Hindawi
Scientific Programming
Volume 2018, Article ID 5494209, 16 pages
https://doi.org/10.1155/2018/5494209

Research Article

A Structure-Driven Method for Information Retrieval-Based Software Change Impact Analysis

Yun He,[1,2] **Tong Li**,[1,2] **Wei Wang**,[1,2] **Wei Lan**,[1] **and Xiang Li**[1]

[1]*Software School, Yunnan University, Kunming, Yunnan, China*
[2]*Key Laboratory for Software Engineering of Yunnan Province, Kunming, Yunnan, China*

Correspondence should be addressed to Tong Li; tli@ynu.edu.cn and Wei Wang; wangwei@ynu.edu.cn

Received 21 March 2018; Revised 8 August 2018; Accepted 3 September 2018; Published 4 October 2018

Academic Editor: Danilo Pianini

An important application of information retrieval technology is software change impact analysis. Existing information retrieval-based change impact analysis methods select a single method to transform the source code corpus into vectors in a process known as indexing. The single method is chosen from two primary methods, known as the bag-of-words and word embedding models, each having their specific advantages and disadvantages. The bag-of-words model records every word in the source code but ignores contextual information in the corpus. The word embedding model records the contextual information but loses detail for individual words. To address this problem, we propose a structure-driven method for information retrieval-based change impact analysis (named SDM-CIA). SDM-CIA integrates the bag-of-words and word embedding models based on the software's structure. Our experiments using a standard benchmark shows that when compared with the existing methods, SDM-CIA improves on precision performance, recall performance, F-score performance, and MRR performance by an average of 3.65%, 3.82%, 3.6%, and 10.28%, respectively. Our experiments confirm the effectiveness of SDM-CIA.

1. Introduction

The main activities in software maintenance are modifications to software source code units [1]. These modifications are called changes. Change impact analysis (CIA) is the process of identifying the mapping relationship between a change request and corresponding source code units, which makes it possible to find new and changed functionality in the source code over time [2]. Programmers will do change impact analysis prior to making any changes to the source code [1]; then, they can decide which source code should be modified. Wilde et al. [3] proposed the earliest CIA method, known as software reconnaissance. After more than 20 years of development, it has become the foundational technology in many fields, including components retrieval [4], software reuse [5], and requirements traceability analysis [6].

Current CIA methods [3, 7–9] based on information retrieval build on the hypothesis that identifiers, comments, and string literals in the source code contain semantic information associated with the software's functions. In typical use, a developer submits a query that describes the change request in natural language. Using the semantic similarities between the query and source code units calculated by the retrieval algorithm, the algorithm identifies the source code units that implement the change request. The calculation requires the transformation of both the source code and the query into numeric vector representations. This process is called indexing. Indexing methods fall into two categories: those based on the bag-of-words model [10], and those based on the word embedding model [11, 12]. The bag-of-words model is easy to perform and completely preserves each word's occurrence in the source code. However, this model builds upon the premise of exchangeability and ignores contextual information of the words. Other researchers [8, 13] argue that source code contains contextual information and prove it by experiment. The word embedding model records contextual information effectively but requires the setting of many parameters that greatly influence the model's performance [8]. In fact, setting

the parameters correctly requires skill and experience of the developers. The word embedding model is quite sensitive to context, but unlike traditional text data, source code grammar is not always strict. Thus, the word embedding model does not always describe source code accurately in the vector space. Existing CIA methods use a single indexing method, either bag-of-words or word embedding with their respective strengths and weaknesses, and do not accurately describe the similarity between source code units and query.

To solve this problem, we propose a structure-driven method for information retrieval-based change impact analysis (SDM-CIA). SDM-CIA is a textual CIA method based on information retrieval technology. SDM-CIA integrates the bag-of-words model and word embedding model based on the software's structure.

The main contributions of this paper are as follows:

(1) We propose the SDM-CIA, which integrates two different indexing models in the IR-based change impact analysis process. SDM-CIA's integration process relies upon the degree of cohesion and coupling of the source code's structure. By using the complementary advantages of both indexing methods, it achieves better performance than existing algorithms that use a single method.

(2) We verify by experiment that the structure information of software source code can help estimate the performance of a CIA method.

Our paper has the following structure. Section 2 provides background information on existing methods and related works. Section 3 describes our proposed SDM-CIA method. Section 4 describes our evaluation and results from using SDM-CIA against five open-source packages. In Section 5, we discuss the applicability of SDM-CIA, present the conclusions of our work, and provide directions for further study.

2. Background

Existing CIA technologies fall into four categories [2]: static [14, 15], dynamic [16, 17], textual [18, 19], and hybrid [20, 21].

Static methods analyze structural information, such as control or data flow dependencies, to point programmers to potentially relevant code [22].

Dynamic methods examine a software system's execution, and they are often needed to record the execution traces of software [2].

Textual methods make use of pattern matching [23], information retrieval (IR) [7, 8], or natural language processing (NLP) [19]. Pattern matching is the act of checking a given sequence of source code for the presence of the constituents of some pattern, usually involving a textual search of source code [2]. IR techniques are essentially statistical methods [2]; they need users to submit a query that describes the change request by text. IR techniques transform the source code texts and query text to vectors, then calculate the similarities between source code texts and query (change request) in vector space. Users can decide which source code is really related to the query based on the similarities. NLP approaches can also exploit a query, but they analyze the parts of speech of the words used in source code [2].

Some researchers [20, 21] combine two or more types of methods because this allows them to realize better CIA performance than either technique alone; these kinds of methods are called hybrid methods. For instance, we can use static and textual methods to help determine which execution traces are useful when using dynamic methods [21, 24].

The textual method is the dominant method because of its usability and low overhead [25]. In recent years, developments in machine learning and deep learning have advanced research into information retrieval. With those developments, information retrieval-based methods have become the focus of current research into CIA technology [7].

2.1. Information Retrieval-Based Change Impact Analysis. As shown in Figure 1, the process of CIA using information retrieval consists of three general steps: preprocessing, indexing, and calculating similarity.

Step 1: Preprocessing. This step includes creation of a document for each source code unit (method) in the source code followed by the extraction of key words from each unit to the corresponding document, with the key words including identifiers, comments, and string literals. Abebe et al. [26] defined an identifier as the name of a class, attribute, method, or parameter. Comments generally are used either to map requirements to code or to describe the code [27]. Moreover, copyright notices in the comments are not included among the key words. The algorithm then performs word splitting, stemming, and stop-word removal on all the documents. Word splitting divides special or combined forms of words into individual words (e.g., "openFile" splits into "open" and "file"). Stemming identifies different forms of common cognate words, matching them with the same key words (e.g., "inserting" and "inserted" are both forms of "insert"). Stop-word removal strips meaningless words from the corpus. Meaningless words contain no semantic knowledge, such as "to," "which," "that," and so on. After extraction, word splitting, stemming, and stop-word removal, the collection of documents formed the corpus.

Suppose we have software consisting of two classes, with each class consisting of two methods as shown in Figure 2. In this example, we have four software units (methods). When performing the extraction, we create a document for each method in the software and extract the identifiers, comments, and string literals from each method to the corresponding document. After extraction, we can obtain four documents which are shown in Figure 3(a). Words in the Doc_1 are extracted from method "OpenFile(FileName)," Doc_2 are from "CloseFile()," Doc_3 are from "DivisionOperation()," and Doc_4 are from "Output()." Then, we perform word splitting, stemming, and stop-word removal. In this example's word splitting, "OpenFile" splits into "open" and

