Change-Patterns Mapping: A Boosting Way for Change Impact Analysis

Yuan Huang, Jinyu Jiang, Xiapu Luo, Xiangping Chen, *Member, IEEE*, Zibin Zheng, *Senior Member, IEEE*, Nan Jia, and Gang Huang, *Senior Member, IEEE*

Abstract—Change impact analysis (CIA) is a specialized process of program comprehension that investigates the ripple effects of a code change in a software system. In this paper, we present a boosting way for change impact analysis via mapping the historical change-patterns to current CIA task in a cross-project scenario. The change-patterns reflect the coupling dependencies between changed entities in a change set. A traditional CIA tool (such as ImpactMiner) outputs an initial impact set for a starting entity. To boost the traditional CIA tool, our approach retrieves an equivalent entity from various historical change sets for the starting entity. Then, the change-patterns between the equivalent entity and the rest of entities in the change set are mapped to the CIA task at hand. For current CIA task, if an entity in the initial impact set involves the similar change-pattern with the starting entity when comparing with the mapped change-pattern, we will reward the impacted confidence of the entity. Accuracy improvements are observed in the experiments when applying our boosting method to three famous CIA tools, i.e., ImpactMiner, JRipples and ROSE.

Index Terms—Change impact analysis, change-patterns, coupling dependency, boosting method

1 Introduction

Software change plays a vital role in software evolution, agile software development, and maintenance [1]. Change impact analysis (i.e., CIA) is a special topic of program comprehension, where the programmers attempt to understand the ripple effect to a software system when making a particular software change. The software entities in a software system usually have direct or indirect dependencies after a long-time evolution, a slight change may rise ripple effects from an entity to another [2]. As a result, change impact analysis is a particularly complex task in software systems, especially for the ones with years of development history, many of developers, and a multitude of software artifacts including millions of lines of code.

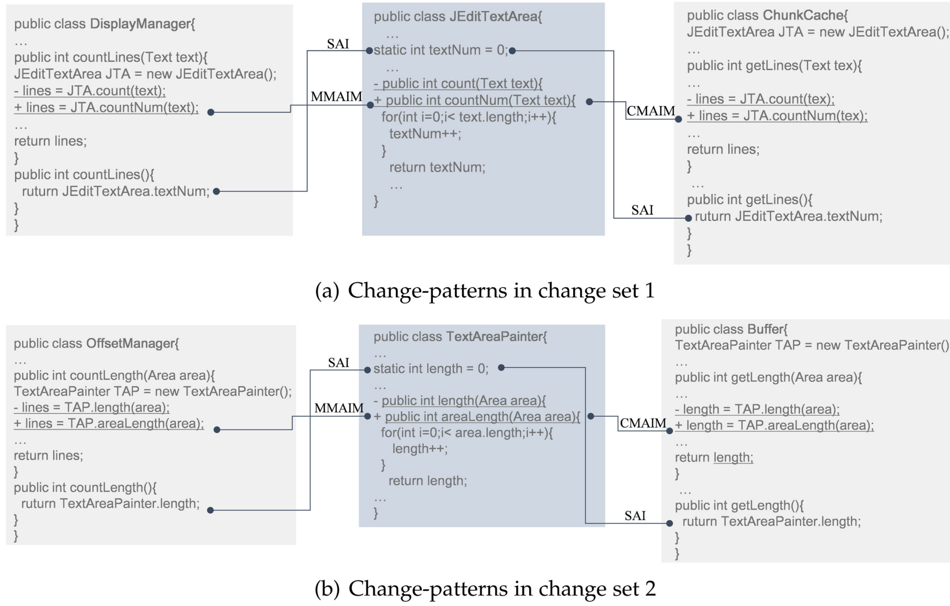
- *Yuan Huang and Zibin Zheng are with the School of Software Engineering, Sun Yat-sen University, Guangzhou 510006, China. E-mail: huangyjn@gmail.com, zhzibin@mail.sysu.edu.cn.*
- *Jinyu Jiang is with the School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou 510006, China. E-mail: exinpie@163.com.*
- *Xiangping Chen is with the Guangdong Key Laboratory for Big Data Analysis and Simulation of Public Opinion, School of Communication and Design, Sun Yat-sen University, Guangzhou 510006, China. E-mail: chenxp8@mail.sysu.edu.cn.*
- *Xiapu Luo is with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong. E-mail: csxluo@comp.polyu.edu.hk.*
- *Nan Jia is with the School of Information Engineering, Hebei GEO University, Shijiazhuang 050031, China. E-mail: jianan_0101@hgu.edu.cn.*
- *Gang Huang is with the Key Laboratory of High Confidence Software Technologies, Ministry of Education, Peking University, Beijing 100871, China. E-mail: hg@pku.edu.cn.*

Manuscript received 19 Mar. 2020; revised 4 Feb. 2021; accepted 9 Feb. 2021. Date of publication 16 Feb. 2021; date of current version 18 July 2022. (Corresponding author: Xiangping Chen.) Recommended for acceptance by S. Apel. Digital Object Identifier no. 10.1109/TSE.2021.3059481

Decades of research efforts have produced a wide spectrum of CIA approaches, ranging from the traditional static and dynamic analysis techniques [3], [4], [5] to the contemporary methods such as those based on coupling dependency analysis [6], [7], [8] and mining software repositories (MSR) [9], [10], [11]. Although ample progress has been made, there still remains much work to be done for further improving the accuracy of the state-of-the-art CIA techniques. For example, the MSR based CIA methods try to uncover important historical dependencies between software entities, e.g., co-changed software entities in the evolutionary history of a project. The operating principle of the MSR based CIA methods has the limitation that they only utilize the historical dependencies of software entities in a project (i.e., within-project scenario), rather than the historical dependencies information from other projects. To overcome this limitation, we generalize the problem to a cross-project scenario, i.e., the dependency information we referenced is not only from the current project, but also from other projects.

For two entities with similar functionality in different projects, it is possible that their changes will have a similar ripple effect in the systems. Since these two entities have similar functionality, they are likely to build similar logical dependencies with the other entities through data interaction or coupling. And because code change can propagate along coupling dependencies (i.e., the changes will first ripple to the entities that have logical dependencies with the change starting entities) [12], as a result, these two entities have similar ripple effects in the systems when they are the change starting entities. We call the coupling dependencies between the change starting entity and the impacted entities as change-patterns (will be presented in Section 4.3). The change-patterns may be used to facilitate the change impact analysis. For example, there are two classes c_{init} and $\tilde{c}_{init}$, and $\tilde{c}_{init}$ builds a change-pattern with

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/



(a) Change-patterns in change set 1

(b) Change-patterns in change set 2

Fig. 1. Different change sets involved in similar change-patterns.

its impacted class $\tilde{c}_n$, and class c_n has similar change-pattern with c_{init} when comparing with one of $\tilde{c}_{init}$ and $\tilde{c}_n$. Then, c_n is likely to be an impacted class by c_{init} when c_{init} is the change starting point.

Based on such an idea, we propose the change-patterns to boost the performance of the traditional CIA tools in this paper. Specifically, we first download a number of change sets from the evolutionary history of more than one hundred projects, and identify the starting changed class (i.e., starting class) in each change set. For a starting class c_{init} in the CIA task at hand, we retrieve an equivalent starting class $\tilde{c}_{init}$ with similar functionality from the historical change sets to c_{init}, and the change-patterns of $\tilde{c}_{init}$ are mapped to the CIA task of c_{init}. Meanwhile, we employ a traditional CIA tool to get an initial impact set for c_{init}. After that, we boost the classes in the initial impact set when they involve similar change-patterns as comparing with the ones between $\tilde{c}_{init}$ and its impacted classes. We chose 7 open-source projects for CIA tasks, which are considered to be high-quality, well evolved and widely used in related studies [13], [14], [15]. We found that the combination of traditional CIA methods and our boosting method is superior to the standalone CIA methods, i.e., the maximum relative improvements of precision for ImpactMiner, JRipples and ROSE are 21.98, 6.52 and 11.81 percent, respectively.

The contributions of our work are shown as follows:

- We propose the concept of change-pattern to measure the coupling dependencies between changed software entities, and using vectorized representation make the change-pattern measurable.
- Due to the metrizability, the change-patterns from different projects can be used to assist the CIA task at hand, and then the MSR based CIA method is generalized to a cross-project scenario by us.
- We demonstrate that there are a lot of similar change-patterns (i.e., 213,554) across projects, which provides a solid foundation for us using the similar change-patterns to boost CIA methods.
- We introduce a set of criteria for evaluating the usefulness of our CIA boosting method. Accuracy improvements are observed when applying our boosting method to ImpactMiner, JRipples and ROSE.

We have uploaded the source code of the boosting method to the Github, and the URL is: https://github.com/CIABoosting/Change-Patterns-Mapping. We describe the basic requirements and steps for running the proposed method. Besides, we upload the key dataset to the Github, such as, the starting classes of the commits, the change-patterns, and the original impacted sets generated by the tools Impactminer, Jripples and Rose, etc. The more detailed information about the key dataset can be found at the URL: https://github.com/CIABoosting/IntermediaryData-for-ChangePatternsMapping.

The rest of this paper is organized as follows. Section 2 shows a motivating scenario and the overview of main steps. Section 3 introduces the traditional change impact analysis. Section 4 describes the process of seeking similar change-patterns from the cross-project scenario. Section 5 introduces the setups of the case study. The result of the case study is presented in Section 6. Section 7 shows a qualitative analysis. Section 8 is the discussion and threats to validity. Section 9 summarizes the related works. Section 10 summarizes our approach and outlines directions of future work.

2 Motivating Scenario and Overview of Main Steps

2.1 Motivating Scenario

This section overviews a motivating scenario with an example that uses the similar change-patterns to determine the impacted entities in CIA task. As shown in Fig. 1, the starting points `JEditTextArea` (Fig. 1a) and `TextAreaPainter` (Fig. 1b) in the two change sets have similar functionality. The source code of these classes is adapted and simplified for presentation purposes. Due to the

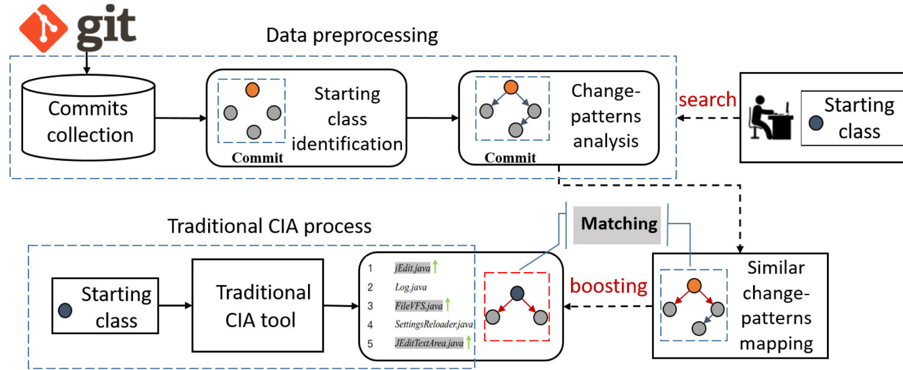


Fig. 2. Overview of main steps.

method renaming (i.e., code lines with underlines are the changed lines, and "-" denotes code removing, and "+" denotes code adding) in `JEditTextArea` and `TextAreaPainter`, `JEditTextArea` transfers the change ripple effect to `ChunkCache` and `DisplayManager`, and `TextAreaPainter` transfers the change ripple effect to `Buffer` and `OffsetManager`.

We found that classes `JEditTextArea` and `TextAreaPainter` have similar ripple effects on the systems, and they have similar change-patterns with their impacted entities. As Fig. 1 shows, `JEditTextArea` builds change-pattern <SAI,[1] MMAIM[2]> with `DisplayManager`, which is the same with the change-pattern between `TextAreaPainter` and `OffsetManager`. Another change-pattern <CMAIM,[3] SAI> between `JEditTextArea` and `ChunkCache` is also similar with the one (i.e., <CMAIM, SAI>) between `TextAreaPainter` and `Buffer`.

The change-patterns can be used to facilitate the change impact analysis. For example, if `JEditTextArea` is a starting point that developer try to change, and he (or she) finds a similar class `TextAreaPainter` to `JEditTextArea` from the historical change set, and the change-patterns between `TextAreaPainter` and its impacted classes (i.e., `Buffer` and `OffsetManager`) are given. Then, the developer can regard the classes that have similar change-patterns (comparing with the change-patterns between `TextAreaPainter` and its impacted classes) to `JEditTextArea` as the potentially impacted entities, i.e., `ChunkCache` and `DisplayManager`.

2.2 Overview of Main Steps

Fig. 2 shows the main steps of the proposed approach. The approach includes three phases: data preprocessing, traditional CIA processing and boosting phase (i.e., the parts connected by dotted arrows in Fig. 2). In the data preprocessing phase, our goal is to identify a starting class for each change set, and analyze the change-patterns between the starting class and the rest of classes in the change set. In the traditional CIA processing phase, we employ the traditional CIA tools to get the initial impact set of a given starting class, and generate a ranking list for the impacted classes according to their change confidence. In the boosting phase, we retrieve a similar starting class from the historical change sets for the given starting class, and map the change-patterns in the historical change sets to the CIA task at hand, and use the change-patterns to boost the rankings of impacted classes obtained by the traditional CIA tools.

To identify a starting class of each change set, we employ the *ISC* tool proposed in our previous study [12], [16], which can identify the root change that causes the change of the rest of entities in a change set. Meanwhile, we employ the traditional CIA tools, ImpactMiner [17], JRipples [18] and ROSE [9], to obtain an initial impact set for a given starting class. To retrieve a similar starting class from the historical change sets for the given starting class, we apply code semantic and syntactic information to measure the similarity between classes.

1. SAI: static variable invoking.
2. MMAIM: method invoking through defining a method member variable.
3. CMAIM: method invoking through defining a class member variable.

3 Change Impact Analysis

3.1 Basic Conceptions

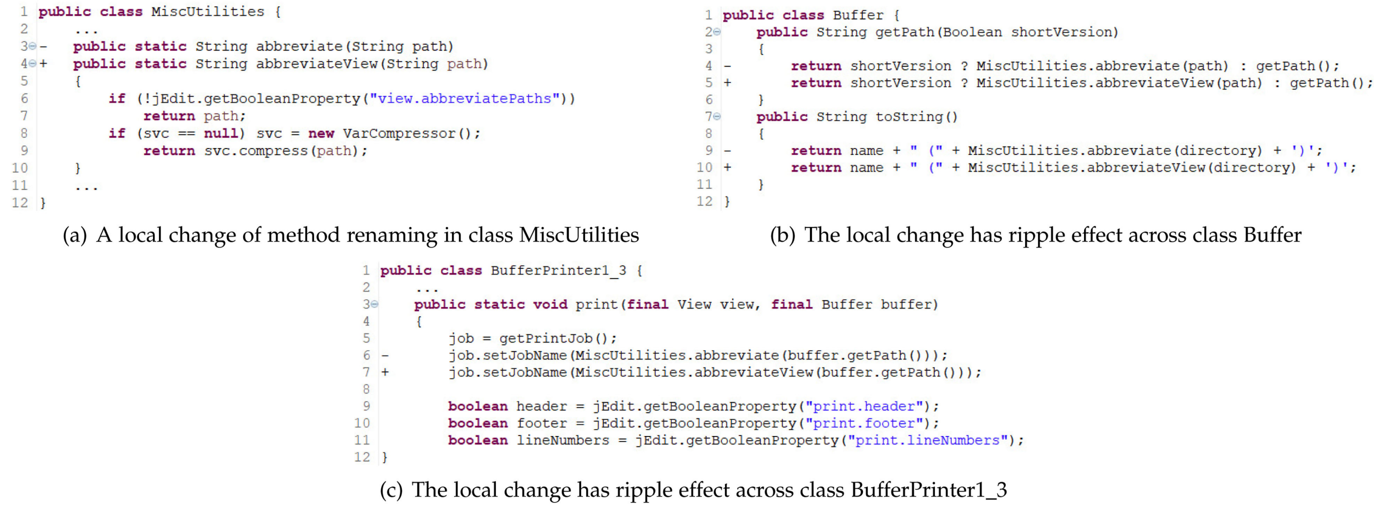
Given a starting entity (such as a class: c_{init}), a traditional change impact analysis approach can infer the software entities (such as a class: c_i) that need to be further changed in the software system [17], [18]. In general, the change impact analysis approach gives a confidence [2], [19] for each entity, that is, the likelihood that further changes be applied to the software entity

$$conf(c_{init}, c_i) = P(c_{init}, c_i). \tag{1}$$

The notation $P(c_{init}, c_i)$ indicates the likelihood of c_i to be changed when c_{init} is changed. Different CIA methods employ different mechanisms to estimate the $P(c_{init}, c_i)$. For example, the default model of ImpactMiner [17] employs latent semantic indexing to estimate the $P(c_{init}, c_i)$, and JRipples [18] analyzes the software entity dependency graph to estimate the $P(c_{init}, c_i)$. ROSE [9] estimates the $P(c_{init}, c_i)$ by analyzing the co-changed entities in the evolutionary history of the project.

