833+0.0500+0.0042), and divide these three values by this sum separately, getting the new feature. Distance of the three code elements {0.6058, 0.3636, 0.0305}. If the sum is 0, then features will remain as before.

Modified classifier II. This classifier calculates relative scores of the code elements in a software document based on original scores. For example, there are three candidate code elements in a document, and their original output scores are 0.6, 0.45, 0.3, respectively. We divide these three values by the max score (0.6) among them, getting the relative scores 1.0, 0.75, 0.5. This classifier will use these relative scores to classify instead of original scores.

4 Experiments

4.1 Research questions

All the experiments in this paper are used to address the following questions:

Q1: Is our approach efficient in recognizing the salient code element? Recognizing the salient code elements and the contextual code elements in a software document is the premise of our work.

Q2: What is the performance of our work compared with existing work? Rigby et al. [7] have done some work on recognizing the essential code elements. Here we will compare with their approach in our experiments.

Q3: Which feature impacts the results mostly? We wonder what the influence of a single feature is in salient code elements recognition.

Q4: What is the performance of our work in filtering out noise traceability links between source code and software documentation? The purpose of our work is to refine the traceability links between code and software document, and to improve the links quality. Thus we will investigate the effect of our approach in filtering out noise traceability links between source code and software documents.

4.2 Data set

We set up our experimental data set using 1899 StackOverflow answer documents which are about open source project Lucene. As a famous Q&A website, StackOverflow has gained a lot of questions and corresponding answers. These question and answer documents are typical and representative software documents. Here we select open source project Lucene for example. We collect its related documents on StackOverflow and recover the traceability links between its source code and these documents.

We select the questions tagged with "lucene" from StackOverflow, and gather all the corresponding answers. We totally collect 9375 questions and 12034 answers from August 2008 to March 2016. We randomly select and filter out some inappropriate answers, such as those which have no code element and those which are not related to java. Finally, we get 1899 answers and we will use those answers documents into our experiments.

After we get these documents, we preprocess these answer documents and extract 5097 candidate code elements from these documents. In average, there are 2.68 code elements per document, and 19 code elements in a document at most and 1 code element at least. We invite 8 graduate students to annotate these code elements. These students are familiar with Lucene and have more than 2 years experience in java programming. They are asked to mark the salient code elements of each document. The marking task assignment ensures that each document will be marked at least by two students. If these two students tag different salient code elements for a document, they are asked to refer to the javadoc for these code elements, discuss and give a final result. In total, annotators annotate 37 of these 1899 documents with disagreement. The ratio of these controversial data is less than 2%.

Finally, 2137 of these candidates are labeled as salient code element. In average, 1.13 code elements are labeled as salient per document, and there are 3 salient code elements in a document at most and 1 salient code element at least. The annotation result indicates that most of the documents contain only one salient code element, but there are also many documents having more than one salient code elements.

4.3 Evaluation metrics

To verify the experimental results, we use 10-fold cross-validation to calculate the precision, recall, and F-value of salient code element recognition. We measure the filtering rate of noise links between code and software document to evaluate our contribution's efficiency. The Precision, Recall, F-value and the Filtering Rate are calculated as formula (4)–(7). All 4 parameters are listed in Table 1, where tp represents the number of code elements correctly classified as salient by our approach; fp represents the number of code elements wrongly classified as salient by our approach; fn represents the number of code elements wrongly classified as non-salient by our approach; tn represents the number of code elements correctly classified as non-salient by our approach.

