SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

June 2020, Vol. 63 162101:1–162101:22
https://doi.org/10.1007/s11432-019-2720-1

# Learning a graph-based classifier for fault localization

Hao ZHONG* & Hong MEI

*Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai* 200240, *China*

Received 21 July 2019/Revised 22 September 2019/Accepted 21 November 2019/Published online 9 May 2020

**Abstract** Because software emerged, locating software faults has been intensively researched, culminating in various approaches and tools that have been applied in real development. Despite the success of these developments, improved tools are still demanded by programmers. Meanwhile, some programmers are reluctant to use any tools when locating faults in their development. The state-of-the-art situation can be naturally improved by learning how programmers locate faults. The rapid development of open-source software has accumulated many bug fixes. A bug fix is a specific type of comments containing a set of buggy files and their corresponding fixed files, which reveal how programmers repair bugs. Feasibly, an automatic model can learn fault locations from bug fixes, but prior attempts to achieve this vision have been prevented by various technical challenges. For example, most bug fixes are not compilable after checking out, which hinders analyzing bug fixes by most advanced static/dynamic tools. This paper proposes an approach called ClaFa that trains a graph-based fault classifier from bug fixes. ClaFa is built on a recent partial-code tool called Grapa, which enables the analysis of partial programs by the complete code tool called WALA. Once Grapa has built a program dependency graph from a bug fix, ClaFa compares the graph from the buggy code with the graph from the fixed code, locates the buggy nodes, and extracts the various graph features of the buggy and clean nodes. Based on the extraction result, ClaFa trains a classifier that combines Adaboost and decision tree learning. The trained ClaFa can predict whether a node of a program dependency graph is buggy or clean. We evaluate ClaFa on thousands of buggy files collected from four open-source projects: Aries, Mahout, Derby, and Cassandra. The $f$-scores of ClaFa achieves are approximately 80% on all projects.

**Keywords** fault classifier, partial code analysis, bug fix analysis

**Citation** Zhong H, Mei H. Learning a graph-based classifier for fault localization. Sci China Inf Sci, 2020, 63(6): 162101, https://doi.org/10.1007/s11432-019-2720-1

## 1 Introduction

As man-made artifacts, software systems can contain bugs that return incorrect values, compromise security, or even crash the systems. In critical applications, bugs can incur huge losses. To improve the quality of software, the software engineering community has devoted much time and attention to bug location (see Section 6 for a detailed survey). Researchers have demonstrated the effectiveness of their proposed approaches, and several tools (e.g., FindBugs [1] and PMD[1)]) have been widely applied in modern software development.

Despite the success of their developments, further improvements are greatly demanded by programmers. For example, DiGiuseppe and Jones [2] considered that spectra-based approaches are less effective at locating multiple faults than single faults, as the execution of one fault can hinder the execution of other faults (e.g., crashes). Although possible solutions to this problem have been proposed, their true effectiveness remains in questionable. For example, Abreu et al. [3] proposed an approach that locates

* Corresponding author (email: zhonghao@sjtu.edu.cn)
1) PMD. https://pmd.github.io.

multiple faults, but evaluated it on only the Siemens benchmark [4], in which faults are manually constructed and each faulty version has exactly one fault. As another example, Wang et al. [5] found that the effectiveness of information retrieval (IR)-based approaches is significantly reduced, when bug reports are poorly written or contain misleading descriptions. Johnson et al. [6] argued that many programmers are dissuaded by the limitations of bug-detection tools, and prefer to rely on their own programming experiences in code debugging. Attempts to improve the state-of-the-art situation are ongoing (see Section 6 for details). In this paper, we explore bug detection by mining programming experiences from real bug fixes. Our approach locates multiple faults with more fidelity than the above dynamic approaches, and with finer accuracy than the above static approaches.

Software programmers often coordinate their development progress with source-code control systems [7]. After repairing bugs, programmers commit their changes to such systems. Meanwhile, the rapid progress of open-source software has promoted bug fixing by open-source communities, and many real bug fixes have accumulated. By examining the commit histories, researchers [8, 9] can identify the commits with repaired bugs, i.e., the bug fixes. Mei and Zhang [10] advocated that big data analysis can revolutionize software automation. Following their visions, we believe that the accumulated data open new research opportunities for locating bugs. Indeed, several researchers [11–13] have conducted empirical studies on bug fixes to understand how programmers repair bugs and make code changes.

**Our insight.** Inspired by the state-of-the-art situation and the availability of bug fixes, we contemplated mining a graph-based fault classifier developed from thousands of real bug fixes. Data cleaning is a hot research hotspot in the data mining area [14]. The output quality of many mining algorithms depends on the input quality. If a tool cannot extract accurate code facts from the source files, it cannot produce accurate bug predictions. In the software engineering and programming language communities, source files are often presented as dependency graphs [15], which more easily present facts of the source files than raw texts. Meanwhile, similar software engineering tasks (e.g., predicting bug fixes from commits [9]) can be assisted by classification techniques. Based on the above observations, we make the following proposition: In a proper presentation format (e.g., dependency graphs), one can accurately display related facts of the buggy code, and can train a classifier to locate bugs.

**The challenges.** To our knowledge, no previous researchers have attempted our proposition despite its benefits, because they are daunted by the following challenges.

Challenge 1. The first challenge is extracting the accurate facts from bug fixes. Most bug fixes are non-compilable, but most static tools (e.g., WALA[2)]) need compilable code. An empirical study [16] showed that only 38% of commits are compilable. When checking out the whole project of a commit, researchers must manually repair many compilation errors, before applying complete-code tools in analyses. Although bug signatures have been mined from buggy code [17] (see Section 6 for details), repairing the compilation errors in checked out buggy code requires much efforts; consequently, prior approaches typically analyze only a small number of known buggy samples. For example, Sun and Khoo [18] evaluated their approach only on the Siemens benchmark [19], in which the bugs are manually constructed and known. Li and Ernst [20] mined the bug signatures from only 53 bugs. This small training set is insufficient for training a reliable classifier. Meanwhile, empirical studies on bug fixes (e.g., [12,13]) have analyzed only the buggy files and their corresponding fixed files, with partial-code tools (e.g., [21]), which are too imprecise and lightweight [22, 23] to extract accurate facts from bug fixes. Without accurate facts, one cannot train an accurate classifier.

Challenge 2. Even when the facts can be accurately extracted from bug fixes, training a reasonably accurate classifier presents additional challenges. An empirical study [12] showed that faults account for only a small portion of source files; that is, most of the lines in a faulty source file are still clean. From a classifier perspective, the training set is quite imbalanced, because few locations are positive (contain faults). According to Yang and Wu [24], the problem of mining imbalanced data is one of ten challenging problems in data mining research. As most classifiers are designed for balanced data, their effectiveness is reduced when applied on imbalanced data.

2) WALA. http://wala.sf.net.

**Our contributions.** To handle the first challenge, we build ClaFa on Grapa and thereby analyze thousands of real bug fixes. Grapa [25] extends the inference strategies of PPA [23], and resolves the remaining unknown code names using released binary files that are related to a partial program. Grapa enables source-file analysis (even in the presence of compilation errors) by the state-of-the-art Java analysis tool WALA. In this way, it builds program dependency graphs (PDGs) from bug fixes.

**Definition 1.** A PDG is defined as $g = \langle V, E_1, E_2 \rangle$, where $V$ is a set of nodes corresponding to variables or expressions, and $E_1, E_2 \subseteq V \times V$ are two sets of edges. A $\langle s_1, s_2 \rangle \in E_1$ edge denotes a data dependency from $s_1$ to $s_2$, and a $\langle s_1, s_2 \rangle \in E_2$ edge denotes a control dependency from $s_1$ to $s_2$.

With Grapa, researchers in [26, 27] built PDGs from bug fixes to explore open questions in automatic program repair and code change patterns. To handle the first challenge, ClaFa uses Grapa to build PDGs from bug fixes that are not compilable. To handle the second challenge, we reduce the bias by a cost function that punishes a classifier, if it wrongly predicts a minority instance (Subsection 3.3.1). The major contributions of this paper are described below:

• We propose the first approach, called ClaFa (classifying faults), that predicts faulty nodes in a given PDG. The internal component of ClaFa is a classifier trained on thousands of bug fixes. During the training phase, ClaFa builds PDGs from the bug fixes, and extracts the feature vector of each node via graph analysis. To prepare labeled training data, ClaFa identifies the buggy nodes by comparing the PDGs of the buggy nodes with those of the fixed nodes. During the prediction phase, ClaFa uses its trained classifier to predict whether a node in a PDG is buggy or clean based on its extracted feature vector.

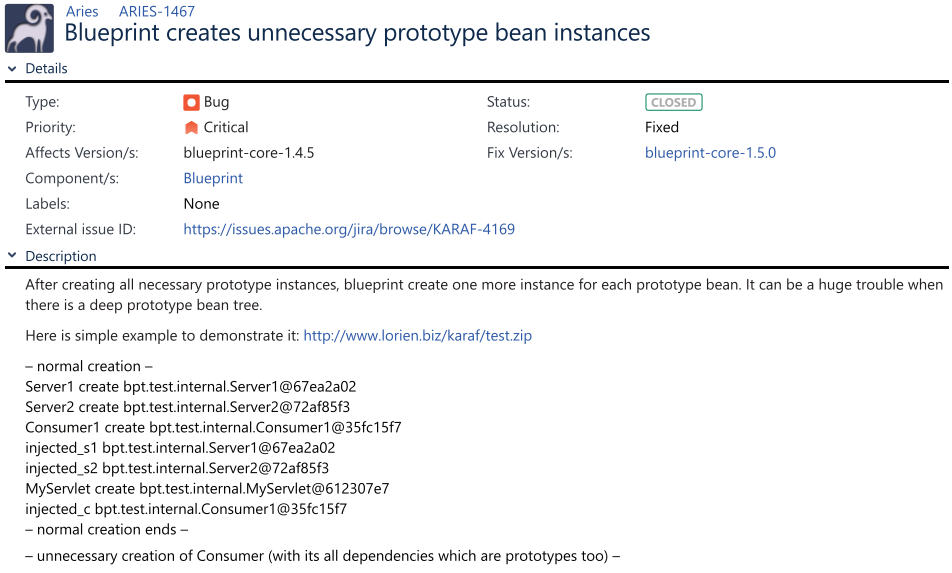
• We evaluate ClaFa on four popular open source projects. In total, we extract the feature vectors of 19343 nodes, covering 3534 buggy methods. To ensure the reliability of our results, we perform within-project and cross-project fault predictions in a time-aware evaluation. ClaFa reasonably predicts the within-project faults (with $f$-scores 70%), but is less effective in cross-project fault prediction, unless the projects are similar. We further found that the best features can differ among projects, which explains why cross-project fault prediction is less effective than within-project fault prediction. Despite these differences, our results show that bug reports and their called application programming interfaces (APIs) are more useful for locating faults than other features; also, that the outgoing control-dependent nodes of a node often determine whether the node is buggy or clean. Furthermore, our results reveal the best parameter and classification techniques, but their impacts prove subtle. In summary, our results reveal plenty of scope for improvement, especially by adding more features.

## 2 Motivating example

Suppose that a programmer Mary needs to repair the reported bug[3)] shown in Figure 1. If she locates the program's faults by a spectra-based approach, she must prepare many test cases that trigger both buggy and normal behaviors. ClaFa negates the need to prepare such test cases. Instead, Mary needs to feed only the reported bug to ClaFa. If using IR-based approaches, Mary must manually analyze the faults in the obtained buggy files. Here she can be further assisted by ClaFa, which can locate the finer faults in buggy files.