Then, for a starting entity c_{init}, a CIA method will output an impact set that contains all the potentially affected software entities

$$CIA(c_{init}) = Impact_Set\{c_1, c_2, \ldots, c_n\}. \tag{2}$$



(a) A local change of method renaming in class MiscUtilities

(b) The local change has ripple effect across class Buffer

(c) The local change has ripple effect across class BufferPrinter1_3

Fig. 3. An example of identifying the starting entity by *ISC*.

Also, we can get a ranking list for the impacted entities according to their change confidence in descending order. Then, several evaluation metrics (e.g., *precision* and *recall*) can be introduced to evaluate the performance of a CIA method with considering a fixed size of the impacted entities in the ranking list, known as cut-off points in [2], [20].

3.2 Ranking Boosting

In order to optimize the performance of the CIA methods, researchers always want to boost the rankings of truly impacted entities in the list as much as possible. For example, Gethers *et al.*, [20] introduce an adaptive approach to improve the rankings of truly impacted entities via combining various boosting mechanisms such as information retrieval, dynamic analysis, etc. Kagdi *et al.*, [2] introduce conceptual couplings to improve the rankings of truly impacted entities.

Intuitively, the co-changed entities in the evolution history of a project can be used as a vital reference for the CIA task. Based on this idea, Zimmermann *et al.* propose to use historical co-changed entities for CIA task [9]. Different from the previous works, we introduce the historical change-patterns to optimize the rankings of truly impacted entities. We call the coupling dependencies from a starting entity to impacted entities as change-pattern. For example, in a historical change set, the change-pattern between a starting entity $\tilde{c}_{init}$ and an impacted entity $\tilde{c}_j$ is denoted by $< \tilde{c}_{init} \overset{cd}{\longrightarrow} \tilde{c}_j >$ when $\tilde{c}_{init}$ builds coupling dependency (i.e., $\overset{cd}{\longrightarrow}$) with $\tilde{c}_j$.

A CIA method can recommend an impact set for a starting entity c_{init}, and the starting entity c_{init} may have coupling dependencies with an entity c_i in the impact set. Then, if c_{init} and c_i have a similar change-pattern when comparing with $\tilde{c}_{init}$ and $\tilde{c}_j$ (which come from historical change set), we can improve the confidence of the impacted entity c_i by rewarding a value of P_{scp}. Then, the final confidence (i.e., $\overline{conf}$) of entity c_i is

$$\overline{conf}(c_{init}, c_i) = P(c_{init}, c_i) + P_{scp}(c_{init}, c_i). \tag{3}$$

After applying P_{scp} to the confidence of c_i, the ranking of c_i in the list will be improved.

4 Seeking Similar Change-Patterns

To determine whether a starting entity c_{init} and its impacted entity c_i have similar change-patterns in historical change sets, we need to first identify the starting entity $\tilde{c}_{init}$ in each historical change set, and then generate the change-pattern $< \tilde{c}_{init} \overset{cd}{\longrightarrow} \tilde{c}_j >$, then we can compare the similarity of change-patterns via $< c_{init} \overset{cd}{\longrightarrow} c_i >$ and $< \tilde{c}_{init} \overset{cd}{\longrightarrow} \tilde{c}_j >$. At last, the similar change-pattern $< \tilde{c}_{init} \overset{cd}{\longrightarrow} \tilde{c}_j >$ can be mapped to the CIA task for c_{init}.

4.1 Identifying Starting Entity

The algorithm of identifying starting entity $\tilde{c}_{init}$ from a historical change set has been proposed in our previous study [12], [16], known as *ISC*, and we employ *ISC* to identify the starting entity in a historical change set in this paper. The starting entity is also called salient entity in [12], [16], and the salient entity is the root change that causes the modification of the rest of entities in a change set, while the rest of entities are the dependency modification along with the salient one. Therefore, the salient entity is the starting entity in this study.

We added a real world example from jEdit (i.e., commit 22,828) project to illustrate how *ISC* identifies the starting entity in a historical change set in this paper, as shown in Figs. 3a, 3b, and 3c. The commit is adapted and simplified for presentation purposes. As Fig. 3a shows, developer updates the method abbreviate() to abbreviateView() (the minus "−" represents the deleted code and the plus "+" represents the added code) in class MiscUtilities, and the code change has ripple effect cross another 2 classes, i.e., Buffer, BufferPrinter1_3 as Figs. 3b and 3c show. Obviously, MiscUtilities is the starting entity.

Because a code change tends to propagate from the initial node of change occurrence to the node that has coupling relationship with the initial node, *ISC* uses structural coupling information between entities as one of the features to identify the starting entity. For example, the coupling relationship of method invocation (i.e., abbreviateView()) between classes MiscUtilities and Buffer. Meanwhile, since the starting entity in a change set is initially modified, starting and non-starting entities differ in the amount of code

involved in the modification. Therefore, *ISC* employs the update degree (e.g., the number of changed code statements) of an entity in a change set to identify the starting entity. For example, the number of changed code statements in MiscUtilities and Buffer are different. In addition, *ISC* also uses the types of commit to distinguish the starting and non-starting entities in a change set. For example, the commit 22,828 is a "Re-Engineering" [12], [16] change set as it involves method renaming. *ISC* will identify the class MiscUtilities as the starting entity via analyzing these features extracted from the commit. *ISC* achieves an accuracy of 87 percent in identifying the starting entity. More introduction of *ISC* can be found in [12], [16].

4.2 Seeking Equivalent Starting Entity

For a current CIA task, we have a starting entity c_{init}, and try to seek a starting entity $\tilde{c}_{init}$ with similar functionality to c_{init} from the historical change sets, and $\tilde{c}_{init}$ is an equivalent starting entity for c_{init}, and the change-patterns caused by $\tilde{c}_{init}$ are used as a reference for the CIA task of c_{init}. Then, we need to compare the similarity of c_{init} and $\tilde{c}_{init}$. In general, we can determine the similarity from the code semantic and syntactic information. Code syntax can catch the program functional information from a perspective of program logic, while code semantics can intuitively catch the program functional information from the word-choices of the source code.

Not all the words in the source code play a positive role for the semantic similarity analysis, some noisy words have no actual semantics and may weaken the original code semantics. To eliminate these noisy words, three preprocessing rules are applied on the original source code: 1) filter out the function words in the source code, such as: *'and'*, *'the'*, *'an'*, etc. 2) filter out the keywords of Java, such as: *'public'*, *'void'*, *'if'*, *'import'*, *'static'*, etc. 3) filter out the letter sequence which does not denote a word, such as: *'tttt'*, *'hhhk'*, *'kkkk'*, etc. In addition, we split the camel-case words into single words. Then, each entity corresponds to a text, e.g., c_{init} and $\tilde{c}_{init}$ corresponds to $T(c_{init})$ and $T(\tilde{c}_{init})$.

Inspired by Ye *et al.* [21], we use the asymmetric similarity to measure the sematic similarity of $T(c_{init})$ and $T(\tilde{c}_{init})$. In study [21], the similarity between a word w in $T(c_{init})$ and the text $T(\tilde{c}_{init})$ is defined as

$$Sim(w, T(\tilde{c}_{init})) = \max_{w \in T(c_{init})} sim(w, w'), \tag{4}$$

where $w' \in T(\tilde{c}_{init})$ and we use WordNet[4] to measure the normalized similarity $sim(w, w')$ between two words. Then, the similarity between two text $T(c_{init})$ and $T(\tilde{c}_{init})$ is

$$Sim(T(c_{init}), T(\tilde{c}_{init})) = \frac{\sum_{w \in T(c_{init})} Sim(w, T(\tilde{c}_{init})) \times idf(w)}{\sum_{w \in T(c_{init})} idf(w)}. \tag{5}$$

Namely, an asymmetric similarity $Sim(T(c_{init}), T(\tilde{c}_{init}))$ is then computed as a normalized, *idf*-weighted sum of similarities between words in $T(c_{init})$.

To measure the syntactic similarity of c_{init} and $\tilde{c}_{init}$, we use the syntax matching method proposed in our previous study [22]. The syntax matching method first parses each code line of a software entity into syntactic tokens via analyzing the abstract syntax tree, then the software entities c_{init} and $\tilde{c}_{init}$ corresponds to token sequences *TokenList1* and *TokenList2*, respectively. To measure the syntactic similarity of two entities, the syntax matching method looks for the longest matching subsequence from the two token sequences, as shown in Algorithm 1.

Algorithm 1. Syntactic Similarity Computing

Input: *TokenList1*: the token sequence of class 1;
TokenList2: the token sequence of class 2;
Output: *SyntaxSimilarity*
Begin
1: **For** i = 0 to *TokenList1.size* **do**:
2: **For** j = 0 to *TokenList2.size* **do**:
3: **If** (*TokenList1*.get(i) == *TokenList2*.get(j)) **do**:
4: $M[i,j] = 1$;
5: **End If**
6: **End For**
7: **End For**
8: **Foreach** $M[n,m]$ **do**:
9: **While** (*true*) **do**:
10: **If**($M[n,m] == 1$) **do**:
11: $subseq_t$ .add($M[n,m]$)
12: $n = n + 1$;
13: $m = m + 1$;
14: *remove*($M[n,m]$);
15: **Else**:
16: *break*;
17: **End If**
18: **End While**
19: **End Foreach**
20: **Foreach** $subseq_t$ **do**:
21: **While** (*true*) **do**:
22: **If** $gap(subseq_t, subseq_{t+1}) < \xi$ **do**:
23: $subseq_t = link(subseq_t, subseq_{t+1})$;
24: $t = t + 1$;
25: $remove(subseq_{t+1})$;
26: **Else**:
27: *break*;
28: **End If**
29: **End While**
30: **End Foreach**
31: $SyntaxSimilarity = \frac{max_length\{subseq_1, subseq_2, \ldots, subseq_t\}}{max_size\{TokenList1, TokenList2\}}$
32: **Return** *SyntaxSimilarity*;
End

In Algorithm 1, we first compare every token in TokenList1 (corresponding to token sequence 1) with every token in TokenList2 (corresponding to token sequence 2). We use a matrix to store the result, and every matrix cell $M[i, j]$ stores the result of the comparison between the relevant token *i* and the relevant token *j*. $M[i, j] = 1$ means the relevant tokens are matched (i.e., identical). Second, we look for the matched cells from upper left corner of the matrix. From the first matched cell, we will further extend up to the first unmatched cell on the main diagonal direction. The continuously matched cells form a subsequence. We continue to find out all the subsequences in the matrix. Third, we check the gap between any two subsequences in the matrix. If the gap is less than a specific number, we link these two subsequences to form a longer one. In the same manner the gap checking is repeated until we traverse all subsequences, then we can find

4. https://wordnet.princeton.edu/

TABLE 1
Common Instances of a Variable Usage

Instances	Description	Abbrev
Method Member Variable	m_{i*} invokes variable a_{i*}	*MMAUA*
	m_{i*} invokes variable a_{j*} of C_j	*MMAIA*
	m_{i*} invokes method m_{j*} of C_j	*MMAIM*
Class Member Variable	m_{i*} invokes variable a_{i*}	*CMAUA*
	m_{i*} invokes variable a_{j*} of C_j	*CMAIA*
	m_{i*} invokes method m_{j*} of C_j	*CMAIM*
Function Parameter	m_{i*} invokes parameter a_{i*}	*FPUA*
	m_{i*} invokes variable a_{j*} of C_j	*FPIA*
	m_{i*} invokes method m_{j*} of C_j	*FPIM*

a longest subsequences at last. Fourth, dividing the length of the longest subsequences by the max length of token sequence 1 and token sequence 2 is the syntactic similarity.

Since semantic (i.e., *semanSimi*) and syntactic similarities (i.e., *syntdeSimi*) are equally important in determining whether two entities are similar, we combine the semantic and syntactic similarities by given them equal weights when calculating the overall similarity (i.e., *weightedSimi*), i.e., $weightedSimi = \frac{1}{2}(semanSimi + syntdeSimi)$. Then, we can retrieve a number of functionality-similar starting entities from the change sets for c_{init}, which is used to assist the CIA task of c_{init}.

4.3 Change-Patterns Identification

The change-patterns refer to the coupling dependencies from the change starting entity to the impacted entities in a change set. Since there can be various coupling dependencies between the starting entity and impacted entities [23], there are many types of change-patterns. According to the study [23] as well as our observation from the object-oriented programming paradigm in previous study [13], we get 21 common instances of coupling dependencies from the source code, and these coupling dependencies may exist at the class level, method level, and variable level.

Coupling Dependency 1. For a change set S, a starting class $C_i \in S$, and an impacted class $C_j \in S$. If C_i and C_j establish coupling relationship at the class level, they satisfy the Class-to-Class coupling dependency, denoted by

$$CR_1 = \{C_i \Rightarrow C_j |\ C_i \in S, C_j \in S\}, \tag{6}$$

where '⇒' denotes the coupling relationship. In Java program syntax, *Inheritance*(*IH*) and *Implementing Interface*(*II*) [24] are the most common cases that satisfy CR_1.

A class C_i includes a sets of variables A_i and methods M_i. Namely, $C_i = \{A_i, M_i\}$, and $A_i = \{a_{i1}, a_{i2}, \ldots, a_{it}\}$, $M_i = \{m_{i1}, m_{i2}, \ldots, m_{ih}\}$. A_i and M_i are the set of variables and methods of C_i, respectively. Similarly, $C_j = \{A_j, M_j\}$, and $A_j = \{a_{j1}, a_{j2}, \ldots, a_{ju}\}$, $M_j = \{m_{j1}, m_{j2}, \ldots, m_{jv}\}$. A_j and M_j are the set of variables and methods of C_j, respectively.

Coupling Dependency 2. A method m_{i*} of C_i uses the class C_j, but it is not intended to define a variable by C_j or call the static variables and methods of C_j, the situation satisfies the Method-to-Class coupling dependency

$$CR_2 = \{C_i \cdot m_{i*} \Rightarrow C_j \mid m_{i*} \in M_i\}. \tag{7}$$

The instances of CR_2 are: *Type-Casting* (*TC*), *Instanceof* (*IO*), *Return Type* (*RT*), and *Exception Throws* (*ET*), etc.

However, in most cases finer-grained coupling dependencies often exist at the method and variable levels. The finer-grained CR_{fg} is defined as

$$CR_{fg} = \{C_i \cdot e_i \Rightarrow C_j \cdot e_j \mid e_i \in (A_i \cup M_i) \ \wedge \ e_j \in (A_j \cup M_j)\}, \tag{8}$$

where e_i and e_j are the variables or methods contained in C_i and C_j, respectively. $e_i \Rightarrow e_j$ denotes that e_i and e_j couple together at the variable or method levels. According to the definition, there are 4 possible instances of the finer granularity CR_{fg}, namely, $C_i \cdot a_{i*} \Rightarrow C_j \cdot a_{j*}$, $C_i \cdot a_{i*} \Rightarrow C_j \cdot m_{j*}$, $C_i \cdot m_{i*} \Rightarrow C_j \cdot a_{j*}$, $C_i \cdot m_{i*} \Rightarrow C_j \cdot m_{j*}$.