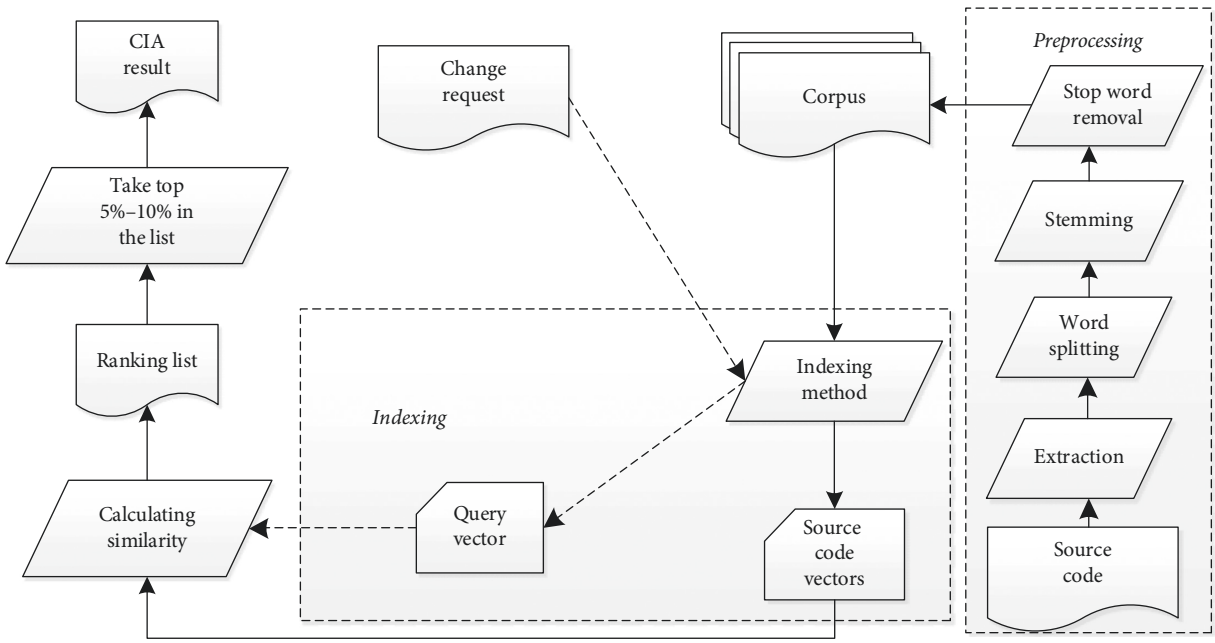

Figure 1: Basic flow of existing information retrieval-based change impact analysis methods.

```
public class FileOperation{
    InputStream in = null;
    public OpenFile(FileName){
        in = new FileInputStream(FileName);
        //Open file which is named FileName
    }
    public CloseFile(){
        in.close();
        //Close the file
    }
}
```

```
public class DivisionOperation(){
    float Divisor = 2.0;
    float Dividend = 2.0;
    float Result = 0.0;
    public Division(){
        Result = Dividend/ Divisor;
    }
     public Output(){
        System.out.println(Result);
        //print the result
    }
}
```

Figure 2: Example of software source code.

"file," "CloseFile" splits in to "close" and "file," and so on. The stemming step, "named," is stemmed to "name," and "division" and "divisor" are both stemmed to "divi." The words "which," "is," and "the" are stop words, so they are removed from the documents in the stop-word removal step. Finally, we can obtain the corpus, which consists of four documents as shown in Figure 3(b).

Step 2: Indexing. Indexing transforms the corpus into numeric vectors, one per document. These vectors are called source code vectors. When a developer submits a query describing a change request, that query is also transformed into a vector.

The bag-of-words model can transform the query into a query vector directly, but the word embedding model Doc2vec is based on a multilayer neural network, and it divides the word and document vector layers. There are two ways to transform the query into query vector for doc2vec:

(1) Treat the training query as a document, transforming it in the document vector layer.

(2) Train each word in the query, obtaining many word vectors in the word vector layer. Then calculate the mean vector of the word vectors as the query vector.

Corley et al. [8] evaluated the performance of both methods and concluded that the second approach achieves

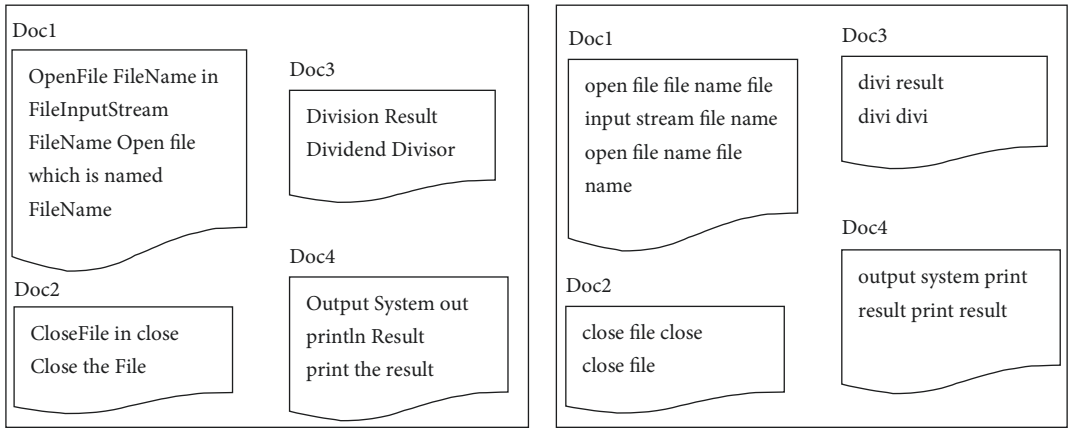

Figure 3: Preprocessing example. (a) Documents after extraction. (b) Corpus after preprocessing.

better performance. Thus, we chose the second way to index the query, calculating it as follows.

Definition 1: Query Vector in doc2vec. For a query Q consisting of multiple words, $Q = \{w_1, w_2, \ldots, w_m\}$, where wi is the ith word in the query. Using doc2vec to train the words in Q, we obtain many word vectors $\{\mathbf{wv}_1, \mathbf{wv}_2, \ldots, \mathbf{wv}_m\}$. We calculate the query vector as

$$\mathbf{q}_{\mathbf{d2v}} = \frac{\sum_{i=0}^{m} \mathbf{wv}_i}{m}. \quad (1)$$

Suppose we want to change the software's open file function to the software shown in Figure 2, we can submit a query as "open file." We index its corpus and the query by using the most basic bag-of-words model. Then, we can obtain a matrix as shown in Table 1. Each row in the matrix is a vector that presents the corresponding documents and the query. Each column represents a word. The element in the matrix represents a word's occurrence number in the corresponding document. For instance, the word "open" occurs twice in the Doc_1, so the value of the first element in the vector of Doc_1 is 2.

Step 3: Calculating Similarity. Next, the algorithm calculates the similarity between the query vector and each source code vector and then ranks the similarities accordingly. The source code vector with the greatest similarity to the query vector is the one most likely related to the change request. Calculating the cosine distance between each document vector and query vector, we can get the set of similarities. Using the $Sim(\text{Doc}_i, \text{query})$ represents the similarity between Doc_i vector and query vector, and we can then obtain the similarities as follows: $Sim(\text{Doc}_1, \text{query}) = 0.74$, $Sim(\text{Doc}_2, \text{query}) = 0.39$, $Sim(\text{Doc}_3, \text{query}) = 0.0$, and $Sim(\text{Doc}_4, \text{query}) = 0.0$. The Doc_1 vector has the greatest similarity to the query vector, and the document Doc_1 is extracted from the method "OpenFile(FileName)." Thus, we can conclude that the "OpenFile(FileName)" is the most likely method that relates to our change request "open file."

2.2. Related Works. Marcus et al. [9] proposed the earliest information retrieval-based change impact analysis (CIA) method in 2004. For the first time, they successfully used information retrieval technology to do change impact analysis of software source code. They defined the most basic process of information retrieval-based change impact analysis as three steps, which include preprocess, indexing, and calculating similarity. Since then, many researchers have conducted further studies, but their works are still built on the three-step process defined by Marcus et al. This paper's work is also built on the basic process defined by Marcus et al., but we found a better way to index and calculate the similarity; thus, we can achieve better change impact analysis performance.

Because indexing is the most important step in the CIA process, researchers have spent the most time on this step. Marcus et al. [9] used Latent Semantic Indexing (LSI). The lightweight LSI identifies and eliminates the influence of synonyms and reduces the dimension of the vectors at the same time. Biggers et al. [7] used Latent Dirichlet Allocation (LDA) to index source code, achieving better performance on some benchmarks. However, both LSI and LDA use the bag-of-words model, which assumes that keywords have no context information [10]. Thus, exchanging the locations of keywords does not lead to detection of the source code's change in function. In 2015, Corley et al. [8] introduced doc2vec, based on the word embedding model [12], in their CIA research and achieved better performance than the LDA method in their tests. The word embedding model [11, 12] not only compresses the dimensions of the source code vectors but also records the contextual relationships between key words. It indexes the corpus based on the co-occurrence relationships of key words. The most famous example of its performance is that it can conclude that "king-queen" is similar to "man-woman." Because the word embedding method is based on a deep learning model, it requires setting many parameters such as vector dimension, number of training epochs, number of training windows, learning rate, and so on. All these parameters require the user to have background knowledge about deep learning. At the same

Table 1: Matrix after indexing example corpus.

	Open	File	Name	Input	Stream	Close	Divi	Result	Output	System	Print
Doc_1 vector	2	6	4	1	1	0	0	0	0	0	0
Doc_2 vector	0	2	0	0	0	3	0	0	0	0	0
Doc_3 vector	0	0	0	0	0	0	3	1	0	0	0
Doc_4 vector	0	0	0	0	0	0	0	2	1	1	2
Query vector	1	1	0	0	0	0	0	0	0	0	0

time, the syntax and format of source code differ greatly from natural language texts. In a source code corpus, co-occurrence relationships between key words do not necessarily mean the words are similar, which is different from words in natural language. Thus, the word embedding model has limitations that arise from describing similarities between source code based solely on the co-occurrence relationships of key words.

Marcus and Biggers' studies [7, 9] are based on a single bag-of-words indexing model, and Corley's research [8] is based on a single word embedding indexing model. Each uses a single indexing method, but different indexing methods have specific advantages and disadvantages in performance. These differences mean that current methods do not accurately describe differences between source code units, which limits the performance of software change impact analysis. Thus, our work in this paper is that we combined two different kinds of indexing methods in SDM-CIA; this combination can eliminate some limitations that are brought about by a single indexing method. SDM-CIA can then achieve better performance than these CIA methods with the same input and preprocessing.