Although ClaFa and IR-based approaches have the same inputs, they operate by different techniques. IR-based approaches compare the bug reports and source files to locate similar pairs. Instead of comparing the two types of files, ClaFa builds graphs from the code, and inspects their detailed features. The techniques of ClaFa are similar to those programmers who manually review code for bugs. During a code review, programmers often read the corresponding bug reports to understand the buggy behaviors. Similarly, ClaFa reads the bug reports first using its trained topic model. In the above example, ClaFa identifies the topics of the bug report in Figure 1 as "blueprint", "service", "dependency", "server", and "create". Based on the topics, ClaFa then generates a vector ($\boldsymbol{b}$) presenting the bug report. In this vector, each bit denotes a topic, and "1" denotes that the bug report is related to the topic. After reading

3) ARIES-1612. https://issues.apache.org/jira/browse/ARIES-1612.


Aries ARIES-1467

Blueprint creates unnecessary prototype bean instances

Details

| | | | |
|---|---|---|---|
| Type: | Bug | Status: | CLOSED |
| Priority: | Critical | Resolution: | Fixed |
| Affects Version/s: | blueprint-core-1.4.5 | Fix Version/s: | blueprint-core-1.5.0 |
| Component/s: | Blueprint | | |
| Labels: | None | | |
| External issue ID: | https://issues.apache.org/jira/browse/KARAF-4169 | | |

Description

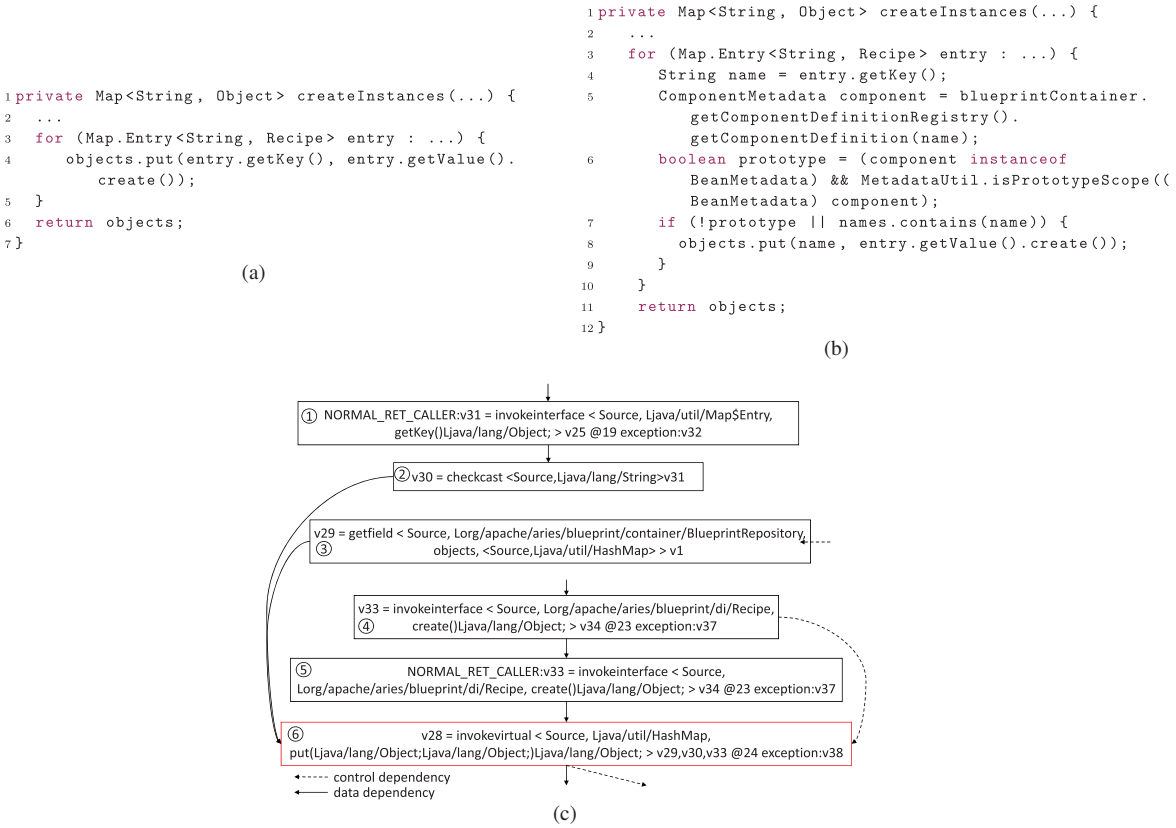
After creating all necessary prototype instances, blueprint create one more instance for each prototype bean. It can be a huge trouble when there is a deep prototype bean tree.

Here is simple example to demonstrate it: http://www.lorien.biz/karaf/test.zip

– normal creation –
Server1 create bpt.test.internal.Server1@67ea2a02
Server2 create bpt.test.internal.Server2@72af85f3
Consumer1 create bpt.test.internal.Consumer1@35fc15f7
injected_s1 bpt.test.internal.Server1@67ea2a02
injected_s2 bpt.test.internal.Server2@72af85f3
MyServlet create bpt.test.internal.MyServlet@612307e7
injected_c bpt.test.internal.Consumer1@35fc15f7
– normal creation ends –

– unnecessary creation of Consumer (with its all dependencies which are prototypes too) –

**Figure 1** (Color online) The bug report of ARIES-1467.


```
private Map<String, Object> createInstances(...) {
  ...
  for (Map.Entry<String, Recipe> entry : ...) {
     objects.put(entry.getKey(), entry.getValue().
        create());
  }
  return objects;
}
```

(a)

```
private Map<String, Object> createInstances(...) {
  ...
  for (Map.Entry<String, Recipe> entry : ...) {
     String name = entry.getKey();
     ComponentMetadata component = blueprintContainer.
        getComponentDefinitionRegistry().
        getComponentDefinition(name);
     boolean prototype = (component instanceof
        BeanMetadata) && MetadataUtil.isPrototypeScope((
        BeanMetadata) component);
     if (!prototype || names.contains(name)) {
       objects.put(name, entry.getValue().create());
     }
   }
   return objects;
}
```

(b)



(c)

**Figure 2** (Color online) The source code of ARIES-1467. (a) The buggy file of ARIES-1467; (b) the fixed file of ARIES-1467; (c) the program dependency graph of (a) (partial).

a bug report, programmers often look through source files to locate the bug. During this process, they can check many relevant aspects of the problem (e.g., the called APIs). Similarly, CLAFA understands code by building PDGs from source files. Figure 2(a) shows the buggy code, and Figure 2(c) shows the built PDG. For each node of the PDG, CLAFA extracts a vector ($\boldsymbol{c}$) based on the node features (e.g., node names), the local features (i.e., features of the $k$-deep nodes before and after the node), and the global features (i.e., features of all the nodes before and after the node). The node vector generated by

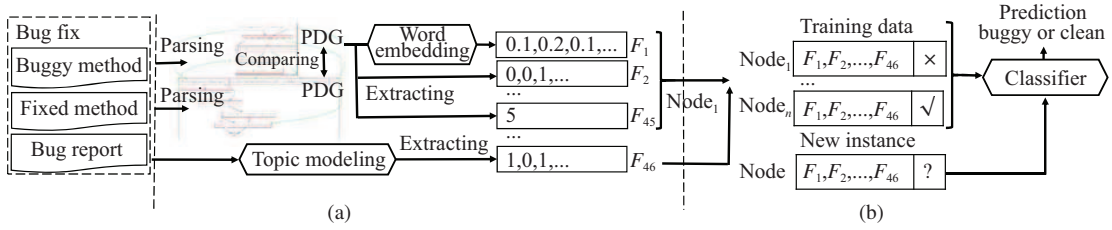



**Figure 3** (Color online) The overview of our approach. (a) Extracting features; (b) training and predicting faults.

ClaFa combines the $\boldsymbol{b}$ and $\boldsymbol{c}$ vectors. When input with the vector of the red-boxed node in Figure 2(c), the internal classifier of ClaFa predicts that the node is buggy (see Section 3 for details).

Mary does not need to understand the above details, because ClaFa automatically determines that the red-boxed node in Figure 2(c) is buggy. If she is unfamiliar with PDGs, ClaFa can locate the buggy lines through the WALA interface[4)], which allows the location of the code elements corresponding to a given node of a PDG. In this example, the buggy node is mapped to Line 4 in Figure 2(a). Figure 2(b) shows the fixed code, which checks some conditions before calling Line 8 (the buggy line). This example of a bug report illustrates the common use of ClaFa. However, our evaluation results show that even without a bug report, ClaFa achieves reasonably high $f$-scores when identifying buggy nodes in buggy PDGs (see Subsection 4.3.6 for details).

Despite the differences, between IR-based approaches and ClaFa, the two approaches can be easily combined. For example, IR-based approaches (e.g., [28]) can reduce the effort of locating buggy files prior to detecting their faults by ClaFa. In addition, ClaFa can be integrated with other tools such as automatic-program-repair tools, which locate faults by spectra-based approaches (e.g., [29]). In each iteration, many candidate patches are generated. If none of these patches pass all test cases, automatic-program-repair approaches must locate new faults in all candidates by spectra-based approaches, which is time-consuming. As ClaFa does not need to execute test cases, it can significantly reduce the time of locating faults.

## 3 Approach

The term, fault, defines different granularities of buggy code. In IR-based approaches, a fault is a buggy file, whereas in spectra-based approaches, it often defines a buggy code line. Although spectra-based approaches can locate faults with finer granularity than IR-based ones, existing studies (e.g., [30]) typically evaluate spectra-based approaches on code-line level, because code lines are more common than other granularities. In our approach, a fault is a buggy node in a PDG. When a code line calls multiple methods, each method invocation is encoded into a node. Consequently, a code line is often encoded into multiple nodes of a PDG, creating a finer granularity. Intuitively, locating finer faults is more challenging than coarser ones.

Our approach involves two stages: model training and fault localization. In the model training stage, a topic model is trained to classify bug reports, and a word embedding model is trained to encode code names into vectors. The faults are then located by a classification model. As the word embedding model and the topic model are learned by unsupervised approaches, they do not need labeled data. Meanwhile, as the classification model is learnt by a supervised approach, it requires labeled data with each node marked as buggy or clean.

Figure 3 shows the overview of our approach. Its major steps are (1) feature extraction (Subsection 3.2), and (2) training of the classification model and fault prediction (Subsection 3.3). In Figure 3, the word embedding model and the topic model are already learnt, and the labels for the classification model are prepared as follows. First, ClaFa builds a PDG ($S_b$) for the buggy method and a PDG ($S_f$) for its corresponding fixed method, and compares $S_b$ and $S_f$ to locate the modified nodes of $S_b$. When preparing

4) The usage guide of WALA. http://wala.sourceforge.net/wiki/index.php/UserGuide:MappingToSourceCode.