In our investigation, the instances of $C_i \cdot a_{i*} \Rightarrow C_j \cdot a_{j*}$ and $C_i \cdot a_{i*} \Rightarrow C_j \cdot m_{j*}$ barely occur in real encoding, while $C_i \cdot m_{i*} \Rightarrow C_j \cdot a_{j*}$ and $C_i \cdot m_{i*} \Rightarrow C_j \cdot m_{j*}$ occur in most cases. Thus, we only consider the latter two in the next.

Coupling Dependency 3. This dependency builds a coupling relation between m_{i*} of C_i and a_{j*} of C_j, namely, a Method-to-Variable dependency

$$CR_3 = \{C_i \cdot m_{i*} \Rightarrow C_j \cdot a_{j*} \mid m_{i*} \in M_i \ \wedge \ a_{j*} \in A_j\}. \tag{9}$$

The most representative instance for CR_3 is *Static Variable Invoking* (*SAI*).

Coupling Dependency 4. This dependency builds a coupling relation between m_{i*} of C_i and m_{j*} of C_j, namely, a Method-to-Method dependency

$$CR_4 = \{C_i \cdot m_{i*} \Rightarrow C_j \cdot m_{j*} \mid m_{i*} \in M_i \ \wedge \ m_{j*} \in M_j\}. \tag{10}$$

CR_4 builds coupling relation at the method level. E.g., *Static Method Invoking* (*SMI*); *Construction Method Invoking* (*CMI*).

In addition to the above mentioned instances for each coupling dependencies, there is a most common instance in real encoding [25]. This kind of instance uses C_j to define an variable a_{i*} contained in C_i, where a_{i*} may be a *Method Member Variable* or *Class Member Variable* or *Function Parameter*. Generally, the defined variable a_{i*} is usually used in a method m_{i*} of C_i, and there are three usages: ① using a_{i*} directly in a certain method m_{i*} of C_i. Because a_{i*} is a variable declared by C_j and a_{i*} is used in a certain method m_{i*} of C_i, thus we can regard the coupling relation between C_i and C_j as Method-to-Class. E.g., the instances *MMAUA*, *CMAUA* and *FPUA* satisfy this case, and detailed description is shown in Table 1; ② invoking the variable a_{j*} of C_j via a_{i*} (because a_{i*} is declared by C_j), and C_i and C_j builds a Method-to-Variable coupling relation. As Table 1 shows, the instances *MMAIA*, *CMAIA* and *FPIA* satisfy this case; ③ invoking the method m_{j*} of C_j via a_{i*}, and C_i and C_j builds a Method-to-Method coupling relation. As Table 1 shows, the instances *MMAIM*, *CMAIM* and *FPIM* satisfy this case.

We have classified the coupling dependencies into 4 types at class, variable and method levels. Table 2 shows the 4 coupling dependencies and their corresponding formalization expressions and 18 instances. To identify the coupling dependencies that two classes satisfy, we get the variable and method definitions and the variable and method invocations in the two classes via analyzing their abstract syntax trees by applying the JavaParser[5] APIs.

5. https://javaparser.org/

TABLE 2
Coupling Dependencies

Name	Coupling Dependencies	Formalization	Instances
CR_1	Class-to-Class	$C_i \Rightarrow C_j$	*IH*, *II*
CR_2	Method-to-Class	$C_i \cdot m_{i*} \Rightarrow C_j$	*TC*, *IO*, *RT*, *ET*, *MMAUA*, *CMAUA*, *FPUA*
CR_3	Method-to-Variable	$C_i \cdot m_{i*} \Rightarrow C_j \cdot a_{j*}$	*SAI*, *MMAIA*, *CMAIA*, *FPIA*
CR_4	Method-to-Method	$C_i \cdot m_{i*} \Rightarrow C_j \cdot m_{j*}$	*SMI*, *CMI*, *FPIM*, *MMAIM*, *CMAIM*

After getting the coupling dependencies between the starting class C_i and an impacted class C_j, the change-pattern between C_i and C_j can be represented by a vector $V_{cp}(C_i, C_j)$ with 18 dimensions

$$V_{cp}(C_i, C_j) = < IH, II, TC, IO, \ldots, CMAIM >. \tag{11}$$

The value of each dimension in $V_{cp}(C_i, C_j)$ is the number of a certain coupling dependency that the two classes satisfy.

4.4 Change-Patterns Mapping

For a starting entity c_{init} in the current CIA task, we can retrieve t equivalent starting entities $\tilde{c}_{init}$ from t historical change sets. Then, the change-patterns between $\tilde{c}_{init}$ and the rest of the entities (i.e., $\tilde{c}_j$) in a change set can be mapped to the current CIA task to boost the rankings of the entities that are truly impacted by c_{init}.

We employ Jaccard similarity [26] to measure the similarity of change-patterns. Jaccard similarity is suitable to deal with the case in this study, because the positive matching of the coupling dependency on a certain dimension (a certain dimension in two vectors have the same value) is more meaningful than negative matching (a certain dimension in two vectors have different values). In the similarity measurement, the number of negative matching is considered to be insignificant, and Jaccard similarity can ignore the influence of negative matching.

We first find t equivalent starting entities from t historical change sets for a starting entity c_{init} according to the code semantic and syntactic similarities. Then, for a change-pattern $< c_{init} \xrightarrow{cd} c_i >$, we try to find its similar change-patterns $< \tilde{c}_{init} \xrightarrow{cd} \tilde{c}_j >$ from the t change sets via the Jaccard similarity

$$JSimi(V_{cp}(c_{init}, c_i), V_{cp}(\tilde{c}_{init}, \tilde{c}_j)) = \frac{V_{cp}(c_{init}, c_i) \cap V_{cp}(\tilde{c}_{init}, \tilde{c}_j)}{V_{cp}(c_{init}, c_i) \cup V_{cp}(\tilde{c}_{init}, \tilde{c}_j)}, \tag{12}$$

where, $i \in 1, \ldots, n, j \in 1, \ldots, m$. When the value of $JSimi(V_{cp}(c_{init}, c_i), V_{cp}(\tilde{c}_{init}, \tilde{c}_j))$ is greater than η, we use the change-pattern $< \tilde{c}_{init} \xrightarrow{cd} \tilde{c}_j >$ to boost the final confidence of entity c_i. Then, the $P_{scp}(c_{init}, c_i)$ in the formula (3) is

$$P_{scp}(c_{init}, c_i) = \sum_{1}^{k} JSimi(V_{cp}(c_{init}, c_i), V_{cp}(\tilde{c}_{init}, \tilde{c}_j)), \tag{13}$$

where, k is the number of change-patterns that their $JSimi(V_{cp}(c_{init}, c_i), V_{cp}(\tilde{c}_{init}, \tilde{c}_j))$ values are greater than η. The final confidence for c_i is

$$\overline{conf}(c_{init}, c_i) = P(c_{init}, c_i) + \sum_{1}^{k} JSimi(V_{cp,k}(c_{init}, c_i), V_{cp}(\tilde{c}_{init}, \tilde{c}_j)). \tag{14}$$

We will determine the values of t and η via the comparison experiments, and the optimal values of t and η are 20 and 0.8. More discussion regarding the t and η can be found in Sections 8.1 and 8.2.

5 Case Study Design

To have a meaningful evaluation, we outline 3 change impact analysis methods (namely, ImpactMiner, JRipples and ROSE) after applying our boosting mechanism. Besides, we empirically investigate how our approach performs by comparing it with a random approach.

5.1 Datasets

To collect the historical change sets, we downloaded 182 projects from Github[6] and SourceForge,[7] the historical change set is also known as commit in most studies [9], [27]. The projects are selected based on the following rules: First, we require that all the selected projects are open source and implemented in Java. Second, we selected projects that have active updates in its evolution history. Third, we selected projects belonging to different domains including graphics editor, text editor, game, programming frame, middleware system, etc. Combining all the above factors, we selected the 182 projects from different domains. Not all the commits are useful due to the number of classes contained in the commits. Since our approach tries to find the similar change-patterns from the historical commits, we filter out the commits with less than 2 classes files (including the commits containing non-source code files). After applying the filter rule, there are 94,778 commits in total. A local repository is built to save these commits, and we use the method described in previous sections to identify the starting class in each commit, and generate the change-patterns for each commit.

We conduct change impact analysis for 7 Java open-source projects in our evaluation. The selected projects are commonly evaluated projects in most of the software engineering related research [13], [14], [15], i.e., FreeCol [28], HSQLDB [29], JAMWiki [30], jEdit [31], JHotDraw [32], Makagiga [33], OmegaT [34]. Since these 7 projects have more than ten years evolution history, they have a relatively high number of commits. The detailed information of the projects is shown in Table 3.

6. https://github.com/
7. https://sourceforge.net/

TABLE 3
Projects Used in the Case Study

Projects	Version	Commits Amount	Projects	Version	Commits Amount
FreeCol	0.3-1.0	2,136	HSQLDB	1.80-2.33	1,137
JAMWiki	0.5-1.3	899	jEdit	4.0-4.5	1,665
JHotDraw	7.1-7.5	378	Makagiga	2.0-4.12	4,072
OmegaT	2.05-3.54	837			

Note that the 7 datasets are included in the 182 projects, because the change-patterns in previous commits of the 7 projects can be used to guide the current CIA tasks, just like the traditional MSR based CIA methods, i.e., within-project scenario.

5.2 Evaluation Methodology

We first employ *ISC* [12] to identify a starting class for each commit in Table 3, then the other classes in the same commit are regarded as the truly impacted ones by the starting class (ground truth). After that, we use the 3 traditional methods (i.e., ImpactMiner, JRipples and ROSE) to analyze the impact set in each project system for each starting class, and generate a ranking list of the impacted classes. The three CIA tools are selected owing to their open-source availability. At last, we apply our boosting mechanism, and try to improve the rankings of the truly impacted classes in the list.

When employing ImpactMiner for the CIA tasks, we use commit message as textual information to estimate an impact set via IR technique, known as IR_{CR} component in ImpactMiner, then we use the second component (i.e., $Hist_{seed}$) of ImpactMiner to mine the past commits of the starting class to estimate the impact set. To apply JRipples to the CIA tasks, we download a complete code of the version the commit locates in, then we can employ JRipples to generate the class dependencies for this version. For the starting class in a commit, the "next" classes given by JRipples are regarded as impact set. JRipples is a plugin of Eclipse, and we use the default setting of JRipples for CIA tasks. Fig. 4 shows the operation interface of ImpactMiner and JRipples. To run ROSE, we download all the commits in the evolutionary histories of the 7 projects, then we utilize ROSE to detect the impact set by analyzing the evolutionary histories of the projects. We also use the default setting of ROSE for CIA tasks.

Recall and Precision are widely used to evaluate the performance of impact analysis methods [2], [20], [23], which can evaluate the difference between the impact set estimated by tools and the actual impact set. Given a starting class c_{init}, E_{imp} represents the impact set estimated by tools, and R_{imp} represents the actual impact set. Then, the recall and precision are defined as

$$recall = \frac{|E_{imp} \cap R_{imp}|}{|R_{imp}|} \times 100\% \tag{15}$$

$$precision = \frac{|E_{imp} \cap R_{imp}|}{|E_{imp}|} \times 100\%. \tag{16}$$

In many cases, we will evaluate the performance of an impact analysis approach under a given cut-off point [2] (e.g., 5, 10, 20, 30, 40, etc.). That is, we rank the estimated impacted classes in descending order, and a given cut-off point will derive an estimated impact set (E.g., a cut-off point of 5 indicates that the estimated impact set with our approach contains 5 entities.), then we can evaluate the recall and precision of the impact analysis approach under such an estimated impact set. *f-measure* is the harmonic mean of *precision* and *recall*, which can be also used to evaluate the performance of the CIA tools

$$f - measure = \frac{2 * precision * recall}{precision + recall} \times 100\%. \tag{17}$$

6 Results Analysis

In this paper, we focus on the performance of applying our boosting method to the traditional CIA methods, and we investigate the likely benefits of our boosting method for the CIA tasks. The cut-off points in this study are 5, 10, 20, 30 and 40, and the η is 0.8, and the value of t for ImpactMiner, JRipples and ROSE is 20, and we will discuss the t and η choice effect in Sections 8.1 and 8.2.

Table 4 presents the results for ImpactMiner as well as the results after applying our boosting method, where "Delta Improv" represents the delta improvement of the boosting method versus the traditional CIA tool under a specific cut-

(a) ImpactMiner

(b) JRipples

Fig. 4. Change impact analysis tools.

TABLE 4
Accuracy Boosting for *ImpactMiner* on all Datasets Using Various Cut-Off Points

Datasets	Approach	Recall(%)					Precision(%)				
		5	10	20	30	40	5	10	20	30	40
FreeCol	ImpactMiner	9.51	16.81	27.20	36.79	45.73	7.33	6.75	5.67	5.27	4.95
	Boosting	15.27	22.76	33.32	41.15	48.27	12.38	9.55	7.07	5.91	5.23
	Delta Improv	+5.76	+5.95	+6.12	+4.36	+2.54	+5.05	+2.80	+1.40	+0.64	+0.28
HSQLDB	ImpactMiner	9.49	16.13	24.66	29.82	34.60	10.90	9.45	7.41	6.15	5.48
	Boosting	15.09	21.68	29.89	34.92	38.21	16.85	12.64	9.35	7.45	6.24
	Delta Improv	+5.60	+5.55	+5.23	+5.10	+3.61	+5.95	+3.19	+1.94	+1.3	+0.76
JAMWiki	ImpactMiner	14.76	20.18	27.24	30.07	31.11	11.69	8.62	6.13	4.67	3.67
	Boosting	16.26	22.65	28.55	30.54	31.50	13.81	9.88	6.59	4.85	3.75
	Delta Improv	+1.5	+2.47	+1.31	+0.47	+0.39	+2.21	+1.26	+0.46	+0.18	+0.08
jEdit	ImpactMiner	14.71	23.26	30.82	34.70	37.71	11.44	10.22	7.13	5.56	4.58
	Boosting	19.79	26.12	33.21	36.02	38.50	16.46	11.30	7.74	5.79	4.72
	Delta Improv	+5.08	+2.86	+2.39	+1.32	+0.79	+5.02	+1.08	+0.61	+0.23	+0.14
JHotDraw	ImpactMiner	7.32	12.20	19.08	23.86	27.38	9.26	8.16	6.62	5.49	4.52
	Boosting	9.53	14.26	20.80	24.27	27.67	11.47	9.41	7.24	5.59	4.61
	Delta Improv	+2.21	+2.06	+1.72	+0.41	+0.29	+2.21	+1.25	+0.62	+0.10	+0.09
Makagiga	ImpactMiner	19.38	28.11	38.94	46.16	51.89	12.52	9.34	6.67	5.46	4.66
	Boosting	26.66	34.57	44.19	50.28	54.69	17.59	11.89	7.78	6.02	4.99
	Delta Improv	+7.28	+6.46	+5.25	+4.12	+2.80	+5.07	+2.55	+1.11	+0.56	+0.33
OmegaT	ImpactMiner	26.24	38.09	48.77	53.23	56.39	16.53	12.26	8.05	6.11	4.91
	Boosting	35.36	43.71	51.00	54.91	57.93	21.98	14.21	8.48	6.30	5.06
	Delta Improv	+9.12	+5.62	+2.23	+1.68	+1.54	+5.45	+1.95	+0.43	+0.19	+0.15
Avg Improv (%)		5.22	4.42	3.46	1.07	1.71	4.41	2.01	0.94	0.46	0.26
Rlt Improv (%)		36.03	19.99	11.18	2.94	4.20	21.98	14.21	8.48	6.30	5.06

off point. We found that applying our boosting method on ImpactMiner improves the performance over any dataset comparing with the standalone ImpactMiner. The maximum improvement for recall is 9.12 percent when the cut-off point is 5 on the dataset OmegaT, while the minimum improvement for recall is 0.29 percent when the cut-off point is 40 on the dataset JHotDraw. Meanwhile, the maximum improvement for precision is 5.59 percent when the cut-off point is 5 on the dataset HSQLDB, while the minimum improvement for precision is 0.08 percent when the cut-off point is 40 on the dataset JAMWiki. The average improvements (i.e., Avg Improv, also called absolute improvement) for recall and precision are 5.22 percent and 4.41 percent at the cut-off point of 5, which means that our method can improve the rankings of the actual impacted classes. Table 4 also shows the relative improvements of the recall or precision achieved by our boosting method when comparing with those of ImpactMiner, i.e., Rlt Improv. The maximum relative improvements of recall and precision for ImpactMiner are 36.03 and 21.98 percent at the cut-off point of 5.

