Research Article

An Integrated Model for Information Retrieval Based Change Impact Analysis

Wei Wang , Yun He, Tong Li, Jiajun Zhu, and Jinzhao Liu

Software School, Yunnan University, Kunming, Yunnan, China

Correspondence should be addressed to Jinzhao Liu; jinzhuo.liu@hotmail.com

Received 23 May 2017; Revised 8 September 2017; Accepted 4 October 2017; Published 1 March 2018

Academic Editor: Thomas Leich

Copyright © 2018 Wei Wang et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

The paper presents an approach to combine multiple existing information retrieval (IR) techniques to support change impact analysis, which seeks to identify the possible outcomes of a change or determine the necessary modifications for affecting a desired change. The approach integrates a *bag-of-words* based IR technique, where each class or method is abstracted as a set of words, and a neural network based IR technique to derive conceptual couplings from the source code of a software system. We report rigorous empirical assessments of the changes of three open source systems: jEdit, muCommander, and JabRef. The impact sets obtained are evaluated at the method level of granularity, and the results show that our integrated approach provides statistically significant improvements in accuracy across several cut points relative to the accuracies provided by the individual methods employed independently. Improvements in *F*-score values of up to 7.3%, 10.9%, and 17.3% are obtained over a baseline technique for jEdit, muCommander, and JabRef, respectively.

1. Introduction

Change impact analysis (CIA) seeks to identify the possible outcomes of a change in systems, or determine the necessary modifications for affecting a desired change [1]. CIA is widely regarded as a basis of systems engineering research regarding program understanding, cost estimation, the tracing of ripple effects, test case selection, and change propagation [1–6]. The software lifecycle impacts different aspects of a software system: the software modules that may be defective [1], the supporting documentation [3], and developers that must be informed about the impact [7]. In this paper, we focus on the first aspect: impacts at the code level. Conceptual couplings capture the extent to which domain concepts and software artifacts are related [8]. Information retrieval (IR) techniques can obtain conceptual coupling information from textual software artifacts (e.g., comments and identifiers in a single snapshot of source code). IR-based CIA is a type of static CIA that can extract many implicit conceptual couplings encoded by developers in identifiers and comments in source code. This feature allows researchers to analyze the impacts of

changes from a semantic perspective, which represents a new perspective for existing CIA research.

Conducting CIA based on conceptual coupling involves two main challenges. First, the process must scale to the current size of software systems, which often consist of thousands of modules. Second, impact estimations must be sufficiently accurate. The accuracy of IR-based CIA involves two aspects: whether the elements in the estimated impact set (EIS) are actually impacted and whether all truly impacted elements are predicted. Our motivation is to design a CIA system based on conceptual coupling that solves these challenges by maintaining high estimation performance while scaling to large systems.

To facilitate the proposed CIA system design, an approach that integrates a number of IR algorithms is expected to improve performance significantly in contrast to using just a single algorithm. This is because different IR algorithms abstract the source code into different matrices. For example, latent semantic indexing (LSI) [9] and latent Dirichlet allocation (LDA) [10] consider source code as a *bag-of-words*, where each class or method is abstracted as a set of words. While

the *bag-of-words* model is easy to understand and process, this mechanism disregards the order or syntactic structure in the source code. Recent research has shown that source code contains context and flow that are even more pronounced than natural language text [11]. A method based on a neural network that transforms the variable-length source code into a fixed length vector was later introduced to address the weaknesses of the *bag-of-words* model by capturing the influence of context on each term [12]. However, techniques based on neural networks are difficult to apply owing to their use of numerous parameters. Here, the key to the performance of these techniques is the appropriate selection of parameter values, and no single parameter selection method is suitable for all types of software. Therefore, the present work integrates *bag-of-words* based techniques with neural network based techniques in an effort to minimize the drawbacks of existing methods.

To evaluate our approach, we consider a case study involving three open source software systems: jEdit (a well-known text editor), muCommander (a cross-platform file manager), and JabRef (a reference manager), totaling 250,000+ lines of code. We introduce a total of 404 changes into the software systems and analyze their actual impacts in the case study. The precision, recall, and *F*-score of each change request are computed. The proposed approach, respectively, increases the *F*-score for the jEdit, muCommander, and JabRef systems up to 7.3%, 10.9%, and 17.3% greater than an LSI-based method [13], which served as the baseline technique, at the method granularity.

The contributions of the present work can be summarized as follows.

- (i) We propose an algorithm for learning the similarity metrics. The algorithm can directly compute the following cosine (1) and Euclidean (2) similarity functions, which are the most commonly used similarity measurements in CIA, and no learning procedure is required.

$$\cos(c, r) = \frac{\langle c, r \rangle}{\|c\| \|r\|}, \quad (1)$$

$$\text{dis}(c, r) = \sqrt{(c - r)^T (c - r)}. \quad (2)$$

Here, c and r represent a source code vector and a change request vector, respectively. For the cosine metric, if any i th component of c or r (denoted as c_i or r_i) is 0, then the similarity between in the i th dimension is 0. Source code has a greater sparseness than natural language, so the occurrence of c_i or r_i being equal to 0 is more frequent than for natural language. Therefore, the cosine metric cannot accurately characterize the similarity between source code and change request. The Euclidean metric has demonstrated good performance on data with a hypersphere structure. However, the performance of the metric is poor for data with a hypercube or super-ellipsoidal structure. Therefore, it is unreasonable to employ the

Euclidean metric to measure the similarity between source code and change request because source code matrices cannot be guaranteed to always have a hypersphere structure.

- (ii) We propose an algorithm for integrating multiple IR techniques to improve the performance of CIA.
- (iii) Experimental results obtained for three open source software systems totaling 250,000+ lines of code indicate that our approach is promising for CIA applications.

The remainder of this paper is organized as follows. In Section 2, we present the background concepts employed in this paper. In Section 3, we introduce the proposed approach. The effectiveness of the proposed approach is validated in Section 4, where we perform an extensive empirical study that compares the performance of the proposed method with the performance of the baseline approach for the three open source software systems based on several metrics. In Section 5, we discuss the factors affecting the validity of our empirical results. Related work is presented in Section 6, and Section 7 presents the conclusions of our work and provides directions for further study.

2. Background

This section discusses the background concepts employed in this paper and related research work, which includes CIA and conceptual coupling, IR-based conceptual coupling measurement, LSI, and doc2vec.

2.1. CIA and Conceptual Coupling. The process of CIA includes two primary steps. It begins with identifying an initial change set that could be affected by a change request [13, 14]. The initial change set is usually identified by feature location, software comprehension, and other techniques [15]. Other artifacts that are likely to be affected by the initial change set are estimated by a CIA technique, which then produces an EIS as the output of CIA [8]. Secondly, the actual modified artifact set, denoted as the actual impact set (AIS), is determined [8]. The goal of CIA is to provide an EIS that is as close as possible to the AIS.

The core of CIA is to identify how software modules relate to each other. Coupling is one of the fundamental properties of software. Coupling involves several factors contained in the source code, such as control and data flow, and it can be measured in a number of different ways. Hence, researchers have proposed numerous coupling measures such as structural couplings metrics [15–17], data flow coupling metrics [18], dynamic coupling metrics [19], and evolution couplings [20].