**Table 1** The features extracted by ClaFa

| ID | Node feature |
|---|---|
| $F_1$ | Main code name (full method name or field name) |
| $F_2$ | Node type |
| $F_3$ | Number of API names |
| $F_4$ | Number of client code names |
| $F_5$ | Occurrences of code names in bug reports |
| ID | Local feature |
| $F_6$ | Number of $k$-depth incoming nodes |
| $F_7$ | Number of $k$-depth outgoing nodes |
| $F_8$ | Similar to $F_2$ but for $k$-depth incoming nodes |
| $F_9$ | Similar to $F_2$ but for $k$-depth outgoing nodes |
| $F_{10}$ | Similar to $F_3$ but for $k$-depth incoming nodes |
| $F_{11}$ | Similar to $F_3$ but for $k$-depth outgoing nodes |
| $F_{12}$ | Similar to $F_4$ but for $k$-depth incoming nodes |
| $F_{13}$ | Similar to $F_4$ but for $k$-depth outgoing nodes |
| $F_{14}$ | Similar to $F_5$ but for $k$-depth incoming nodes |
| $F_{15}$ | Similar to $F_5$ but for $k$-depth outgoing nodes |
| | ($F_6 - F_{15}$ are calculated for data-dependent nodes) |
| $F_{16} - F_{25}$ | Similar to $F_6 - F_{15}$ but for control dependency |
| ID | Global feature |
| $F_{26} - F_{45}$ | Similar to $F_6 - F_{25}$ but the depth is maximized |
| ID | Bug report feature |
| $F_{46}$ | Classification result |

the training data, it marks any modified node as buggy ($\times$), and any unmodified node as clean ($\surd$). Our underlying Grapa tool detects a modified node, if its name or edges are changed. Modifications include the addition and deletion of code lines by programmers. For example, the buggy line in Figure 2(a) is not modified, but its control edges in Figure 2(c) are changed by the code lines added to Figure 2(b). Accordingly, ClaFa detects the modified node in Figure 2(c).

Table 1 lists our extracted features. Note that ClaFa extracts $F_{46}$ from each bug report. For each node, ClaFa extracts a set of code features, and combines them with $F_{46}$ to produce a vector. Once its classifier is trained, ClaFa can predict whether the vector of a node indicates a bug.

### 3.1 Data acquisition

The data are acquired from Apache projects, in which bugs are easily identified. Zhong and Su [12] showed that the links of most Apache projects are carefully maintained, meaning that most bug fixes in Apache projects can be identified by extracting and comparing their issue numbers. The issue number is often included in the title of an Apache commit (e.g., "ARIES-960" in Figure 4). In our setting, bug fixes can be identified without complicated techniques (e.g., [9]). We determine that this commit is a bug fix, because its bug report[5)] says that "ARIES-960" is a bug. As shown in Figure 4, a commit provides a link to its changed files. The commits are extracted and compared by our extension of the eGit tool[6)]. It checks out the buggy files and fixed files of each bug fix.

The bug reports are collected by a web crawler based on XPath [31]. This technical choice reduces the effort of handling different styles of bug reports. Even when composed by the same open source community, bug reports can have subtle differences. For example, although Derby and Cassandra are both from the Apache foundation, the "Detail" section of a Derby report[7)] includes an item called "Affects

5) ARIES-960. https://issues.apache.org/jira/browse/ARIES-960.
6) eGit. http://www.eclipse.org/egit/.
7) DERBY-5396. https://issues.apache.org/jira/browse/DERBY-5396.

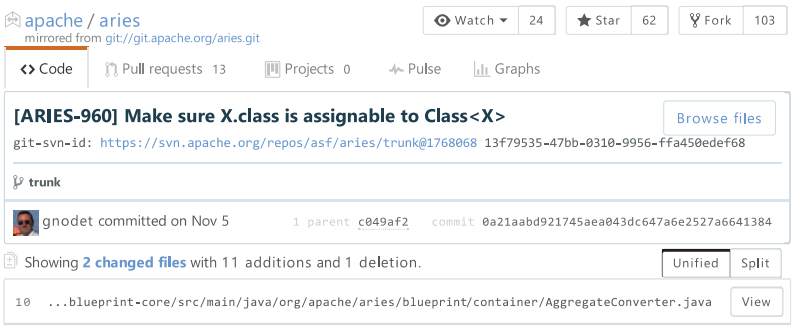



**Figure 4** (Color online) A sample commit.

Version/s", which is absent in a Cassandra report[8]. Our web crawler can extract bug reports of different styles after minor modifications on XPath queries.

## 3.2 Feature extraction

### 3.2.1 *The analysis of bug reports*

ClaFa combines the title, description, and comments of each bug report into a bag of words. Stop words (e.g., "is", "are", and "would") are removed, as these words are used in most bug reports and thus have little discriminative power. The remaining words are reduced to their roots by a stemming operation [32]. After that, ClaFa extracts the topics of each bug report using latent Dirichlet allocation (LDA) [33].

We choose LDA to classify bug reports, because it is a widely used technique to classify texts [33] including IR-based fault localization (e.g., [34]). LDA treats each topic as a multinomial distribution of a vocabulary of $w$ words. Topics are not predefined, but are learnt by unsupervised labeling. LDA models each document as a mixture of $k$ latent topics, and produces a relevance score between a topic and a document. ClaFa considers that a document contains a topic if the relevance score exceeds 0.3. Based on the LDA results, ClaFa encodes a bug report as a $k$-dimensional vector, with each bit denoting a topic. If a bug report is related to a topic, the corresponding bit is set to 1; otherwise, it is set to 0.

### 3.2.2 *The analysis of source files*

Source files in ClaFa are analyzed by Grapa [25], which extends the state-of-the-art WALA to build PDGs of bug fixes. Whereas WALA does not discriminate between the control dependencies and data dependencies of a PDG, Grapa extracts both types of dependencies. When ClaFa extracts the local and global features of a node (see Table 1) it identifies the nodes before and after the node in the PDG. The node labels are extracted by the Hungarian algorithm [35], which compares PDGs to locate their modified nodes (i.e., buggy nodes).

### 3.2.3 *The analysis of code names*

In a built PDG, a node denotes an instruction that can call a method or access a field. For each node, ClaFa extracts code names by parsing its label. As such labels are generated by WALA, they follow specific name patterns. More than one code name can be extracted from a node name. For example, if a node denotes a method invocation, ClaFa extracts its return type, full method name, and parameter type names as its code names. For the buggy node in Figure 2(c), it extracts two code names such as `java/lang/Object` and `java/util/HashMap.put`. Here, we ignore duplicated code names. For each node, ClaFa identifies a main code name: the full method name is the main code name of a method invocation, and the field name is the main code name of a field access. We collect all code names, and use word embedding [36] to train a word model. With the model, ClaFa generates a vector for each code name.

8) CASSANDRA-2044. https://issues.apache.org/jira/browse/CASSANDRA-2044.

#### 3.2.4 *Extracted features*

Table 1 lists our extracted features. For $F_1$, ClaFa encodes the code names into vectors, with the support of word embedding [36]. For $F_2$, ClaFa identifies the instruction types of nodes. In total, ClaFa identifies 33 node types (e.g., `invokevirtual`, `getfield`, and `putfield`). The type of a node is denoted as one bit in a vector. As mentioned in Subsection 3.2.3, ClaFa can extract multiple code names from one node. $F_3$ and $F_4$ count its API code names and its client code names, respectively. Here, API code names are code names that are declared by third party libraries. For example, when analyzing bug fixes of Aries[9)], the prefix `org/apache/aries` determines whether a code name is an API name. Wang et al. [5] showed that both IR-based approaches and programmers locate buggy files by searching code names in bug reports. As such code names are useful for locating bugs, we determine $F_4$ if code names in nodes overlap the code names in bug reports. Local features are derived from node features, and are extracted from the $k$-depth nodes before and after a current node. For example, when calculating $F_6$ of the red node in Figure 2(c) and the depth is set to one, we sum up the incoming data-dependent nodes of ②, ③, and ⑤, which have direct data dependencies to ⑥. As ① and ④ are the directly incoming nodes of the three data-dependent nodes, $F_6$ of ⑥ is 2, when the depth is one. The global features are the local features, whose values are obtained at maximum $k$-depths. For $F_{46}$, ClaFa encodes each bug report into a vector based on its topics. Here, the size of the vector is set to one hundred. Whereas other features capture the structure of the code, $F_{46}$ sets the type of a reported bug.

When extracting the features from a bug report, the comments are ignored as they reveal the true locations of faults. For example, some faulty locations are fixed repetitively [37] and can be supplemented with comments added by programmers with true locations. When extracting our features from source files, we also ignore code comments. After these exclusions, no known labels exist in our extracted features. Our evaluation results are encouraging (see Section 4), but we expect that collecting more features in PDGs would improve the effectiveness. We further discuss this issue in Section 5.

### 3.3 Model training and bug localization

#### 3.3.1 *Training the classifier*

He and Garcia [38] divided the state-of-the-art solutions for imbalanced learning into sampling approaches and cost-sensitive approaches. Sampling approaches modify an imbalanced data set either by adding instances to minority classes (e.g., [39]) or by removing instances from majority classes (e.g., [40]). Although sampling is easily understood, it can lead to overfitting or omission of important concepts [38, 41]. Cost-sensitive approaches (e.g., [42]) impose costs on misclassified instances. Determining whether a node is buggy or clean is a binary classification problem. Accordingly, we define $c$(min, maj) as the cost of misclassifying a majority class instance as a minority class instance, and $c$(maj, min) to denote the cost of the contrary case. In traditional classification techniques, both costs are the same. In cost-sensitive approaches, the cost of misclassifying minority instances is usually higher than the cost of misclassifying majority instances, i.e., $c$(maj, min) $>$ $c$(min, maj). He and Garcia [38] claimed that cost-sensitive approaches are superior to sampling approaches, because costs can be naturally imposed on imbalanced learning problems [43]. To balance our data, we therefore define the cost of a node ($i_t$) as follows:

$$\text{cost}(i_t) = \sum_{i}^{n} \frac{w_i}{n} w_t \frac{|C_m|}{|C|}, \quad i_t \in C_m, \tag{1}$$

where $w_i$ denotes the weight of the $i$th node; $w_t$ denotes the weight of node $i_t$; $C_m$ denotes all nodes (buggy and clean) in the $m$th class; and $C$ denotes all nodes. For simplicity, we assign all $w_i$ as one.

Our classifier comprises two classification techniques: the decision tree learning [44] and AdaBoost [45]. Decision tree is a supervised classification technique that classifies instances by constructing an if-else tree. Each interior node denotes a variable, and each leaf denotes a class. Adaboost is a meta-level learning technique that combines the outputs of weak classifiers into a weighted sum to predict the final

9) Apache Aries. http://aries.apache.org.

**Table 2** Subject

| Project | Single | Multiple | Graph | Fix | Percentage (%) |
|---|---|---|---|---|---|
| Aries | 37 | 263 | 1192 | 394 | 76.1 |
| Mahout | 47 | 253 | 1573 | 313 | 95.8 |
| Derby | 32 | 268 | 1981 | 1134 | 26.5 |
| Cassandra | 30 | 270 | 1468 | 2536 | 11.8 |
| Total | 146 | 1054 | 6214 | 4377 | 27.4 |

output. The training set of ClaFa is a set of labeled data ($\boldsymbol{f}_i$, $l_i$), where $\boldsymbol{f}_i$ is the feature vector, and $l_i$ is the label of a node. Adaboost repeatedly tunes its weights. We defined by $d_t(i)$ to denote the weight of the $i$th instance of the training data in the $t$th iteration. In the next iteration $t+1$, the weight is updated as follows:

$$d_{t+1}(i) = \frac{d_t(i)\exp(-\alpha_t h_t(\boldsymbol{f}_i) l_i)}{z_t}, \tag{2}$$

where $\alpha_t = \frac{1}{2}\ln(\frac{1-\varepsilon_t}{\varepsilon_t})$ is the weight updating parameter, $h_t(\boldsymbol{f}_i)$ is the prediction on feature vector $\boldsymbol{f}_i$, and $z_t$ is a normalization factor that ensures that the all new weights sum to one. Here, $\varepsilon_t$ is the error in the current model over the training set. After imposing cost $\text{cost}(i)$ on the $i$th instance, the above equation is modified to

$$d_{t+1}(i) = \frac{d_t(i)\exp(-\alpha_t \text{cost}(i) h_t(\boldsymbol{f}_i) l_i)}{z_t}. \tag{3}$$

#### 3.3.2 *Locating bugs*

For simplicity, a trained classifier is considered as a function ($y = f(\boldsymbol{x})$), where $\boldsymbol{x}$ denotes the vector of a node and $y$ denotes the prediction. To predict whether a node is buggy, ClaFa extracts its $\boldsymbol{x}$ and feeds the vector to its trained classifier. In particular, for a newly reported bug, ClaFa first uses its trained topic model to generate a feature vector ($F_{46}$). For each source file, it builds a program dependency graph, and for each node of the graph, it then extracts its node features, local features, and global features. Furthermore, ClaFa combines the above features into a vector ($\boldsymbol{x}$). Given $\boldsymbol{x}$ as the input, ClaFa predicts whether the node is buggy or clean. In particular, if a prediction value of a node is greater than a threshold (0.5), ClaFa determines that the node is buggy.