2.3. Bag-of-Words and Word Embedding Models

2.3.1. Bag-of-Words Model. The bag-of-words model is a simplifying indexing method using information retrieval (IR) [10]. In this model, text (such as a sentence or a document) is represented as an unordered collection of words, disregarding grammar, and even word order. The bag-of-words model generates long, sparse vectors like [1, 0, 2, 0, . . ., 0, 0, 1], where each dimension represents the frequency of a word in the corpus. The vectors record only the frequency of key words without any context information.

For example, if we want to index two sentences, we proceed as follows:

(a) Clicking the "open" button can open a file.

(b) Clicking the "close" button can close the file.

There are eight total distinct words in these two sentences, and we can build a dictionary for these two sentences:

{"clicking": 1, "the": 2, "open": 3, "button": 4, "can": 5, "a": 6, "file": 7, "close": 8}.

Using the indexes of the dictionary, each sentence is represented by an eight-dimensional vector:

(a) [1, 1, 2, 1, 1, 1, 1, 0],

(b) [1, 2, 0, 1, 1, 0, 1, 2],

where each dimension represents the frequency of a word in the sentences. This vector representation does not preserve the order of the words in the original sentences.

There are three typical bag-of-words models in IR technologies: Term Frequency-Inverse Document Frequency (TF-IDF), Latent Semantic Indexing (LSI) [28], and Latent Dirichlet Allocation (LDA) [29]. In SDM-CIA, we chose TF-IDF as the bag-of-words model. TF-IDF is the most basic bag-of-words method. TF-IDF performs no dimensional reduction, recording the maximum number of key word occurrences when indexing. Other bag-of-words models, such as LSI and LDA, are all built on the basis of TF-IDF, and they all reduce the dimension of vectors in a very low range. The dimensional reduction may lead to some individual words' information being lost. Therefore, we chose TF-IDF in SDM-CIA.

TF-IDF is an indexing model that is intended to reflect how important a word is to a document in a collection or corpus. It is a term-weighting scheme, and it can assign a weight to each element in the bag-of-word matrix. It is the product of two statistics, term frequency and inverse document frequency. Term frequency is the number of times a given term appears in the document. In general, it is divided by the length of the document:

$$tf_{i,j} = \frac{n_{i,j}}{|d_j|}, \tag{2}$$

where $n_{i,j}$ is the number of occurrences of the term i in the document j, and $|d_j|$ is the total number of terms in the document j. Inverse document frequency indicates the general importance of the term i in the collection:

$$idf_i = \log \frac{|D|}{|\{d : i \in d\}|}, \tag{3}$$

where $|D|$ is the number of documents in the collection, while the denominator is the number of documents that contain the term i. In this way, we can find the TF-IDF weight as follows:

$$(tf - idf)_{i,j} = tf_{i,j} * idf_i. \tag{4}$$

At last, every element in the bag-of-word matrix will be multiplied by corresponding $(tf - idf)_{i,j}$.

2.3.2. Word Embedding Model. The word embedding model generates short real-value vectors like [0.723, 0.051, . . ., 0.231 0.321, 0.4231, 0.448], which record the co-occurrence relationships between words mapped in a low-dimensional space to describe the corpus [12]. The transformation

preserves context at the expense of information about many independent key words, which is lost.

Word embedding models include, e.g., word2vec [11], doc2vc [12], and autoencoder [30]. Doc2vec is a new word embedding method using deep learning technology [8]. It is currently the most popular indexing model in natural language processing and text information retrieval research. In addition, researchers [8] have already verified the effectiveness of doc2vec in IR-based CIA. Therefore, we chose doc2vec in SDM-CIA.

doc2vec is a word embedding model based on a multistage neural network, consisting of several hidden layers in addition to single input and output layers. The input layer consists of an ordered sequence of identifiers extracted from the code. The multiple hidden layers serve to capture the context for each encountered term, representing the complex patterns of term contexts occurring in the corpus. The output layer consists of a vector for each term, which has been shown to carry semantic meaning [8]. doc2vec can transform a paragraph or a document to vectors, and it is based on the word2vec model, which can transform words into vectors [12]. We can use doc2vec to transform the documents of the corpus to vectors. The transformation preserves context, but it will lose the information of many individual words in the corpus.

3. Proposed Approach

In this section, we present our proposed structure-driven method for information retrieval-based change impact analysis (SDM-CIA). As shown in Figure 4, the black boxes highlight the main differences between our SDM-CIA and traditional IR-based CIA methods. Preprocessing is unchanged. The differences are in the indexing and calculation of similarity steps.

IR-based CIA methods rely on quantifying the semantic similarity between source code units and a query. The indexing process transforms the source code and queries into vectors, and the distance in the vector space determines the similarity. As mentioned previously, individual indexing methods have strengths and weaknesses. Our goal is to integrate both methods to achieve better performance.

3.1. Module Structure. Some researchers [31, 32] have proven that the linear combination of distances in multiple vector spaces is still a distance. For a given source code vector $\mathbf{m}_i$ and query vector $\mathbf{q}$, the distance (similarity) can be described as the linear combination of distances in different spaces and calculated as follows:

$$d(\mathbf{m}_i, \mathbf{q}) = \sum_{k=1}^{p} \omega_k \cdot d_k(\mathbf{m}_i, \mathbf{q}). \tag{5}$$

In this formula, $d_k(\mathbf{m}_i, \mathbf{q})$ is the distance between $\mathbf{m}_i$ and $\mathbf{q}$ in the (different) vector space $\boldsymbol{k}$, and ω_k is the corresponding weight. The most important problem is finding the appropriate weight ω_k, which is difficult to calculate directly. Consequently, we turn the problem to calculating the optimal structure of the vector space, which is what we mean by the term structure-driven.

We aim for high cohesion and low coupling as our most basic structural principle when we are organizing the software source code modules. Source code within the same module should have high cohesion, but source code in different modules should have low coupling. If software source code is strictly organized by the principle of high cohesion and low coupling, then we can consider that it to have good structure. Well-structured software should have the following two characteristics. First, the source code in the same unit implements the same single function, and this source code should have common keywords in identifiers, comments, and string literals. Second, source code in different units implements different functions, and this source code should have different keywords in identifiers, comments, and string literals. Thus, the best similarity (distance) calculation method should accord with this structural principle. In the IR-based CIA method, similarity (distance) is calculated in vector space, and all the source code units are transformed to vectors in vector space. In this way, the distribution of source code vectors in vector space should also reflect the source code units' structure. When we are evaluating the distribution of source code vectors, we can use the Internal Distance and External Distance.

In order to clearly introduce the SDM-CIA method, we need to introduce several definitions, as follows:

Definition 2: Module. In software with source code organized by good structure, a module is a set of source code units performing related functions. A software system consists of many modules. For example, in an object-oriented software system, each class is a module consisting of multiple methods, each of which is a source code unit.

Definition 3: Internal Distance. Given a software system with source code units distributed among different modules, after indexing all the source code units in the indexing space, we can calculate the internal distance "interDis" between modules using the following formulas:

$$\text{interDis} = \sum_{j=1}^{m} ModuleDis\left(module_j\right),$$

$$ModuleDis\left(model_j\right) = \begin{cases} \dfrac{\sum_{l<h}^{n} d\left(\mathbf{code}_{j,l}, \mathbf{code}_{j,h}\right)}{C_n^2}, & n > 2, \\ d\left(\mathbf{code}_{j,1}, \mathbf{code}_{j,2}\right), & n = 2, \\ 1, & n = 1, \end{cases} \tag{6}$$

where m is the total number of modules (classes) in the software; $ModuleDis(module_j)$ calculates the mean distance between the source code units in the $module_j$; $d(\mathbf{code}_{j,l}, \mathbf{code}_{j,h})$ is the distance between the lth source code unit and hth source code unit in module j; and $h \le n$; and C_n^2 is the combination formula. A combination is an unordered