Conceptual coupling is modeled based on the textual information shared between the modules of a source code. The elements of the source code written in a programming language help identify control or data flow between software modules, and the comments and identifiers express the intent

FIGURE 1: Overview of the proposed integrated change impact analysis (ICIA) algorithm.

of the software. Two sections of the software with similar intents will most likely refer to the same (or related) concepts in the problem or solution domains of the system. Hence, they are conceptually related. This has been also confirmed by the earlier work of other researchers who examined the overlap of semantic information in comments and identifiers among different software modules [21]. The CIA research community recognizes the importance of textual elements and applies many IR methods to compute conceptual couplings [22]. Numerous factors have made the study of CIA based on conceptual coupling a topic of great interest. For example, many useful intermediate artifacts (e.g., requirement documents, design documents, and traceability information) between change requests and source code are commonly unavailable. Change requests (e.g., bug reports submitted by programmers or users) are often the only source of information available for conducting CIA.

2.2. IR-Based Conceptual Coupling Measurement. IR is an advanced technique that can be used to transform source code or change requests into corresponding digital formats, matrices, or vectors. IR methods are commonly employed in the field of software engineering. For example, latent semantic analysis and concept lattices have been employed to construct software libraries [23] and support reuse tasks [24], while more recent work has focused on specific software maintenance and development tasks such as the recovery of traceability links [25] and the identification of duplicate bug reports [26]. Recently, IR techniques have also been applied to CIA [26, 27].

Here, we present a formal definition of conceptual coupling.

Definition 1 (conceptual coupling between methods—CCM). The conceptual coupling between two methods m_i and m_j , denoted as $CCM(m_i, m_j)$, is computed by a similarity metric

between the vectors vm_i and vm_j , corresponding to m_i and m_j .

2.3. LSI. LSI is a *bag-of-words* based technique. It maps both words and documents into a real valued vector space using the singular value decomposition (SVD) technique. The documents abstracted by LSI are linear combinations of the original term frequency-inverse document frequency (TF-IDF) features. This method can capture various linguistic factors such as synonymy and polysemy, which is a valuable advancement compared with TF-IDF. Additional technical details regarding LSI are available elsewhere [28].

2.4. Doc2vec. As discussed, the *bag-of-words* scheme does not account for context in documents. Distributed memory representation is a newly reported scheme that can capture the influence of the surrounding context and order in documents and therefore addresses some of the shortcomings of LSI. Here, doc2vec is a typical implementation of distributed memory representation that can learn a fixed length vector representation for each segment of text. This scheme is general and applicable to texts of any length, such as sentences, paragraphs, and entire documents. Additional technical details regarding doc2vec are available elsewhere [12].

3. Proposed Approach

In this section, we present the proposed integrated CIA (ICIA) algorithm. As shown in Figure 1, the ICIA process involves the following four steps.

Step 1 (preprocessing). Extract all identifiers, comments, and other artifacts from the source code to create a corpus for a software system.

Step 2 (indexing). Transform the corpus and change request into their corresponding matrix and vector forms by IR

FIGURE 2: Call graph of FreeMind source codes related to the problem *cannot automatically save file*.

techniques. The present work integrates LSI [28] and doc2vec [29].

Step 3 (similarity learning). Employ a learning paradigm to generate a similarity metric.

Step 4. Integrate the IR-generated similarity metrics and set the cut point. Any document with a similarity greater than the cut point is taken as the EIS.

This algorithm has two primary differences compared with other CIA algorithms: similarity learning and similarity integration.

- (i) We propose a new method for measuring the similarity between source code and change request based on a learning paradigm.
- (ii) To overcome the drawbacks associated with IR techniques based on *bag-of-words* and neural networks, we combine the similarity results generated by these two IR techniques.

The above four steps are discussed in detail in the following subsections.

3.1. Preprocessing. The process of preprocessing involves the following two substeps.

Step 1 (build the corpus). A corpus consists of a set of documents, where each document is created for a single class or a method in the source code. As an example, we analyze the problem “cannot automatically save file” in the FreeMind source code (Freemind: <http://freemind.sourceforge.net>).

Figure 2 shows the call graph of FreeMind. Here, only the shaded methods in the graph are related to the problem. According to Figure 2, the corpus contains 8 documents. This step does not differ from existing techniques, and more detailed information can be found elsewhere [26, 28].

Step 2. This step involves three components: stop word removal, keyword segmentation, and stemming. The stop words, such as *int* and *new*, are usually operators, programming language keywords, and constants. Keyword segmentation splits the compound identifiers into several words. For example, *timerForAutomaticSaving* must be split into four words: *timer*, *for*, *automatic*, and *saving*. Stemming is performed to reduce words to their root forms. For example, we transform the keyword *tools* into *tool*.

3.2. Indexing. We selected LSI as an appropriate IR technique for three main reasons. First, LSI has been shown to address the issues of polysemy and synonymy quite well, which is important with respect to the CIA process because developers typically construct change request descriptions without precise prior knowledge regarding the vocabulary used in the evolving system. Second, source code is a special subset of natural language. This is eligible for using IR techniques to facilitate the CIA process. Thirdly, identifiers in source code have no clearly defined grammar. LSI is well-suited to accommodate this characteristic because it does not use predefined grammar or vocabulary. The meanings of the words in LSI are derived from their usage rather than from a dictionary. Equation (3) shows the LSI transformation of Figure 2, where the change request “cannot automatically save file” is transformed to a vector $\mathbf{r} = [0.5481 \ -0.0373 \ -0.5541 \ 0.0160 \ 0.0950 \ 0.3691 \ -0.8635]^T$.

$$A = \begin{bmatrix} d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 & d_8 \\ 0.5679 & 0.5346 & 0.4147 & 0.8033 & 0.3148 & 0.3148 & 0.8431 & 0.1076 \\ -0.1909 & -0.1728 & -0.1037 & -0.2400 & 0.9487 & 0.9487 & -0.1855 & -0.0399 \\ -0.6977 & -0.7299 & 0.3108 & 0.4200 & -0.0266 & -0.0266 & 0.3864 & 0.1031 \\ -0.0153 & -0.0157 & 0.1571 & 0.0314 & -0.0068 & -0.0068 & 0.0433 & -0.9815 \\ 0.0353 & 0.0604 & 0.8342 & -0.2691 & 0.0004 & 0.0004 & -0.2308 & 0.1132 \\ -0.3909 & 0.3838 & -0.0052 & -0.0107 & -0.0026 & -0.0026 & 0.0346 & 0.0004 \\ -0.0082 & 0.0180 & 0.0080 & 0.2171 & 0.0070 & 0.0070 & -0.2217 & -0.0018 \end{bmatrix}. \quad (3)$$

Equation (4) shows the doc2vec abstraction of Figure 2, where the change request “cannot automatically save file” is

transformed to vector $\mathbf{r} = [-0.0146 \ 0.0075 \ -0.0201 \ 0.0319 \ 0.0410 \ -0.0493 \ 0.0292 \ -0.0175 \ -0.0047 \ -0.0102]^T$.

$$A = \begin{bmatrix} d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 & d_8 \\ -0.0163 & 0.0012 & 0.0038 & 0.0518 & 0.0230 & 0.0159 & -0.0136 & 0.0445 \\ -0.0148 & -0.0280 & -0.0433 & -0.0063 & 0.0242 & -0.0287 & -0.0361 & -0.0355 \\ 0.0185 & -0.0351 & -0.0204 & -0.0017 & 0.0164 & 0.0456 & 0.0340 & -0.0486 \\ 0.0030 & -0.0356 & 0.0128 & -0.0479 & 0.0201 & 0.0088 & 0.0127 & 0.0266 \\ 0.0345 & 0.0210 & 0.0146 & -0.0319 & -0.0164 & -0.0247 & -0.0065 & -0.0179 \\ -0.0226 & 0.0222 & 0.0407 & 0.0222 & 0.0419 & -0.0333 & 0.0048 & 0.0068 \\ -0.0223 & 0.0499 & 0.0138 & 0.0243 & -0.0370 & 0.0026 & -0.0089 & 0.0056 \\ -0.0435 & -0.0342 & -0.0018 & -0.0126 & -0.0433 & -0.0100 & -0.0461 & 0.0030 \\ 0.0190 & -0.0028 & -0.0043 & -0.0406 & 0.0118 & -0.0309 & 0.0455 & -0.0085 \\ -0.0294 & 0.0028 & 0.0164 & -0.0266 & 0.0080 & 0.0043 & -0.0469 & 0.0036 \end{bmatrix}. \quad (4)$$

3.3. Similarity Learning. Distance metrics are often employed in the field of CIA to measure the similarity between source code and change request. In this section, we provide a distance metric learning algorithm. For each change request vector $\mathbf{r}$, the algorithm generates two similarity vectors $\mathbf{s}_{\text{LSI}}$ and $\mathbf{s}_{\text{doc2vec}}$. Distance metrics are defined as follows.