$$P = \frac{\mathrm{tp}}{\mathrm{tp} + \mathrm{fp}}, \tag{4}$$

$$R = \frac{\mathrm{tp}}{\mathrm{tp} + \mathrm{fn}}, \tag{5}$$

Table 1 Method metric description

	Salient (positive)	Non-salient (negative)
True results	tp	fp
False results	fn	tn

Table 2 The results of our classifiers

Normal classifier				Modified classifier I				Modified classifier II			
t	P	R	F-value	t	P	R	F-value	t	P	R	F-value
0.2	0.498	0.923	0.647	0.2	0.558	0.923	0.696	0.5	0.638	0.814	0.715
0.3	0.562	0.904	0.693	**0.3**	**0.643**	**0.853**	**0.733**	0.6	0.646	0.795	0.713
0.4	**0.623**	**0.846**	**0.717**	0.4	0.698	0.712	0.705	0.7	0.659	0.769	0.71
0.45	0.679	0.718	0.698	0.45	0.786	0.66	0.718	0.8	0.689	0.737	0.712
0.5	0.783	0.532	0.634	0.5	0.814	0.59	0.684	**0.85**	**0.707**	**0.712**	**0.709**
0.55	0.857	0.423	0.567	0.55	0.846	0.564	0.677	0.9	0.711	0.692	0.701
0.6	0.862	0.359	0.507	0.6	0.871	0.474	0.614	0.95	0.721	0.647	0.682

$$F = \frac{2 \times P \times R}{P + R}, \tag{6}$$

$$C = \frac{\text{tn}}{\text{fp} + \text{tn}}. \tag{7}$$

4.4 Evaluation setup and results

4.4.1 *Q1: Is our work efficient in recognizing the salient code element?*

In the experiments, the classifier is trained based on the train data set. We use this classifier to classify the code elements for each document. Table 2 shows three classifiers' classification results based on different thresholds. In Table 2, t stands for threshold, P stands for precision, R stands for recall.

The left part of Table 2 shows the result of the normal classifier. We can see that the precision increases as the threshold increases, but the recall decreases. The F-value of the classifier is the highest when the threshold t is equal to 0.4. Meanwhile, we get a precision of 62.3% and a recall of 84.6%. However, the large gap (22.3% = 84.6 % − 62.3%) between precision and recall makes the classifier maybe not the optimal choice. In order to optimize the model, we build up another two classifiers: modified classifier I and modified classifier II.

The middle part of Table 2 shows the classification results of modified classifier I. The F-value is the highest when the threshold t is equal to 0.3. At the same time, the classifier achieves a 64.3% precision and a 85.3% recall.

The right part of Table 2 shows the classification results of modified classifier II. The F-value is the highest when the threshold t is equal to 0.85. At the same time, the classifier achieves a 70.7% precision and a 71.2% recall.

Seeing from the results of the classifier of modified classifier II, both of precision and recall are larger than 0.7 when we set the threshold to be 0.85. Compared with the results of normal classifier and modified classifier I, modified classifier II's F-value is more stable when threshold t changes. We regard 0.85 as the optimal threshold for modified classifier II, though it does not bring us with the largest F-value. The reason is that it brings us with a little difference between precision and recall, when the threshold is 0.85.

In Table 3, we illustrate the process of modifier classifier II scoring each code element in the motive example document in Figure 1. The different feature values of each code element are also presented. There are four code elements in this document, including ScoreDoc, TopDocs, totalHits and searchAfter. All of them have direct relations with each other in Lucene's source code, so they have the same Relation value 3. ScoreDoc gets a lower TF-IDF value than searchAfter though it appears more times in this document, which is mainly because there are too many ScoreDoc and TopDocs in the documents of data set. Finally, the classifier recognizes searchAfter as the salient code element of this document and suggests to build a

Table 3 The scores of code elements extracted from the motive example document

	TF-IDF	Location	Expwords	DocCos	Type	Relation	Distance	Origianl score	Normalized score
ScoreDoc	2.142	2	2	0.106	0	3	0.239	0.396	0.534
TopDocs	1.527	1	0	0.04	0	3	0.279	0.209	0.282
totalHits	2.231	1	0	0.008	1	3	0.488	0.135	0.182
searchAfter	3.311	2	2	0.184	1	3	0.026	0.741	1.0

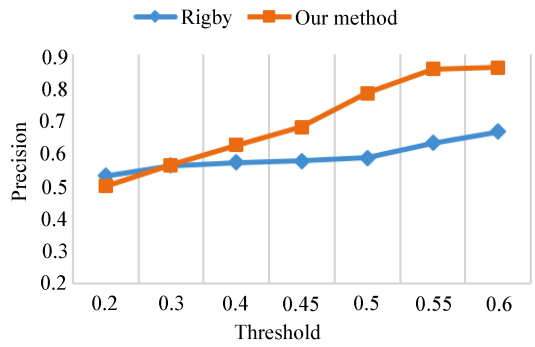

Figure 3 (Color online) The precision of our approach and Rigby's work.

