The Journal of Systems and Software 148 (2019) 88–104

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Does the fault reside in a stack trace? Assisting crash localization by predicting crashing fault residence

Yongfeng Gu[a], Jifeng Xuan[a,*], Hongyu Zhang[b], Lanxin Zhang[a], Qingna Fan[c], Xiaoyuan Xie[a], Tieyun Qian[a]

[a] School of Computer Science, Wuhan University, Wuhan 430072, China
[b] School of Electrical Engineering and Computer Science, The University of Newcastle, Callaghan NSW2308, Australia
[c] Ruanmou Edu, Wuhan 430079, China

ARTICLE INFO

Article history:
Received 27 February 2018
Revised 7 October 2018
Accepted 6 November 2018
Available online 6 November 2018

Keywords:
Crash localization
Stack trace
Predictive model
Crashing fault residence

ABSTRACT

Given a stack trace reported at the time of software crash, crash localization aims to pinpoint the root cause of the crash. Crash localization is known as a time-consuming and labor-intensive task. Without tool support, developers have to spend tedious manual effort examining a large amount of source code based on their experience. In this paper, we propose an automatic approach, namely CraTer, which predicts whether a crashing fault resides in stack traces or not (referred to as *predicting crashing fault residence*). We extract 89 features from stack traces and source code to train a predictive model based on known crashes. We then use the model to predict the residence of newly-submitted crashes. CraTer can reduce the search space for crashing faults and help prioritize crash localization efforts. Experimental results on crashes of seven real-world projects demonstrate that CraTer can achieve an average accuracy of over 92%.

1. Introduction

Software faults can hide everywhere in source code and could cause software crashes. Once a crash happens, a stack trace of the crash is logged to record the status of program execution. To fix the crash-causing fault (*crashing fault* or *fault* for short), developers need to locate the root cause of the crash in the source code. Such localization is known as *crash localization* (Wu et al., 2014).

Crash localization is challenging. A stack trace consists of a runtime exception and a function call sequence. Crash localization takes the stack trace and the source code as input and outputs the location of the fault. Besides the stack trace, crash localization can leverage bug reports from bug tracking systems such as Bugzilla, or consulting websites such as StackOverflow. However, the information obtained from bug tracking systems or consulting websites can be incomplete or inaccurate; this makes it difficult to automate collection and validation. Hence, in this paper we only focus on the stack traces and source code. In an empirical study of Mozilla crash data, Wu et al. (2014) found that 59% to 67% of the crashing faults can be found in functions that are inside stack traces while 33% to 41% of crashing faults are outside stack traces. In general, localizing crashing faults that reside outside of stack traces requires more effort since developers need to examine the function call graph and to check a larger amount of source code. For example, Gong et al. (2014) studied crashes of Firefox 3.6. They found that by expanding the call depth by 1 (i.e., checking all the functions that are directly called by any functions in the stack trace), developers have to examine 624 more functions and only eight more crashing faults can be discovered. By expanding the call depth by 2 (i.e., checking all the functions that are two call steps away from any function in the stack trace), 964 more functions have to be examined and only five more crashing faults can be discovered. Although the number of discovered faults increases, the developers have to spend much more effort in examining a large number of functions outside the stack trace. Additionally, a crashing fault may associate with a hidden or private function that does not appear in the API reference document, which increases the difficulty of crash localization. Hence, predicting whether a crashing fault resides in a stack trace or not can assist developers to speed-up crash localization and to prioritize debugging efforts (Theisen et al., 2015).