Definition 2 (distance metric). The metric $\mathcal{D}(x, y)$ is denoted as a distance metric if and only if it satisfies the following four properties:

- (1) Nonnegativity: $\mathcal{D}(x, y) \geq 0$;
- (2) Coincidence: $\mathcal{D}(x, y) = 0$ if and only if $x = y$;
- (3) Symmetry: $\mathcal{D}(x, y) = \mathcal{D}(y, x)$;
- (4) Subadditivity: $\mathcal{D}(x, y) + \mathcal{D}(y, z) \geq \mathcal{D}(x, z)$.

Any metric meeting the above properties can be considered a distance metric. In CIA research, Euclidean and cosine distances are used to compute the similarity between source code and change request. However, Euclidean and cosine distance metrics have distinct disadvantages in CIA

research. Both source code and change request vectors are typically multidimensional and sparse, and the Euclidean and cosine distance metrics attribute equivalent importance to any direction (word) of these vectors. Without prior knowledge of what is being specifically searched for in high-dimensional space, irrelevant vectors cannot be removed (i.e., penalized). Therefore, Euclidean and cosine distance metrics are weakly discriminant.

This section proposes a distance metric learning technique that provides prior knowledge regarding the necessary search parameters to facilitate searching in high-dimensional space. Here, we mainly focus on learning the generalized Mahalanobis distance.

Definition 3 (generalized Mahalanobis distance). The generalized Mahalanobis distance measures the distance between source code vector $\mathbf{c}$ and query vector $\mathbf{r}$ as

$$\mathcal{D}(\mathbf{c}, \mathbf{r}) = \sqrt{(\mathbf{c} - \mathbf{r})^T \mathbf{M} (\mathbf{c} - \mathbf{r})}, \quad (5)$$

where $\mathbf{M}$ is some arbitrary symmetric positive semidefinite (SPSD) matrix. From the expression given in (5), we can

decompose $\mathbf{M}$ as $\mathbf{M} = \mathbf{U}\mathbf{A}\mathbf{U}^T$ using eigenvalue decomposition, where $\mathbf{U}$ is a matrix collecting all the eigenvectors of $\mathbf{M}$, and $\mathbf{A}$ is a diagonal matrix formed of all the eigenvalues of $\mathbf{M}$. Letting $\mathbf{W} = \mathbf{U}\mathbf{A}^{1/2}$ yields the following.

$$\begin{aligned} \mathcal{D}(\mathbf{c}, \mathbf{r}) &= \sqrt{(\mathbf{c} - \mathbf{r})^T (\mathbf{U}\mathbf{A}^{1/2}) (\mathbf{U}\mathbf{A}^{1/2})^T (\mathbf{c} - \mathbf{r})} \\ &= \sqrt{(\mathbf{c} - \mathbf{r})^T \mathbf{W}\mathbf{W}^T (\mathbf{c} - \mathbf{r})} \\ &= \sqrt{(\mathbf{W}^T (\mathbf{c} - \mathbf{r}))^T (\mathbf{W}^T (\mathbf{c} - \mathbf{r}))}. \end{aligned} \quad (6)$$

By comparing the expression given by (6) with that given by (2), we note that the generalized Mahalanobis distance is equivalent to the Euclidean distance of the data in the projected space transformed by matrix $\mathbf{W}$. Therefore, learning an optimal matrix $\mathbf{M}$ is equivalent to learning a projection matrix $\mathbf{W}$ that transforms the source code matrix from the original space to the projected space. We believe that the similarity between two code vectors in original space should be maintained in projected space.

Formally, let $\mathbf{X}$ be an original space matrix with t rows and d columns, where t is the number of terms and d is the number of documents in the source code. The i th column of $\mathbf{X}$ represents the i th document in the corpus. Let $\mathbf{S} = \mathbf{X}\mathbf{X}^T$ be a square SPSP matrix. The term $S_{i,j}$ is the inner product similarity between the i th and j th documents in original space. The projection matrix $\mathbf{W}$ maps documents in the t -dimensional original space to new representations in a k -dimensional projected space. Let $\mathbf{W}$ be the projection matrix implementing such mapping. The product $\mathbf{W}\mathbf{X}^T$ is a $k \times d$ matrix of source code represented in the projected space. Therefore, $\hat{\mathbf{S}} = (\mathbf{W}\mathbf{X})(\mathbf{W}\mathbf{X})^T = \mathbf{W}\mathbf{X}\mathbf{X}^T\mathbf{W}$ is the inner product similarity between source code vectors in projected space. The goal of similarity learning is to find a $\mathbf{W}$ that minimizes the similarity difference $(\mathbf{S} - \hat{\mathbf{S}})^2$, that is, to find $\mathbf{W}$ with the following property:

$$\begin{aligned} \arg \min \quad & (\mathbf{S} - \hat{\mathbf{S}})^2, \\ \text{s.t.} \quad & \mathbf{W}\mathbf{W}^T = \mathbf{I}, \end{aligned} \quad (7)$$

where $\mathbf{I}$ is an identity matrix. The critical point of (7) is of the form

$$\mathbf{W} = \mathbf{P}_k^T \mathbf{C}\mathbf{X}^+. \quad (8)$$

Here, $\mathbf{P}_k^T$ is the transpose of $\mathbf{P}_k$ where $\mathbf{P}_k$ is derived from $\mathbf{P}$ by retaining the k columns of $\mathbf{P}$, $\mathbf{C}$ is obtained using the SVD of $\mathbf{S}$ into $\mathbf{S} = \mathbf{C}^T\mathbf{C}$, and $\mathbf{X}^+$ is the pseudoinverse of $\mathbf{X}$. Let the SVD of $\mathbf{X}$ be $\mathbf{X} = \mathbf{U}\mathbf{L}\mathbf{A}^T$, where $\mathbf{L}$ and $\mathbf{A}$ are Eigen matrix and singular vectors, respectively; then, $\mathbf{X}^+ = \mathbf{A}\mathbf{L}^{-1}\mathbf{U}^T$. Finally, $\mathbf{M}$ is provided by the SVD of the product $\mathbf{C}\mathbf{A}\mathbf{A}^T$ into $\mathbf{C}\mathbf{A}\mathbf{A}^T = \mathbf{P}\mathbf{\Sigma}\mathbf{N}^T$, where $\mathbf{\Sigma}$ and $\mathbf{N}$ are Eigen matrix and singular vectors, respectively. The details for computing the critical point can be found elsewhere [30]. The use of SVD for achieving the proposed similarity learning is given in Algorithm 1.