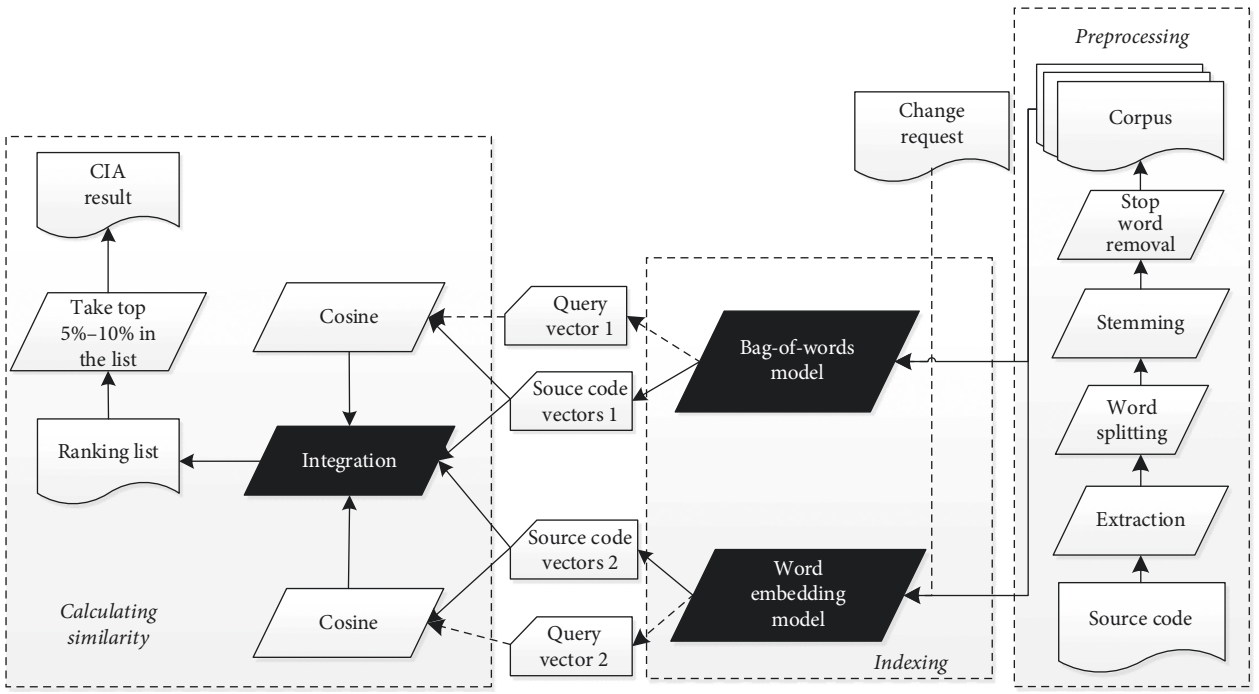

Figure 4: Structure-driven method for information retrieval-based software change impact analysis.

collection of distinct elements, usually of a prescribed size and taken from a given set, with $C_n^2 = n!/[2!(n-2)!]$. C_n^2 calculates the total number of distances between each pair of the n total source code units in the module j. Generally, if a module consists of more than two source code units, then we have $n > 2$. However, if there are some modules that consist of two or one source code units, namely $n = 2$, we use $d(\mathbf{code}_{j,1}, \mathbf{code}_{j,2})$ to replace $\sum_{l<h}^{n} d(\mathbf{code}_{j,l}, \mathbf{code}_{j,h})/C_n^2$ in the formula. If $n = 1$, we use 1 to replace $\sum_{l<h}^{n} d(\mathbf{code}_{j,l}, \mathbf{code}_{j,h})/C_n^2$ in the formula.

Definition 4: External Distance. For a system consisting of many source code units belonging to different modules, after indexing all the source code units in the indexing space, we calculate the external distance between modules as

$$\text{exterDis} = \sum_{j<i}^{m} d\left(\mathbf{cen}_i, \mathbf{cen}_j\right), \tag{7}$$

where m is the total number of modules in the software; $\mathbf{cen}_i$ is the center vector of module i calculated as $\mathbf{cen}_i = (1/n)\sum_{j=1}^{n} \mathbf{code}_{i,j}$; $\mathbf{code}_{i,j}$ represents the jth source code unit in the module i; and n is the number of source code units in the module i.

Figure 5 is a graphical representation of Definitions 1, 2, 3, and 4, using a software system with two modules. In the case of an object-oriented system, each module is a class, with dots representing the methods in module 1 and the cross stars representing the methods in module 2. The dotted line within the module 2 boundary represents the distance between two methods of module 2. We calculate the distance between each pair of methods in the module 2 and then the mean of all the distances. The mean of distances is the internal distance of module 2. The square and triangle represent the center vectors of module 1 and module 2, respectively. The distance between the triangle and the square is the external distance of these two modules.

3.2. Structure-Driven Method for Information Retrieval-Based Software Change Impact Analysis. Consider a software system consisting of two classes (modules), with each class consisting of many methods (source code units). We create a document for each source code unit in the software and then extract the key words from each source code unit to the corresponding document. These documents make up the corpus of this software. Then we apply two different models to index the corpus, calling them indexing model 1 and indexing model 2. The resulting two-dimensional source code vectors are distributed as shown in Figures 6(a) and 6(b) and denoted as indexing space 1 and indexing space 2. In this example, the distribution in indexing space 1 is more regular than in indexing space 2. The source code vectors in indexing space 1 closely align with the software's structure, with source code units for each module cleanly separated. In contrast, the distribution of source code vectors in indexing space 2 is chaotic, with a large overlap between the source code vectors of the two modules. In this situation, we see that the indexing space 1 better describes the software's source code than indexing space 2, which indicates that indexing model 1 is more suitable for this software's source code. Thus, when considering the distances in these two indexing spaces, we should give more weight ω_1 to $d_1(\mathbf{m}_i, \mathbf{q})$ in space 1 and less weight ω_2 to $d_2(\mathbf{m}_i, \mathbf{q})$ in space 2. In this manner, we can use the internal distance and external distance of modules to calculate the weight ω_k in the linear combination formula.

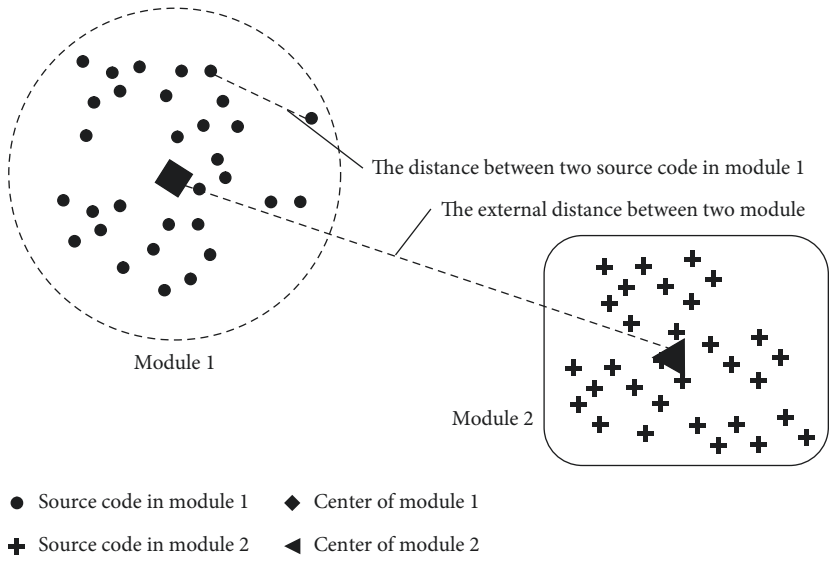

Figure 5: Graphical representation of module distances.

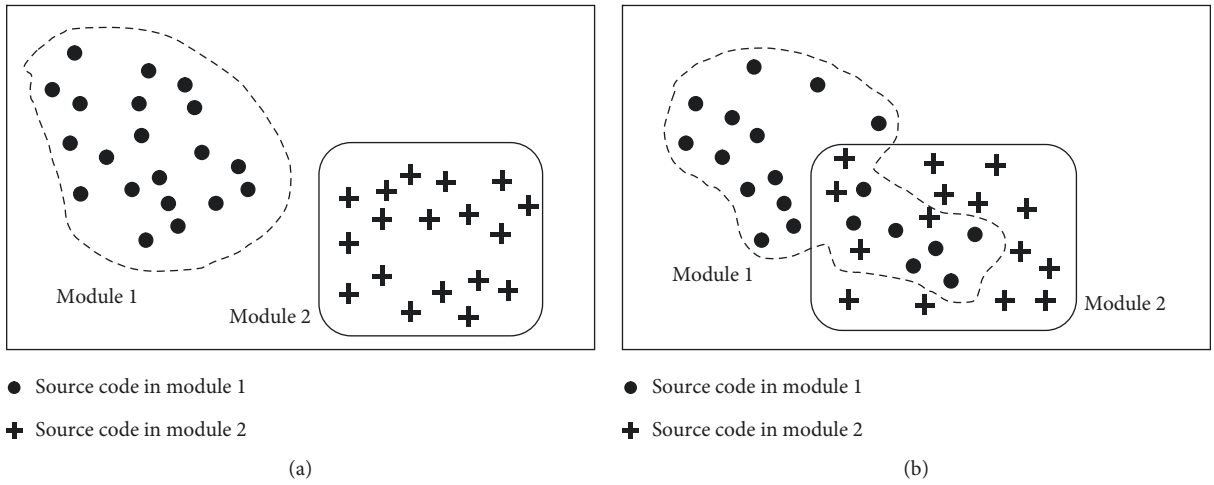

Figure 6: Source code distribution in different indexing spaces. (a) Source code distribution in indexing space 1. (b) Source code distribution in indexing space 2.

Thus, we propose a structure-driven method for IR-based CIA. We first calculate the linear combination weight based on the degree of consistency between the vectors distribution in the indexing spaces and the software source code's structure. Then, we calculate the similarity between source code units and a query based on the linear combination formula. We calculate the degree of consistency using the internal distance and external distance of modules in the software source code.