Table 5 presents the recall and precision after applying the boosting method on JRipples. The results indicate a positive improvement for the recall in most of datasets with the cut-off point of 5 when applying the boosting method. When setting the cut-off points as 10 and 20, the recall also shows a positive improvement in most of the datasets except for the datasets JHotDraw and OmegaT, where the recall shows a slight descent (e.g., 0.44 percent of decline on JHotDraw). When setting the cut-off point as 30, the recall shows a positive improvement. When setting the cut-off point as 40, we can also observe some recalls remain unchanged (e.g., in FreeCol, JAMWiki, JHotDraw, Makagiga, OmegaT), this is because the number of impacted entities estimated by JRipples is less than or equal to 40, and our boosting method works on an initial impact set estimated by JRipples, then our method cannot be superior to JRipples when the total impacted entities estimated by JRipples is less than the cut-off points, i.e., 40. Table 5 also presents the precision for JRipples as well as the precision after applying the boosting method. The results show a positive improvement for the precision at the cut-off points of 5, 10, 20. Table 5 also shows the relative improvements (i.e., Rlt Improv). The maximum relative improvements of recall and precision for JRipples are 6.48 and 6.52 percent at the cut-off point of 5.

At the same time, we can observe that the average boosting improvements on ImpactMiner are more significant than those on JRipples. This is because the initial impact set estimated by ImpactMiner contains more truly impacted classes, and then our method has more chance to boost the sorting positions of the actual impact class forward in the ranking list.

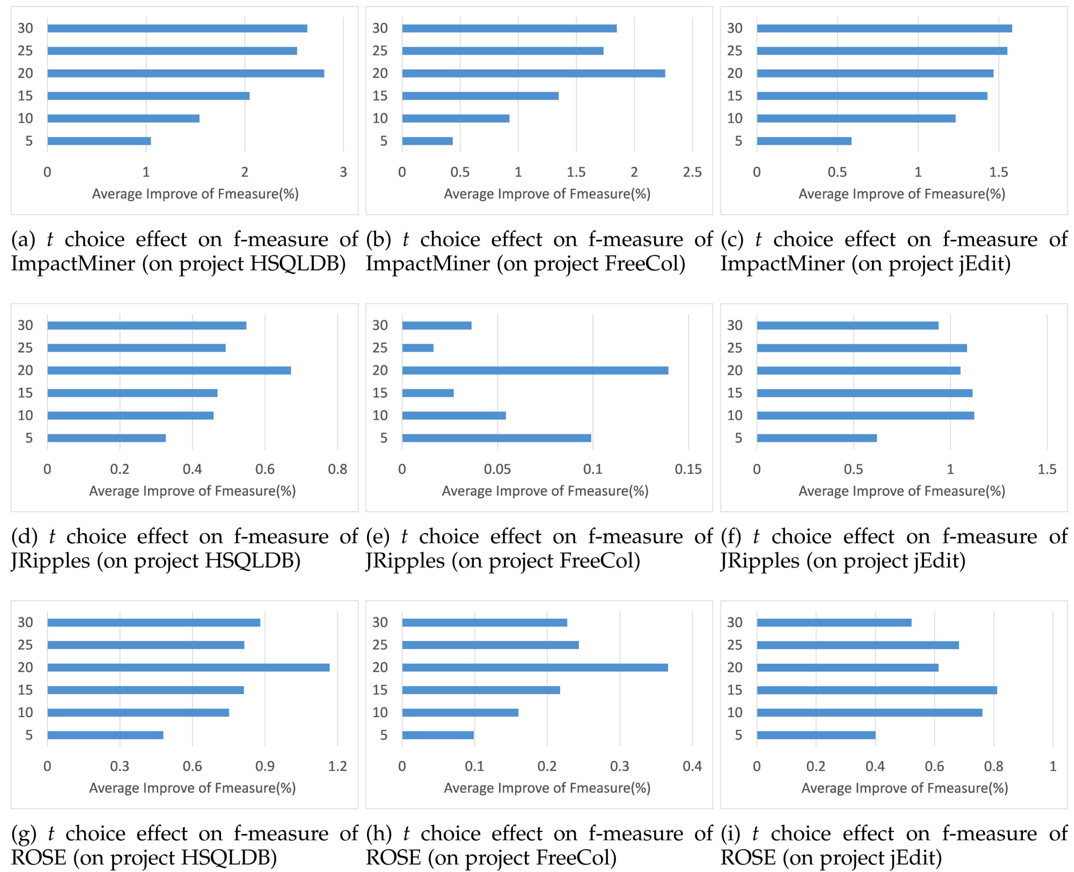
Table 6 presents the recall and precision for ROSE before and after applying the boosting method, and the maximum relative improvements of recall and precision for ROSE are 12.39 and 11.81 percent at the cut-off point of 5. We can observe that most of the recall and precision are improved when we apply the boosting method to ROSE, while there are some exceptions, such as the recall and precision after applying the boosting method at the cut-off points of 5 and 10 on project FreeCol. In this case, the boosting method shows a negative effect on the identification of the actual impacted classes of ROSE. However, with the improvement of the cut-off point, the boosting method shows a positive

TABLE 5
Accuracy Boosting for *JRipples* on all Datasets Using Various Cut-Off Points

Datasets	Approach	Recall(%)					Precision(%)				
		5	10	20	30	40	5	10	20	30	40
FreeCol	JRipples	16.40	25.50	31.13	35.01	35.49	15.06	11.57	7.06	5.26	4.01
	Boosting	17.38	26.18	31.77	35.08	35.49	14.98	11.66	7.19	5.28	4.01
	Delta Improv	+0.98	+0.68	+0.64	+0.07	+0.00	–0.07	+0.09	+0.13	+0.02	+0.00
HSQLDB	JRipples	15.63	22.49	31.41	37.92	40.79	18.23	13.96	10.12	8.21	6.81
	Boosting	16.84	24.96	32.99	38.18	40.89	19.71	15.46	10.60	8.28	6.77
	Delta Improv	+1.21	+2.47	+1.58	+0.26	+0.10	+1.48	+1.50	+0.48	+0.07	–0.04
JAMWiki	JRipples	24.62	32.48	36.10	37.12	37.18	15.79	11.11	6.46	4.48	3.38
	Boosting	29.63	33.65	36.24	37.12	37.18	20.58	12.05	6.52	4.48	3.38
	Delta Improv	+5.01	+1.17	+0.14	0.00	0.00	+4.79	+0.94	+0.06	0.00	0.00
jEdit	JRipples	26.10	35.64	44.83	51.26	54.32	23.31	16.43	11.02	8.83	7.22
	Boosting	28.26	37.60	46.09	51.85	54.56	24.48	17.40	11.37	8.93	7.23
	Delta Improv	+2.16	+1.96	+1.26	+0.59	+0.24	+1.17	+0.97	+0.35	+0.10	+0.01
JHotDraw	JRipples	11.63	12.72	12.72	13.17	13.17	13.10	8.28	4.14	2.99	2.24
	Boosting	11.63	12.28	12.72	12.72	13.17	13.10	7.59	4.14	2.76	2.24
	Delta Improv	0.00	–0.44	0.00	–0.35	0.00	0.00	–0.69	0.00	–0.23	0.00
Makagiga	JRipples	24.33	27.57	29.47	30.28	30.57	14.77	8.67	4.74	3.27	2.48
	Boosting	25.05	27.90	29.49	30.39	30.57	15.40	8.87	4.75	3.29	2.48
	Delta Improv	+0.72	+0.33	+ 0.02	+0.11	0.00	+0.63	+0.30	+0.01	+0.02	0.00
OmegaT	JRipples	33.55	35.91	36.10	36.10	36.10	20.99	11.41	5.73	3.82	2.86
	Boosting	33.35	35.91	36.10	36.10	36.10	20.92	11.41	5.73	3.82	2.86
	Delta Improv	–0.20	0.00	0.00	0.00	0.00	–0.07	0.00	0.00	0.00	0.00
Avg Improv (%)		1.41	0.88	0.52	0.08	0.05	1.13	0.43	0.15	0.00	0.00
Rlt Improv (%)		6.48	3.20	1.64	0.23	0.14	6.52	3.70	2.13	0.00	0.00

TABLE 6
Accuracy Boosting for *ROSE* on all Datasets Using Various Cut-Off Points

Datasets	Approach	Recall(%)					Precision(%)				
		5	10	20	30	40	5	10	20	30	40
FreeCol	ROSE	22.09	33.04	44.19	50.26	54.00	17.39	13.33	9.11	7.02	5.75
	Boosting	19.53	31.60	46.33	52.50	56.13	15.45	12.76	9.58	7.37	6.01
	Delta Improv	–2.56	–1.44	+2.14	+2.24	+2.13	–1.94	–0.57	+0.47	+0.35	+0.26
HSQLDB	ROSE	16.12	23.14	30.69	36.15	40.07	16.23	12.56	8.93	7.22	6.15
	Boosting	16.77	25.23	34.12	39.09	42.72	16.90	14.03	10.17	7.99	6.65
	Delta Improv	+0.65	+2.09	+3.43	+2.94	+2.65	+0.67	+1.47	+1.24	+0.77	+0.50
JAMWiki	ROSE	34.29	45.19	55.01	61.57	65.26	28.73	19.46	12.06	9.19	7.35
	Boosting	37.96	49.33	60.32	64.13	66.91	31.81	21.54	13.42	9.63	7.59
	Delta Improv	+3.67	+4.14	+5.31	+2.56	+1.65	+3.08	+2.08	+1.36	+0.44	+0.24
jEdit	ROSE	26.62	39.52	50.98	57.95	61.74	22.11	16.76	11.22	8.69	7.03
	Boosting	31.72	42.76	53.31	59.38	63.46	25.76	18.31	11.62	8.85	7.25
	Delta Improv	+5.1	+3.24	+2.33	+1.43	+1.72	+3.65	+1.55	+0.40	+0.16	+0.22
JHotDraw	ROSE	27.64	37.52	45.48	50.87	53.67	23.56	16.53	10.30	7.83	6.22
	Boosting	31.38	41.09	49.78	53.02	55.32	27.12	18.49	11.42	8.22	6.48
	Delta Improv	+3.74	+3.57	+4.30	+2.15	+1.65	+3.56	+1.96	+1.12	+0.39	+0.24
Makagiga	ROSE	22.10	29.20	37.24	42.07	45.94	14.15	9.78	6.51	5.00	4.13
	Boosting	27.51	34.07	40.83	44.56	48.00	18.03	11.61	7.23	5.35	4.34
	Delta Improv	+5.41	+4.87	+3.59	+2.49	+2.06	+3.88	+1.83	+0.72	+0.35	+0.21
OmegaT	ROSE	31.91	40.74	48.76	51.75	54.26	21.90	14.44	9.02	6.49	5.08
	Boosting	38.30	44.45	51.03	53.30	55.11	26.03	15.71	9.37	6.63	5.16
	Delta Improv	+6.39	+3.71	+2.27	+1.55	+0.85	+4.13	+1.27	+0.35	+0.14	+0.08
Avg Improv (%)		3.2	2.88	3.33	2.19	1.82	2.43	1.37	0.81	0.37	0.25
Rlt Improv (%)		12.39	8.13	7.48	4.38	3.39	11.81	9.32	8.44	5.03	4.2



(a) t choice effect on f-measure of ImpactMiner (on project HSQLDB) (b) t choice effect on f-measure of ImpactMiner (on project FreeCol) (c) t choice effect on f-measure of ImpactMiner (on project jEdit)

(d) t choice effect on f-measure of JRipples (on project HSQLDB) (e) t choice effect on f-measure of JRipples (on project FreeCol) (f) t choice effect on f-measure of JRipples (on project jEdit)

(g) t choice effect on f-measure of ROSE (on project HSQLDB) (h) t choice effect on f-measure of ROSE (on project FreeCol) (i) t choice effect on f-measure of ROSE (on project jEdit)

Fig. 5. t choice effects on the performance of ImpactMiner, JRipples, and ROSE.

effect on the CIA task of ROSE. Meanwhile, the average recall and precision on all the datasets are improved.

Based on these results, we conclude that the combination of traditional CIA methods and our boosting method is shown to be superior to the standalone CIA methods in most cases, and our boosting method does improve the average accuracy of the traditional CIA methods in the task of change impact analysis on multiple datasets.

7 Qualitative Analysis

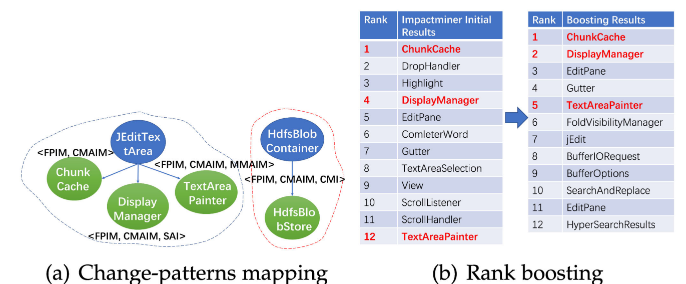
In this section, we try to systematically analyze where do our boosting method benefits come from. First, we illustrate several examples regarding how the change-patterns affect the positions of the impacted classes in the ranking list. Second, we empirically analyze the frequency of similar change-patterns in historical change sets and further provide an empirical evidence to support our boosting method. Third, we empirically analyze how many similar change-patterns coming from across-project and within-project cases can be utilized by our boosting method.

7.1 Ranking Boosting Examples

We perform a qualitative analysis on the results in order to better understand where our boosting method benefits come from. Specifically, we analyze several cases with comparing the initial ranking lists by the CIA tools (e.g., Impactminer) and the reordered ranking lists by our boosting method.

Fig. 6 shows an example of applying our boosting method on Impactminer. In Fig. 6a, the blue dotted circle is the commit (#12780) from jEdit and the red dotted circle is the commit (#1776) from Flyway (Flyway is included in the 182 projects). Class `JEditTextArea` is the change starting point, and classes `ChunkCache`, `DisplayManager` and `TextAreaPainter` are the actual impacted classes by `JEditTextArea`. In Fig. 6b (left side), Impactminer gives an initial list of the classes impacted by `JEditTextArea`. We can observe that the ranks of the three actual impacted classes are 1, 4, and 12. Then, we apply our boosting method to the initial result given by Impactminer, and the boosting method recommends the change-pattern (i.e., $<FPIM, CMAIM, CMI>$) between `HdfsBlobContainer` and `HdfsBlobContainer` to the CIA task. Since this change-pattern is similar with the ones between `JEditTextArea` and its actual impacted classes, our boosting method improves the ranks of the three classes in the list, as shows in In Fig. 6b (right side).