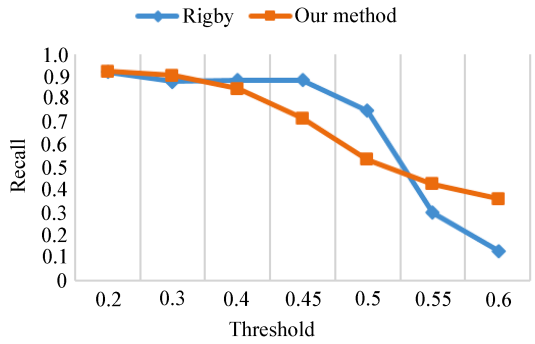

Figure 4 (Color online) The recall of our approach and Rigby's work.

traceability link between searchAfter and this document. That is mainly because: (1) The context of this document is much more similar to searchAfter's comments in source code; (2) searchAfter and ScoreDoc have both appeared in natural language text and code snippet, while the other two code elements only appear in the code snippet; (3) There are some positive words in the context of searchAfter and ScoreDoc, bringing them with higher Expwords values; (4) Compared to other code elements, searchAfter is closer to the semantic center.

4.4.2 *Q2: What is the performance of our work compared with existing work?*

Rigby et al. [7] have done some work on detecting which of the code elements in a document are salient. Different from Rigby's work, we add some new features, which are mainly related to relations among the code elements in the document, to classifier. Here we will compare our work (normal classifier) with their approach.

We mainly compare our approach and Rigby's work from 2 metrics: precision and recall. As we can see from Figure 3, our approach outperforms Rigby's work a lot in precision when the threshold is larger than 0.3. With a high threshold, our approach can classify the code elements with high certainty. From the aspect of recall (the results are shown in Figure 4), Rigby's approach is better than our work on some thresholds. But when the threshold is bigger than 0.55, our work is better than Rigby's approach. At the same time, our work shows us with a stable recall when threshold changes.

4.4.3 *Q3: Which feature impacts the results mostly?*

In our approach, we use 3 code-related features. The result shows our approach is efficient for recognizing the salient code element of a document. However, we have not figured out whether these code-related features are more important in our work.

To answer this question, we build 7 different classifiers. Every classifier is built on only one feature and will use the original output value to classify code elements. Table 4 shows the precisions and Table 5 shows the recalls of each classifier with different thresholds. The null entries in the tables indicate that no code element is scored more than the corresponding threshold value by the corresponding classifier. For example, 0.525 (in boldface) means when we use feature DocCos to build a classifier and set the threshold to be 0.4, the classifier get the 0.525 precision.

Table 4 The precisions of classifiers based on different features and thresholds

Feature	Threshold						
	0.2	0.3	0.4	0.45	0.5	0.55	0.6
DocCos	0.344	0.478	**0.525**	0.553	0.565	0.588	0.595
TF-IDF	0.341	0.344	0.517	0.526	0.833	–	–
Type	0.341	0.345	0.500	–	–	–	–
Location	0.563	0.563	0.563	0.563	0.563	0.64	0.64
Expwords	0.341	0.34	0.402	0.455	–	–	–
Relation	0.374	0.483	0.533	0.533	0.533	–	–
Distance	0.344	0.434	0.8125	0.925	0.925	0.925	0.925
Normal classifer	0.498	0.562	0.623	0.679	0.783	0.857	0.862