Similarity learning

Input: source code matrix $\mathbf{X}$, dimension parameter k

Output: distance matrix $\mathbf{M}$

- (1) $\mathbf{S} \leftarrow \mathbf{X}\mathbf{X}^T$
- (2) Compute $\text{SVD}(\mathbf{X}) = \mathbf{U}\mathbf{L}\mathbf{A}^T$
- (3) $\mathbf{X}^+ \leftarrow \mathbf{A}\mathbf{L}^{-1}\mathbf{U}^T$
- (4) Compute $\text{SVD}(\mathbf{S}) = \mathbf{C}\mathbf{C}^T$
- (5) Compute $\text{SVD}(\mathbf{C}\mathbf{A}\mathbf{A}^T) = \mathbf{P}\mathbf{\Sigma}\mathbf{N}^T$
- (6) $\mathbf{W} \leftarrow \mathbf{P}_k^T \mathbf{C}\mathbf{X}^+$
- (7) *return* $\mathbf{M} \leftarrow \mathbf{W}\mathbf{W}^T$

ALGORITHM 1: The use of SVD to achieve similarity learning.

The first step of Algorithm 1 produces a square SPSP matrix $\mathbf{S} = \mathbf{X}\mathbf{X}^T$, where $S_{i,j}$ is the inner product similarity between the i th and j th documents in original space and $\mathbf{X}^+$ is the pseudoinverse of $\mathbf{X}$, which can be calculated by using SVD (steps (2) and (3) of the algorithm). Let the SVD of $\mathbf{X}$ be $\mathbf{X} = \mathbf{U}\mathbf{L}\mathbf{A}^T$; then, $\mathbf{X}^+ = \mathbf{A}\mathbf{L}^{-1}\mathbf{U}^T$, $\mathbf{C}$ is obtained by the SVD of $\mathbf{S}$ into $\mathbf{S} = \mathbf{C}\mathbf{C}^T$ (step (4)), $\mathbf{P}$ is obtained by the SVD of the product $\mathbf{C}\mathbf{A}\mathbf{A}^T = \mathbf{P}\mathbf{\Sigma}\mathbf{N}^T$, where $\mathbf{A}$ is given by the SVD of $\mathbf{X}$ (step (5)), and $\mathbf{P}_k$ is derived from $\mathbf{P}$ by retaining the k columns of $\mathbf{P}$ corresponding to the largest singular values in $\mathbf{\Sigma}$ and discarding the rest. Finally, $\mathbf{W}$ is given by the product of $\mathbf{P}_k^T \mathbf{C}\mathbf{X}^+$ (step (6)).

The output of Algorithm 1 is the SPSP matrix $\mathbf{M}$. Any similarity between query vector $\mathbf{r}$ and source code $\mathbf{X}_i$ can be computed as $\mathcal{D}(\mathbf{S}_i, \mathbf{r}) = \sqrt{(\mathbf{S}_i - \mathbf{r})^T \mathbf{M} (\mathbf{S}_i - \mathbf{r})}$. Therefore, the similarity vectors of the query “cannot automatically save file” and the LSI and doc2vec matrices (equations (3) and (4), respectively) are $[0.46 \ 0.68 \ 0.11 \ 0.0 \ 0.12 \ 0.12 \ 0.36 \ 0.0]^T$, and $[0.44 \ 0.11 \ 0.12 \ 0.40 \ 0.42 \ 0.02 \ 0.02 \ 0.06]^T$, respectively.

3.4. Integration. As discussed in Section 2.2, each query vector $\mathbf{r}$ corresponds to two similarity vectors $\mathbf{s}_{\text{LSI}}$ and $\mathbf{s}_{\text{doc2vec}}$. In this subsection, we integrate the two similarity vectors in the form of $\mathbf{IS} = w_{\text{LSI}} \cdot \mathbf{s}_{\text{LSI}} + w_{\text{doc2vec}} \cdot \mathbf{s}_{\text{doc2vec}}$. Therefore, the core step of the integration is to identify the coefficients (weights) w_{LSI} and w_{doc2vec} of $\mathbf{s}_{\text{LSI}}$ and $\mathbf{s}_{\text{doc2vec}}$, respectively.

We use variance to calculate w_{LSI} and w_{doc2vec} . As shown in Figure 3, the orange triangle represents the query vector and the blue circles represent source code vectors. Comparing Figures 3(a) and 3(b), we note that it is more difficult to determine which of the source code vectors that are close to the query vector in Figure 3(a) is more similar to the query than the case in Figure 3(b). Formally speaking, the variance of source code vectors in Figure 3(a) is less than that of Figure 3(b).

We employ the following steps to calculate w_{LSI} and w_{doc2vec} .

- (i) Normalization: we scale $\mathbf{s}_{\text{LSI}}$ and $\mathbf{s}_{\text{doc2vec}}$ in the range $[0, 1]$ using the equation $(\mathbf{s}_{i,j} - \mathbf{s}_{i,\min})/(\mathbf{s}_{i,\max} - \mathbf{s}_{i,\min})$, where $i \in \{\text{LSI}, \text{doc2vec}\}$, and $\mathbf{s}_{i,\max}$ and $\mathbf{s}_{i,\min}$ are the maximum and minimum elements of $\mathbf{s}_i$.

FIGURE 3: Two collections of source code vectors with different variances.

- (ii) Calculate the variance of $\mathbf{s}_{\text{LSI}}$ and $\mathbf{s}_{\text{doc2vec}}$: the variance sv_i is calculated as follows:

$$sv_i = \frac{1}{n} \left[(\mathbf{s}_{i,1} - \mathbf{s}_{i,\text{avg}})^2 + (\mathbf{s}_{i,2} - \mathbf{s}_{i,\text{avg}})^2 + \dots + (\mathbf{s}_{i,c} - \mathbf{s}_{i,\text{avg}})^2 \right], \quad (9)$$

where $\mathbf{s}_{i,\text{avg}} = (\mathbf{s}_{i,1} + \mathbf{s}_{i,2} + \dots + \mathbf{s}_{i,n})/n$ and n is the number of documents.

- (iii) Compute w_{LSI} and w_{doc2vec} : the coefficients are calculated as

$$w_i = \frac{sv_i}{sv_{\text{LSI}} + sv_{\text{doc2vec}}}. \quad (10)$$

- (iv) Integrate $\mathbf{s}_{\text{LSI}}$ and $\mathbf{s}_{\text{doc2vec}}$ according to $\mathbf{IS} = w_{\text{LSI}} \cdot \mathbf{s}_{\text{LSI}} + w_{\text{doc2vec}} \cdot \mathbf{s}_{\text{doc2vec}}$: take the top c elements of $\mathbf{IS}$ as the EIS, where the parameter c is denoted as the cut point.

Based on the integration procedure, we set the cut point as $c = 3$. The variances of LSI and doc2vec similarity vectors for the example given in Figure 2 are 0.11 and 0.17, and $w_{\text{LSI}} = 0.39$ and $w_{\text{doc2vec}} = 0.61$. The integrated similarity vector $\mathbf{IS} = [0.45 \ 0.34 \ 0.12 \ 0.24 \ 0.30 \ 0.06 \ 0.15 \ 0.04]^T$. We then obtain the source code vectors with the top c similarities in $\mathbf{IS}$ as the EIS. For example, if $c = 4$, then the 1st, 2nd, 5th, and 4th methods are taken as the EIS.

4. Case Study

The case study was designed to evaluate the effectiveness of our approach. In particular, we seek to obtain answers to the following research questions (RQs).

- (i) RQ1: does our integration approach provide an improvement over the individual methods?
- (ii) RQ2: does the choice of c affect the accuracy of the proposed approach?
- (iii) RQ3: does the size of the software affect the comprehensive performance of the proposed approach?