Fig. 7 shows an example (coming from jEdit, commit #2684 (blue dotted circle) and OKhttp, commit #2132 (red

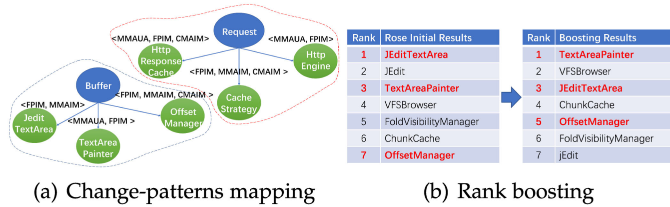


Rank	Impactminer Initial Results
1	ChunkCache
2	DropHandler
3	Highlight
4	DisplayManager
5	EditPane
6	ComleterWord
7	Gutter
8	TextAreaSelection
9	View
10	ScrollListener
11	ScrollHandler
12	TextAreaPainter

Rank	Boosting Results
1	ChunkCache
2	DisplayManager
3	EditPane
4	Gutter
5	TextAreaPainter
6	FoldVisibilityManager
7	jEdit
8	BufferIORequest
9	BufferOptions
10	SearchAndReplace
11	EditPane
12	HyperSearchResults

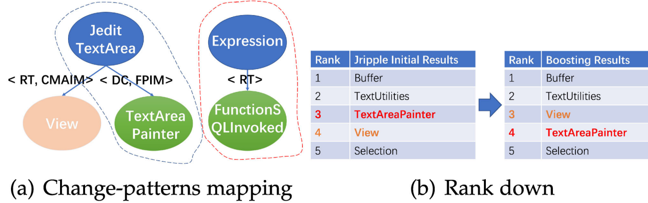
(a) Change-patterns mapping (b) Rank boosting

Fig. 6. A boosting case for Impactminer.



(a) Change-patterns mapping (b) Rank boosting

Fig. 7. A boosting case for ROSE.



(a) Change-patterns mapping (b) Rank down

Fig. 8. A bad case for JRipples.

dotted circle)) of applying our boosting method on ROSE. As Fig. 7a shows, `Buffer` is the change starting point, and it builds a change-pattern of $<$ *MMAUA*, *FPIM* $>$ with `TextAreaPainter`, and a change-pattern of $<$ *FPIM*, *MMAIM*, *CMAIM* $>$ with `OffsetManager`, and a change-pattern of $<$ *FPIM*, *MMAIM* $>$ with `JeditTextArea`. Our boosting method recommends the change-patterns of commit #2132 from project OKhttp to the CIA task, and both the change-patterns between `Request` and `HttpResponseCache`, `Request` and `HttpEngine`, are similar with the change-pattern between `Buffer` and `TextAreaPainter`. As a result, `TextAreaPainter` is promoted to the first ranking in the list, as Fig. 7b shows. Meanwhile, benefitting from the change-pattern between `Request` and `CacheStrategy`, the rank of `OffsetManager` is also improved.

Fig. 8 shows a bad example (coming from jEdit, commit #2160 (blue dotted circle) and HSQLDB, commit #4931 (red dotted circle)) of applying our boosting method on JRipples, where the rank of the actual impacted class goes down when applying the boosting method. As Fig. 8a shows, `TextAreaPainter` is the actual impacted class by the change starting point `JeditTextArea`, while `View` is not. Due to the code similarity between `JeditTextArea` and `Expression`, our boosting method recommends the change-patterns of commit #4931 from project HSQLDB to the CIA task. Then, the change-pattern between `Expression` and `FunctionSQLInvoked` is similar with the one between `JeditTextArea` and `View`, and `View` is promoted to the 3rd ranking, and `TextAreaPainter` is demoted to the 4th ranking in the list. Then, the actual impacted class is pulled down in the list. This would explain why our boosting method has side effects on some datasets such as JHotDraw in Table 5.

7.2 Similar Change-Patterns

We state that the proposed boosting method can use the historical change-patterns in a cross-project scenario to boost CIA tools. The premise of this claim is that there are a lot of similar (or same) change-patterns across projects, and then the proposed boosting method can utilize the similar change-patterns coming from other projects.

TABLE 7
The Top 10 Change-Patterns Across Projects

No.	Change-patterns (18 dimens)	Amount	# of Projects Crossed
1	$<0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0>$	18,555	173
2	$<0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0>$	15,049	159
3	$<0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0>$	7,061	156
4	$<0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,0>$	5,531	154
5	$<0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,2\,0\,0\,0\,0\,0\,0>$	3,494	145
6	$<0\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0>$	3,415	143
7	$<0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,1\,0\,0\,0\,0>$	3,385	143
8	$<0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,2\,0\,1\,0\,0\,0\,0>$	1,930	129
9	$<0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0>$	1,915	113
10	$<0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,1>$	1,879	130

TABLE 8
The Percentage of Valid Change-Patterns Coming From Cross-Project and Within-Project Cases

Datasets	ImpactMiner		JRipples		ROSE	
	within	cross	within	cross	within	cross
FreeCol	14,579	7,115	1,853	924	13,762	6,375
HSQLDB	7,997	858	7,557	666	5,936	707
JAMWiki	972	478	1,052	423	1,399	704
jEdit	6,402	4,663	6,416	4,599	4,819	3,874
JHotDraw	416	998	55	175	326	513
Makagiga	10,086	4,836	7,283	3,195	6,839	3,377
OmegaT	1,200	2,124	871	1,378	766	1,349
Percentage	66.2%	33.8%	68.8%	31.2%	66.7%	33.3%

We count the frequency of the change-patterns across different projects. Specifically, for the commits coming from the 182 projects (i.e., the dataset), we extract the change-patterns between the starting class and the rest of classes in a commit, and then we cluster the same change-pattern in the commits of the 182 projects. In total, we find 233,348 change-patterns in these projects, in which 213,554 change-patterns can find at least one identical change-pattern, up to 91.5 percent. Table 7 shows the top 10 most frequent change-patterns, and the number of change-patterns and the number of projects involving in the change-patterns can also be found in the table.

We can observe that many frequent change-patterns are across many different projects. E.g., the change-pattern 1 (i.e., *Implementing Interface, II*) is across 173 projects. This statistical result provides a solid foundation for us using the similar change-patterns coming from other projects to boost change impact analysis methods.

7.3 The Percentage of Valid Change-Patterns

We allow the boosting method to find the similar change-patterns from 20 commits in the experiment. If the change-patterns in the 20 commits can directly affect the rankings of the classes in the original impact set, we regard it as a valid one. Then, the valid change-patterns may come from the historical commits of the current project, i.e., within-project case, or coming from other projects, i.e., cross-project case.

We count the number of the valid change-patterns coming from the two cases, respectively. We can observe from Table 8 that about one third of the valid change-patterns are

TABLE 9
Accuracy Boosting for ImpactMiner When Applying Cross- and Within-Project Change-Patterns

Datasets	Approach	Recall					Precision				
		5	10	20	30	40	5	10	20	30	40
FreeCol	ImpactMiner	9.51	16.81	27.2	36.79	45.73	7.33	6.75	5.67	5.27	4.95
	cross	12.88	20.04	30.8	39.44	47.13	10.17	8.26	6.46	5.64	5.10
	within	14.15	21.91	32.26	40.96	47.92	11.59	9.22	6.79	5.86	5.17
HSQLDB	ImpactMiner	9.49	16.13	24.66	29.82	34.60	10.90	9.45	7.41	6.15	5.48
	cross	11.51	18.22	26.85	31.63	35.75	13.35	10.64	8.12	6.58	5.75
	within	14.44	21.25	28.88	34.09	37.97	15.61	12.23	8.97	7.19	6.15
JAMWiki	ImpactMiner	14.76	20.18	27.24	30.07	31.11	11.69	8.62	6.13	4.67	3.67
	cross	15.26	22.15	27.88	30.22	31.33	12.78	9.48	6.33	4.74	3.74
	within	16.41	22.15	28.31	30.28	31.34	13.70	9.57	6.45	4.76	3.75
Jedit	ImpactMiner	14.71	23.26	30.82	34.7	37.71	11.44	10.22	7.13	5.56	4.58
	cross	16.73	24.83	32.35	35.54	38.1	13.58	10.81	7.51	5.69	4.66
	within	19.32	25.93	32.84	35.99	38.72	15.78	11.24	7.65	5.79	4.73
JhotDraw	ImpactMiner	7.32	12.2	19.08	23.86	27.38	9.26	8.16	6.62	5.49	4.52
	cross	8.27	14.02	20.62	24.61	27.33	10.00	8.75	7.06	5.51	4.56
	within	9.60	14.69	21.93	24.45	27.10	10.59	8.75	7.13	5.61	4.52
Makagiga	ImpactMiner	19.38	28.11	38.94	46.16	51.89	12.52	9.34	6.67	5.46	4.66
	cross	24.90	32.38	41.68	48.71	53.89	16.22	10.84	7.23	5.77	4.86
	within	25.66	33.95	43.57	49.78	54.74	16.71	11.42	7.57	5.92	4.95
OmegaT	ImpactMiner	26.24	38.09	48.77	53.23	56.39	16.53	12.26	8.05	6.11	4.91
	cross	33.24	42.58	50.94	54.54	57.55	20.74	13.78	8.42	6.22	5.02
	within	34.90	43.96	50.52	54.31	57.50	21.55	14.12	8.36	6.22	5.02
Avg Improv (cross)		3.05	2.78	2.06	1.44	0.90	2.45	1.11	0.49	0.21	0.13
Rlt Improv (cross)		20.28	12.94	7.00	3.90	1.98	21.75	12.34	7.39	3.67	2.72
Avg Improv (within)		4.72	4.15	3.09	2.18	1.50	3.69	1.68	0.75	0.38	0.22
Rlt Improv (within)		34.29	19.99	10.94	6.06	3.48	33.52	18.81	11.20	6.65	4.37

from cross-project case, i.e., 33.8, 31.2 and 33.3 percent for ImpactMiner, JRipples and ROSE, respectively. Therefore, this result confirms that the proposed method can utilize the change-patterns coming from other projects to boosting the CIA tasks. In addition, it demonstrates that the historical change-patterns applying to CIA tasks can be generalized to a cross-project scenario by our boosting method.

7.4 Within- and Cross-Project Change-Patterns Effect

Section 7.3 shows that about 1/3 of the valid change-patterns come from cross-projects, and we want to evaluate the performance of the proposed boosting method when only the cross-projects information is available. If our boosting method works with only the cross-projects information, our method can be generalized to many more application scenarios, such as starting a new project without within-project information. Meanwhile, we also want to evaluate the performance of the proposed boosting method when only the within-projects information is available. If our boosting method works well, it demonstrates that within-project change-patterns are also useful to improve the performance of the traditional CIA tools.

Tables 9, 10 and 11 show the accuracy boosting for ImpactMiner, JRipples and ROSE when only using cross-project or within-project change-patterns. We can observe the average improvements in term of Avg Improv and Rlt Improv in Tables 9 and 11. Such improvements indicate that the proposed boosting method can improve the performance of the traditional CIA method even if only the cross-projects (or within-project) information is available. It is worth noting that a slight drop in recall is observed on some projects when cut-off point is 5 in Table 10 when only using the cross-projects information. For example, the recall of JRipples on project HSQLDB is 15.63, while the recall is 15.59. Meanwhile, the precision of JRipples on project HSQLDB is improved from 18.23 to 18.73 when cut-off point is 5. In general, the F-measure (i.e., a weighted harmonic mean of precision and recall) of JRipples has no significant difference when cut-off point is 5. When the cut-off points go up, we can see the average improvements in term of Avg Improv(cross) and Rlt Improv(cross) in Table 10.

In summary, the recall and precision improvements can be observed on most of the projects when we only utilize the cross-project information, and the proposed boosting method is applicable to new project without within-project information.

7.5 Practical Significance of the Improvements

Tables 4, 5, and 6 show the average improvements in both precision and recall. Since the average improvements of precision and recall may not directly illustrate the effectiveness of our approach for CIA tasks in practice, we demonstrate the effectiveness of our approach from another perspective. More precisely, we count the average improved positions of the truly impacted classes in the CIA tasks when applying our boosting method in each project. For example, when we apply our boosting method to ImpactMiner on the project of

TABLE 10
Accuracy Boosting for JRipples When Applying Cross- and Within-Project Change-Patterns

Datasets	Approach	Recall					Precision				
		5	10	20	30	40	5	10	20	30	40
FreeCol	Jripples	16.4	25.5	31.13	35.01	35.49	15.06	11.57	7.06	5.26	4.01
	cross	17.31	26.13	31.36	35.08	35.49	14.81	11.62	7.13	5.28	4.01
	within	17.50	25.76	31.80	34.80	35.49	14.98	11.53	7.19	5.25	4.01
HSQLDB	Jripples	15.63	22.49	31.41	37.92	40.79	18.23	13.96	10.12	8.21	6.81
	cross	15.59	24.1	32.49	38.4	40.74	18.73	14.76	10.48	8.34	6.78
	within	16.22	24.68	32.91	38.42	40.88	18.85	14.95	10.49	8.30	6.78
JAMWiki	Jripples	24.62	32.48	36.1	37.12	37.18	15.79	11.11	6.46	4.48	3.38
	cross	26.11	32.27	36.15	37.12	37.18	18.01	11.35	6.49	4.48	3.38
	within	28.92	32.73	36.14	37.12	37.18	19.65	11.58	6.49	4.48	3.38
Jedit	Jripples	26.10	35.64	44.83	51.26	54.32	23.31	16.43	11.02	8.83	7.22
	cross	24.23	35.79	45.76	51.55	54.38	21.55	16.38	11.31	8.91	7.21
	within	27.12	37.14	45.54	51.41	54.38	23.86	17.13	11.20	8.83	7.20
JhotDraw	Jripples	11.63	12.72	12.72	13.17	13.17	13.10	8.28	4.14	2.99	2.24
	cross	10.69	12.28	12.72	12.72	13.17	13.10	7.59	4.14	2.76	2.24
	within	11.43	12.28	12.72	12.72	13.17	12.41	7.59	4.14	2.76	2.24
Makagiga	Jripples	24.33	27.57	29.47	30.28	30.57	14.77	8.67	4.74	3.27	2.48
	cross	23.96	27.62	29.61	30.34	30.56	14.54	8.74	4.76	3.28	2.48
	within	23.34	26.52	28.15	28.71	28.91	13.67	8.02	4.34	2.96	2.23
OmegaT	Jripples	33.55	35.91	36.10	36.10	36.10	20.99	11.41	5.73	3.82	2.86
	cross	32.38	35.78	36.10	36.10	36.10	20.46	11.37	5.73	3.82	2.86
	within	33.72	36.02	36.10	36.10	36.10	21.22	11.41	5.73	3.82	2.86
Avg Improv (cross)		–0.19	0.39	0.55	0.30	0.24	0.13	0.15	0.17	0.05	0.03
Rlt Improv (cross)		–0.87	1.38	1.69	0.66	0.81	1.47	1.25	2.55	0.86	1.45
Avg Improv (within)		0.95	0.55	0.43	0.01	0.02	0.62	0.21	0.10	–0.02	–0.01
Rlt Improv (within)		4.18	1.79	1.27	–0.30	0.05	3.49	1.04	1.15	–0.97	–0.17

TABLE 11
Accuracy Boosting for ROSE When Applying Cross- and Within-Project Change-Patterns

Datasets	Approach	Recall					Precision				
		5	10	20	30	40	5	10	20	30	40
FreeCol	ROSE	22.09	33.04	44.19	50.26	54.00	17.39	13.33	9.11	7.02	5.75
	cross	19.74	32.00	44.69	51.52	55.48	15.7	12.79	9.23	7.19	5.90
	within	18.78	30.68	45.24	51.84	56.09	14.98	12.33	9.32	7.26	5.98
HSQLDB	ROSE	16.12	23.14	30.69	36.15	40.07	16.23	12.56	8.93	7.22	6.15
	cross	16.65	25.18	32.22	37.54	40.8	16.8	13.72	9.46	7.57	6.28
	within	16.44	24.51	33.64	38.55	42.07	16.74	13.39	9.97	7.78	6.47
JAMWiki	ROSE	34.29	45.19	55.01	61.57	65.26	28.73	19.46	12.05	9.19	7.35
	cross	35.88	47.57	59.43	63.36	66.53	29.96	20.81	13.19	9.47	7.53
	within	37.93	48.67	59.83	63.73	67.13	31.27	21.16	13.26	9.56	7.60
Jedit	ROSE	23.93	37.87	49.41	56.56	60.54	19.93	16.15	11.1	8.67	7.06
	cross	26.30	37.79	50.00	57.24	61.51	21.68	16.37	11.15	8.71	7.16
	within	28.25	38.93	50.21	57.25	61.75	22.84	16.78	11.00	8.67	7.17
JhotDraw	ROSE	5.57	12.07	13.85	15.20	15.20	6.41	6.79	4.49	3.33	2.50
	cross	8.34	13.14	14.49	15.2	15.33	10.51	7.56	4.55	3.33	2.53
	within	8.32	13.32	14.98	15.88	16.72	11.03	8.08	4.68	3.46	2.76
Makagiga	ROSE	22.10	29.20	37.24	42.07	45.94	14.15	9.78	6.51	5.00	4.13
	cross	24.93	32.06	39.51	44.01	47.33	15.84	10.72	6.99	5.23	4.26
	within	26.21	32.80	40.00	44.21	47.60	16.88	11.16	7.07	5.29	4.30
OmegaT	ROSE	31.91	40.74	48.76	51.75	54.26	21.90	14.55	9.02	6.49	5.08
	cross	37.19	44.29	50.81	52.9	55.05	25.19	15.66	9.31	6.60	5.15
	within	38.24	44.56	50.45	53.22	55.05	25.82	15.77	9.29	6.60	5.15
Avg Improv (cross)		1.86	1.54	1.71	1.17	0.97	1.56	0.72	0.38	0.17	0.11
Rlt Improv (cross)		12.32	5.44	4.32	2.47	1.92	13.97	6.01	4.16	2.44	2.04
Avg Improv (within)		2.59	1.75	2.17	1.59	1.59	2.12	0.86	0.48	0.24	0.20
Rlt Improv (within)		14.78	5.91	5.92	3.84	4.11	17.43	7.61	5.56	3.80	4.29