Table 5 The recalls of classifiers based on different features and thresholds

Feature	Threshold						
	0.2	0.3	0.4	0.45	0.5	0.55	0.6
DocCos	1.000	0.705	0.609	0.532	0.474	0.321	0.160
TF-IDF	1.000	0.974	0.288	0.064	0.032	–	–
Type	1.000	0.994	0.083	–	–	–	–
Location	0.885	0.885	0.885	0.885	0.885	0.205	0.205
Expwords	1.000	0.987	0.423	0.064	–	–	–
Relation	1.000	0.718	0.564	0.506	0.506	–	–
Distance	0.968	0.763	0.250	0.237	0.237	0.237	0.237
Normal classifier	0.923	0.904	0.846	0.718	0.532	0.423	0.359

Seen from Tables 4 and 5, we can find:

Feature Distance is the most important among these 7 features. As we can see from Table 4, the precision of the classifier, which is built on feature Distance, changes significantly (from 43.4% to 81.25%), when the threshold changes from 0.3 to 0.4. However, the precision is without any change when the threshold change between 0.45 and 0.6. So the feature Distance can score the code elements with a higher score ($\geqslant 0.6$) or a lower score ($\leqslant 0.4$). That is to say, feature Distance can separate the code element with a big margin, and the result is reliable when the score is bigger than 0.6. Therefor, feature Distance will be perfect for binary classification.

DocCos, Location and Relation play important roles in classifying code elements. We totally annotate 2137 code elements as salient from all the 5097 candidate code elements. The precision will be 0.419 (2137/5097) if we randomly classify code elements. As Table 4 shows, all the 3 features' corresponding classifiers' precisions are bigger than 0.419 when threshold increases. Meanwhile, these classifiers' recall are bigger than 0.205. These results indicate all these 3 features are helpful.

TF-IDF, Type and Expwords make fewer contributions than other features. Although the precisions of these 3 features' corresponding classifiers are bigger than 0.419 when threshold increases, the recalls are too low to make the results reliable. However, it is hard to conclude that these 3 features make no difference. We usually use some special words to emphasize something, and it is strange that feature Expwords does not take effect. One possible reason is that: salient and contextual code elements always occur together, and we do not make it clear which code element these special words emphasize. So, we need to do more researches to make better use of these features or abandon them.

The results also indicate: the semantic information of code snippets is important when extract the features of a document. Among all the classifiers, the classifiers built on feature Relation and Distance get relatively better results. These two features are calculated based on relation graph and are related to the semantic of a document. It indicates our work on code relation graph building is helpful in salient code element recognition.

We also illustrate F-values of all 7 classifiers, which are shown in Figure 5. The F-value of Righy's work and our normal classifier are also shown in Figure 5. The horizontal axis represents the threshold,

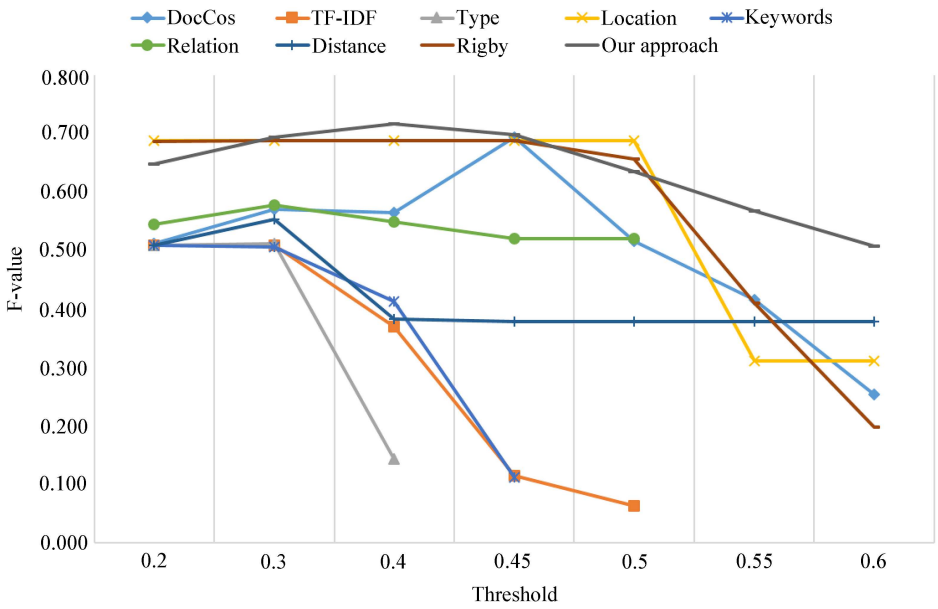

Figure 5 (Color online) The F-values of classifiers based on different features and thresholds.