4.1. Experimental Systems. To guarantee the objectivity of the case study, we built a benchmark that collects data from previous related work [12, 31, 32]. The jEdit, muCommander, and JabRef software systems employed for testing have between 4607 and 8187 methods, and each dataset includes gold sets (program elements that are actually related to the change requests) at method granularity, change request descriptions constructed by extracting the title and description from the issue tracking system, and a corpus extracted from the source code. Table 1 lists further details of these three software systems.

4.2. Accuracy Metrics. To measure the accuracy of the proposed CIA technology, we must determine how close the EIS is to the AIS. For each change request, the AIS is equivalent to the gold set. Previous studies have proposed various metrics for quantitative analysis. Precision and recall are two widely accepted metrics in CIA [13, 31, 33, 34]. Precision reflects the consistency between the EIS and AIS, and recall reflects the proportion of the EIS in the AIS. A high precision indicates that software engineers will require less effort to determine the impact set for a change request, and a high recall indicates that the EIS is credible. Precision and recall are defined as $P = |\text{EIS} \cap \text{AIS}|/|\text{EIS}| \times 100\%$, and $R = |\text{EIS} \cap \text{AIS}|/|\text{AIS}| \times 100\%$, respectively. However, we note that because P and R are reciprocal, they are unable to reflect the comprehensive performance of a CIA technique. Therefore, we also employ

TABLE 1: Overview of the software systems employed in the case study.

System	Version	Lines of code	Number of methods	Number of change requests	Average number of gold sets for each change request
JEdit	4.3	103,896	6,413	272	4
muCommander	0.8.5	76,649	8,187	92	8
JabRef	2.6	74,182	4,607	40	7

the F -score to measure the performance, which is defined as follows.

Definition 4 (F-score). The F -score is defined as $F = 2 \times (P \times R / (P + R)) \times 100\%$.

4.3. Evaluation Process. The evaluation process involves the following steps.

- (1) Build the benchmark from datasets published by other researchers [12, 31, 32].
- (2) Generate the EIS for each change request by ICIA and the LSI-based baseline technique, respectively. We employed *gensim* (<https://radimrehurek.com/gensim/models/lsmi.html>) to implement LSI-based CIA and set all parameters involved in the LSI as the default values. We employed *numpy* (<http://www.numpy.org/>) to implement SVD and variance computations for ICIA.
- (3) Examine the effectiveness of the proposed approach. Compute P , R , and the F -score for each change request. Compare the metrics obtained for the various methods and evaluate the effectiveness of our approach.
- (4) Identify the effects of c and the size of software on our approach.

4.4. Experimental Results. The comprehensive results for ICIA and the baseline technique with different values of c are presented in Table 2. The results for each software system are discussed in detail as follows.

4.4.1. Results for jEdit. The results given in Table 2 indicate that the proposed approach comprehensively outperforms the baseline technique for jEdit. The table indicates an improvement of as much as 2.24% in precision and 21.88% in recall. A maximum precision of 34.33% was obtained for the change request ID 1546200 and $c = 50$. A maximum recall of 100% was obtained for the change request ID 1541372 and $c = 200$. The change descriptions of IDs 1546200 and 1541372 are given as follows (please note that these examples, and all subsequent examples, are given verbatim to better illustrate the nature of the data employed): “Add filter to recent files menu This patch helps to find a items in the recent “files menu”, by typing the first letter of the filename in a new textfield in recent files menu. Files which are not matching are enabled in menu. Please add it to org/gjt/sp/jedit/menu/RecentFilesProvider.java Joerg” and “If the caret is in a fold that

gets collapsed, the caret should be placed at first line of that fold, i. e. the remaining visible line, otherwise it reexpands the fold on direction key press. Sorry if this is a dupe, but I have many bugs to post and am too lazy to check them all for dupes currently.-) OS: Windows XP Java Version: Sun Java 1.5.0_06-b05 jEdit Version: SVN Revision 6684”.

4.4.2. Results for muCommander. The results given in Table 2 also indicate that the proposed approach comprehensively outperforms the baseline technique for muCommander. The table indicates an improvement of as much as 11.05% in precision and 10.89% in recall. A maximum precision of 28.67% was obtained for change request ID 39 and $c = 50$. A maximum recall of 100% was obtained for change request ID 245 and $c = 200$. The change descriptions of IDs 39 and 245 are ““Hey, When there are no elements at the combobox (after “clear history” action, for example), opening the popup of the combobox throws the following exception: java.lang.IllegalArgumentException: setSelectedIndex: 0 out of bounds You should probably add the check “if (getItemCount() > 0)” before the “setSelectedIndex(0)” is made at the “popupMenuWillBecomeVisible function. Arik” and “Currently, the Unicode byte-order mark (BOM) receives no special treatment and is thus considered as an editable character by the text editor. Therefore, files encoded in UTF-8/16/32 with a leading byte-order mark (BOM) will start with an invisible character in the editor, as evidenced by the caret (the right-arrow needs to be pressed twice to move past the first real character). To prevent this artifact from occurring, the Unicode BOM should not be considered as an editable character”.

4.4.3. Results for JabRef. The results given in Table 2 indicate that the proposed approach comprehensively outperforms the baseline technique for JabRef. The table indicates an improvement of as much as 14.2% in precision and 19% in recall. A maximum precision of 28.57% was obtained for change request ID 1297576 and $c = 50$. A maximum recall of 100% was obtained for change request ID 2119059 and $c = 200$. The change descriptions of IDs 1297576 and 2119059 are “Sending the content of the entry preview to a printer (simple File→Print) would be cool. Customizing the preview allows to have notes or comments displayed and nicely formatted. But printing out this information is not easily possible (copy-paste of the entry preview can be a solution, but depending on the target programm may fail to preserve text formatting).” and “The keyword A1 is used for first author instead of AU for other authors in the RIS file of the APS (American Physical Society)

TABLE 2: Precision (P), recall (R), and harmonic mean (F) results of ICIA and the baseline approach for the three software systems considered.

Cut point		50			100			150			200		
jEdit													
Accuracy	P	R	F	P	R	F	P	R	F	P	R	F	
LSI (%)	8.51	22.76	12.39	3.92	17.28	6.39	3.13	17.46	5.31	3.51	30.38	6.30	
ICIA (%)	10.12	19.69	19.69	6.16	35.05	10.48	5.23	39.33	9.23	4.12	45.43	7.56	
muCommander													
Accuracy	P	R	F	P	R	F	P	R	F	P	R	F	
LSI (%)	11.28	11.55	11.41	3.92	17.28	6.39	3.67	27.68	6.48	3.05	34.33	5.60	
ICIA (%)	22.33	22.29	22.31	9.86	28.17	14.61	5.72	34.22	9.80	4.23	40.58	7.66	
JabRef													
Accuracy	P	R	F	P	R	F	P	R	F	P	R	F	
LSI (%)	10.78	5.82	7.56	8.08	16.22	10.79	4.95	16.17	7.58	4.51	22.69	7.53	
ICIA (%)	24.98	24.82	24.90	9.68	23.64	13.74	7.21	27.22	11.40	4.80	23.03	7.95	

journals. This keyword A1 is not detected by JabRef RIS import filter. emmanuel.clouet@cea.fr.”