## 4 Evaluation

### 4.1 Research question

For a fair comparison, we must align inputs and outputs in a controlled experiment. The inputs of ClaFa differ from both spectra-based approaches (which require test cases), and the outputs of ClaFa differ from IR-based approaches (which cannot detect faults within source files). The different inputs and outputs preclude a fair comparison of ClaFa and prior spectra-based or IR-based approaches. Instead, we pose the following research questions.

(RQ1) How effectively does ClaFa detect faults in source files (Subsection 4.3.1)?
(RQ2) How does the local-feature depth affect the effectiveness (Subsection 4.3.2)?
(RQ3) How effectively does ClaFa detect faults, after learning from other projects (Subsection 4.3.3)?
(RQ4) What are the best discriminating features of buggy nodes (Subsection 4.3.4)?
(RQ5) How does ClaFa compare with other classification techniques (Subsection 4.3.5)?
(RQ6) What is the impact of bug reports (Subsection 4.3.6)?

### 4.2 Setup

#### 4.2.1 *Dataset*

Table 2 shows the subjects of our evaluations (300 randomly selected bug fixes from each project). The "Single" and "Multiple" columns list the number of bug fixes that modify single methods and more than

**Table 3** Overall effectiveness of ClaFa

| Project | Precision | Recall | $f$-score | The area under ROC |
|---|---|---|---|---|
| Aries | 0.772 | 0.818 | 0.787 | 0.647 |
| Mahout | 0.856 | 0.888 | 0.871 | 0.619 |
| Derby | 0.902 | 0.924 | 0.912 | 0.647 |
| Cassandra | 0.892 | 0.917 | 0.903 | 0.650 |

one method, respectively. Column "Graph" list the number of PDGs (over six thousand PDGs in total). Column "Fix" lists the total number of bug fixes. Column "Percentage" is calculated as $\frac{300}{\text{Fix}}$.

#### 4.2.2 *Metric*

Comparing our predictions against the gold standard of the faults (taken as the modified nodes of each bug fix), we classify all the nodes into false negatives (FNs), false positives (FPs), true negatives (TNs), and true positives (TPs). Based on the results, we measured our classifier by the following metrics:

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \tag{4}$$

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \tag{5}$$

$$f\text{-score} = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}. \tag{6}$$

ClaFa classifies nodes into clean and buggy ones. For each type of nodes, we calculate the precisions, recalls, and $f$-scores, followed by their weighted averages. The weight of each type is the proportion of its nodes over the total number of nodes.

### 4.3 Empirical result

#### 4.3.1 *RQ1 Effectiveness of* ClaFa

**Setting.** This research question explores the effectiveness of ClaFa in bug detection. For this purpose, we apply ClaFa to bug detection for each project listed in Table 2. Di Nucci et al. [46] mentioned that the traditional $n$-fold cross validation ignores the time sequences of the data instances. To avoid this problem, we conduct a time-aware evaluation [47, 48]. In particular, we sort the bug fixes in Table 2 by their issue numbers. After sorting, we use the top 80% bug fixes as the training data, and reserve the remaining bug fixes as the testing data. The effectiveness of ClaFa is evaluated by the precision, recall, $f$-scores, and the area under the receiver operating curve (ROC).

**Results.** The training data and the testing data consume approximately 1 gigabyte of storage per project. The classifier is trained after several hours, but once trained, it predicts the state of a node within seconds. Table 3 shows the overall results. Our ClaFa achieve high $f$-scores for all the projects. Although a fair comparison with spectra-based and IR-based approaches is infeasible, an indirect comparison can provide a reference. The prior studies [2, 49, 50] show that spectra-based approaches can predict only one fault location effectively. We suspect that when a buggy file has multiple faults their $f$-scores shall be low. Meanwhile, the $f$-score of a recent IR-based approach [51] is 0.641. As ClaFa locates finer faults than IR-based approaches or spectra-based approaches, our results show that ClaFa already achieves reasonably accuracies. In summary, our results show that ClaFa is able to predict buggy nodes with reasonable precisions and recalls.

#### 4.3.2 *RQ2 Impact of the depth parameter*

**Setting.** As mentioned in Subsection 3.2.3, $F_6$ to $F_{25}$ are extracted from the $k$ deep incoming or outgoing nodes. To understand the impact of the $k$-depth parameter, we change the $k$-depth from one to ten and investigate its impact on ClaFa. At each step, we record the precisions, the recalls, and the $f$-scores in the four projects, and analyze the optimized depth.

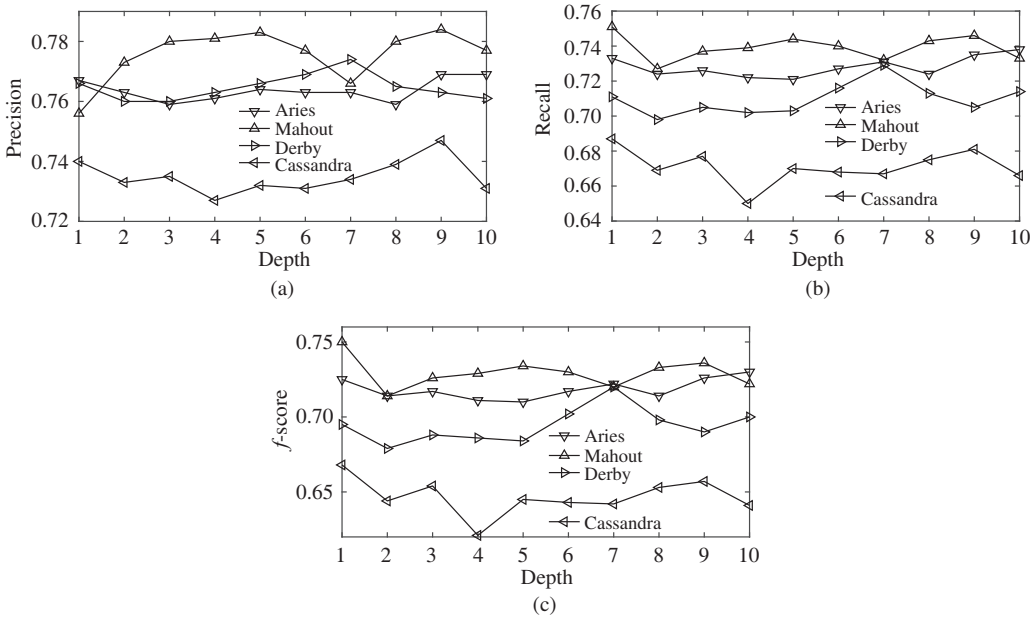



**Figure 5** The impacts of our depth parameter. (a) Precision; (b) recall; (c) $f$-score.

**Table 4** Learning from other projects

| Project | Aries | Mahout | Derby | Cassandra | Combination |
|---|---|---|---|---|---|
| Aries | – | 0.445 | 0.431 | 0.485 | 0.459 |
| Mahout | 0.467 | – | 0.440 | 0.428 | 0.439 |
| Derby | 0.519 | 0.439 | – | 0.507 | 0.479 |
| Cassandra | 0.496 | 0.468 | 0.457 | – | 0.463 |

**Results.** The impacts of varying the $k$-depth are shown in Figure 5. The $k$-depth parameter little affects the precisions, recalls and $f$-scores of the four projects. The three measures follow different trends. For example, the $f$-scores of Aries and Derby are optimized at a depth of ten, whereas those of Mahout and Cassandra are maximized at unity depth. We suspect that the depth reflects the complexity of bugs in different projects. Most projects yield favorable results at a depth of nine. Consequently, the depth is set to 9 in other research questions.

#### 4.3.3 *RQ3 Learning from other projects*

**Setting.** To explore this research question, we use the data of each project as the testing data, and the data from other projects and their combinations as the training data. We then analyze the changes in the $f$-scores under different settings.

**Results.** Table 4 shows the $f$-scores during learning from other projects. Each row and column denote the source of the testing and training data, respectively. For example, when the classifier is trained on Mahout and applied to the prediction of Aries faults, the obtained $f$-score is 0.445. Comparing the results of Tables 4 and 3, we find that the effectiveness is reduced when the model is learnt from the date of other projects. Column "combination" lists the $f$-scores when all the other projects are used as the training data. For example, when the data of Aries are predicted by the model that is trained on the data of Mahout, Derby, and Cassandra, the $f$-score is 0.459. Introducing more data for training does not always improve $f$-scores of the projects. In summary, our results show that predicting the buggy nodes of projects other than the training projects reduces the effectiveness of a trained model. Furthermore, the effectiveness may not be improved, by naively adding more training data. However, some training-testing pairs yield higher $f$-scores than others. This suggests that when a project does not have previous bug fixes, ClaFa can be feasibly trained on the data of similar projects.

**Table 5** Top ten features, ranked in the order of their importance in the clean versus buggy classification[a)b)c)d)]

| Rank | Aries | Mahout | Derby | Cassandra |
|---|---|---|---|---|
| 1 | g, $F_5$, o, $\overset{c}{\leftarrow}$ | $F_{46}$ | g, $F_5$, o, $\overset{c}{\leftarrow}$ | g, $F_5$, o, $\overset{c}{\leftarrow}$ |
| 2 | l, $F_5$, o, $\overset{c}{\leftarrow}$ | g, $F_1$, o, $\overset{d}{\leftarrow}$ | g, $F_3$, o, $\overset{c}{\leftarrow}$ | g, $F_5$, o, $\overset{c}{\leftarrow}$ |
| 3 | l, $F_7$, o, $\overset{c}{\leftarrow}$ | l, $F_1$, o, $\overset{d}{\leftarrow}$ | l, $F_5$, o, $\overset{c}{\leftarrow}$ | g, $F_5$, i, $\overset{c}{\leftarrow}$ |
| 4 | l, $F_6$, i, $\overset{c}{\leftarrow}$ | l, $F_1$, o, $\overset{c}{\leftarrow}$ | l, $F_3$, o, $\overset{c}{\leftarrow}$ | g, $F_5$, i, $\overset{c}{\leftarrow}$ |
| 5 | g, $F_3$, i, $\overset{c}{\leftarrow}$ | g, $F_4$, o, $\overset{c}{\leftarrow}$ | l, $F_7$, o, $\overset{c}{\leftarrow}$ | l, $F_5$, i, $\overset{c}{\leftarrow}$ |
| 6 | l, $F_3$, i, $\overset{c}{\leftarrow}$ | g, $F_1$, i, $\overset{c}{\leftarrow}$ | g, $F_2$, o, $\overset{c}{\leftarrow}$ | $F_{46}$ |
| 7 | l, $F_2$, o, $\overset{c}{\leftarrow}$ | n, $F_1$ | l, $F_2$, o, $\overset{c}{\leftarrow}$ | l, $F_5$, i, $\overset{d}{\leftarrow}$ |
| 8 | g, $F_2$, o, $\overset{c}{\leftarrow}$ | l, $F_1$, i, $\overset{c}{\leftarrow}$ | g, $F_4$, o, $\overset{c}{\leftarrow}$ | n, $F_1$ |
| 9 | l, $F_3$, o, $\overset{c}{\leftarrow}$ | g, $F_5$, o, $\overset{c}{\leftarrow}$ | l, $F_4$, o, $\overset{c}{\leftarrow}$ | g, $F_4$, o, $\overset{c}{\leftarrow}$ |
| 10 | g, $F_3$, o, $\overset{c}{\leftarrow}$ | g, $F_1$, o, $\overset{c}{\leftarrow}$ | g, $F_2$, o, $\overset{c}{\leftarrow}$ | l, $F_7$, o, $\overset{c}{\leftarrow}$ |

a) n: node feature; l: local feature; g: global feature.

b) $F_1$: main code name; $F_2$: node type; $F_3$: number of API code names; $F_4$: number of client code names; $F_5$: code names mentioned in bug reports; $F_6$: in-degree; $F_7$: out-degree; $F_{46}$: categories of bug reports.

c) i: incoming; o: outgoing.

d) $\overset{c}{\leftarrow}$: control dependency; $\overset{d}{\leftarrow}$: data dependency.