We measure the software's cohesion and coupling degree using the internal and external distances according to the formula

$$\text{Dis} = \frac{\text{exterDis}}{\text{interDis}}, \tag{8}$$

where exterDis and interDis are the external and internal distances, respectively. After indexing the source code corpus, we calculate the source code vectors' Dis. Larger values of Dis correspond to better descriptions of the corpus. Thus, if the vectors generated by indexing model 1 have a larger Dis than those generated by indexing model 2, we consider indexing model 1 preferable to indexing model 2. Thus, when integrating the similarities (distances) calculated by the two indexing models, we make ω_1 larger than ω_2. Because the linear combination of distances is a relative weighting, the weighting ω_k needs to satisfy the constraint

$$\sum_{k=1}^{p} \omega_k = 1. \tag{9}$$

Based on the preceding ideas, we propose the SDM-CIA method as follows. After preprocessing, we use the bag-of-words model and the word embedding model to index the corpus and query separately. After that, we calculate the similarities between the query vector and source code vectors in the two indexing spaces. We use Sim_1 and Sim_2 to denote

the similarities in the bag-of-words and word embedding indexing spaces, respectively. Thus, $Sim_1 = \{s_1^1, s_2^1, \ldots, s_n^1\}$ and $Sim_2 = \{s_1^2, s_2^2, \ldots, s_n^1\}$, where S_p^1 represents the similarity between query vector and the p_{th} source code vector in the bag-of-words indexing space, and S_p^2 represents the similarity between query vector and the p_{th} source code vector in the word embedding indexing space. We use D_1 and D_2 to represent the source code vectors in the bag-of-words and word embedding indexing spaces, respectively. All source code units are divided into k modules by the structure of the software itself, so $D_1 = \{module_1^1, module_2^1, \ldots, module_k^1\}$ and $D_2 = \{module_1^2, module_2^2, \ldots, module_k^2\}$, where $module_p^1$ represents all the source code vectors for the p_{th} module in the bag-of-words indexing space, and $module_p^2$ represents all the source code vectors for the p_{th} module in the word embedding indexing space. Algorithm 1 presents the similarity calculation algorithm of the SDM-CIA method.

Step 1. Lines 1 and 2 of the algorithm calculate the internal distances in the two indexing spaces separately, and this step uses the Internal Distance formula in Definition 3.

Step 2. Lines 3 and 4 calculate the external distances likewise. This step uses the External Distance formula in Definition 4; and $cen_{\mathbf{i}}^1$ is the center vector of $module_i^1$, and $cen_{\mathbf{i}}^2$ is the center vector of $module_i^2$.

Step 3. Line 5 calculates the linear combination weight ω_k based on the internal distances and external distances.

Step 4. Lines 6, calculates the final similarities (distance) based on the linear combination formula.

Finally, we use the similarities to rank the source code units. The source code unit with the maximum similarity has the greatest possibility mapping to the change request.

4. Case Study

We designed a case study to evaluate the effectiveness of our approach. In particular, we wanted to obtain answers to the following research questions (RQs).

(1) RQ1: Does our approach achieve better performance than the existing IR-based CIA methods based on a single indexing model?

(2) RQ2: Can we predict the performance of the CIA method based on the internal and external distances of the vectors in the indexing space?

To answer RQ1, we need to compare our approach's performance with existing methods. As introduced in Section 2.2, three typical indexing methods have been used in CIA studies. The representative CIA studies corresponding to these three indexing methods are the word embedding model doc2vec published by Corley et al. [8], the bag-of-words LDA model published by Biggers et al. [7], and the bag-of-words LSI model published by Marcus et al. [9]. These three studies were all the first time the corresponding indexing method was used in IR-based CIA, and they all used the same preprocessing technologies. Thus, we compared our approach with these three IR-based CIA methods based on three single indexing models.

4.1. Experimental Systems. To guarantee the objectivity of the case study, we used the software maintenance tasks benchmark published by Poshyvanyk [2] for testing. Table 2 presents details of the benchmark, which consists of five software products and their related changes. Each software package in the benchmark may have multiple changes, and each change corresponds to several software units. For example, we were able to modify the file function and edit function in the same version update of the jEdit. There are two changes in this updated version. Modifying the file function may require modifying two or three methods, and modifying the edit function may require modifying even more methods. All the changes exist in each product's official issue tracking system. Each change has an ID number and two components:

(1) Description: a natural language description of the change request, which we used as a query

(2) Gold set: a record of the source code units actually related to the change request, which we use to verify the results of our method

The evaluation of our approach requires the source code to be of high quality according to the following requirements:

(1) The source code should have good semantic meaning, i.e., the key words in the source codes are meaningful. Meaningless vocabulary refers to unconventional words with no clear meaning, such as single letters, variables, or method names like "c2," "m2," and so on. We counted the number of meaningless words in each software package's source code. As shown in Table 1, the proportion of meaningless words within the total is 4.56% to 7.43%. Thus, more than ninety percent of the key words in the source code have good semantic meaning. We conclude that the source code in the benchmark is of high quality.

(2) The source code should be organized according to the principle of high cohesion and low coupling. Special teams manage changes to the benchmark products, which ensures the quality of the source code structure. If the structures of the products in the benchmark are of high quality, the answer to RQ2 will be affirmative. Thus, we can indirectly confirm the quality of the products' software structures with the answer to RQ2.

4.2. Performance Metric. We also need a measure to evaluate our method's performance.

Many researchers [2, 21, 33] have used precision and recall as metrics to evaluate CIA performance. These metrics calculate the fraction of relevant elements generated by an approach. They are defined as

Input: Sim_1, Sim_2, D_1, D_2
Output: Final similarities set $\mathrm{Sim}_{\mathrm{int}}$
(1) $\mathrm{interDis}_1 = \sum_{j=1}^{m} ModuleDis(\mathrm{module}_j^1)$
(2) $\mathrm{interDis}_2 = \sum_{j=1}^{m} ModuleDis(\mathrm{module}_j^2)$
(3) $\mathrm{exterDis}_1 = \sum_{\mathbf{j<i}}^{\mathbf{m}} d(\mathbf{cen}_i^1, \mathbf{cen}_j^1)$
(4) $\mathrm{exterDis}_2 = \sum_{\mathbf{j<i}}^{\mathbf{m}} d(\mathbf{cen}_i^2, \mathbf{cen}_j^2)$
(5) $\omega_1 = ((\mathrm{exterDis}_1/\mathrm{interDis}_1)/((\mathrm{exterDis}_1/\mathrm{interDis}_1) + (\mathrm{exterDis}_2/\mathrm{interDis}_2)))$,
$\omega_2 = ((\mathrm{exterDis}_2/\mathrm{interDis}_2)/((\mathrm{exterDis}_1/\mathrm{interDis}_1) + (\mathrm{exterDis}_2/\mathrm{interDis}_2)))$,
(6) $Sim_{\mathrm{int}} = \omega_1 * Sim_1 + \omega_2 * Sim_2$
(7) Return Sim_{int}

Algorithm 1: Similarity calculation algorithm of SDM-CIA method.

Table 2: Benchmark.

Software	Information						
	Version	Methods	Modules	Changes	Corpus	Number of meaningless words	Meaningless words (%)
JabRef	2.6	4604	532	39	286271	17771	6.21
jEdit	4.3	6413	1031	150	330705	21599	6.53
muCommander	0.85	8187	975	92	400120	18242	4.56
ArgoUML	0.22	11000	729	91	521463	28135	5.40
Eclipse	3.0	121216	6641	45	7602447	565124	7.43

$$\mathrm{precision} = \frac{D_c \cap D_r}{D_r} * 100\%, \qquad \mathrm{recall} = \frac{D_c \cap D_r}{D_c} * 100\%, \tag{10}$$

where D_c is the set of all correct source code documents related to the change request, and D_r is the set of source code documents retrieved by the CIA method. When calculating the precision and recall metrics, we use the top 5% of the source code documents from the result ranking list as the CIA results. However, since precision and recall are reciprocal, they are unable to reflect the comprehensive performance of a CIA technique [2]. Thus, we also employ the F-score to measure the performance. F-score is the harmonic average of the precision and recall, where an F-score reaches its best value at 1 (perfect precision and recall) and worst at 0. F-score is defined as

$$\mathrm{F-score} = 2 * \frac{\mathrm{precesion} * \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}} * 100\%. \tag{11}$$

At the same time, CIA is essentially information retrieval, so we use an information retrieval metric to evaluate performance. Similar to related studies [8, 33, 34], we use the rank of the first relevant document as the measure of effectiveness. The rank represents the number of source code entities a developer would have to view before reaching a relevant one. A MRR reaches its best value at 1 and its worst at 0. The Mean Reciprocal Rank (MRR) is defined as

$$\mathrm{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{e_i}, \tag{12}$$

where Q is the set of queries and e_i is the effectiveness measure for a given query Q_i. Larger MRR values indicate better performance.

For the sake of generality, we use all four metrics to evaluate the performance of our approach in the case study.

4.3. Evaluation Process

(1) *Preprocessing*. The preprocessing was the same as the existing IR-based CIA methods, including extraction, word splitting, word stemming, and stop-word removal.

(2) *Indexing*. We perform TF-IDF and doc2vec to indexing the corpus, respectively. We generated the source code vectors in the two indexing spaces. D_{tfidf} represents the vectors in the TF-IDF indexing space, and D_{d2v} represents the vectors in the doc2vec indexing space. Then, we need to perform TF-IDF and doc2vec indexing of the query as well. It is important to note that when we were performing the doc2vec to index the query, we used the method in Definition 1. After indexing the query with both TF-IDF and doc2vec, we have two query vectors $\mathbf{q_{tfidf}}$ and $\mathbf{q_{d2v}}$.