4.5. Discussion of RQs

4.5.1. Discussion for RQ1. The experimental results in Table 2 show that the accuracy of the proposed CIA approach is better than that of the baseline technique. The average F -scores of ICIA for the four values of c considered are 22.30%, 12.94%, 10.14%, and 7.72%, while the corresponding average F -scores of the baseline technique are 10.45%, 7.86%, 6.46% and 6.48%, respectively. There are positive improvements in P and R for all three software systems. We explored the EISs generated by the baseline and ICIA approaches. We find that many false positive methods are included in the EIS obtained by LSI. It is evident that the methods ranked by ICIA are more relevant to the change request compared with those of the LSI-based approach because the positions of such false positive methods reside closer to the bottom of the similarity vectors. Moreover, the integration approach promoted the relevant methods ranked at the bottom by LSI. To illustrate the benefits of ICIA, we present change request ID 1535044 from JabRef as an example, which is given as “I don’t know if it’s a bug or a feature request but it would be great to have month sorting based on the calendar rather than on alphabet:-). Now I use year as first sort criterion and month as second sort criterion and I get the paper sorted from “A”pril to “S”eptember.” This change request involves 7 methods in the gold set, `net.sf.jabref.FieldComparator.FieldComparator(String,boolean)`, `net.sf.jabref.FieldComparator.compare(Object, Object)`, `net.sf.jabref.Util.toFourDigitYear(String)`, `net.sf.jabref.Util.toFourDigitYear(String,int)`, `net.sf.jabref.Util.getMonthNumber(String)`, `tests.net.sf.jabref.UtilTest.test2to4DigitsYear()`, `tests.net.sf.jabref.UtilTest.testToMonthNumber()`. Using LSI, the 7 relevant methods in the gold set are ranked at positions 52, 476, 58, 46, 46, 190, and 28. Using the ICIA approach, the corresponding methods in the gold set are ranked at positions 8, 12, 4, 41, 41, 15, and 1, respectively.

In addition, ICIA is a change request driven technique. Change requests are usually depicted using natural language, and it is easy to understand and distill them from existing

resources such as requirement specifications, design specifications, and user manuals. Hence, engineers without any domain knowledge of evolving systems can conduct CIA easily. Many other CIA techniques take a code driven analysis approach that requires engineers to have a good domain knowledge of the evolving software system to determine the evolving target code in advance. In the absence of effective software design documents, user manuals, and other data support, it is very difficult to define the target code of an evolving task, and the subsequent CIA will also be difficult to achieve.

From the above discussion, we conclude that the proposed approach is effective, in that it provides an improvement over the individual methods.

4.5.2. Discussion for RQ2. With respect to the effect of the value of c on the accuracy of the proposed approach, we note from Table 2 that a smaller value of c generates fewer elements in the ESI, which will increase P , but this will occur at the expense of a decreased R . Alternatively, employing a larger value of c can result in an increased R , but this occurs at the expense of a decreased P . The selection of an optimum value of c depends on the domain knowledge of the engineers involved, and no unified method exists for selecting a reasonable cut point.

4.5.3. Discussion for RQ3. With respect to the effect of the size of the software on the accuracy of the proposed approach, we note from Table 1 that jEdit is the largest software with 103,896 lines of code, muCommander is the second largest software with 76,649 lines of code, and JabRef is the smallest one with 74,182 lines of code. Here, we employ the variance of the average F -scores to measure the performance differences for equivalent values of c . A small variance indicates little performance difference. From Figure 4, we note that the average F -scores of our approach for the three software systems are nearly identical. The largest variance of F -scores for the ICIA approach for the three benchmark software systems is 6.8% when $c = 50$. The smallest variance of F -scores for our approach on the three benchmark software is 0.04% for

FIGURE 4: The comprehensive performance of the ICIA approach for the different sized software systems with different cut points.

$c = 200$. We therefore conclude that the performance of the ICIA approach is not affected by the size of the software system.

5. Factors Affecting the Validity of Results

Here, we identify the factors affecting the validity of the results of our empirical case study, and that limits our ability to generalize our findings. Our case study was performed using three java software systems. Although we employed a diverse set of software systems in terms of application domains, further empirical evaluation on systems implemented in other programming languages and different development paradigms would be required to claim generalization and external validity of our results. In addition, the performance of our approach is influenced by the value of c , which is determined by the domain knowledge of the evolving software. As discussed, no unified method exists for setting the value of c automatically. In addition, language habits will affect the experimental results. Word preferences in change requests and source code may be different, and the source code matrices abstracted by LSI and doc2vec are affected by these differences. Such complications may affect the generalization of our results.

6. Related Work

A wide range of research work regarding CIA has been conducted, and this work can be roughly divided into four main types: conventional dependency analysis, software repositories mining, coupling measurement, and execution information analysis [8]. Each of these types can be further divided into subtypes. Conventional dependency analysis includes structural and textual dependency analysis. Subtypes of software repositories mining include historical repositories analysis, and runtime repositories and code repositories analyses. Subtypes of coupling measurement include structural

coupling, conceptual coupling, dynamic function coupling, and relational topic based coupling [13, 16, 35–37]. Execution information analysis include offline and online paradigms [38–41].

Different types of CIA have their own characteristics. Conventional dependency analysis may capture program dependence, such as program control flow, data flow, and syntax or semantic dependence, in a section of code to identify relevant segments of the subject software [37, 42]. Most dependency analysis techniques focus on the source code level, whereas a few focus on the requirement and design level [40, 43]. Requirement and design level CIA techniques depend on highly abstracted models such as unified modeling language diagrams or use case maps [44, 45]. Other efforts applied to CIA techniques based on dependency analysis focus on the source code level. In 1996, Bohner [2] proposed the first dependency-based CIA technique, which focused on the structural dependence of software artifacts, and the approach successfully conducted CIA. Inspired by Bohner's contribution, a number of other techniques were proposed [40, 43, 46] such as reachability graph, reachability matrix, object oriented class, and member dependence graph. Conceptual dependence is typically reflected by the textual elements in the source code such as comments, identifiers, and API names. The conceptual dependence is employed to implement CIA [13, 47]. Dependency-based CIA techniques are easily understood and used. While the use of static analysis for CIA is very close to the activities engaged in by a developer manually searching for code, this approach often overestimates what is relevant to a change request and is prone to returning many false positive results, resulting in a relatively poor precision. To eliminate false positive elements, engineers are usually required to evaluate the results manually to determine which results are, indeed, correct. Therefore, the precision of this method is greatly affected by engineers' domain knowledge of the evolving software.

Recently, a number of CIA techniques have been developed based on mining information from software repositories. These repositories contain some rich historical resources (e.g., source control information and bug reports), runtime resources (e.g., deployment logs), and code resources (e.g., SourceForge and GitHub), which include abundant knowledge of software evolution, such as software architecture, evolutionary decisions, version control, bug reports, and configurations. Some dependencies between software artifacts that cannot be distilled by conventional program analysis techniques can be captured by mining from software repositories. These dependencies are employed to identify cochange phenomena in software repositories, and then to implement CIA. Researchers have conducted CIA by mining the information from multiple version control systems [48–50]. One group [48] mined the association rules from software repositories to conduct CIA. Another [27] utilized both the source code of the current program version and previous versions from software repositories to obtain better impact results, when compared with using the current and previous versions independently. Because software repositories mining uses the actual evolutionary information contained in historical, runtime, and code repositories to generate the EIS, the precision of these techniques is better compared with the other

techniques. However, the results of any of these mining-based techniques are heavily influenced by the quality of the repositories (i.e., the quality of source code, comments, bug reports, and version control information). In other words, CIA based on repositories mining could produce more accurate results if maintainers of repositories improved the quality of information by refining or modifying existing repositories.