#### 4.3.4 *RQ4 Identification of important features*

**Setting.** This research question explores the features that contribute to our high effectiveness. The results will provide insights into the nature of bugs. In particular, we rank the features by their Pearson correlation coefficients [52]. We then compare the top ten features of all projects and select those with commonality to all projects. This study is not intended to further tune the effectiveness of ClaFa, which can be better achieved by the correlation-based feature selection [53]. Indeed, even the default settings of the classifier yield satisfactory results in RQ1. We believe that all features are valuable, and removing any one of them will reduce the overall effectiveness, as observed in Subsection 4.3.6.

**Results.** Table 5 shows the top ten features in the four projects. As shown in Table 1, our features are derived from several basic features (i.e., $F_1$ to $F_7$). To improve the presentation, Table 5 shows how features are derived from the features, instead of giving their number. For example, $F_8$ is written as "l, $F_2$, i, $\overset{d}{\leftarrow}$" meaning that $F_8$ is derived from $F_2$ and is a local feature calculated from the incoming-data-dependent nodes. From Table 5, we obtain the following findings.

(1) The node features alone cannot satisfactorily determine buggy nodes from clean ones. In total, only two node features rank among the top features, indicating whether a node is buggy is difficult to determine by inspecting only the node. This result highlights the importance of ClaFa, which accurately extracts local and global features. Without such features, the effectiveness of our approach can be significantly reduced.

(2) The quality of bug reports is important. In Table 5, $F_5$ and $F_{46}$ are related to bug reports. As the effectiveness of IR-based approaches relies on the quality of bug reports [5], the quality of bug reports matters, when locating faults in buggy files. As an extreme case, seven of the top ten features in the Cassandra project are derived from $F_5$, but this feature is non-dominant in other projects, confirming that only bug reports alone are insufficient for locating finer faults.

(3) Calling APIs can introduce bugs. Although reusing APIs significantly reduces the programming effort, existing empirical studies (e.g., [12]) show that many bugs are related to wrong API usages. Our results show that API-related features such as $F_3$ and $F_1$ are useful for locating faults. In particular, $F_3$ appears in the top ten features of Aries and Derby, but is not ranked in Mahout. In fact, six of the ten top features in the Mahout project are derived from $F_1$, which can be related to APIs (note that a code name can be an API code name).

(4) Most of the features are related to outgoing nodes. In all projects, most of the features are more related to outgoing nodes. As an extreme case, all top ten features of Derby are related to outgoing nodes. After inspecting some bug fixes, we identify two types of bug reports: outsider reports (often submitted by users) and insider reports (usually submitted by the project programmers). Although users

have limited knowledge of the implementation details, their reports typically provide error messages for locating faults. As faults often appear before the messages, the outgoing nodes have stronger impacts ion the fault localization than the incoming nodes. In contrast, the programmers submitting the insider reports fully understand the implementation details, so their reports typically explain why faults occur. As such explanations often appear before faults, the incoming nodes have stronger impacts on the fault localization than the outgoing nodes. Other important factors, such as code structures and semantics, are also worthy exploring in future work.

(5) Most of the features are related to control dependencies. Typically, control dependencies are related to code structures, whereas data dependencies are related to input and output values. For example, various approaches that detect legal call sequences of APIs (e.g., [54]) are more related to control dependencies. Our results indicate that exploring more control-dependency bugs is a promising prospect.

In summary, whether a node is buggy or clean cannot be discerned from the node features alone. The features in bug reports and the called APIs are more effective in fault localization than other types of features. The outgoing control dependencies of a node can also usefully determine whether that node is buggy.

#### 4.3.5 *RQ5 Other classification techniques*

**Setting.** In this research question, we exchange our classifier with related classifiers. As the Adaboost boosts various classifiers, we eliminate it from this study, and compare the effectiveness of various internal classifiers. Many off-the-shelf classifiers are implemented in the WEKA suite of machine learning algorithms. Among these classifiers, we select the following for comparison with ClaFa.

(1) SMO [55] is a sequential optimization algorithm that trains a support vector machine (SVM). An SVM [56] represents a data point as a vector, and the training process searches for one or more hyperplanes that split the data points. The hyperplanes are used for classifying new data points.

(2) Naive Bayes [57] is a probabilistic classifier based on Bayes' theorem. During training, a naive Bayes classifier constructs a graph model representing the dependencies among observations, and then estimates the probability distributions of the observations. The graph model is used for predicting new data points.

(3) Decision table [58] is a compact model for constructing complex rule sets and their corresponding actions. Given a set of data points, it constructs rules based the distributions of their features. The constructed decision table is used for predicting new data points.

(4) Logistic [59] is a multinomial logistic regression model. During training, the regression model minimizes the estimated prediction errors. Given a set of independent variables, it predicts the probabilities of the categories of dependent variables.

These classifiers are selected because they are widely applied in software engineering papers (e.g., [9,28]). In this study, all classifiers are operated with their default settings. For different thresholds, we compare ROC and precision-recall (PR) curves of the five classifiers.

**Results.** Figures 6 and 7 show the ROC curves and the PR curves, respectively. As mentioned in Subsection 3.3.1, the internal classifier of ClaFa is a decision tree. The findings are summarized below.

(1) The logistic classifier outperformed the other classifiers on Cassandra. The top ten features of Cassandra differ from those of the other three projects (see Table 5). We suspect that the features in this project are intrinsically more suited to logistic regression than to other classifiers. When programmers apply ClaFa to bug detections in their own projects, they can tune its internal classifier to suite the nature of their debug features.

(2) ClaFa is the optimum classifier on the other three projects. The left sides of our ROC and PR curves are above the left sides of the other classifiers. Although the curves intertwine, Fawcett's analysis [60] implies that the shape of our curves is optimal in the case of highly imbalanced data.

(3) There remains much space for improvement. As the ROC areas are not maximized, all classifiers are sub-optimal. The ROC curves are intertwined, suggesting that multiple classifiers can be interpolated

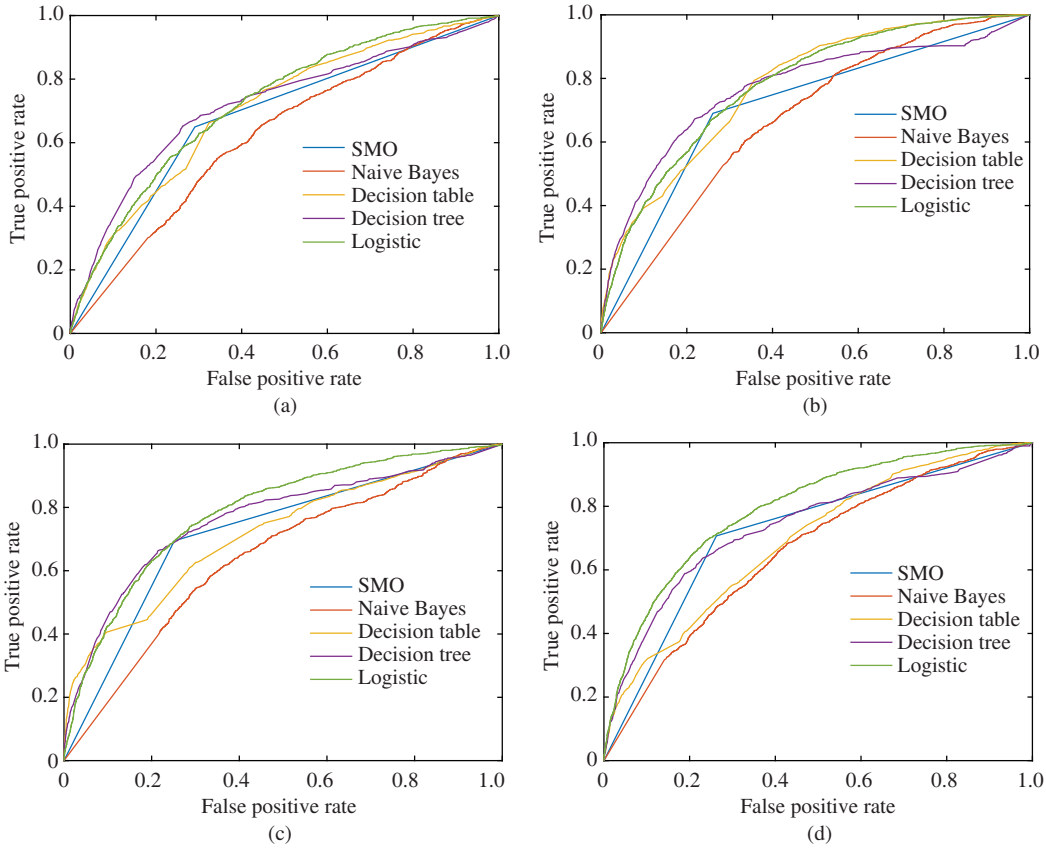



**Figure 6** (Color online) ROC. (a) Aries; (b) Mahout; (c) Derby; (d) Cassandra.

by existing techniques (e.g., [61]). However, as the curves closely resemble each other, we suspect that the combining the classifiers will little improve the results. This issue is further discussed in Section 5.

Figures 6 and 7 show that the differences among several classifiers (e.g., decision tree and logistic) are minor, and the classifiers could not be distinguished by their $f$-scores and AUC values. Consequently, we answer this research question by analyzing the ROC and PR graphs.

Ghotra et al. [62] evaluated the effectiveness of buggy-file prediction by different classifiers. Although their research goal differs from ours, some of their findings are consistent with ours. For example, they report that logistic (SL) is one of the best classifiers, and SMO is among the bottom classifiers. Other findings of Ghotra et al. [62] differed from ours, owing to the different settings in the two studies. Whereas we compared individual classifiers, they compared both individual classifiers and their combinations. They report that some combinations are significantly better than individual ones (e.g., Bag+J48).

In summary, ClaFa is the best classifier for Aries, Derby, and Mahout, and Logistic is the best classifier for Cassandra. However, the differences are minor, indicating that tuning classifiers may not achieve significant improvement. This finding is unsurprising, because Hall et al. [63] also reported that simple models such as logistic achieved even better results than complicated models such as SVM. Instead, we expect that exploring more features is effective for better results.