(3) *Similarity Calculation*. Calculating the cosine distance between $\mathbf{q_{tfidf}}$ and each vector in D_{tfidf}, we obtain the similarities set Sim_{tfidf}. Calculating the cosine distance between $\mathbf{q_{d2v}}$ and each vector in the D_{d2v}, we obtain the similarities set Sim_{d2v}. We calculate and rank the final similarities as given in Algorithm 1. We used the gold sets to verify the performance and recorded the results.

4.4. Experimental Results. Figure 7 presents the comprehensive results for SDM-CIA and the baseline techniques with different values of precision and recall metrics. Figure 8 presents the results using the F-score metrics. Figure 9 presents the results using the MRR metrics. We discuss these results further below.

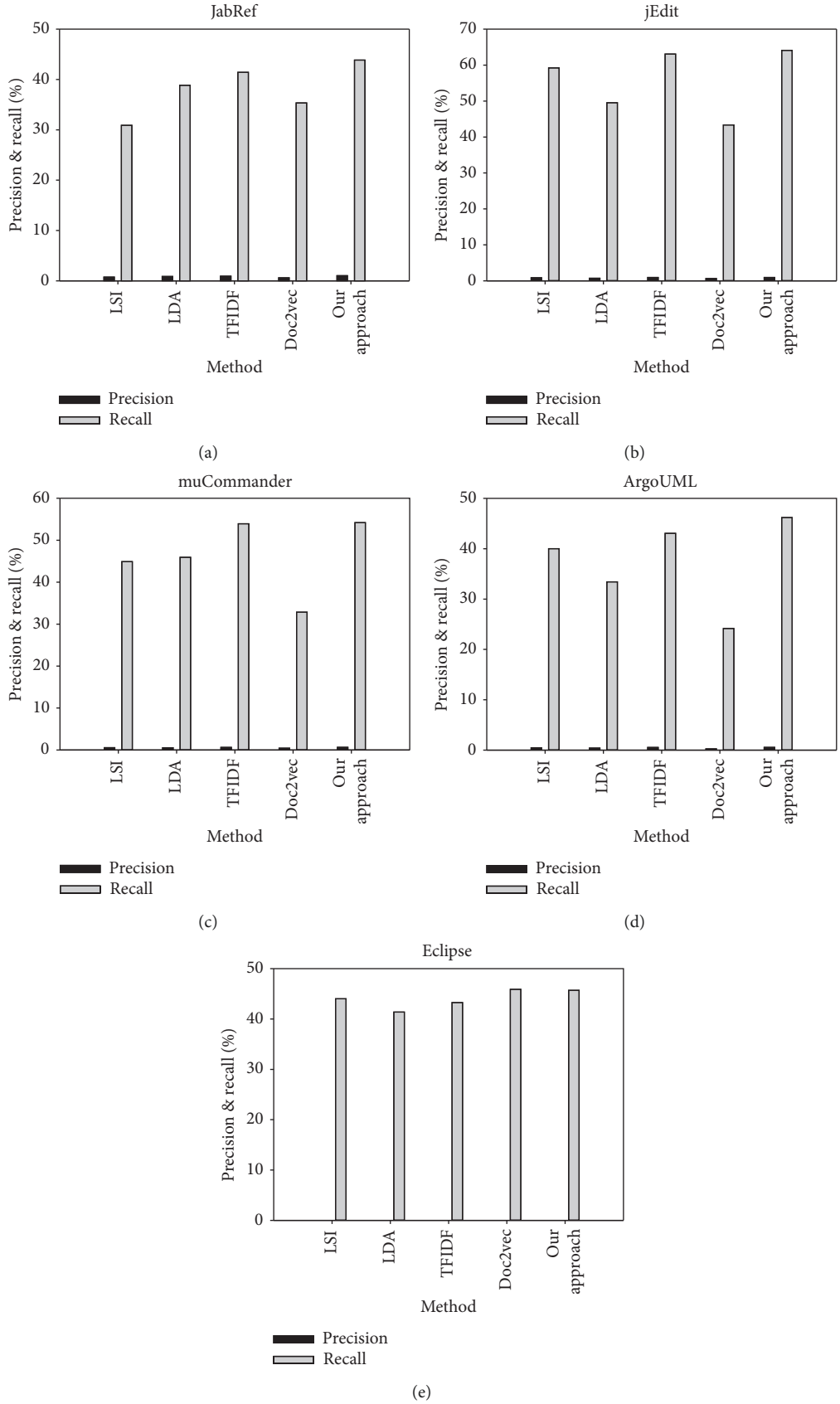

Figure 7: Recall and precision performance with different methods. (a) Recall and precision on JabRef. (b) Recall and precision on jEdit. (c) Recall and precision muCommander. (d) Recall and precision on ArgoUML. (e) Recall and precision on Eclipse.

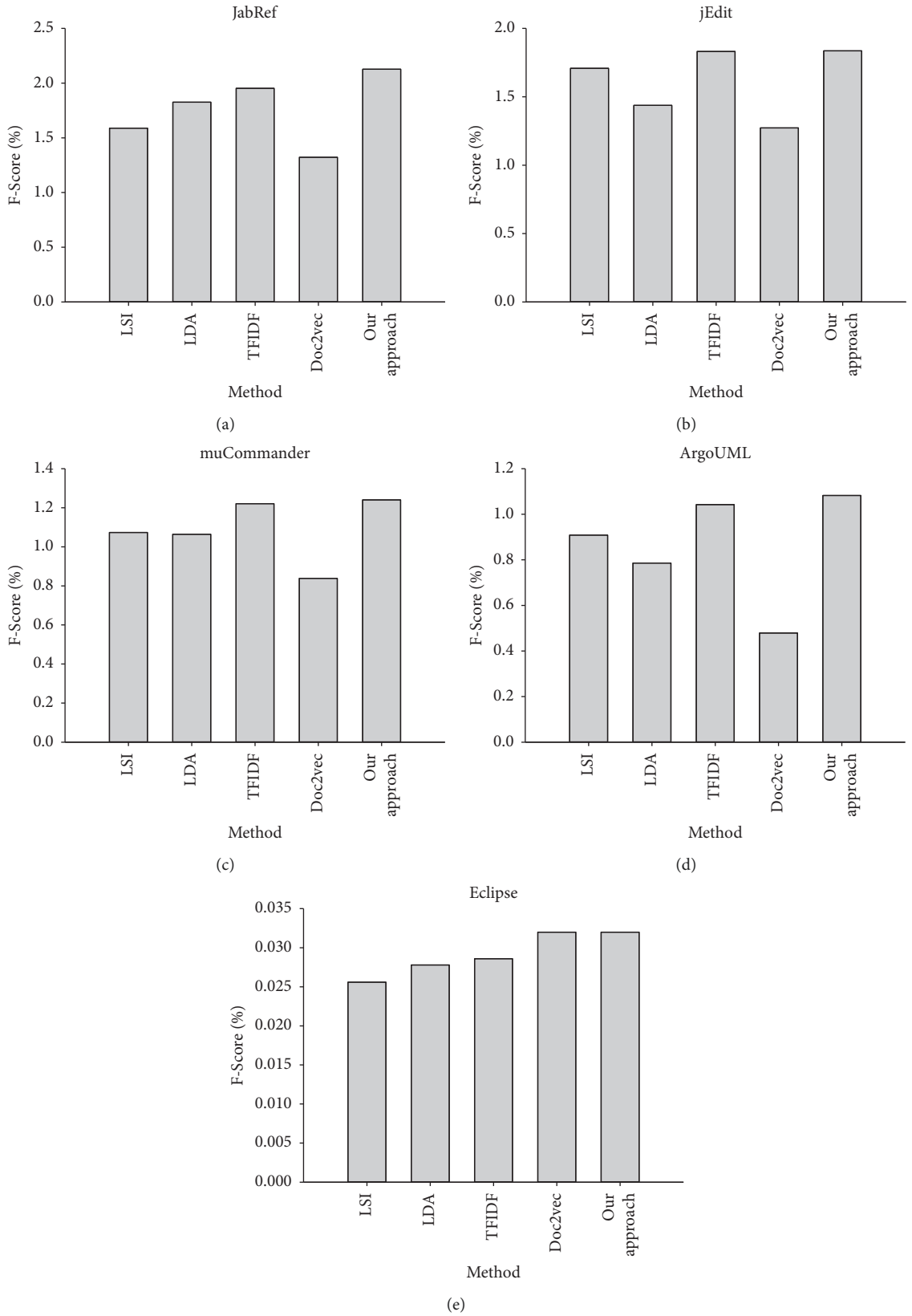

Figure 8: F-score performance with different methods. (a) F-score on JabRef. (b) F-score on jEdit. (c) F-score on muCommander. (d) F-score on ArgoUML. (e) F-score on Eclipse.