TABLE 12
The Average Improved Positions of the Truly Impacted Classes

Projects	Boosting		
	ImpactMiner	JRipples	ROSE
FreeCol	5.60	0.15	3.31
HSQLDB	6.13	0.71	3.87
JAMWiki	1.73	1.02	2.71
JEdit	2.85	0.79	1.62
JhotDraw	2.68	0.00	3.34
Makagiga	3.94	0.21	4.07
OmegaT	2.55	0.00	1.89

TABLE 13
Recall Comparison for ROSE

Datasets	η	Recall					AVG
		5	10	20	30	40	
FreeCol	0.5	17.6	29.06	44.31	52.09	56.38	39.89
	0.6	18.65	29.12	45.16	52.49	56.43	40.37
	0.7	19.08	30.01	46.16	52.60	56.41	40.85
	0.8	19.53	31.60	46.33	52.50	56.13	41.22
	0.9	21.41	33.61	46.16	52.17	55.66	**41.80**
HSQLDB	0.5	16.71	25.36	34.82	38.97	42.52	31.68
	0.6	17.30	25.45	34.5	39.28	42.96	**31.89**
	0.7	16.98	25.28	34.34	39.17	42.49	31.65
	0.8	16.77	25.23	34.12	39.09	42.72	31.59
	0.9	17.31	24.75	32.96	37.71	41.56	30.86
JEdit	0.5	27.93	39.51	50.27	57.15	62.03	47.38
	0.6	28.54	40.04	50.39	57.39	62.21	47.71
	0.7	28.37	39.94	50.48	57.51	62.31	47.72
	0.8	29.09	39.69	50.34	57.58	62.22	**47.78**
	0.9	28.37	39.94	50.48	57.51	62.31	47.72

FreeCol, the average improved positions of the truly impacted classes is 5.6, which is practical significance for the developers when they do CIA tasks in practice. Imagine that ImpactMiner recommends a list of classes to a developer, and the truly impacted classes rank at 7 and 8. If a developer needs 1 minute (on average) to determine whether a recommended class is a truly impacted one. Then, before applying our boosting method, the developer needs at least 8 minutes to determine the two truly impacted classes. After applying our boosting method, the truly impacted classes rank at 2 and 3, and thus the developer only needs 2 minutes to determine the two truly impacted classes. That is, our boosting method can improve the efficiency of developers doing CIA tasks. Table 12 shows the average improved positions of the truly impacted classes for each project when we apply the boosting method to traditional CIA methods.

8 Discussion and Threats to Validity

In this section, we analyze the parameters that may affect the experimental results. It is worth noting that the parameters calibration is performed before we conduct the experiments in Section 6. Meanwhile, the historical change-patterns coming from a cross- and within-project scenarios are also discussed in this section.

8.1 *t* Choice Effect

t is the number of commits that allows our boosting method to find the similar change-patterns from them. To evaluate the *t* choice effect on the performance of our boosting method, we compare the performance of ImpactMiner, JRipples and ROSE on three randomly selected datasets (i.e., HSQLDB, FreeCol, jEdit) across the values of *t* with 5, 10, 15, 20, 25 and 30.

Fig. 5 shows the average improvement of the f-measure when applying different values of *t* to ImpactMiner, JRipples and ROSE on the projects of HSQLDB, FreeCol and jEdit across the cut-off points of 5, 10, 20, 30 and 40. In general, we can observe that the average improvements of the f-measure achieved by ImpactMiner keep growing when the value of *t* is less than 20, and ImpactMiner achieves best improvement when *t* equals 20 on the projects of HSQLDB and FreeCol, as shows in Figs. 5a and 5b. Similarly, JRipples and Rose also achieve the best improvement of the f-measure when *t* equals 20 on the projects of HSQLDB and FreeCol (Figs. 5d and 5e, 5g and 5h). Meanwhile, we can observe that the performance of the three CIA tools show little difference when the value of *t* is greater than 15 on the project jEdit, as shown in Figs. 5c, 5f and 5i. Since the three projects are randomly selected and the three CIA tools show best performance when *t* equals 20 on the projects in most of the cases, we set the value of *t* as 20 when we employ ImpactMiner, JRipples and Rose to conduct the CIA tasks in the experiment.

8.2 η Choice Effect

To evaluate the η choice effect on the performance of our boosting method, we compare the performance of ImpactMiner, JRipples and ROSE on the three randomly selected datasets (i.e., HSQLDB, FreeCol, jEdit) across the values of η with 0.5, 0.6, 0.7, 0.8 and 0.9. Tables 13, 14, 15, 16, 17, and 18 show the recall and precision when applying different values of η to ImpactMiner, JRipples and ROSE on the projects of HSQLDB, FreeCol and jEdit across the cut-off points of 5, 10, 20, 30 and 40.

We can observe that ROSE achieves the best average (i.e., AVG in Tables 13 and 14) recall and precision when η is 0.9 on dataset FreeCol, and η is 0.6 on dataset HSQLDB, and η is 0.8 on dataset JEdit. It seems that there is no fixed value for η that can make ROSE always achieve the best results. However, there is a pattern on the η choice effect to the performance of JRipples, i.e., when η is 0.8, JRipples achieves the best average recall and precision on the three datasets (i.e., AVG in Tables 15 and 16). Similarly, ImpactMiner performs better when η is 0.8 (i.e., AVG in Tables 17 and 18) half of the time (e.g., average recall on jEdit, average precision on HSQLDB and jEdit). Therefore, we use 0.8 as the default value of η in the experiment.

8.3 Equivalent Class Choice Effect

In the process of seeking an equivalent class $\tilde{c}_{init}$ for the starting class c_{init}, we regard the starting class identified by *ISC* in each historical commit as the equivalent class, known as the boosting strategy. To have a meaningful comparison, we also randomly select a class from each historical commit as the equivalent class, and mapping the change-pattern between the randomly selected class and the rest of classes

TABLE 14
Precision Comparison for ROSE

Datasets	η	Precision					AVG
		5	10	20	30	40	
FreeCol	0.5	14.26	11.69	9.09	7.31	6.04	9.68
	0.6	14.80	11.79	9.31	7.37	6.05	9.86
	0.7	15.07	12.09	9.52	7.39	6.06	10.03
	0.8	15.45	12.76	9.58	7.37	6.01	10.23
	0.9	16.87	13.51	9.55	7.34	5.97	**10.65**
HSQLDB	0.5	16.85	13.51	10.26	7.92	6.58	11.02
	0.6	17.26	13.67	10.26	8.02	6.65	**11.17**
	0.7	17.11	13.62	10.13	7.93	6.57	11.07
	0.8	16.9	14.03	10.17	7.99	6.65	11.15
	0.9	17.47	13.8	9.72	7.61	6.41	11.00
jEdit	0.5	22.77	16.88	10.96	8.60	7.17	13.28
	0.6	23.08	17.2	10.98	8.62	7.2	13.42
	0.7	23.18	17.26	11.09	8.68	7.23	13.49
	0.8	23.66	17.21	11.07	8.71	7.26	**13.58**
	0.9	23.18	17.26	11.09	8.68	7.23	13.49

TABLE 16
Precision Comparison for JRipples

Datasets	η	Precision					AVG
		5	10	20	30	40	
FreeCol	0.5	14.13	11.57	7.15	5.28	4.01	8.43
	0.6	14.72	11.62	7.17	5.28	4.01	8.56
	0.7	14.55	11.87	7.17	5.28	4.01	8.58
	0.8	14.98	11.66	7.19	5.28	4.01	**8.62**
	0.9	14.89	11.53	7.19	5.28	4.01	8.58
HSQLDB	0.5	18.11	14.87	10.69	8.25	6.82	11.75
	0.6	18.32	15.03	10.74	8.28	6.80	11.83
	0.7	18.89	14.97	10.66	8.27	6.78	11.91
	0.8	19.71	15.46	10.60	8.28	6.77	**12.16**
	0.9	19.75	15.01	10.67	8.27	6.75	12.09
jEdit	0.5	21.61	16.49	11.38	8.89	7.16	13.10
	0.6	22.91	16.65	11.38	8.87	7.16	13.39
	0.7	23.71	17.03	11.38	8.89	7.21	13.64
	0.8	24.48	17.40	11.37	8.93	7.23	**13.88**
	0.9	23.80	17.08	11.24	8.92	7.20	13.65

TABLE 15
Recall Comparison for JRipples

Datasets	η	Recall					AVG
		5	10	20	30	40	
FreeCol	0.5	16.59	25.97	31.58	35.08	35.49	28.94
	0.6	17.04	25.90	31.66	35.08	35.49	29.03
	0.7	16.60	26.63	31.56	35.08	35.49	29.07
	0.8	17.38	26.18	31.77	35.08	35.49	**29.18**
	0.9	17.12	25.43	31.71	35.08	35.49	28.97
HSQLDB	0.5	15.74	24.38	33.81	38.11	40.86	30.58
	0.6	15.71	24.7	33.73	38.14	40.79	**30.61**
	0.7	16.33	24.46	33.53	38.16	40.80	30.65
	0.8	16.84	24.96	32.99	38.18	40.89	**30.77**
	0.9	17.15	24.63	33.00	38.13	40.77	30.74
jEdit	0.5	25.19	36.07	46.13	51.74	54.00	42.63
	0.6	26.61	36.29	46.32	51.58	54.18	42.99
	0.7	27.22	36.72	46.16	51.74	54.28	43.22
	0.8	28.26	37.60	46.09	51.85	54.56	**43.67**
	0.9	26.59	37.00	45.56	51.62	54.34	43.02

TABLE 17
Recall Comparison for ImpactMiner

Datasets	η	Recall					AVG
		5	10	20	30	40	
FreeCol	0.5	13.62	22.18	33.88	43.08	49.36	32.42
	0.6	14.54	23.11	34.61	43.22	49.82	**33.06**
	0.7	15.28	22.78	33.99	42.51	49.25	32.76
	0.8	15.31	22.80	33.34	41.18	48.28	32.18
	0.9	14.26	21.08	31.76	39.91	47.36	30.87
HSQLDB	0.5	16.15	24.13	31.94	35.95	39.17	29.47
	0.6	17.18	24.42	32.12	36.15	38.87	**29.75**
	0.7	17.27	24.49	31.73	35.90	38.65	29.61
	0.8	17.92	24.35	31.88	35.58	38.38	29.62
	0.9	17.15	23.06	30.65	34.79	37.83	28.69
jEdit	0.5	17.53	25.59	33.51	36.25	39.16	30.41
	0.6	18.64	26.00	33.41	36.46	38.81	30.66
	0.7	18.96	26.42	33.13	36.30	38.71	30.70
	0.8	19.79	26.12	33.21	36.02	38.50	**30.73**
	0.9	19.18	24.84	32.30	36.63	38.23	30.24

in the commit to the current CIA task (known as the random strategy). We apply the original CIA tools (i.e., ImpactMiner, JRipples and ROSE), the CIA tools with boosting strategy, and the CIA tools with random strategy to the randomly selected dataset of jEdit for CIA tasks, and we evaluate the average precision and recall of these methods across 5, 10, 20, 30 and 40 cut-off points, with η equal to 0.8.

Fig. 9 shows the accuracy comparison between the boosting method and random method for ImpactMiner, JRipples and ROSE, respectively. Figs. 9a and 9b show ImpactMiner with the boosting strategy achieves the best precision and recall, and the ImpactMiner with the random strategy achieves the second best precision and recall. Similar results can be observed in Figs. 9c, 9d, 9e, and 9f, i.e., JRipples and ROSE with the boosting strategy achieve the best results, while original JRipples and ROSE achieve worst results.

We can observe from Fig. 9 that the CIA tools with the random strategy outperform the original ones in most cases, especially for those whose cut-off point is less than 30. The benefit of the CIA tools with the random strategy comes from the following two cases. First, if the randomly selected class from a commit is exactly the starting class, then CIA tools with the random strategy is equivalent to the boosting strategy proposed in this paper. Second, if the randomly selected class from a commit is not a starting class, then the change-pattern between the randomly selected class and the rest of classes in the commit is mapped to the current CIA task. Noting that we require the similarity between change-patterns (i.e., η) should be greater than 0.8 in any case. We would use the recommended change-patterns to boost the final confidence of an impacted class. Therefore, even though we randomly select the equivalent class, the recommended change-

TABLE 18
Precision Comparison for ImpactMiner

Datasets	η	Precision					AVG
		5	10	20	30	40	
FreeCol	0.5	10.83	9.20	7.11	6.14	5.33	7.72
	0.6	11.87	9.69	7.31	6.18	5.40	8.09
	0.7	12.39	9.61	7.21	6.07	5.33	**8.12**
	0.8	12.41	9.57	7.07	5.92	5.23	8.04
	0.9	11.47	8.83	6.74	5.76	5.15	7.59
HSQLDB	0.5	14.60	11.79	8.42	6.51	5.44	9.35
	0.6	15.63	11.91	8.47	6.54	5.40	9.59
	0.7	15.84	11.94	8.35	6.49	5.36	9.59
	0.8	16.61	11.83	8.38	6.45	5.33	**9.72**
	0.9	15.68	11.22	8.01	6.23	5.21	9.27
jEdit	0.5	13.93	11.18	7.90	5.87	4.84	8.74
	0.6	15.01	11.34	7.84	5.87	4.79	8.97
	0.7	15.38	11.51	7.73	5.82	4.76	9.04
	0.8	16.46	11.30	7.74	5.79	4.72	**9.20**
	0.9	15.87	10.87	7.51	5.70	4.67	8.92

patterns is always similar to the one in the current CIA task. As a result, the recommended change-patterns will boost the accuracy of the CIA task.

In summary, we can conclude that the CIA tools with the boosting strategy outperform the ones with the random strategy in the CIA tasks on dataset jEdit, while the CIA tools with the random strategy outperform the original ones.

8.4 Runtime Discussion

To estimate the average runtime, we divide the process of running the boosting method into two steps, i.e., similar starting entity identification (step 1) and change-pattern matching (step 2), and we count the average runtime of each step. Table 19 shows the results, and we can observe that the average runtime of identifying the similar starting entity on the three CIA tools is about 13 seconds, and the average runtime of matching the change-pattern ranges from 62.8 seconds to 73.3 seconds on the three CIA tools. The total runtime of the boosting method is about 80 seconds. The evaluation was conducted on a PC with a common configuration (i.e., Intel Core i5, 3.5 GHz, 32G RAM). In a real development environment, the efficiency of the boosting method can be further increased with better computational resources.