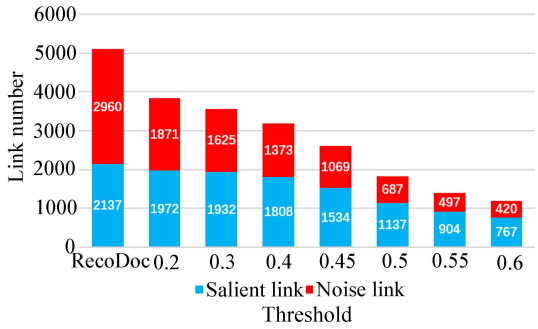

Figure 6 (Color online) Classification result based on original score.

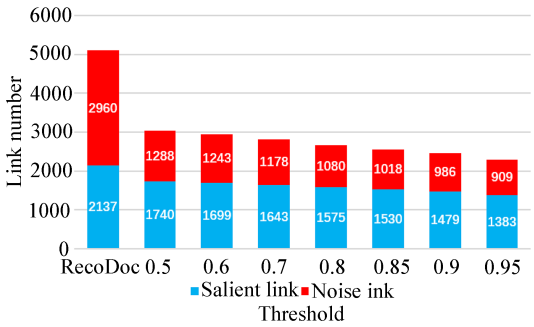

Figure 7 (Color online) Classification result based on relative score.

the vertical axis represents the F-value. Obviously, our normal classifier which combines all these seven features gets the best result, and it gets the best F-value (0.717) when threshold is 0.4. The results indicate these features are complementary. Although a single feature cannot bring a good result, combining them will bring us a better result.

4.4.4 *Q4: What is the performance of our work in filtering out noise traceability links between source code and software documentation?*

In this work, we try to filter out those noise traceability links between source code and software documentation so as to improve the quality of traceability links. We implement the approach of RecoDoc to extract candidate code elements from documents. Totally, we extract 5097 code elements from the experimental data set. Among them, only 2137 code elements are annotated as the salient code elements of particular document. It means that we need only keep these 2137 links between our experimental source code and software documentation, the remaining 2960 (5097−2137) links are noise links, which is about 58.1% (2960/5097) of the total links, should be filtered out.

We illustrate the experimental results of our approach on noise link filtering with different thresholds. Respectively, the normal classifer's classification results are shown in Figure 6 and the modified classifier II's classification results are shown in Figure 7. The bars stand for the code elements which are classified as salient. The blue parts stand for the salient code elements which are correctly classified, called salient

(strong) links between code and documentation. The red parts stand for the contextual code elements which are wrongly kept, called noise (weak) links between code and documentation.

As Figure 6 shows, normal classifier filters out 1089 (2960−1871) noise links and keeps 1972 salient links when threshold is 0.2. 2540 (2960−420) noise links are filtered out and 767 salient links are kept when threshold is 0.6. Along with threshold changing from 0.2 to 0.6, noise link filtering rate changes from 36.8% (1089/2960) to 85.8% (2540/2960), salient link keeping rate changes from 92.3% (1972/2137) to 35.9% (767/2137).

As Figure 7 shows, modified classifier II filters out 1672 (2960−1288) noise links and keeps 1740 salient links when threshold is 0.5. 2051 (2960−909) noise links are filtered out and 1383 salient links are kept when threshold is 0.95. Along with threshold changing from 0.5 to 0.95, noise link filtering rate changes from 56.5% (1672/2960) to 69.3% (2051/2960), salient link keeping rate changes from 81.4% (1740/2137) to 64.7% (1383/2137).