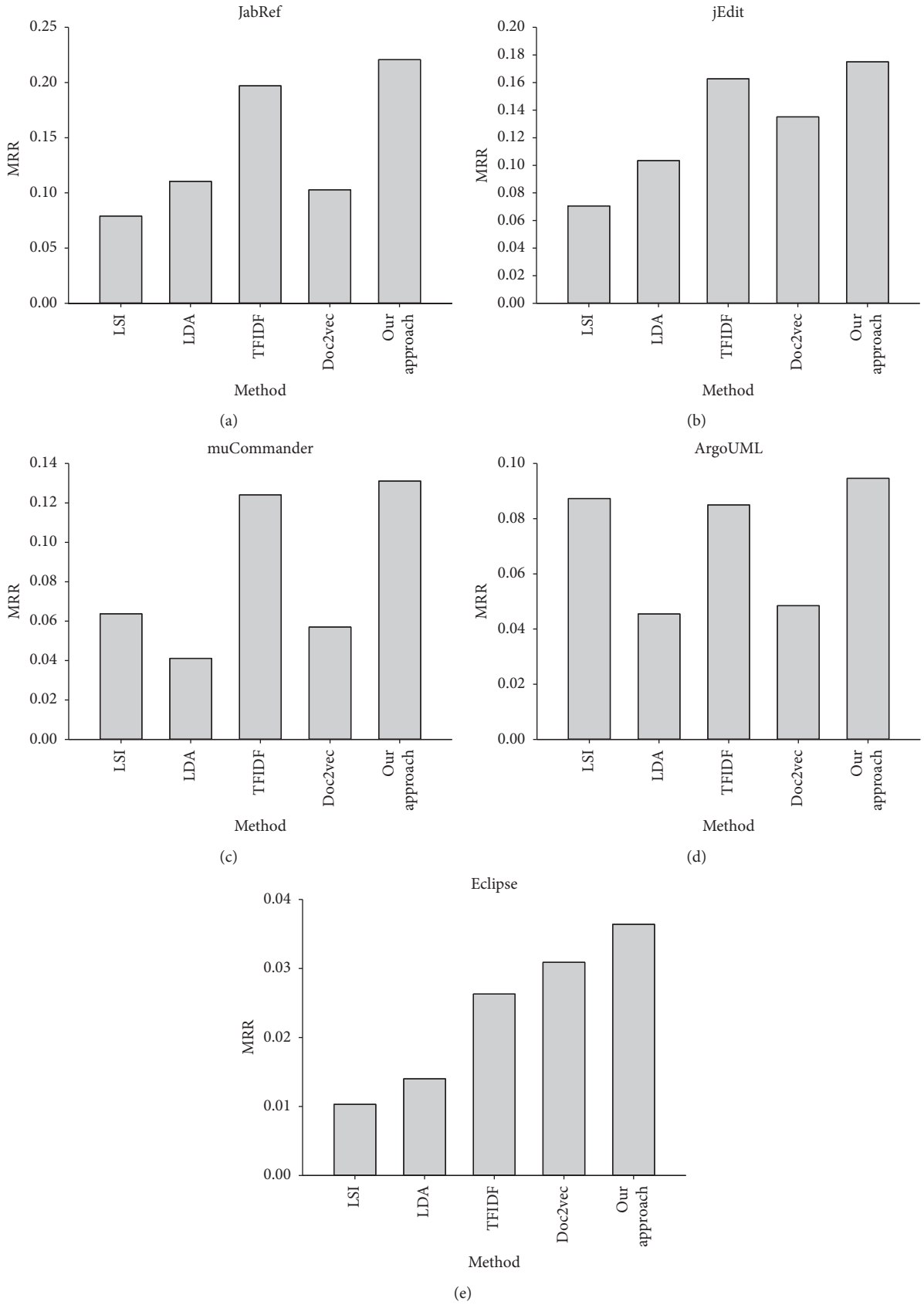

Figure 9: MRR performance with different methods. (a) MRR performance on JabRef. (b) MRR performance on jEdit. (c) MRR performance on muCommander. (d) MRR performance on ArgoUML. (e) MRR performance on Eclipse.

Table 3: CIA performance and weight distribution.

Values	Softwares				
	JabRef	jEdit	muCommander	ArgoUML	Eclipse
TFIDF precision performance (%)	**1.000**	**0.929**	**0.617**	**0.528**	0.014
doc2vec precision performance (%)	0.673	0.646	0.425	0.242	**0.016**
TFIDF recall performance (%)	**43.495**	**63.080**	**53.927**	**43.063**	43.271
doc2vec recall performance (%)	35.409	43.364	32.870	24.167	**45.905**
TFIDF F-score performance (%)	**1.953**	**1.831**	**1.221**	**1.042**	0.285
doc2vec F-score performance (%)	1.323	1.273	0.839	0.479	**0.320**
TFIDF MRR performance	**0.197**	**0.162**	**0.124**	**0.085**	0.026
doc2vec MRR performance	0.102	0.135	0.057	0.048	**0.031**
TFIDF weight	**0.780**	**0.749**	**0.730**	**0.773**	0.458
doc2vec weight	0.220	0.251	0.270	0.227	**0.542**

In Figures 7–9, LSI denotes the method of Marcus et al. [9], LDA denotes the method of Biggers et al. [7], and Doc2vec denotes the method of Corley et al. [8]. We now return to RQ1.

4.4.1. Discussion of RQ1. Figures 7(a)–7(e) show that our approach obtains better precision and recall performance against the JabRef, jEdit, muCommander, and ArgoUML projects, achieving an average 3.65% precision improvement and an average 3.82% recall improvement compared to the least effective approach. In the case of Eclipse, our performance was the same as the method from Corley et al. [16]. This is because when calculating the precision and recall metrics, our algorithm uses only the top 5% of the source code documents from the ranking list as the CIA result [14]. With about 120,000 methods in Eclipse's source code, the top 5% equates to about 6,000 source code units. Because only a few source code units are actually related to each change request, the large number of source code units in the results makes the difference in the precision and recall measurements negligible.

As shown in Figure 8, we find that our approach is always better than approaches using a single indexing method according to the F-score. Figures 8(a)–8(d) show that our method improves F-score by as much as 8.92% (JabRef), 0.24% (jEdit), 1.59% (muCommander), and 3.83% (ArgoUML) when compared to the least effective existing approach in each case. In the case of Eclipse, our performance was the same as the method of Corley et al. [8]. This is because when we were calculating the F-score, we used the precision and recall as the input variables. However, the precision and recall performance of our approach are same as those in Corley's method in the case of Eclipse. Therefore, we obtained the same F-measure performance with Corley's method in the case of Eclipse. Even so, our approach still achieved an average improvement of 3.6% in F-score performance compared to the least effective method using a single indexing model.

As shown in Figure 9, we found that our approach is always better than approaches using a single indexing method according to the Mean Reciprocal Rank (MRR). Figures 9(a)–9(e) show that our method improves by as much as 12.03% (JabRef), 7.56% (jEdit), 5.64% (muCommander), 8.36% (ArgoUML), and 17.8% (eclipse) in MRR when compared to the least effective existing approach in each case. Our approach achieved an average improvement of 10.28% in MRR performance improvement compared to the least effective method using a single indexing model.

SDM-CIA's performance is not very distinctive in the measure of precision, recall, and F-score. The main reason for this is that there are five software products in our benchmark, each consisting of thousands or tens of thousands of source code units. However, for the changes in the benchmark, there are far fewer (1–5) source code units that are truly related to each change. When calculating the precision, recall, and F-score, we needed to use the top 5% of the source code units from the ranking list as the result, because the scale of the software product in the benchmark is too large. We take the top 5% source code units from the ranking list, which may be a very large number. In particular, when calculating the precision by the formula precision = $(D_c \cap D_r / D_r) * 100\%$, D_r will be a very large number, while $D_c \cap D_r$ is very small. This will render *precision* a very small number. Thus, the difference in *precision* between different approaches will be very small.

This is also why we chose the top 5% from the ranking list as the CIA result. Existing studies [21, 35] advise taking the top 5%–15% from the ranking list as the CIA result. The more results we collect, the smaller the precision, which will lead to difference in performance between different methods becoming smaller. Therefore, we chose a minimum of 5% in our experiment.

Compared with the precision, recall, and F-score, MRR would be a more objective measure to evaluate the performance of CIA. Many other researchers [7, 8, 33, 34] have used MRR to evaluate the performance of their CIA approach. We also used this measure in our experiment, and results showed that our approach achieved an average improvement of 10.28% in MRR performance relative to the least effective method using a single indexing model. This could represent far better performance than other methods.

4.4.2. Discussion of RQ2. Table 3 presents the performance results from TF-IDF and doc2vec along with the weights calculated for our own linear combination. The CIA performance reflects the corresponding indexing model alone. The weight represents the cohesion and coupling degree of

the vectors in the corresponding indexing space. The CIA performance of TF-IDF is better than doc2vec's for JabRef, jEdit, muCommander, and ArgoUML, so the combination weight favors TF-IDF for these packages. However, doc2vec's CIA performance is better than the TF-IDF's on Eclipse, so the combination weight favors doc2vec in that case. Thus, we can estimate the performance of the CIA method based on the internal and external distance of the vectors in the indexing space. From Table 3, we can see that the cohesion and coupling degree of the source code vectors is consistent with CIA performance. Thus, we conclude that the structures of these software packages are all of high quality.