TABLE 19
The Average Runtime of the Boosting Method

	ImpactMiner		JRipples		ROSE	
	Step 1	Step 2	Step 1	Step 2	Step 1	Step 2
Average Time (Sec)	13.1	62.8	13	73.3	12.8	69.4

8.5 Scalability Discussion

The proposed boosting method is extensible, then we can take other dependency such as inversion of control (i.e., IoC) [35] into consideration. To detect the IoC change-pattern from the source code, we first find the `spring.xml` file from the root path of the project (if any), and then analyze the mapping relationship between class names and their alias in the spring.xml, e.g., class B may be with an alias of B' in the `spring.xml`. After that, we go back to the source code of a commit to determine whether there is code line of "`applicationContext.getBean(`B'`)`" in class A. If so, we will check to determine if the current commit contains both classes A and B. If all the above conditions are met, IOC dependence between A and B is found in this commit. To add the dependencies of IoC to the change-patterns, the vector dimensions of coupling dependencies (i.e., $V_{cp}(C_i, C_j)$) is changed from 18 to19, and the extra dimension is used to indicate the dependency of IoC between two classes. The effort cost on constructing a new change-patterns mapping repository is low, because we only need to

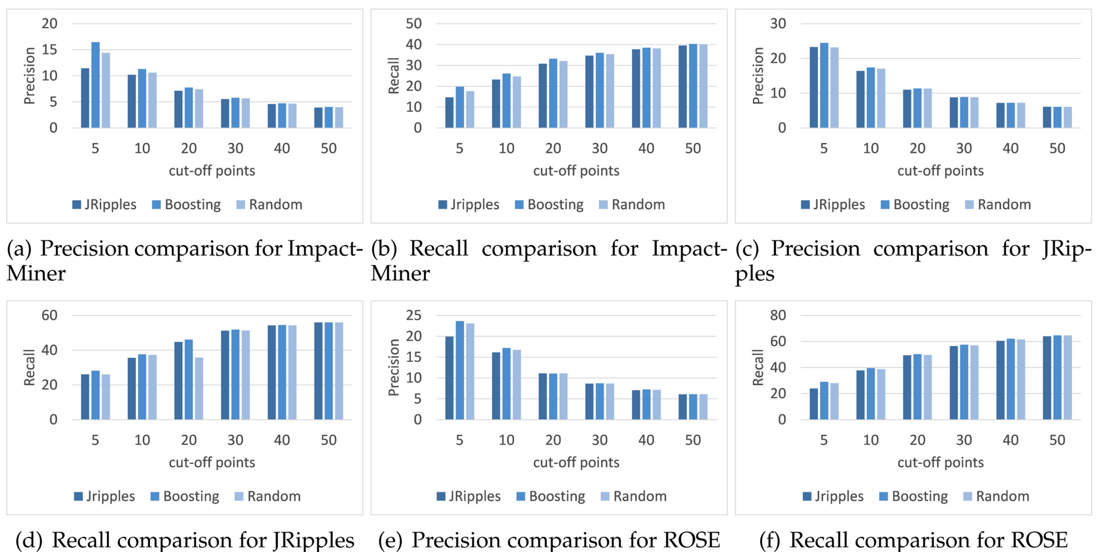


Fig. 9. Boosting strategy versus random strategy.

TABLE 20
IoC Commits Detected by the Boosting Method

No.	Projects	# IoC Commits
1	Spring-Security	1
2	Springframework	2
3	Kablink	3
4	Hazelcast	11
5	CAS	96

TABLE 21
The Code Smells Covered by Jdeodorant and PMD

Code Smells	Jdeodorant	PMD
God Method	✓	✓
Duplicated Code	✓	✓
God Class	✓	✓
Future Envy	✓	X
Type Checking	✓	X
Data Class	X	✓
Long Parameter List	X	✓

re-detect the IoC dependency for each commit, and the original 18 dimensions of the vector can be reused and keep unchanged.

Applying the IoC detection method to dataset of the 182 projects, we find IoC dependency from the commits of 5 projects, and they are Spring-Security, Springframework, Kablink, Hazelcast and CAS, as shown in Table 20. Spring-Security and Springframework provide the IoC feature, and the rest of the 3 projects is built based on the Spring framework.

8.6 Dataset Quality Effect

The dataset used in the experiment includes 182 projects. Inevitably, some projects have good-quality code, while some have bad-quality code. We want to study whether there is any relation between using projects datasets with good-quality code and bad-quality code. To achieve this goal, we conduct a new experiment. We employ two famous code smell detection tools (i.e., JDeodorant [36] and PMD [37]) to detect the code smell in the source code of the 182 subject projects, and then we evaluate the quality of the projects according to the number of the code smell detected from the source code of these projects. Code smells are code fragments that suggest the possibility of refactoring, which can degrade quality aspects of the software system [38]. Therefore, the number of code smells in a project can reflect the quality of the code.

JDeodorant is an open-source Eclipse plugin for Java that detects five code smells: God Method, Duplicated Code, God Class, Feature Envy, and Type Checking. PMD is an open-source tool for Java and an Eclipse plugin that detects many problems in Java code, including: God Method, Duplicated Code, God Class, Data Class and Long Parameter List. We can see from Table 21 that some code smells can be detected by both tools, while some code smells can be detected by one of them. Then, we chose both tools to detect

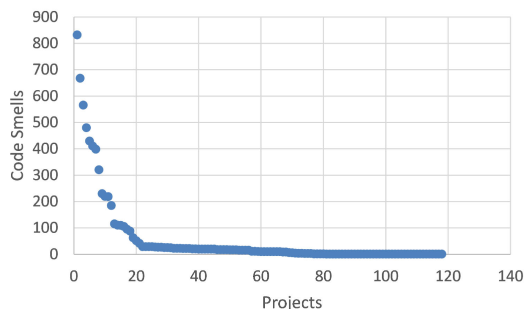


Fig. 10. The number of code smells containing in each project

code smells at the same time, so that they can detect as many code smells in the code as possible.

To detect the code smell from the code of each projects, we need to collect the complete source code of the projects, because the dataset used in the previous experiments only contain the commits of each project. Since a project may have multiple versions of source code, we download the version of source code that can cover the latest commit in our dataset.

After collecting the source code of each project, we use JDeodorant and PMD to detect the code smells. In particular, we move the source code of each project to the workspace of Eclipse because JDeodorant and PMD are both Eclipse plugin. Since a code smell may be detected by two tools, if that happens, we only count once. The two tools found that 118 projects have code smells, and the number of the code smells ranges from 1 to 832, while the two tools found no code smell in the rest of the projects. Fig. 2 shows the number of code smells containing in each project. Fig. 10 shows the number of code smells containing in each project.

To determine whether the code quality can affect the performance of our boosting method, we need two datasets: one containing code smells, i.e., we call it dataset 1, and another one containing no code smell, i.e., we call it dataset 2. Since it is difficult to judge whether the quality of a project is poor if it contains 1 or 2 code smells, we select the projects containing most code smells for analysis. We found 16 projects that contain more than 100 code smells and thus we chose these projects for analysis. These 16 projects contain 8,283 commits in total. For a fair comparison, we randomly select several projects from dataset 2 that they have the similar number of commits as the 16 projects. Specifically, we randomly select 22 projects which contain 8,251 commits from dataset 2.

Because it is easy to determine which projects the valid change-patterns come from, a simple and effective evaluation way is to analyze the existing experimental results to find out how many valid change-patterns come from the 16 projects containing most code smells, and how many valid change-patterns come from the 22 projects containing no code smell. Since the valid change-patterns is an equitable mechanism that can be used to indicate the contribution of the commits of a project to the boosting method, the valid change-patterns can be used to evaluate whether there is any relation between using projects datasets with good-quality code and bad-quality code.

For any commit, the boosting method first finds 20 most similar commits for it, and then we count the valid

TABLE 22
The Valid Change-Patterns From Case 1 and Case 2

Types	# Projects	# Commits	Valid Change-patterns
Case 1	16	8,283	6.63%
Case 2	22	8,251	6.61%

change-patterns in the 20 similar commits and how many valid change-patterns come from the 16 projects containing most code smells (i.e., we call it case 1) and the 22 projects containing no code smell (i.e., we call it case 2), respectively. At last, we calculate the proportion of the valid change-patterns coming from case 1 and case 2 for a commit. For other commits, we repeat the calculation, and we calculate an average proportion of the valid change-patterns coming from case 1 and case 2 for all commits. The results in Table 22 shows that the valid change-patterns coming from case 1 is 6.63 percent, and the valid change-patterns coming from case 2 is 6.61 percent. There is no significant difference between these two cases. In summary, we can conclude that there is no relation between using projects datasets with good-quality code and bad-quality code.

8.7 A Simpler Boosting Method

We found that most of the valid change-patterns come from with-projects, then we want to know whether a simpler approach based on logical coupling would help to boost the performances of the CIA methods that are not based on logical coupling. In the experiment, we evaluate the performance of our boosting method on three CIA tool, i.e., ImpactMiner, JRipples, ROSE. ImpactMiner and ROSE are the CIA methods based on (or partially based on) logical coupling of the past commits in the evolutionary history of a project, while JRipples is not. Then, we can make comparison for the performance of JRipples under a simpler logical coupling boosting method and our boosting method. For a simpler boosting method, JRipples can only use the logical coupling (i.e., change-patterns) come from the evolutionary history of the same project, i.e., within-project. In contrast, our boosting method can use the logical coupling (i.e., change-patterns) come from the same project and other projects (i.e., within-project + cross-projects) to boost the performance of JRipples. As a result, the contrast experiment turns into a comparison between applying the "within-project" and "within-project + cross-projects" change-patterns to boost the performance of JRipples.

We evaluate the performance of the JRipples when applying the "within-project" and "within-project + cross-projects" change-patterns. Table 23 shows the accuracy boosting for JRipples. We can observe the average improvements in term of Avg Improv and Rlt Improv of our boosting method are higher than those of the simpler boosting method. This result indicates that the change-patterns come from other projects are useful to improve the performance of the traditional CIA method. In summary, a simpler boosting method that only using the within-project change-patterns does improve the performance of the CIA tool, but it is even better if we take the change-patterns coming from other projects into consideration.

8.8 Threats to Validity

In this section, we focus on the threats that could affect the results of our case studies.

Threats to External Validity. One of the threats to external validity is the potential data leakage from the dataset of 182 collected projects to the 7 evaluated projects. To avoid the data leakage, given a commit, we only use the commits (both for current projects and other projects) before the current commits as reference to find available change-patterns from them, whereas the commits after the current commit are not used for current CIA task. This can avoid using the change-patterns contained in the future commits to boosting the CIA task.

Another threat to external validity relates to the forks or duplicated code/commits from the 7 evaluated projects. Due to the rise of the pull-request development model, the dataset may contain forks from the evaluated 7 projects. Although we manually checked that none of the 175 projects were forked from any of the 7 projects, it is possible that some of the projects were significantly modified to remove the "fork traces". There might be some duplicated commits in the datasets that were used to boost the CIA tasks. In the future, further investigation by analyzing the forked projects is needed to mitigate this threat.

The third threat to external validity relates to the usability of the boosting method. It is possible to use the same projects (i.e., 182 collected projects) to do the CIA task for other users. For example, the user can first employ the a traditional CIA tool (e.g., ImpactMiner) to determine an initial impacted set according to the starting entity the user provided, and then use the boosting method to find the similar change sets from the evolutionary histories of many projects according to the similarity between starting entities. After that, the change-patterns that have been extracted from the

TABLE 23
Accuracy Improvements for JRipples When Applying Simpler Boosting and Our Boosting Methods

	Recall					Precision				
	5	10	20	30	40	5	10	20	30	40
Avg Improv (simpler boosting)	0.95	0.55	0.43	0.01	0.02	0.62	0.21	0.10	–0.02	–0.01
Rlt Improv (simpler boosting)	4.18	1.79	1.27	–0.30	0.05	3.49	1.04	1.15	–0.97	–0.17
Avg Improv (our boosting)	1.41	0.88	0.52	0.08	0.05	1.13	0.43	0.15	0.00	0.00
Rlt Improv (our boosting)	6.48	3.20	1.64	0.23	0.14	6.52	3.70	2.13	0.00	0.00

change sets can be used to boosting the rankings of the truly impacted entities in the initial impacted set. In the whole process, our boosting method only needs to analyze the coupling dependencies between the starting entities and the ones in the initial impacted set, which is a completely automatic process. On the other hand, the change-patterns from other projects have been pre-extracted and stored by our method. Therefore, any users can use the boosting method and the collected change-patterns for their CIA tasks.

The four threat to external validity relates to language-dependency and project-size dependency. The boosting method mainly aims at Java language, while it can be generalized to other object-oriented programming languages. In the paper, we used four types of coupling dependencies which are observed from the object-oriented programming paradigm. Although the same instance implemented by different object-oriented languages (e.g., C++ and Java) may have difference, it can be covered by our defined coupling dependencies, because these four coupling dependencies are defined at a high level, including Class-to-Class, Method-to-Class, Method-to-Variable and Method-to-Method. These four coupling dependencies can cover most coupling relationships in the object-oriented programming paradigm. In addition, the proposed boosting method has no dependency with the project size, which can be used both for small- and large-scale projects.

Threats to Internal Validity. One of the threats to internal validity relates to a commit containing more than one starting entities. By applying *ISC* method, we employ random forest to predict the probability of a class being a starting one. By default, the class with the highest probability in a commit is the starting one. In the meantime, if the probabilities of more than one classes are greater than 0.5 in a commit, *ISC* regards all these classes as salient ones [12], [16]. Then, our method will extract the change-patterns between each salient class and the rest of the classes in the commit, and apply the change-patterns to the rank boosting step. We found in previous study [39] that if a commit contains 2 starting entities, it contains unrelated changes. Then, one starting entity builds change-patterns between parts of the classes in the commit, while another starting entity builds change-patterns between the other parts of the classes in the commit. As a result, the change-patterns in this commit can be equivalent to the change-patterns coming from two commits. Because we only care the change-patterns rather than where they come from, if a change-pattern comes from a commit containing two starting entities, it can be also used by our boosting method. Thus, we believe there is little threat to a commit containing more than one starting entities.

9 Related Work

According to the impact analysis process, the CIA can be divided into two steps [40], i.e., identifying the starting impact set and identifying the candidate impact set.

9.1 Identifying the Starting Impact Set

The first step of the impact analysis process concerns the identification of the starting impact set that requires the analysis of the change request specification and both the source code and the software documentation [40]. However, mapping the concepts (or features) defined in a change request specification to the source code components is not an easy thing. Existing studies rely on different types of analysis (such as: textual, historical, static and dynamic) for the identification of concepts in the source code [41], [42].

Textual analysis tries to reveal the mappings between the concepts and the domain knowledge already encoded in the form of comments and identifier-names in the source code. To achieve this goal, many analysis techniques such as Natural Language Processing (NLP), Information Retrieval (IR) are the employed in textual analysis [43], [44], [45]. With historical analysis, researchers found the software entities related to a concept by mining the evolutionary history from the version control system such as SVN and Git [46]. The philosophy behind the historical analysis is that if a software entity is known to be in a concept, entities that tend to change in the same commits as that entity might also be likely candidate locations for that concept [47], [48], [49], [50].

Static analysis allows developers to identify the relevant software entities by the data or control flow dependencies between them [46]. For example, if one software entity is known to be part of the concept and it is the only caller of another entity, then it is considered likely that this latter entity is also part of the concept [51], [52]. Dynamic analysis refers to the invocation and observation of concept at execution time: execution traces are analyzed to identify code that is always executed when the concept is exercised in the system, and code that is not executed when the concept is not, thus identifying code that is (exclusively) associated with the concept [53], [54].

9.2 Identifying the Candidate Impact Set

In the second step of the impact analysis process concerns the identification of the candidate impact set based on the starting impact set identified in the first step. In general, dynamic analysis [3], [4], [55], [56], [57] and static analysis [5], [10], [58], [59], [60] can be used to identify the candidate impact set. Dynamic impact analysis obtains the interaction and dependency between software entities via the information collected from the running time of a program, and then estimates the impact set for a changed entity. Static analysis gets the dependencies between software entities by analyzing the static information of the source code. The accuracy of dynamic change impact analysis is relatively higher. However, dynamic change impact analysis needs to collect the dynamic information of the running time of a program, then its running cost is high, while the running cost of static change impact analysis is low.