#### 4.3.6 *RQ6 Impact of bug reports*

**Setting.** We supplement the graph features with the information from bug reports, which are used in existing approaches (e.g., [64]). In Subsection 4.3.4, our results show that the features from bug reports are important in fault localization, which somewhat overshadows the significance of ClaFa. In this research question, we reveal the true effectiveness of the graph features, removing all features from bug reports. Specifically, we remove $F_{46}$, $F_5$ from Table 1, and all features that are derived from $F_5$. We then analyze the precisions, recalls, and $f$-scores to understand the classification potential of the graph

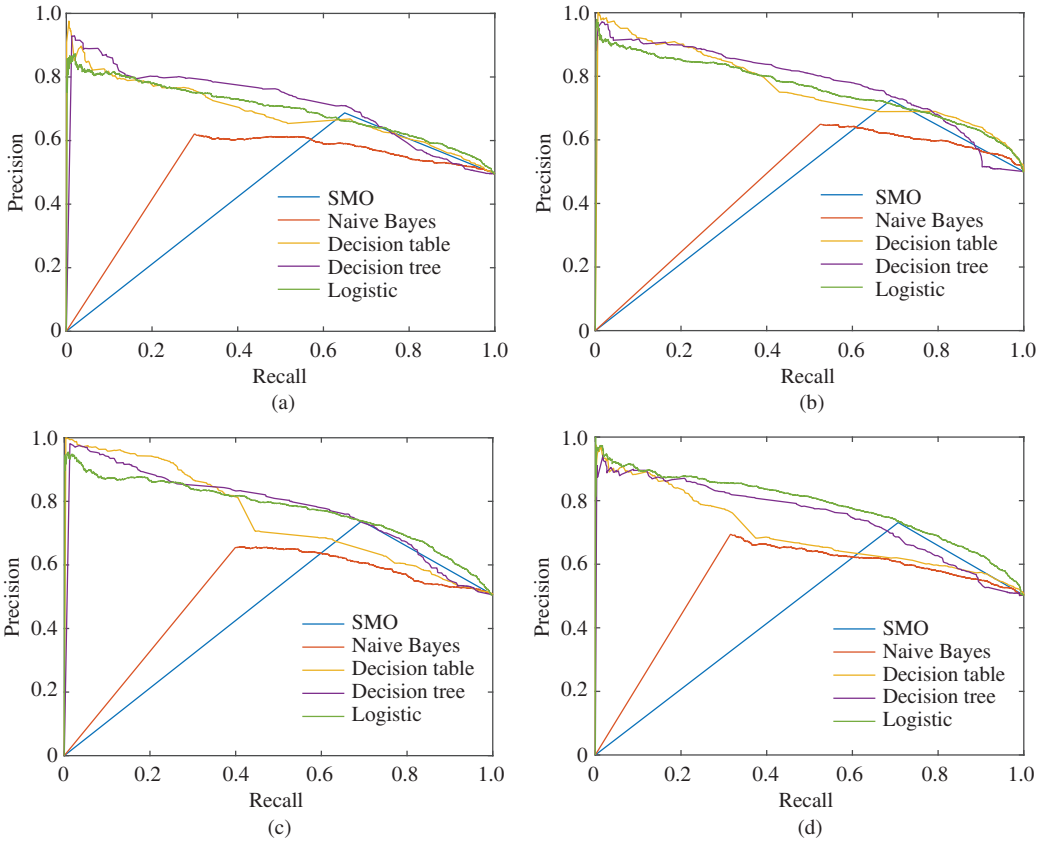



**Figure 7** (Color online) Precision-recall graph. (a) Aries; (b) Mahout; (c) Derby; (d) Cassandra.

**Table 6** The results without bug reports

| Project | Precision | Recall | $f$-score |
|---|---|---|---|
| Aries | 0.691 | 0.664 | 0.651 |
| Mahout | 0.710 | 0.679 | 0.668 |
| Derby | 0.735 | 0.672 | 0.652 |
| Cassandra | 0.706 | 0.657 | 0.631 |

features.

**Results.** Table 6 shows the results. Remove the features from bug reports reduced all the precision, recall, and $f$-score. Wang et al. [5] reported that the overall effectiveness of their compared IR-based approach [65] is severely degraded, when the bug reports lacked code entity names. ClaFa achieve reasonable high $f$-score even without inputs from bug reports (Table 6).

Although Table 5 show that more top features are related to bug reports in Cassandra than other projects, the $f$-score on this project is less reduced by removing the bug-report features than $f$-scores of the other project. Note that Table 5 does not present the weight of each top feature. The results of learning classification model by code features alone are consistent (see Table 6), but the impacts of adding more features are apparently complicated.

#### 4.3.7 *Threats to validity*

The threats to internal validity include the experimental bias of labeling faults in buggy files. For example, when programmers fix a bug, they somewhat also modify clean code lines (e.g., refactoring). Another threat is missed bug fixes in commits. Although such cases can be rare, they will negatively impact on the results. The threat could be reduced by introducing manual inspection in future work. Meanwhile, the threats to external validity include the selected subjects, which are all open-source projects. This

threat could be reduced by replicating our studies on commercial projects.

## 5 Discussion and future work

**Exploring more basic features.** As shown in Table 1, our basic features are limited. Previous researchers explored other data sources such as stack traces [66], version histories [67], and commit activities [68]. They also proposed useful metrics [68] for predicting buggy code. When we introduce more features in future work, we plan to borrow ideas from these approaches. To handle the complexity of features, we must allow features with variable lengths. For example, mined specifications (e.g., [54]) are widely used in the detection of bugs related to wrong API sequences. To detect such bugs, the API call sequences can be encoded into feature vectors. As client code methods can call different APIs, feature vectors of the encoded API call sequences are length-variable. To fully leverage such features in future, we plan to tune our classification technique or introduce more advanced techniques.

**Definition of bugs and its measures.** Outside Apache projects, programmers can fail to maintain the links between the bug reports and their fixes [69]. Even if programmers maintain the links, they can define bugs in different ways, and their definitions can be disputed by researchers [70, 71]. These inconsistent definitions can lead to different labels of faulty nodes. Here, the effectiveness of ClaFa is evaluated only on the definitions of programmers. Evaluate ClaFa under other definitions would help to assess the rationality of these definitions. A reasonable definition shall be consistent, and should therefore yield better results than a dubious definition. We plan to explore this issue in future work.

**Integrating with automatic program repair.** Automatic program repair (e.g., [72]) has recently become a research hotspot, but remains limited in scope (e.g., [73]). Automatic program repair typical invokes many iterations of spectra-based fault localization to locate the buggy lines of mutated candidates. This time-consuming process might partially explain why only limited number of bug fixes by the prior approaches. ClaFa reduces the time of fault localization by removing the need to execute test cases. In addition, spectra-based approaches cannot effectively locate multiple faults. ClaFa resolves this problem by eliminating the interference among multiple bugs. In future work, we plan to integrate ClaFa with automatic program repair. An automatic program-repair approach must know the nodes to be modified, but modified nodes do not necessarily indicate bugs. To assist bug discrimination by the prior approach, we intend to locate the bug-causing nodes in an extended version of ClaFa.

**Tuning ClaFa.** Although ClaFa achieves high $f$-scores in our evaluation analysis, the results are obtained under the default settings of classifiers. The effectiveness of ClaFa could be further improved by fine-tuning. He and Garcia [38] recommended cost-sensitive approaches for handling imbalanced data, but sampling approaches might be more effective for our specific application. Researchers have proposed various techniques to automatically tune the parameters of a classifier [74–77]. In future work, we plan to tune ClaFa using these approaches.

**Combining with other fault-localization approaches.** Bug finding bugs has been widely researched. Even the 100-plus papers cited in Section 6 do not cover all approaches; moreover, many new approaches are proposed each year. As mentioned in Subsection 4.1, a fair controlled experiment is precluded because the inputs and outputs of ClaFa differ from those of other approaches. Despite the large number of available approaches, many programmers prefer to their own debugging experience [6]. To handle the problem, a feasible way is to combine existing approaches, instead of arguing the best approach. For example, Le et al. [78] combined IR-based and spectra-based approaches to improve the fault-localization results. In future work, we will explore the combination of ClaFa and other approaches in a real development context.

**Words and topics outside the vocabulary.** Our present evaluation trains our topic model and word embedding on all subjects, which excludes words or topics outside the vocabulary. In real usage, a bug report can contain new words or topics that never appear in the previous histories, which confound the trained models. This problem could be feasibly solved by re-training the models on the new bug reports. Alternatively, the new words and topics could be replaced with similar ones in the vocabulary.

In future work, we plan to explore both approaches.

## 6 Related work

**Spectra-based fault localization.** A typical spectra-based approach calculates the suspicious scores of buggy lines based on passed and failed tests. These approaches assume that if a code line often appears in failed tests, it probably contains a bug. This research topic has been intensively studied, and some early approaches (e.g., [79]) are published in 1980s [80]. Various research metrics (e.g., [81–83]) can calculate suspicious values of code lines, and their effectiveness has been explored in various empirical studies (e.g., [84, 85]). Besides these explicitly defined metrics for suspicious buggy code, researchers have explored the automated mining of metrics. For example, Wong and Qi [86] learnt a neural network for buggy code prediction. Approaches other than metrics have also improved the state-of-the-art situation. Hao et al. [29] proposed an approach that reduces the number of test cases while minimizing the impacts on fault localization. Mao et al. [87] removed irrelevant code by slicing before calculating suspicious buggy code. However, most of the published papers assumed that one bugs in each buggy file. Although some approaches (e.g., [3,88]) locate multiple faults, these multi-faults are typically assumed as independent [80]. Abreu et al. [3] proposed an approach that locates multiple faults, but their approach is evaluated on only the Siemens benchmark [4], on which faults are manually constructed and each faulty version contains exactly one fault. Although Abreu et al. [3] combined multiple versions to simulate multiple faults, the combination again assumed independent multiple faults, which do not represent true faults. Gao and Wong [89] proposed another approach that locates multiple faults, but their evaluation also generated multiple faults from single fault versions. Pearson et al. [30] argued that artificial faults cannot confirm the true effectiveness of spectra-based localizations. In particular, their collected traces may not reflect the interferences among multiple faults observed in empirical studies (e.g., [2,90]). As the problem lies in the underlying assumption, many researchers (e.g., [2, 90]) considered that spectra-based approaches cannot insufficiently locate multiple faults, although at least one fault can top a suspicious list. Perez et al. [91] claimed that over 82% of bug fixes are single-fault fixes. In their definition, a bug is single-faulted, if all tests affect at least one changed component of the fix [91]. They considered that single faults can modify multiple locations, which is inconsistent with our definitions. Moreover, the best results from spectra-based approaches are typically obtained only after many high-quality test cases. Just et al. [92] showed that if the test cases are of superior quality, the time of generating correct patches can be significantly reduced. Spectra-based approaches have been adapted with fewer test inputs [93, 94], but our approach focuses on a precise static analysis rather than test coverage. The multi-fault detection by our approach, which needs no test inputs, can complement spectra-based approaches.

**IR-based fault localization.** Typical IR-based approaches locate buggy files by comparing the bug reports with the source files. The bug reports are assumed to share similarity with their corresponding buggy files. The similarity is detected by various machine learning techniques such as TF-IDF [64], LDA [95], naive Bayes [28], and SVM [65, 96]. The early papers handled source files such as natural language texts, but more recent papers (e.g., [97]) have extracted the code structures (i.e., class, methods, variables, and comments) from source files to improve fault-localization accuracy. Besides natural language descriptions of bug reports, other data sources such as stack traces [66], version histories [67], and their combination [98] have been explored. Wang et al. [5] argued that the similarity lies in the code names that appear in both bug reports and source files. ClaFa is not an IR-based approach, as it does not calculate the similarity between bug reports and source files, although it extracts features from bug reports. Being built on accurate source analysis, ClaFa can determine the buggy locations in source files, completing the IR-based approaches.