The results show that we can increase the threshold if we want to filter out more noise traceability links. Although some salient links will be filtered out when the threshold increases, more noise/weak links are dropped. For example, when we increase the threshold from 0.3 to 0.4 in Figure 6, we can filter 252 (1625−1373) noises links while discard 124 (1932−1808) strong links. But in application, we advise you should keep strong links as many as possible. As our purpose is to improve the quality of the traceability links between source code and software documents, we want to improve the precision and keep a good recall.

5 Threats to validity

There are still some threats to the validity of our results though we have gotten some new advance compared with existing work.

Our experiments are based on 1899 answer documents from StackOverflow. Although this provides a good starting point, there might be limitations on the representativeness of software documents. It is a better choice to conduct larger scale study in different programming languages and different kinds of projects. We are trying to apply our approach to other kinds of software documents, such as Mailing lists and Application Programming Interface (API) tutorials. At the same time, we also trying to apply our approach to link different projects' source code and their documents.

As mentioned above, we use some features whose values are discrete. For example, a code element can occur in code snippets or context text of a document, so the feature Location can take 3 candidate value: Text, Code and Both (mentioned in Subsection 3.2). From the results of the experiments, we find a single test case whose value of Location is Both will tend to get a higher predict value. However, there are documents without code snippets, which means all the feature Location of code element extracted from these document will be labeled as Text, and these data may get low predict value, no matter the corresponding code elements are salient or not. In order to get a better result, we need to overcome this issue in the feature work.

6 Related work

Recovering traceability links between source code and software documentation is an important research topic in software engineering. Existing work can be divided into two categories: program analysis based approaches and information retrieval based approaches.

Information retrieval based traceability links recovery approaches [1, 2, 18–25] are like to treat the traceability links recovery as an information retrieval process. As there is some similar content between source code and software documentation, researchers try to link code to those software documents with high information similarity. For example, Antoniol et al. [1] construct query using a code component's identifiers in source code, then software documents are searched and sorted against query by words similarity, the search results over similarity threshold (or Top-k search results) are selected and linked.

Tsuchiya et al. [18] recover the links between requirement documents and source code according their similarity on TF-IDF. Xu et al. [20] propose an approach for traceability between Chinese documentation and source code based on Latent Dirichlet Allocation (LDA). These approaches are limited by the fact that some code identifiers (such as the class name or method name) are not self-explanatory. For those software projects whose source code are short of natural language comments, the link results will not be very good. To improve the precision of similarly evaluation, more information is added in. Tsuchiya et al. [19] propose an approach for recovering traceability links between requirements and source code using the configuration management log. Ye et al. [23] map the words and code tokens into a same vector space before searching. Rahimi et al. [24] present Trace Link Evolver (TLE), an approach for automating the evolution of trace links as changes are introduced to source code. Zhang et al. [25] propose an approach to infer links between concerns and methods with Multi-abstraction Vector Space Model.

Compared with the above work, program analysis based traceability links recovery approaches [4–12] could locate more granular code element. Typically, Bacchelli et al. [4, 5] build the traceability links between Email document and code by syntax analysis of code elements. Dagenais et al. [6] implement a tool named RecoDoc, which uses document context to disambiguate and extract code elements from document accurately, its average precision and recall for code element extraction could reach 96%. Rigby et al. [7] give a number of regular expressions based on the island grammar and naming rules and implement a tool Automatic Code element Extractor (ACE) to extract code elements in document without source code. McMillan et al. [8] extract the "rely" and "call" relations among code elements in source code. They use these relations to build traceability link graph (TLG). Based on the TLG, they try to find new traceability links among software documents. Subramanian et al. [10] apply Partial Program Analysis (PPA) technology and island grammar to syntax analyze code snippets and code elements embedded in the document, then uses an iterative inference process to build bi-directional traceability links between API tutorials and forum webpages. These work mainly focus on extracting code elements from software documents and locating its location in source code. As a software document usually contains many code elements, it is not accurate to link a document with its all code elements. This problem motivates our work. Now we can recognize the salient code elements in a software document, filter out those noise links and improve the quality of the traceability links between source code and software documents.