According to different technologies used, the identification of the candidate impact set can be divided into two types: one based on logical dependency analysis [6], [7], [8], [19], [61], [62] and the other based on software repository mining (MSR) [9], [10], [11]. The dependency analysis based impact analysis estimates the impact set through the dependency relationships (such as invocation relationships, control relationships.) of software entities in a single version. The software repository mining based impact analysis gets the dependency relationships between software entities by

mining the update information of software entities in the software evolution.

Ying *et al.* [23] state that mining software repositories can uncover important historical dependencies between software entities, which may not be captured by the methods that based on logical dependency analysis. However, most of the impact analysis methods mine the software repository in a scenario of within-project, which limits the capabilities of current MSR based methods. To break this limitation, we generalize the MSR based method to a cross-project scenario, i.e., the change-patterns we referenced come not only from the current project, but also from other projects.

In addition, some researchers propose other type of techniques to identify the candidate impact set. For example, Ceccarelli *et al.* [63] employ a vector regression model to model the relationship between code changes, and further predict the co-changed entities. Canfora *et al.* [64] use the text similarity to retrieve similar change requests from the code repository to obtain the impact set of code change. Gethers *et al.* [65] use the relational topic model to model the dependencies among software entities and utilize the topic dependencies among the entities to conduct the change impact analysis.

10 Conclusion and Future Work

Change impact analysis plays an important role in code change comprehension and software maintenance. This paper proposes a novel method to boost the performance of traditional CIA tasks. To improve the rankings of actual impacted classes, we retrieve equivalent classes from the historical change sets for a starting class, and map the change-patterns of the equivalent classes to the starting class. Then, the rankings of the classes with similar change-patterns in current CIA tasks will be improved in the list. Experimental results demonstrated the feasibility and effectiveness of our approach. In the future, we will further consider to apply our boosting approach to more CIA tools.

Acknowledgments

This work was supported by the Key Area Research and Development Program of Guang dong Province under Grant 2020B010164002, the National Natural Science Foundation of China under Grants 61902441 and 61722214, Guangdong Basic and Applied Basic Research Foundation under Grant 2020A1515010973, China Postdoctoral Sc ience Foundation (2018M640855), Hong Kong RGC Project No. 152239/18E, the Fundamental Research Funds for the Central Universities under Grant 20wkpy06, 20lgpy129, the National Natural Science Foundation of China under Grant 61725201, and the Beijing Outstanding Young Scientist Program under Grant BJJWZYJH01201910001004.

References

[1] M. Petrenko and V. Rajlich, “Variable granularity for improving precision of impact analysis,” in *Proc. IEEE 17th Int. Conf. Program Comprehension*, 2009, pp. 10–19.

[2] H. Kagdi, M. Gethers, and D. Poshyvanyk, “Integrating conceptual and logical couplings for change impact analysis in software,” *Empir. Softw. Eng.*, vol. 18, pp. 933–969, Oct. 2013.

[3] T. Apiwattanapong, A. Orso, and M. J. Harrold, “Efficient and precise dynamic impact analysis using execute-after sequences,” in *Proc. 27th Int. Conf. Softw. Eng.*, 2005, pp. 432–441.

[4] J. Law and G. Rothermel, “Whole program path-based dynamic impact analysis,” in *Proc. 25th Int. Conf. Softw. Eng.*, 2003, pp. 308–318.

[5] S. Park and D. H. Bae, “An approach to analyzing the software process change impact using process slicing and simulation,” *J. Syst. Softw.*, vol. 84, no. 4, pp. 528–543, 2011.

[6] L. C. Briand, Y. Labiche, and G. Soccar, “Automating impact analysis and regression test selection based on UML designs,” in *Proc. Int. Conf. Softw. Maintenance*, 2002, pp. 252–261.

[7] J. Díaz, J. Pérez, J. Garbajosa, and A. L. Wolf, “Change impact analysis in product-line architectures,” in *Proc. 5th Eur. Conf. Softw. Archit.*, 2011, pp. 114–129.

[8] T. Rolfsnes, S. D. Alesio, R. Behjati, L. Moonen, and D. W. Binkley, “Generalizing the analysis of evolutionary coupling for software change impact analysis,” in *Proc. IEEE 23rd Int. Conf. Softw. Anal. Evol. Reeng.*, 2016, vol. 1, pp. 201–212.

[9] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, “Mining version histories to guide software changes,” *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 429–445, Jun. 2005.

[10] G. Canfora and L. Cerulo, “Fine grained indexing of software repositories to support impact analysis,” in *Proc. Int. Workshop Mining Softw. Repositories*, 2006, pp. 105–111.

[11] M. Torchiano and F. Ricca, “Impact analysis by means of unstructured knowledge in the context of bug repositories,” in *Proc. ACM/IEEE Int. Symp. Empir. Softw. Eng. Meas.*, 2010, pp. 47:1–47:4.

[12] Y. Huang, N. Jia, X. Chen, K. Hong, and Z. Zheng, “Salient-class location: Help developers understand code change in code review,” in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 770–774.

[13] Y. Huang, X. Chen, Z. Liu, X. Luo, and Z. Zheng, “Using discriminative feature in software entities for relevance identification of code changes,” *J. Softw., Evol. Process*, vol. 35, pp. 1258–1277, 2020.

[14] B. Dit, A. Holtzhauer, D. Poshyvanyk, and H. H. Kagdi, “A dataset from change history to support evaluation of software maintenance tasks,” in *Proc. 10th Work. Conf. Mining Softw. Repositories*, 2013, pp. 131–134.

[15] J. Zhang *et al.*, “Search-based inference of polynomial metamorphic relations, ” in *Proc. 29th ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2014, pp. 701–712.

[16] Y. Huang, N. Jia, X. Chen, K. Hong, and Z. Zheng, “Code review knowledge perception: Fusing multi-features for salient-class location,” *IEEE Trans. Softw. Eng.*, early access, Sep. 2020, doi: 10.1109/TSE.2020.3021902.

[17] B. Dit *et al.*, “ImpactMiner: A tool for change impact analysis,” in *Companion Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 540–543.

[18] J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich, “JRipples: A tool for program comprehension during incremental change,” in *Proc. 13th Int. Workshop Program Comprehension*, 2005, pp. 149–152.

[19] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. L. Collard, “Blending conceptual and evolutionary couplings to support change impact analysis in source code,” in *Proc. 17th Work. Conf. Reverse Eng.*, 2010, pp. 119–128.

[20] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, “Integrated impact analysis for managing software changes,” in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 430–440.

[21] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, “From word embeddings to document similarities for improved information retrieval in software engineering,” in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 404–415.

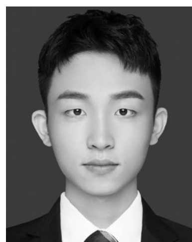
[22] Y. Huang, Q. Zheng, X. Chen, Y. Xiong, Z. Liu, and X. Luo, “Mining version control system for automatically generating commit comment,” in *Proc. ACM/IEEE Int. Symp. Empir. Softw. Eng. Meas.*, 2017, pp. 414–423.

[23] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, “Predicting source code changes by mining change history,” *IEEE Trans. Softw. Eng.*, vol. 30, no. 9, pp. 574–586, Sep. 2004.

[24] E. Nasseri, S. Counsell, and M. Shepperd, “An empirical study of evolution of inheritance in Java oss,” in *Proc. 19th Australian Conf. Softw. Eng.*, 2008, pp. 269–278.

[25] Y. Huang, X. Hu, N. Jia, X. Chen, Y. Xiong, and Z. Zheng, "Learning code context information to predict comment locations," *IEEE Trans. Rel.*, vol. 69, no. 1, pp. 88–105, Mar. 2020.
[26] P. Jaccard, "The distribution of the flora in the alpine zone," *New Phytologist*, vol. 11, no. 2, pp. 37–50, 2010.
[27] M. Barnett, C. Bird, J. A. Brunet, and S. K. Lahiri, "Helping developers help themselves: Automatic decomposition of code review changesets," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 134–144.
[28] Freecol. 2002. [Online]. Available: http://www.freecol.org
[29] Hsqldb. 2001. [Online]. Available: http://hsqldb.org/
[30] Jamwiki. 2006. [Online]. Available: http://www.jamwiki.org/
[31] jedit. 1999. [Online]. Available: http://www.jedit.org
[32] Jhotdraw. 2000. [Online]. Available: http://www.jhotdraw.org
[33] Makagiga. 2005. [Online]. Available: https://makagiga.sourceforge.io/
[34] Omegat. 2002. [Online]. Available: https://omegat.org/
[35] M. Fowler, "Inversion of control containers and the dependency injection pattern," 2004. [Online]. Available: http://www.martinfowler.com/articles/injection.html
[36] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "JDeodorant: Identification and removal of type-checking bad smells," in *Proc. 12th Eur. Conf. Softw. Maintenance Reeng.*, 2008, pp. 329–331.
[37] Pmd. 2017. [Online]. Available: https://pmd.github.io/
[38] A. Yamashita and S. Counsell, "Code smells as system-level indicators of maintainability: An empirical study," *J. Syst. Softw.*, vol. 86, no. 10, pp. 2639–2653, 2013.
[39] Y. Huang, X. Chen, Z. Liu, X. Luo, and Z. Zheng, "Using discriminative feature in software entities for relevance identification of code changes," *J. Softw.: Evol. Process*, vol. 29, no. 7, 2017, Art. no. e1859.
[40] A. De Lucia, F. Fasano, and R. Oliveto, "Traceability management for impact analysis," in *Proc. Front. Softw. Maintenance*, 2008, pp. 21–30.
[41] N. Wilde, M. Buckellew, H. Page, V. Rajlich, and L. Pounds, "A comparison of methods for locating features in legacy software," *J. Syst. Softw.*, vol. 65, pp. 105–114, 2003.
[42] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: A taxonomy and survey," *J. Softw.: Evol. Process*, vol. 25, pp. 53–95, 2013.
[43] D. Binkley, D. Lawrie, C. Uehlinger, and D. Heinz, "Enabling improved IR-based feature location," *J. Syst. Softw.*, vol. 101, pp. 30–42, 2015.
[44] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace," in *Proc. 22nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2007, pp. 234–243.
[45] S. Wang, D. Lo, Z. Xing, and L. Jiang, "Concern localization using information retrieval: An empirical study on linux kernel," in *Proc. 18th Work. Conf. Reverse Eng.*, 2011, pp. 92–96.
[46] A. Razzaq, A. L. Gear, C. Exton, and J. Buckley, "An empirical assessment of baseline feature location techniques," *Empir. Softw. Eng.*, vol. 25, no. 5, pp. 266–321, 2020.
[47] M. Chochlov, M. English, and J. Buckley, "A historical, textual analysis approach to feature location," *Inf. Softw. Technol.*, vol. 88, pp. 110–126, 2017.
[48] S. Wang and D. Lo, "Version history, similar report, and structure: Putting them together for improved bug localization," in *Proc. 22nd Int. Conf. Program Comprehension*, 2014, pp. 53–63.
[49] X. Ye, R. Bunescu, and C. Liu, "Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation," *IEEE Trans. Softw. Eng.*, vol. 42, no. 4, pp. 379–402, Apr. 2016.
[50] T. Zhang, H. Jiang, X. Luo, and A. Chan, "A literature review of research in bug resolution: Tasks, challenges and future directions," *Comput. J.*, vol. 59, no. 5, pp. 741–773, 2016.
[51] B. Bassett and N. A. Kraft, "Structural information based term weighting in text retrieval for feature location," in *Proc. 21st Int. Conf. Program Comprehension*, 2013, pp. 133–141.
[52] G. Scanniello, A. Marcus, and D. Pascale, "Link analysis algorithms for static concept location: An empirical assessment," *Empir. Softw. Eng.*, vol. 20, pp. 1666–1720, Dec. 2015.
[53] D. Poshyvanyk, Y. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE Trans. Softw. Eng.*, vol. 33, no. 6, pp. 420–432, Jun. 2007.
[54] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace," in *Proc. 22nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2007, pp. 234–243.
[55] L. Huang and Y.-T. Song, "Precise dynamic impact analysis with dependency analysis for object-oriented programs," in *Proc. 5th ACIS Int. Conf. Softw. Eng. Res. Manage. Appl.*, 2007, pp. 374–384.
[56] H. Cai and R. Santelices, "Diver: Precise dynamic impact analysis using dependence-based trace pruning," in *Proc. 29th ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2014, pp. 343–348.
[57] H. Cai and D. Thain, "DistiA: A cost-effective dynamic impact analysis for distributed programs," in *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2016, pp. 344–355.
[58] M. Acharya and B. Robinson, "Practical change impact analysis based on static program slicing for industrial software systems," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, pp. 13:1–13:2.
[59] H. Kagdi, M. Gethers, and D. Poshyvanyk, "Integrating conceptual and logical couplings for change impact analysis in software," *Empir. Softw. Eng.*, vol. 18, pp. 933–969, Oct. 2013.
[60] H. Abdeen, K. Bali, H. Sahraoui, and B. Dufour, "Learning dependency-based change impact predictors using independent change histories," *Inf. Softw. Technol.*, vol. 67, pp. 220–235, 2015.
[61] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using information retrieval based coupling measures for impact analysis," *Empir. Softw. Eng.*, vol. 14, pp. 5–32, Feb. 2009.
[62] L. C. Briand, J. Wuest, and H. Lounis, "Using coupling measurement for impact analysis in object-oriented systems," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 1999, pp. 475–482.
[63] M. Ceccarelli, L. Cerulo, G. Canfora, and M. Di Penta, "An eclectic approach for change impact analysis," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*, 2010, pp. 163–166.
[64] G. Canfora and L. Cerulo, "Impact analysis by mining software and change request repositories," in *Proc. 11th IEEE Int. Softw. Metrics Symp.*, 2005, pp. 9 pp.-29.
[65] M. Gethers and D. Poshyvanyk, "Using relational topic models to capture coupling among classes in object-oriented software systems," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2010, pp. 1–10.

Yuan Huang received the PhD degree in computer science from Sun Yat-sen University, China, in 2017. He is currently a research fellow with the School of Software Engineering, Sun Yat-sen University, China. He is particularly interested in software evolution and maintenance, code analysis and comprehension, and mining software repositories.



Jinyu Jiang is currently working toward the undergraduate degree with the Sun Yat-sen University, China. His research interest includes software engineering, code analysis and comprehension, and mining software repositories.

Xiapu Luo received the PhD degree in computer science from The Hong Kong Polytechnic University, China, and was a postdoctoral research fellow with the Georgia Institute of Technology, Atlanta, Georgia. He is currently an associate professor with the Department of Computing, The Hong Kong Polytechnic University, China. His current research focuses on mobile/IoT security and privacy, blockchain/smart contracts, software engineering, network security and privacy, and Internet measurement. His work appeared in top security, software engineering and networking conferences and journals, and he received several best paper awards (e.g., INFOCOM'18, ISPEC'17, ATIS'17, ISSRE'16, etc.).

Xiangping Chen (Member, IEEE) received the PhD degree from Peking University, China, in 2010. She is currently an associate professor with the Sun Yat-sen University, China. Her research interest includes software engineering and mining software repositories.

Zibin Zheng (Senior Member, IEEE) received the PhD degree from the Chinese University of Hong Kong, China, in 2011. He is currently a professor with Sun Yat-sen University, China. He published more than 120 international journal and conference papers, including three ESI highlycited papers. His research interests include blockchain, services computing, software engineering, and financial big data. He was a recipient of several awards, including the Top 50 Influential Papers in Blockchain of 2018, the ACM SIGSOFT Distinguished Paper Award at ICSE2010, the Best Student Paper Award at ICWS2010.

Nan Jia received the PhD degree in computer science from Sun Yat-sen University, China, in 2017. She is currently an associate professor with the School of Information Engineering, Hebei GEO University, China. She is particularly interested in data mining and software engineering.

Gang Huang (Senior Member, IEEE) is currently a full professor with the Institute of Software, Peking University, China. His research interests include in the area of middleware of cloud computing and mobile computing.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.