**Model-based fault localization.** A model can define either legal usages (specs) or illegal usages (bug signatures). Ammons et al. [99] mined automata for APIs. Pandita et al. [100] refined their approach, while other researchers [54,101–103] mined graphs for specs. Robillard et al. [104] showed that automata and graphs are equivalent. These types of model-based fault localization can be reduced to the grammar

inference problem. Method-pair extraction, proposed by Li and Zhou [105], has been improved in more complicated contexts [106]. The approach of Engler et al. [107], which extracts frequent call sequences, has also been improved by advanced techniques [108,109]. Furthermore, mined sequences have been encoded as temporal logic [110,111]. This research line can be reduced to sequence mining [112]. All of the above approaches are based on call sequences. Ernst et al. [113] inferred invariants to define the variable rules. More informative specs can be obtained by combining the invariants with sequences [114], and spec-mining has been enriched in various test cases [115,116]. Brünink and Rosenblum [117] mined performance models from runtime traces. Zhong and Meng [26] empirically analyzed several open questions (e.g., the best formats of specs). Meanwhile, bug signatures have been mined in sequences [17] and graphs [18,20, 118–120]. Most of these approaches analyzed traces [17,18,20,120]; only a few approaches [20] analyzed the buggy source files. Hsu et al. [17] and Sun and Khoo [18] applied a frequency-based mining approach; Li and Ernst [20] extracted subgraphs; and Cheng et al. [119] applied discriminative graph mining. These approaches differ from our approach by applying different techniques and taking different inputs from our approach.

**Bug prediction.** Studies on bug prediction have applied well-known metrics [121] or self-defined metrics [122]. In a bug prediction, a bug is assumed to cause a metric violation. However, defining the metrics is a challenging task. Nagappan et al. [123] predicted module-level bugs using several metrics, but none of the metrics yielded high effectiveness on all projects. ClaFa is not built on known metrics. Bug prediction can also be based on commit activities [48,68], assuming that bugs can be identified by specific commit activities (e.g., bursting). Rahman et al. [124] showed that bug predicting by activities is not substantially better than counting the number of commits touching a source file. ClaFa differs from the above approaches in both its underlying assumptions and technical details. In addition, most of these approaches predict faults on courser levels (e.g., modules and classes) [63].

**Classification in software engineering.** Classification is an intensively studied technique in data mining, and commonly assists software engineering tasks such as requirement elicitation [125], development [126], testing [127], debugging [128], maintenance [129], and software reuse [130]. ClaFa is designed mainly for debugging and maintenance. Because it accurately analyzes source code, ClaFa predicts the faults within source files, complementing existing approaches (e.g., [9]).

## 7 Conclusion

Open-source communities accumulate many real bug fixes. Locating code faults using knowledge mined from bug these fixes, and is desired, but is rendered challenging by various technical limitations. This paper proposed an approach called ClaFa that combines graph analysis and classification to locate multiple faults in source files. First, it builds program dependency graphs from the bug fixes and compares them to detect buggy nodes. The buggy nodes are used to label the faults. ClaFa extracts the graph features of each node, and denotes them in a vector. A classification model is then trained on the labeled data. We have evaluated ClaFa on thousands of real bug fixes extracted from four popular open-source projects. Our results confirmed that ClaFa can locate multiple faults with reasonably high accuracy.

**Acknowledgements** This work was sponsored by National Key R&D Program of China (Grant No. 2018YFC0830500), National Nature Science Foundation of China (Grant No. 61572313), and Science and Technology Commission of Shanghai Municipality (Grant No. 15DZ1100305). We appreciated the anonymous reviewers for their constructive comments.

**References**

1 Hovemeyer D, Pugh W. Finding bugs is easy. In: Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 2004. 132–136
2 DiGiuseppe N, Jones J A. On the influence of multiple faults on coverage-based fault localization. In: Proceedings of International Symposium on Software Testing and Analysis (ISSTA), 2011. 210–220
3 Abreu R, Zoeteweij P, van Gemund A J C. Spectrum-based multiple fault localization. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, 2009. 88–99
4 Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. Empir Softw Eng, 2005, 10: 405–435

5 Wang Q, Parnin C, Orso A. Evaluating the usefulness of IR-based fault localization techniques. In: Proceedings of International Symposium on Software Testing and Analysis (ISSTA), 2015. 1–11
6 Johnson B, Song Y, Murphy-Hill E, et al. Why don't software developers use static analysis tools to find bugs? In: Proceedings of the International Conference on Software Engineering (ICSE), 2013. 672–681
7 Rochkind M J. The source code control system. IEEE Trans Softw Eng, 1975, 1: 364–370
8 Wu R, Zhang H, Kim S, et al. Relink: recovering links between bugs and changes. In: Proceedings of ESEC/FSE, 2011. 15–25
9 Tian Y, Lawall J, Lo D. Identifying linux bug fixing patches. In: Proceedings of the 34th International Conference on Software Engineering (ICSE), 2012. 386–396
10 Mei H, Zhang L. Can big data bring a breakthrough for software automation? Sci China Inf Sci, 2018, 61: 056101
11 Guo P J, Zimmermann T, Nagappan N, et al. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In: Proceedings of the International Conference on Software Engineering (ICSE), 2010. 495–504
12 Zhong H, Su Z. An empirical study on real bug fixes. In: Proceedings of the International Conference on Software Engineering (ICSE), 2015. 913–923
13 Martinez M, Monperrus M. Mining software repair models for reasoning on the search space of automated program fixing. Empir Softw Eng, 2015, 20: 176–205
14 Rahm E, Do H H. Data cleaning: problems and current approaches. IEEE Data Eng Bullet, 2000, 23: 3–13
15 Ottenstein K J, Ottenstein L M. The program dependence graph in a software development environment. ACM SIGPLAN Not, 1984, 19: 177–184
16 Tufano M, Palomba F, Bavota G, et al. There and back again: can you compile that snapshot? J Softw Evol Proc, 2017, 29: e1838
17 Hsu H-Y, Jones J A, Orso A. Rapid: identifying bug signatures to support debugging activities. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008. 439–442
18 Sun C, Khoo S-C. Mining succinct predicated bug signatures. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, 2013. 576–586
19 Hutchins M, Foster H, Goradia T, et al. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In: Proceedings of the International Conference on Software Engineering (ICSE), 1994. 191–200
20 Li J, Ernst M D. CBCD: cloned buggy code detector. In: Proceedings of the International Conference on Software Engineering (ICSE), 2012. 310–320
21 Fluri B, Wuersch M, PInzger M, et al. Change distilling: tree differencing for fine-grained source code change extraction. IEEE Trans Softw Eng, 2007, 33: 725–743
22 Mishne A, Shoham S, Yahav E. Typestate-based semantic code search over partial programs. In: Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 2012. 997–1016
23 Dagenais B, Hendren L J. Enabling static analysis for partial Java programs. In: Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 2008. 313–328
24 Yang Q, Wu X. 10 challenging problems in data mining research. Int J Info Tech Dec Mak, 2006, 05: 597–604
25 Zhong H, Wang X. Boosting complete-code tools for partial program. In: Proceedings of IEEE/ACM International Conference on Automated Software Engineering, 2017. 671–681
26 Zhong H, Meng N. Towards reusing hints from past fixes. Empir Softw Eng, 2018, 23: 2521–2549
27 Wang Y, Meng N, Zhong H. An empirical study of multi-entity changes in real bug fixes. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018
28 Kim D S, Tao Y D, Kim S H, et al. Where should we fix this bug? A two-phase recommendation model. IEEE Trans Softw Eng, 2013, 39: 1597–1610
29 Hao D, Xie T, Zhang L, et al. Test input reduction for result inspection to facilitate fault localization. Autom Softw Eng, 2010, 17: 5–31
30 Pearson S, Campos J, Just R, et al. Evaluating and improving fault localization. In: Proceedings of the International Conference on Software Engineering (ICSE), 2017. 609–620
31 Berglund A, Boag S, Chamberlin D, et al. XML path language (xpath). World Wide Web Consortium (W3C), 2003
32 Lovins J B. Development of a stemming algorithm. Mech Transl Comput Linguist, 1968, 11: 1–10
33 Newman D, Asuncion A, Smyth P, et al. Distributed algorithms for topic models. J Mach Learn Res, 2009, 10: 1801–1828
34 Nguyen A T, Nguyen T T, Al-Kofahi J, et al. A topic-based approach for narrowing the search space of buggy files from a bug report. In: Proceedings of IEEE/ACM International Conference on Automated Software Engineering, 2011. 263–272
35 Kuhn H W. The Hungarian method for the assignment problem. Naval Res Logist, 1955, 2: 83–97
36 Mikolov T, Chen K, Corrado G, et al. Efficient estimation of word representations in vector space. 2013. ArXiv: 1301.3781
37 Gu Z, Barr E T, Hamilton D J, et al. Has the bug really been fixed? In: Proceedings of the 32nd International Conference on Software Engineering (ICSE), 2010. 55–64
38 He H B, Garcia E A. Learning from imbalanced data. IEEE Trans Knowl Data Eng, 2009, 21: 1263–1284
39 Chawla N V, Bowyer K W, Hall L O, et al. SMOTE: synthetic minority over-sampling technique. J Artif Intell Res, 2002, 16: 321–357
40 Liu X-Y, Wu J X, Zhou Z-H. Exploratory undersampling for class-imbalance learning. IEEE Trans Syst Man Cybern