5. Conclusions

In this paper, we have presented a structure-driven method for information retrieval-based change impact analysis. Our approach integrates the bag-of-words and word embedding models during the indexing and similarity calculation steps. Our empirical results using a standard benchmark consisting of five open-source software packages demonstrate that our approach achieves better performance on the precision, recall, F-score, and MRR metrics than existing methods that use a single indexing method.

Several factors affect the validity of the results of our empirical case study and limit our ability to generalize our findings:

(1) We performed our case study with five Java software systems. The applicability of our approach to software written in other languages remains to be verified.

(2) We did not discuss the influence of parameters on the indexing methods; we used default settings. While other studies explore parameter settings, it was not the purpose of our own research.

(3) We evaluated our approach using only high quality source code. Our approach may fail when used with low-quality software.

In the future, we will develop our approach for information retrieval in related fields. Moreover, we plan to conduct additional experiments employing a greater variety of open-source projects to verify the universality of the proposed approach.

Data Availability

The software products' source code and changes data used to support the findings of this study have been deposited in the SEMERU repository [2] (http://www.cs.wm.edu/semeru/data/benchmarks/).

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under Grant nos. 61462092, 61379032, and 61662085, the Key Project of the Natural Science Foundation of Yunnan Province under Grant no. 2015FA014, the Data Driven Software Engineering Research Innovation Team of Yunnan Province under Grant no. 2017HC012, the Talents Training Programme Foundation for Light of West under Grant no. W8090311, and the Graduate Scientific Research Innovation Foundation of Yunnan University under Grant no. YDY17094.

References

[1] S. Y. Jiang, C. McMlillan, and R. Santelices, "Do programmers do change impact analysis in debugging?," *Empirical Software Engineering*, vol. 22, no. 2, pp. 631–669, 2017.

[2] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[3] N. Wilde, J. A. Gomez, T. Gust et al., "Locating user functionality in old code," in *Proceedings of 1992 IEEE International Conference on Software Maintenance*, pp. 200–205, Orlando, FL, USA, November 1992.

[4] W. T. Lee, S. P. Ma, and Y. Y. Tsai, "Retrieval of web service components using UML modeling and term expansion," *Journal of Information Science and Engineering*, vol. 33, no. 1, pp. 17–36, 2017.

[5] M. Linaresvasquez, A. Holtzhauer, and D. Poshyvanyk, "On automatically detecting similar Android apps," in *Proceedings of 24th International Conference on Program Comprehension*, pp. 1–10, Austin, TX, USA, May 2016.

[6] M. K. Hossen, H. Kagdi, and D. Poshyvanyk, "Amalgamating source code authors, maintainers, and change proneness to triage change requests," in *Proceedings of 22nd International Conference on Program Comprehension*, pp. 130–141, Hyderabad, India, May 2014.

[7] L. R. Biggers, C. Bocovich, R. Capshaw, B. P. Eddy, L. H. Etzkorn, and N. A. Kraft, "Configuring latent Dirichlet allocation based feature location," *Empirical Software Engineering*, vol. 19, no. 3, pp. 465–500, 2014.

[8] C. S. Corley, K. Damevski, and N. A. Kraft, "Exploring the use of deep learning for feature location," in *Proceedings of 2015 International Conference on Software Maintenance and Evolution*, pp. 556–560, Bremen, Germany, September-October 2015.

[9] A. Marcus, A. Sergeyev, V. Rajich et al., "An information retrieval approach to concept location in source code," in *Proceedings of 11th Working Conference on Reverse Engineering*, pp. 214–223, Delft, Netherlands, November 2004.

[10] Y. Zhang, R. Jin, and Z. H. Zhou, "Understanding bag-of-words model: a statistical framework," *International Journal of Machine Learning and Cybernetics*, vol. 1, no. 1–4, pp. 43–52, 2010.

[11] T. Mikolov, K. Chen, G. Corrado et al., "Efficient estimation of word representations in vector space," in *Proceedings of International Conference on Learning Representations*, pp. 1–13, Scottsdale, Arizona, USA, May 2013.

[12] T. Mikolov, I. Sstskever, K. Chen et al., "Distributed representations of words and phrases and their compositionality," *Advances in Neural Information Processing Systems*, vol. 26, pp. 3111–3119, 2013.

[13] R. S. Alsuhaibani, C. D. Newman, M. L. Collard et al., "Heuristic-based part-of-speech tagging of source code identifiers and comments," in *Proceedings of IEEE 5th Workshop*

on Mining Unstructured Data (MUD), pp. 1–6, Bremen, Germany, September 2015.

[14] S. Xu, "Modular change impact analysis for configurable software," in *Proceedings of 2016 IEEE International Conference on Software Maintenance and Evolution*, pp. 468–472, Raleigh, NC, USA, October 2016.

[15] G. Scanniello, A. Marcus, and D. Pascale, *Link Analysis Algorithms for Static Concept Location: An Empirical Assessment*, Kluwer Academic Publishers, Boston, MA, USA, 2015.

[16] S. Pugh, "Davied Binkley change impact using dynamic history analysis," in *Proceedings of 49th ACM Technical Symposium on Computer Science Education*, p. 275, Baltimore, MD, USA, February 2018.

[17] M. Sahu and D. P. Mohapatra, "Computing dynamic slices of feature--oriented programs using execution trace file," *ACM SIGSOFT Software Engineering Notes*, vol. 42, no. 2, pp. 1–16, 2017.

[18] M. Borg, K. Wnuk, B. Regnell, and P. Runeson, "Supporting change impact analysis using a recommendation system: an industrial case study in a safety-critical context," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 675–700, 2016.

[19] J. Font and C. Cetina, "Improving feature location by transforming the query from natural language into requirements," in *Proceedings of 20th International Systems and Software Product Line Conference*, pp. 362–369, Beijing, China, September 2016.

[20] T. Savage, M. Revelle, and D. Poshyvanyk, "FLAT 3 : feature location and textual tracing tool," in *Proceedings of 32nd International Conference on Software Engineering*, pp. 255–258, Cape Town, South Africa, May 2010.

[21] W. Wang, T. Li, Y. He et al., "A Hybrid approach for ripple effect analysis of software evolution activities," *Journal of Computer Research and Development*, vol. 53, no. 3, pp. 503–516, 2016.

[22] M. P. Robillard, "Topology analysis of software dependencies," *ACM Transaction on Software Engineering Methodology*, vol. 17, no. 4, pp. 1–36, 2008.

[23] M. Petrenko, V. Rajlich, and R. Vanciu, "Partial domain comprehension in software evolution and maintenance," in *Proceedings of 16th IEEE International Conference on Program Comprehension*, pp. 13–22, Amsterdam, Netherlands, January 2008.

[24] B. Dit, M. Revelle, and D. Poshyvanyk, "Integrating information retrieval, execution and link analysis algorithms to improve feature location in software," *Empirical Software Engineering*, vol. 18, no. 2, pp. 277–309, 2013.

[25] A. Panichella, B. Dit, R. Oliveto et al., "Parameterizing and assembling IR-based solutions for SE tasks using genetic algorithms," in *Proceedings of 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 314–325, Suita, Osaka, Japan, March 2016.

[26] S. L. Abebe, S. Haiduc, A. Marcus et al., "Analyzing the evolution of the source code vocabulary," in *Proceedings of 13th European Conference on Software Maintenance and Reengineering*, pp. 189–198, Kaiserslautern, Germany, 2009.

[27] B. L. Vinz and L. H. Etzkorn, "A synergistic approach to program comprehension," in *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pp. 69–73, Athens, Greece, 2006.

[28] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, vol. 41, no. 6, pp. 391–407, 1990.

[29] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of Machine Learning Research*, vol. 3, pp. 993–1022, 2003.

[30] C. Y. Liou, J. C. Huang, and W. C. Yang, "Modeling word perception using the Elman network," *Neurocomputing*, vol. 71, no. 16–18, pp. 3150–3157, 2008.

[31] Y. J. Guo, S. T. Wang, and L. Xu, "Learning a linear combination of distances based on the maximum-margin theory," *CAAI Transactions on Intelligent Systems*, vol. 10, no. 6, pp. 843–850, 2015.

[32] J. Wang, S. T. Wang, and Z. H. Deng, "A novel text clustering algorithm based on feature weighting distance and soft subspace learning," *Chinese Journal of Computers*, vol. 35, no. 8, pp. 1655–1665, 2012.

[33] M. Chochlov, M. English, and J. Buckley, "A historical, textual analysis approach to feature location," *Information and Software Technology*, vol. 88, pp. 110–126, 2017.

[34] D. Poshyvanyk, Y. G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE Transactions on Software Engineering*, vol. 33, no. 6, pp. 420–432, 2007.

[35] J. Xiaolin, S. Jiang, and Y. Zhang, "Advances in fault localization techniques," *Journal of Frontiers of Computer Science and Technology*, vol. 1, no. 2, pp. 139–176, 2012.