There is still some work offering good reference to our approach. For example, Petrosyan et al. [11] and Jiang et al. [12] use a supervised machine learning approach to extracting the relative description for APIs. Kim et al. [26] carry on a study on recovering traceability between documents of a software project and de Lucia et al. [27] point out that we should enhance artifact management with traceability recovery. Nishikawa et al. [28] recover traceability links between software artifact A and C based on links between A and B as well as links between B and C. Ye et al. [29] propose a semi-supervised machine learning approach to extracting API mentions from informal social text. Sridhara et al. [30] generate descriptive summary comments for Java methods. Besides those work, some studies analyze the project source code and apply its abstract syntax tree information to code or document search [31–37]. In this paper, we use 7 kinds of features to represent our learning example and train our classifier. We use Code Embedding to measure the distances between code elements. It is an effective way to combine code structure and document content. We are trying to semantically understand different software document by Code Embedding, which will help us improve the accuracy of the traceability links between code and software documentation.

7 Conclusion

In this paper, we propose a decision-tree-based approach to identifying the salient code elements and the contextual code elements in a software document. The approach can filter out the noise traceability links between source code and software documents. Our learning model considers not only 4 document-related features, but also 3 code-related features. The experimental results show that our approach can recognize the salient code elements in StackOverflow answer documents with a precision of 70.7%. At the same

time, we can filter out 56.5%∼69.3% noise traceability links in the results of the RecoDoc work. Our approach greatly improves the quality of traceability links between code and software documentation.

Acknowledgements This paper was supported by National Key Research and Development Project of China (Grant No. 2016YFB1000804) and National Natural Science Fund for Distinguished Young Scholars (Grant No. 61525201).

References

1 Antoniol G, Canfora G, Casazza G, et al. Recovering traceability links between code and documentation. IEEE Trans Softw Eng, 2002, 28: 970–983
2 Marcus A, Maletic J I. Recovering documentation-to-source-code traceability links using latent semantic indexing. In: Proceedings of the 25th International Conference on Software Engineering, Portland, 2003. 125–135
3 Robillard M P, Marcus A, Treude C, et al. On-demand developer documentation. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME 2017), Shanghai, 2017. 479–483
4 Bacchelli A, D'Ambros M, Lanza M, et al. Benchmarking lightweight techniques to link e-mails and source code. In: Proceedings of the 16th Working Conference on Reverse Engineering (WCRE 2009), Lille, 2009. 205–214
5 Bacchelli A, Lanza M, Robbes R. Linking e-mails and source code artifacts. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, Cape Town, 2010. 375–384
6 Dagenais B, Robillard M P. Recovering traceability links between an API and its learning resources. In: Proceedings of the 34th International Conference on Software Engineering (ICSE 2012), Zurich, 2012. 47–57
7 Rigby P C, Robillard M P. Discovering essential code elements in informal documentation. In: Proceedings of the 2013 International Conference on Software Engineering, San Francisco, 2013. 832–841
8 McMillan C, Poshyvanyk D, Revelle M. Combining textual and structural analysis of software artifacts for traceability link recovery. In: Proceedings of ICSE Workshop on Traceability in Emerging Forms of Software Engineering. Washington: IEEE Computer Society, 2009. 41–48
9 Panichella A, McMillan C, Moritz E, et al. When and how using structural information to improve ir-based traceability recovery. In: Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR 2013), Genova, 2013. 199–208
10 Subramanian S, Inozemtseva L, Holmes R. Live API documentation. In: Proceedings of the 36th International Conference on Software Engineering (ICSE 2014), Hyderabad, 2014. 643–652
11 Petrosyan G, Robillard M P, Mori R D. Discovering information explaining API types using text classification. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1, Florence, 2015. 869–879
12 Jiang H, Zhang J, Li X, et al. A more accurate model for finding tutorial segments explaining APIs. In: Proceedings of IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016), Suita, 2016. 157–167
13 Zou Y Z, Ye T, Lu Y Y, et al. Learning to rank for question-oriented software text retrieval. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015), Lincoln, 2015. 1–11
14 Lin Z Q, Xie B, Zou Y Z, et al. Intelligent development environment and software knowledge graph. J Comput Sci Technol, 2017, 32: 242–249
15 Mikolov T, Chen K, Corrado G, et al. Efficient estimation of word representations in vector space. ArXiv:1301.3781
16 Friedman J H. Greedy function approximation: a gradient boosting machine. Ann Stat, 2001, 29: 1189–1232
17 Friedman J H. Stochastic gradient boosting. Comput Stat Data Anal, 2002, 38: 367–378
18 Tsuchiya R, Kato T, Washizaki H, et al. Recovering traceability links between requirements and source code in the same series of software products. In: Proceedings of the 17th International Software Product Line Conference, Tokyo, 2013. 121–130
19 Tsuchiya R, Washizaki H, Fukazawa Y, et al. Recovering traceability links between requirements and source code using the configuration management log. IEICE Trans Inf Syst, 2015, 98: 852–862
20 Xu Y, Liu C. Research on retrieval methods for traceability between Chinese documentation and source code based on LDA. Comput Eng Appl, 2013, 49: 70–76
21 Lai G, Wang X, Liu C. Analysis and improvement on retrieval methods for traceability links between source code and documentation. ACTA Electron Sin, 2009, 37: 22–30
22 Yang B, Liu C. Research on traceability recovery between documentation and source code based on software structure. J Front Comput Sci Tech, 2014, 6: 7
23 Ye X, Shen H, Ma X, et al. From word embeddings to document similarities for improved information retrieval in software engineering. In: Proceedings of the 38th International Conference on Software Engineering, Austin, 2016. 404–415
24 Rahimi M, Goss W, Cleland-Huang J. Evolving requirements-to-code trace links across versions of a software system. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME 2016), Raleigh, 2016. 99–109
25 Zhang Y, Lo D, Xia X, et al. Inferring links between concerns and methods with multi-abstraction vector space model. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME 2016), Raleigh, 2016. 110–121