Numerous CIA techniques have been proposed based on coupling measurements between software artifacts, which generate ranked lists of impacted artifacts. Among the various subtypes of coupling measurement measures, structural coupling has many representative styles such as coupling between objects (classes), message passing coupling, and information-flow-based coupling. Conceptual coupling is based on measuring how syntax elements (i.e., identifiers and comments) are related to other artifacts [13]. Poshyvanyk et al. [13] conducted a performance comparison between structural coupling and conceptual coupling measures. The results suggested that one of the conceptual coupling methods considered was superior to existing structural coupling measures. Relational topic based coupling of software artifacts uses relational topic models to capture latent topics in software artifacts and their relationships and is therefore similar to conceptual coupling. In fact, relational coupling has been demonstrated to be a good complement to conceptual coupling [47]. Dynamic function coupling between two source code entities (i.e., objects, functions, and sentences) is proposed based on the hypothesis that if the distance between two source code entities is closer in the call stack, they are more likely to be dependent on each other. Then, impact sets can be computed based on this kind of coupling [51].

Because conventional dependency analysis, software repository mining, and coupling measurement techniques make predictions based on an analysis of static artifacts, some information is unknown until runtime due to aliasing and polymorphism. In contrast, execution information analysis, such as with execution traces and system logs, predicts impact sets by analyzing software execution behaviors with respect to a specified change request. The impact results obtained by execution information analysis can be more accurate than the results obtained by other techniques [43, 52]. Typically, one or more change request related test use cases are developed. Then, the use cases are run and execution information is captured either by source code instrumentation or through profiling. Finally, the execution information is analyzed to determine which segments of an execution are related to change requests. Dynamic CIA can be performed online or offline after program execution [39, 52]. Online CIA techniques are employed to alleviate the need to obtain the complete runtime execution information after instrumented execution [39]. The impact set can be calculated for any number of multiple runs of the same program depending on the set of inputs used for inferring the dynamic behavior of the system. The precision of online impact analysis and offline impact analysis has been empirically validated. The results show that the impact sets obtained by online CIA techniques are no more accurate than those of offline CIA techniques, although offline techniques scale better. Dynamic CIA techniques tend to be more precise than static analyses. However, the computational cost of dynamic techniques is greater than that of static techniques

because of the extensive analysis overhead during program execution. Moreover, their EISs often include false-negatives. Conventional dynamic analysis methods tend to collect the execution trace many times for a single use case. The common elements that emerge in all execution traces are identified as an exact result of analysis, and the remaining elements are regarded as noise data. The overall process relies heavily on manual intervention. Another potential limitation of dynamic analysis is that the results produced are heavily influenced by the quality of the use cases and traces generated.

In recent years, numerous attempts have been made at combining multiple existing CIA techniques [27, 35, 53–55]. For example, software repositories mining and coupling measurement have been combined [27]. In addition, static and dynamic analyses have been integrated [53, 56]. The results of these efforts indicate that a combination of multiple techniques can reveal additional useful and important dependence between software elements, and maintainers can use the dependence to propagate changes to related software elements rather than solely relying on conventional static or dynamic analysis, which may fail to capture accurate dependence. Hence, combining existed CIA techniques may be a good choice for improving the performance of conventional impact analysis paradigms.

7. Conclusions

This paper presented an integrated approach for conducting CIA. The approach integrates a *bag-of-words* based technique with a neural network based technique. The empirical results for three open source software systems support our hypothesis that the combined CIA techniques helps to counter the comprehensive deficits of the individual methods as well as improve the precision at the appropriate expense of recall. In future work, we will develop methods for automatically setting the cut point parameter, with the purpose of improving the precision and recall. Moreover, additional experiments employing a greater variety of open source projects will be conducted to verify the universality of the proposed approach.

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, and Data Driven Software Engineering Research Innovation Team of Yunnan Province under Grant no. 2017HC012.

References

- [1] R. Göritzner, *Software Change Impact Analysis*, IEEE Xplore, 1996.

- [2] S. A. Bohner, "Impact analysis in the software change process: a year 2000 perspective," in *proceedings of the International Conference on Software Maintenance*, vol. 703, pp. 42–51, 1996.
- [3] R. J. Turver and M. Munro, "An early impact analysis technique for software maintenance," *Journal of Software Maintenance: Research and Practice*, vol. 6, no. 1, pp. 35–52, 1994.
- [4] P. Rovegård, L. Angelis, and C. Wohlin, "An empirical study on views of importance of change impact analysis issues," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 516–530, 2008.
- [5] S. Black, "Deriving an approximation algorithm for automatic computation of ripple effect measures," *Information and Software Technology*, vol. 50, no. 7-8, pp. 723–736, 2008.
- [6] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: A tool for change impact analysis of java programs," *ACM SIGPLAN Notices*, vol. 39, no. 10, pp. 432–448, 2004.
- [7] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proceedings of the Joint 12th European Software Engineering Conference and 17th ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC-FSE'09*, pp. 111–120, Amsterdam, Netherlands, August 2009.
- [8] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," *Software Testing, Verification and Reliability*, vol. 23, no. 8, pp. 613–646, 2013.
- [9] M. Gethers, H. Kagdi, B. Dit, and D. Poshyvanyk, "An adaptive approach to impact analysis from change requests to source code," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011*, pp. 540–543, Lawrence, KS, USA, November 2011.
- [10] D. Binkley and D. Lawrie, "Information retrieval applications in software maintenance and evolution," *Encyclopedia of Software Engineering*, p. 10, 2009.
- [11] K. Ohara, R. Harper, H. Mentis, A. Sellen, and A. Taylor, "On the naturalness of touchless," *ACM Transactions on Computer-Human Interactions*, vol. 20, no. 1, pp. 1–25, 2013.
- [12] C. S. Corley, K. Damevski, and N. A. Kraft, "Exploring the use of deep learning for feature location," in *Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution, ICSME 2015*, pp. 556–560, Bremen, Germany, October 2015.
- [13] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using information retrieval based coupling measures for impact analysis," *Empirical Software Engineering*, vol. 14, no. 1, pp. 5–32, 2009.
- [14] L. Hattori, D. Guerrero, J. Figueiredo, J. Brunet, and J. Damásio, "On the precision and accuracy of impact analysis techniques," in *Proceedings of the 7th IEEE/ACIS International Conference on Computer and Information Science, IEEE/ACIS ICIS 2008*, pp. 513–518, Portland, OR, USA, May 2008.
- [15] 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.
- [16] L. C. Briand and J. W. Daly, "A unified framework for coupling measurement in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 25, no. 1, pp. 91–121, 1999.
- [17] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1992.
- [18] L. Briand, P. Devanbu, and W. Melo, "Investigation into coupling measures for C++," in *Proceedings of the 1997 IEEE 19th International Conference on Software Engineering*, pp. 412–421, May 1999.
- [19] E. Arisholm, L. C. Briand, and A. Føyen, "Dynamic coupling measurement for object-oriented software," *IEEE Transactions on Software Engineering*, vol. 30, no. 8, pp. 491–506, 2004.
- [20] H. Gall, M. Jazayeri, and J. Krajewski, "CVS release history data for detecting logical couplings," in *Proceedings of the 6th International Workshop on Principles of Software Evolution, IWPSE 2003, in Association with ESEC/FSE 2003*, pp. 13–23, IEEE, Helsinki, Finland, September 2003.
- [21] L. Etzkorn and H. Delugach, "Towards a semantic metrics suite for object-oriented design," in *Proceedings of the Proceedings 34th International Conference on Technology of Object-Oriented Languages and Systems TOOLS 34*, pp. 71–80, Santa Barbara, CA, USA, August 2000.
- [22] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *Proceedings of the ICSM 2006: 22nd IEEE International Conference on Software Maintenance*, pp. 469–478, Philadelphia, PA, USA, September 2006.
- [23] B. Fischer, "Specification-Based Browsing of Software Component Libraries," *Automated Software Engineering*, vol. 7, no. 2, pp. 179–200, 2000.
- [24] Y. Ye and G. Fischer, "Reuse-conductive development environments," *Automated Software Engineering*, vol. 12, no. 2, pp. 199–235, 2005.
- [25] M. Eaddy, A. V. Aho, G. Antoniol, and Y.-G. Guéhéneuc, "CERBERUS: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis," in *Proceedings of the 16th IEEE International Conference on Program Comprehension, ICPC*, pp. 53–62, Amsterdam, The Netherlands, June 2008.
- [26] W. Xiaoyin, Z. Lu, X. Tao, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of the 30th International Conference on Software Engineering 2008, ICSE'08*, pp. 461–470, Leipzig, Germany, 2009.
- [27] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. L. Collard, "Blending conceptual and evolutionary couplings to support change impact analysis in source code," in *Proceedings of the 17th Working Conference on Reverse Engineering, WCRE 2010*, pp. 119–128, Beverly, MA, USA, October 2010.
- [28] C. H. Papadimitriou, P. Raghavan, H. Tamaki, and S. Vempala, "Latent semantic indexing: a probabilistic analysis," *Journal of Computer and System Sciences*, vol. 61, no. 2, pp. 217–235, 2000.
- [29] Q. Le and T. Mikolov, "Distributed Representations of Sentences and Documents," in *Proceedings of the in International Conference on Machine Learning*, vol. 32, pp. 1188–1196, 2014.
- [30] B. T. Bartell, G. W. Cottrell, and R. K. Belew, "Latent semantic indexing is an optimal special case of multidimensional scaling," in *Proceedings of the Fifteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval - SIGIR '95*, pp. 161–167, 1995.
- [31] S. A. Bohner, "Software change impacts - An evolving perspective," in *Proceedings of the 2002 IEEE International Conference on Software Maintenance*, pp. 263–272, October 2002.
- [32] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, ASE'07*, pp. 234–243, Atlanta, Georgia, USA, November 2007.
- [33] M. Ceccarelli, L. Cerulo, G. Canfora, and M. Di Penta, "An eclectic approach for change impact analysis," in *Proceedings*