B, 2009, 39: 539–550
41 Mease D, Wyner A J, Buja A. Boosted classification trees and class probability/quantile estimation. J Mach Learn Res, 2007, 8: 409–439
42 Sun Y, Kamel M S, Wong A K C, et al. Cost-sensitive boosting for classification of imbalanced data. Pattern Recogn, 2007, 40: 3358–3378
43 Weiss G M. Mining with rarity: a unifying framework. ACM SIGKDD Explor Newsletter, 2004, 6: 7–19
44 Frank E. Pruning decision trees and lists. Dissertation for Ph.D. Degree. Hamilton: University of Waikato, 2000
45 Freund Y, Schapire R E. Experiments with a new boosting algorithm. In: Proceedings of International Conference on Machine Learning, San Francisco, 1996. 148–156
46 Di Nucci D, Palomba F, Tamburri D A, et al. Detecting code smells using machine learning techniques: are we there yet? In: Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution, and Reengineering, 2018. 612–621
47 Di Nucci D, Palomba F, de Rosa G, et al. A developer centered bug prediction model. IEEE Trans Softw Eng, 2018, 44: 5–24
48 Hassan A E. Predicting faults using the complexity of code changes. In: Proceedings of the International Conference on Software Engineering (ICSE), 2009. 78–88
49 Lucia L, Lo D, Jiang L, et al. Extended comprehensive study of association measures for fault localization. J Softw Evol Proc, 2014, 26: 172–219
50 Di Giuseppe N, Jones J A. Fault density, fault types, and spectra-based fault localization. Empir Softw Eng, 2015, 20: 928–967
51 Wang S, Liu T, Tan L. Automatically learning semantic features for defect prediction. In: Proceedings of the International Conference on Software Engineering (ICSE), 2016. 297–308
52 Benesty J, Chen J, Huang Y, et al. Pearson correlation coefficient. In: Noise Reduction in Speech Processing. Berlin: Springer, 2009. 1–4
53 Hall M A. Correlation-based feature selection for machine learning. Dissertation for Ph.D. Degree. 1999
54 Zhong H, Zhang L, Xie T, et al. Inferring resource specifications from natural language API documentation. In: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering, 2009. 307–318
55 Platt J C. Fast training of support vector machines using sequential minimal optimization. Advances in Kernel Methods, 1999. 185–208
56 Suykens J A K, Vandewalle J. Least squares support vector machine classifiers. Neural Process Lett, 1999, 9: 293–300
57 John G H, Langley P. Estimating continuous distributions in bayesian classifiers. In: Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence, 1995. 338–345
58 Kohavi R. The power of decision tables. In: Proceedings of the 8th European Conference on Machine Learning, 1995. 174–189
59 Le Cessie S, van Houwelingen J C. Ridge estimators in logistic regression. Appl Stat, 1992, 41: 191–201
60 Fawcett T. An introduction to ROC analysis. Pattern Recogn Lett, 2006, 27: 861–874
61 Flach P A, Wu S. Repairing concavities in roc curves. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence, 2005. 702–707
62 Ghotra B, McIntosh S, Hassan A E. Revisiting the impact of classification techniques on the performance of defect prediction models. In: Proceedings of the International Conference on Software Engineering (ICSE), 2015. 789–800
63 Hall T, Beecham S, Bowes D, et al. A systematic literature review on fault prediction performance in software engineering. IEEE Trans Softw Eng, 2012, 38: 1276–1304
64 Rao S, Kak A. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In: Proceedings of the 8th International Working Conference on Mining Software Repositories, 2011. 43–52
65 Zhou J, Zhang H, Lo D. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In: Proceedings of the International Conference on Software Engineering (ICSE), 2012. 14–24
66 Wong C, Xiong Y, Zhang H, et al. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME), 2014. 181–190
67 Sisman B, Kak A C. Incorporating version histories in information retrieval based bug localization. In: Proceedings of 9th IEEE Working Conference on Mining Software Repositories, 2012. 50–59
68 Kim S, Zimmermann T, Whitehead Jr E J, et al. Predicting faults from cached history. In: Proceedings of the 29th International Conference on Software Engineering (ICSE), 2007. 489–498
69 Bachmann A, Bird C, Rahman F, et al. The missing links: bugs and bug-fix commits. In: Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010. 97–106
70 Antoniol G, Ayari K, Di Penta M D, et al. Is it a bug or an enhancement? a text-based approach to classify change requests. In: Proceedings of Conference of the Center for Advanced Studies on Collaborative Research, 2008. 304–318
71 Herzig K, Just S, Zeller A. It's not a bug, it's a feature: how misclassification impacts bug prediction. In: Proceedings of the International Conference on Software Engineering (ICSE), 2013. 392–401
72 Weimer W, Nguyen T, Le Goues C, et al. Automatically finding patches using genetic programming. In: Proceedings of the International Conference on Software Engineering (ICSE), 2009. 364–374
73 Qi Y, Mao X, Lei Y, et al. The strength of random search on automated program repair. In: Proceedings of the 36th International Conference on Software Engineering (ICSE), 2014. 254–265

74 Sarro F, Di Martino S, Ferrucci F, et al. A further analysis on the use of genetic algorithm to configure support vector machines for inter-release fault prediction. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing, 2012. 1215–1220

75 Tantithamthavorn C, McIntosh S, Hassan A E, et al. Automated parameter optimization of classification techniques for defect prediction models. In: Proceedings of the International Conference on Software Engineering (ICSE), 2016. 321–332

76 Thornton C, Hutter F, Hoos H H, et al. Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2013. 847–855

77 Tantithamthavorn C, McIntosh S, Hassan A E, et al. The impact of automated parameter optimization on defect prediction models. IEEE Trans Softw Eng, 2019, 45: 683–711

78 Le T-D B, Oentaryo R J, Lo D. Information retrieval and spectrum based bug localization: better together. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015. 579–590

79 Shapiro E. Algorithmic program debugging. Dissertation for Ph.D. Degree. New Haven: Yale University, 1983

80 Wong W E, Gao R, Li Y, et al. A survey on software fault localization. IEEE Trans Softw Eng, 2016, 42: 707–740

81 Jones J A, Harrold M J, Stasko J. Visualization of test information to assist fault localization. In: Proceedings of the International Conference on Software Engineering (ICSE), 2002. 467–477

82 Naish L, Lee H J, Ramamohanarao K. A model for spectra-based software diagnosis. ACM Trans Softw Eng Methodol, 2011, 20: 1–32

83 Wong W E, Debroy V, Xu D. Towards better fault localization: a crosstab-based statistical approach. IEEE Trans Syst Man Cybern C, 2012, 42: 378–396

84 Abreu R, Zoeteweij P, van Gemund A J C. An evaluation of similarity coefficients for software fault localization. In: Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing, 2006. 39–46

85 Abreu R, Zoeteweij P, Golsteijn R, et al. A practical evaluation of spectrum-based fault localization. J Syst Softw, 2009, 82: 1780–1792

86 Wong W E, Qi Y. BP neural network-based effective fault localization. Int J Soft Eng Knowl Eng, 2009, 19: 573–597

87 Mao X, Lei Y, Dai Z, et al. Slice-based statistical fault localization. J Syst Softw, 2014, 89: 51–62

88 Dickinson W, Leon D, Podgurski A. Finding failures by cluster analysis of execution profiles. In: Proceedings of the International Conference on Software Engineering (ICSE), 2001. 339–348

89 Gao R, Wong W E. MSeer-an advanced technique for locating multiple bugs in parallel. IEEE Trans Softw Eng, 2019, 45: 301–318

90 Debroy V, Wong W E. Insights on fault interference for programs with multiple bugs. In: Proceedings of IEEE International Conference on Software Reliability Engineering, 2009. 165–174

91 Perez A, Abreu R, d'Amorim M. Prevalence of single-fault fixes and its impact on fault localization. In: Proceedings of IEEE International Conference on Software Testing, 2017. 12–22

92 Just R, Parnin C, Drosos I, et al. Comparing developer-provided to user-provided tests for fault localization and automated program repair. In: Proceedings of International Symposium on Software Testing and Analysis (ISSTA), 2018. 287–297

93 Campos J, Abreu R, Fraser G, et al. Entropy-based test generation for improved fault localization. In: Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE), 2013. 257–267

94 Perez A, Abreu R, van Deursen A. A test-suite diagnosability metric for spectrum-based fault localization approaches. In: Proceedings of the International Conference on Software Engineering (ICSE), 2017. 654–664

95 Lukins S K, Kraft N A, Etzkorn L H. Bug localization using latent dirichlet allocation. Inf Softw Tech, 2010, 52: 972–990

96 Wang S, Lo D, Lawall J. Compositional vector space models for improved bug localization. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME), 2014. 171–180

97 Saha R K, Lease M, Khurshid S, et al. Improving bug localization using structured information retrieval. In: Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE), 2013. 345–355

98 Wang S, Lo D. AmaLgam+: composing rich information sources for accurate bug localization. J Softw Evol Proc, 2016, 28: 921–942

99 Ammons G, Bodík R, Larus J R. Mining specifications. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2002. 4–16

100 Pandita R, Xiao X, Zhong H, et al. Inferring method specifications from natural language API descriptions. In: Proceedings of the 34th International Conference on Software Engineering (ICSE), 2012. 815–825

101 Nguyen T T, Nguyen H A, Pham N H, et al. Graph-based mining of multiple object usage patterns. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, 2009. 383–392

102 Nguyen H V, Nguyen H A, Nguyen A T, et al. Mining interprocedural, data-oriented usage patterns in JavaScript web applications. In: Proceedings of the International Conference on Software Engineering (ICSE), 2014. 791–802

103 Corbett J C, Dwyer M B, Hatcliff J, et al. Bandera: Extracting finite-state models from Java source code. In: Proceedings of the 22nd International Conference on Software Engineering (ICSE), 2000. 439–448

104 Robillard M P, Bodden E, Kawrykow D, et al. Automated API property inference techniques. IEEE Trans Softw Eng, 2013, 39: 613–637

105 Li Z, Zhou Y. PR-Miner: automatically extracting implicit programming rules and detecting violations in large

software code. In: Proceedings of the 10th European Software Engineering Conference Held Jointly With 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005. 306–315
106 Saied A, Benomar O, Abdeen H, et al. Mining multi-level API usage patterns. In: Proceedings of IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2015. 23–32
107 Engler D, Chen D, Chou A. Bugs as inconsistent behavior: a general approach to inferring errors in systems code. In: Proceedings of 18th Symposium on Operating Systems Principles, 2001. 57–72
108 Wasylkowski A, Zeller A, Lindig C. Detecting object usage anomalies. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, 2007. 35–44
109 Ramanathan M, Grama A, Jagannathan S. Path-sensitive inference of function precedence protocols. In: Proceedings of the 29th International Conference on Software Engineering (ICSE), 2007. 240–250
110 Maoz S, Ringert J O. GR(1) synthesis for LTL specification patterns. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, 2015. 96–106
111 Lemieux C, Park D, Beschastnikh I. General LTL specification mining. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015. 81–92
112 Agrawal R, Srikant R. Mining sequential patterns. In: Proceedings of the 11th International Conference on Data Engineering, 1995. 3–14
113 Ernst M D, Perkins J H, Guo P J, et al. The Daikon system for dynamic detection of likely invariants. Sci Comput Programm, 2007, 69: 35–45
114 Le T, Le X, Lo D, et al. Synergizing specification miners through model fissions and fusions. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015. 115–125
115 Dallmeier V, Knopp N, Mallon C, et al. Generating test cases for specification mining. In: Proceedings of the 19th International Symposium on Software Testing and Analysis, 2010. 85–96
116 Pradel M, Gross T R. Leveraging test generation and specification mining for automated bug detection without false positives. In: Proceedings of the International Conference on Software Engineering (ICSE), 2012. 288–298
117 Brünink M, Rosenblum D S. Mining performance specifications. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016. 39–49
118 Pham N H, Nguyen T T, Nguyen H A, et al. Detecting recurring and similar software vulnerabilities. In: Proceedings of the International Conference on Software Engineering (ICSE), 2010. 227–230
119 Cheng H, Lo D, Zhou Y, et al. Identifying bug signatures using discriminative graph mining. In: Proceedings of International Symposium on Software Testing and Analysis (ISSTA), 2009. 141–152
120 Zuo Z, Khoo S-C, Sun C. Efficient predicated bug signature mining via hierarchical instrumentation. In: Proceedings of International Symposium on Software Testing and Analysis (ISSTA), 2014. 215–224
121 El Emam K, Melo W, Machado J C. The prediction of faulty classes using object-oriented design metrics. J Syst Softw, 2001, 56: 63–75
122 Marcus A, Poshyvanyk D, Ferenc R. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. IEEE Trans Softw Eng, 2008, 34: 287–300
123 Nagappan N, Ball T, Zeller A. Mining metrics to predict component failures. In: Proceedings of the International Conference on Software Engineering (ICSE), 2006. 452–461
124 Rahman F, Posnett D, Hindle A, et al. Bugcache for inspections: hit or miss? In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, 2011. 322–331
125 Hayes J H, Dekhtyar A, Osborne J. Improving requirements tracing via information retrieval. In: Proceedings of 11th IEEE International Requirements Engineering Conference, 2003. 138–147
126 Williams C C, Hollingsworth J K. Automatic mining of source code repositories to improve bug finding techniques. IEEE Trans Softw Eng, 2005, 31: 466–480
127 Last M, Friedman M, Kandel A. The data mining approach to automated software testing. In: Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2003. 388–396
128 Podgurski A, Leon D, Francis P, et al. Automated support for classifying software failure reports. In: Proceedings of the 25th International Conference on Software Engineering (ICSE), 2003. 465–475
129 Hindle A, German D M, Holt R. What do large commits tell us? a taxonomical study of large commits. In: Proceedings of the 2008 International Working Conference on Mining Software Repositories, 2008. 99–108
130 Menzies T, Di Stefano J S. More success and failure factors in software reuse. IEEE Trans Softw Eng, 2003, 29: 474–477