26 Kim S, Kim H Y, Kim J A, et al. A study on traceability between documents of a software R&D project. In: Advanced Multimedia and Ubiquitous Engineering. Berlin: Springer, 2016. 203–210

27 de Lucia A, Fasano F, Oliveto R, et al. Enhancing an artefact management system with traceability recovery features. In: Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004), Chicago, 2004. 306–315

28 Nishikawa K, Washizaki H, Fukazawa Y, et al. Recovering transitive traceability links among software artifacts. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME 2015), Bremen, 2015. 576–580

29 Ye D, Xing Z, Foo C Y, et al. Learning to extract api mentions from informal natural language discussions. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME 2016), Raleigh, 2016. 389–399

30 Sridhara G, Hill E, Muppaneni D, et al. Towards automatically generating summary comments for java methods. In: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2010), Antwerp, 2010. 43–52

31 Eddy B P, Kraft N A. Using structured queries for source code search. In: Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution, Victoria, 2014. 431–435

32 Ponzanelli L, Mocci A, Bacchelli A, et al. Improving low quality stack overflow post detection. In: Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution, Victoria, 2014. 541–544

33 Lin Y, Liu Z, Sun M, et al. Learning entity and relation embeddings for knowledge graph completion. In: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, Austin, 2015. 2181–2187

34 Creation O W L. To generate the ontology from Java source code. Int J Adv Comput Sci Appl, 2011, 2: 111–116

35 McMillan C, Grechanik M, Poshyvanyk D, et al. Portfolio: finding relevant functions and their usage. In: Proceedings of the 33rd International Conference on Software Engineering, Waikiki, 2011. 111–120

36 Bajracharya S K, Ossher J, Lopes C V. Leveraging usage similarity for effective retrieval of examples in code repositories. In: Proceedings of the 18th ACM SIGSOFT international symposium on Foundations of software engineering, Santa Fe, 2010. 157–166

37 Butler S, Wermelinger M, Yu Y J. Investigating naming convention adherence in Java references. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME 2015), Bremen, 2015. 41–50