- of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE 2010, pp. 163–166, Cape Town, South Africa, May 2010.
- [34] J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich, “JRipples: A tool for program comprehension during incremental change,” in *Proceedings of the 13th International Workshop on Program Comprehension, IWPC 2005*, pp. 149–152, St. Louis, MO, USA, May 2005.
 - [35] M. Reville, M. Gethers, and D. Poshyvanyk, “Using structural and textual information to capture feature coupling in object-oriented software,” *Empirical Software Engineering*, vol. 16, no. 6, pp. 773–811, 2011.
 - [36] B. Li, X. Sun, and J. Keung, “FCA-CIA: an approach of using FCA to support cross-level change impact analysis for object oriented Java programs,” *Information and Software Technology*, vol. 55, no. 8, pp. 1437–1449, 2013.
 - [37] X. Sun, B. Li, H. Leung, B. Li, and J. Zhu, “Static change impact analysis techniques: A comparative study,” *The Journal of Systems and Software*, vol. 109, pp. 137–149, 2015.
 - [38] T. Apiwattanapong, A. Orso, and M. J. Harrold, “Efficient and precise dynamic impact analysis using execute-after sequences,” in *Proceedings of the 27th International Conference on Software Engineering (ICSE ’05)*, pp. 432–441, May 2005.
 - [39] B. Breech, A. Danalis, S. Shindo, and L. Pollock, “Online impact analysis via dynamic compilation technology,” in *Proceedings of the Proceedings - 20th IEEE International Conference on Software Maintenance, ICSM 2004*, pp. 453–457, Chicago, IL, USA, September 2004.
 - [40] A. Orso, T. Apiwattanapong, and M. J. Harrold, “Leveraging field data for impact analysis and regression testing,” *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 5, pp. 128–137, 2003.
 - [41] H. Cai and R. Santelices, “A comprehensive study of the predictive accuracy of dynamic change-impact analysis,” *The Journal of Systems and Software*, vol. 103, pp. 248–265, 2015.
 - [42] H. Cai and R. Santelices, “Method-level program dependence abstraction and its application to impact analysis,” *The Journal of Systems and Software*, vol. 122, pp. 311–326, 2016.
 - [43] J. Law and G. Rothermel, “Whole program path-based dynamic impact analysis,” in *Proceedings of the 25th International Conference on Software Engineering*, vol. 6, pp. 308–318, May 2003.
 - [44] L. C. Briand, Y. Labiche, and L. O’Sullivan, “Impact analysis and change management of UML models,” in *Proceedings of the International Conference on Software Maintenance, ICSM 2003*, pp. 256–265, Amsterdam, The Netherlands, September 2003.
 - [45] L. C. Briand, Y. Labiche, L. O’Sullivan, and M. M. Wka, “Automated impact analysis of UML models,” *The Journal of Systems and Software*, vol. 79, no. 3, pp. 339–352, 2006.
 - [46] X. Sun, B. Li, W. Wen, and S. Zhang, “Analyzing impact rules of different change types to support change impact analysis,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 23, no. 3, pp. 259–288, 2013.
 - [47] M. Gethers and D. Poshyvanyk, “Using Relational Topic Models to capture coupling among classes in object-oriented software systems,” in *Proceedings of the 2010 IEEE International Conference on Software Maintenance, ICSM 2010*, pp. 1–10, Timisoara, Romania, September 2010.
 - [48] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, “Mining version histories to guide software changes,” *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.
 - [49] M.-A. Jashki, R. Zafarani, and E. Bagheri, “Towards a more efficient static software change impact analysis method,” in *Proceedings of the 2008 SIGSOFT/SIGPLAN Workshop on Program Analysis for Software Tools and Engineering, PASTE ’08*, pp. 84–90, Atlanta, Georgia, USA, November 2008.
 - [50] G. Canfora and L. Cerulo, “Fine grained indexing of software repositories to support impact analysis,” in *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR ’06, Co-located with the 28th International Conference on Software Engineering, ICSE 2006*, pp. 105–111, Shanghai, China, May 2006.
 - [51] Á. Beszédes, T. Gergely, S. Faragó, T. Gyimóthy, and F. Fischer, “The dynamic function coupling metric and its use in software evolution,” in *Proceedings of the CSMR 2007: 11th European Conference on Software Maintenance and Reengineering*, pp. 103–112, IEEE, Amsterdam, Netherlands, March 2007.
 - [52] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold, “An empirical comparison of dynamic impact analysis algorithms,” in *Proceedings of the 26th International Conference on Software Engineering (ICSE ’04)*, pp. 491–500, May 2004.
 - [53] H. Cai, R. Santelices, and S. Jiang, “Prioritizing Change-Impact Analysis via Semantic Program-Dependence Quantification,” *IEEE Transactions on Reliability*, vol. 65, no. 3, pp. 1114–1132, 2016.
 - [54] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, “Integrated impact analysis for managing software changes,” in *Proceedings of the 34th International Conference on Software Engineering, ICSE 2012*, pp. 430–440, Zurich, Switzerland, June 2012.
 - [55] H. Cai, R. Santelices, and D. Thain, “DiaPro: Unifying dynamic impact analyses for improved and variable cost-effectiveness,” *ACM Transactions on Software Engineering and Methodology*, vol. 25, no. 2, pp. 1–50, 2016.
 - [56] L. Huang and Y. T. Song, “Precise Dynamic Impact Analysis with Dependency Analysis for Object-oriented Programs,” in *Proceedings of the in Acis International Conference on Software Engineering Research, Management Applications*, pp. 374–384, 2007.