In this paper, we proposed an automatic approach, namely CraTer (short for **Cra**sh de**Te**ctor), to address the problem of predicting crashing fault residence, which aims to predict whether a crashing fault resides in a stack trace. This problem is modeled as a binary classification problem with two class labels: InTrace and

* Corresponding author.
E-mail addresses: yongfenggu@whu.edu.cn (Y. Gu), jxuan@whu.edu.cn (J. Xuan), hongyu.zhang@newcastle.edu.au (H. Zhang), lanxinzhang@whu.edu.cn (L. Zhang), qingna@ruanmou.net (Q. Fan), xxie@whu.edu.cn (X. Xie), qty@whu.edu.cn (T. Qian).

https://doi.org/10.1016/j.jss.2018.11.004
0164-1212/

OutTrace. That is, a crashing fault resides in or out of the statements that are recorded in a stack trace. For each crash, CraTer extracts 89 features from the faulty program as well as its stack trace to characterize the residence of the crashing fault in the stack trace. Examples of the features include the type of the exception in the stack trace and the number of files included in the source code. Each crash corresponds to a vector of 89 feature values. CraTer consists of two major phases: the training phase and the deployment phase. In the training phase, a predictive model is built by combining a decision tree classifier with a strategy of imbalanced data processing. In the deployment phase, the trained model is used to predict whether the crashing fault resides in the stack trace for a newly-submitted crash. The predicted results can assist the manual crash localization work performed by developers. For instance, consider a stack trace with 10 lines. Analyzing all frames and method calls that are listed in the stack trace requires the review of many lines of code. If our approach CraTer can identify the prediction result of InTrace, i.e., the crashing fault is predicted inside the stack trace, then we can focus on the review of the 10 lines of code recorded in the stack trace. This can save the time cost and the human labor.

We evaluated our approach CraTer on seven real-world, open-source Java projects: Apache Commons Codec, Ormlite-Core, JSqlParser, Apache Commons Collections, Apache Commons IO, Jsoup, and Mango. We seeded faults using program mutation techniques to mimic real crashes and randomly sample 500 crashes for ten times. The overall accuracy on all the crashes of seven projects reaches 92.7%. Experiments on each individual project show that our approach can correctly predict the residence of crashing faults with the accuracy ranging from 86.0% to 95.7%. The F-measure of InTrace and OutTrace for individual projects are from 65.0% to 87.9% and from 90.8% to 97.9%%, respectively. In our experiments, we also analyzed the most dominant features among the 89 features, which have the strongest correlation with the residence of the crashing faults. To find out the impact of different classifiers on the prediction results, we compared six classification algorithms as well as four imbalanced data processing strategies. We also showed the time cost and the saved effort of using CraTer.

This paper makes the following main contributions:

- We proposed an automatic approach, namely CraTer, to predict whether a crashing fault resides in a stack trace or not.
- We empirically evaluated CraTer on crashes from seven real-world, open-source projects. The results demonstrate that this approach can achieve an accuracy of over 92% on all the crashes under evaluation.

The rest of paper is organized as follows. Section 2 provides the background of our work. Section 3 details our predictive approach and its feature extraction. Experimental setup and results are presented in Section 4 and Section 5, respectively. We brief the threats to the validity in Section 6 and describe the related work in Section 7. Finally, we conclude the paper in Section 8.

2. Background

In this section, we briefly introduce the background of stack traces and crash localization.

2.1. Crashes and stack traces

Software may crash if an internal fault is triggered. Mainstream programming languages have their own exception handling mechanism that can throw exceptions due to internal faults and catch exceptions for further processing (Oliveira et al., 2018). Developers have to write source code to specify their steps to deal with exceptions (Li et al., 2018). Taking Java programs as an example, Java Virtual Machine (JVM) pushes a function into the stack if a function is called by the main program.[1] Once a crash appears, JVM aborts the program execution and outputs the function calls stored in the stack based on their call sequences. Modern software projects collect software crashes to facilitate program debugging and bug fixing. Many projects deploy a bug tracking system (such as Bugzilla[2] and Jira[3]) to enable users to submit a bug report to record the crash of projects. Large-scale crash reporting systems are also used to automatically collect crash reports from end users. Examples of such systems include Microsoft Windows Error Reporting System (Dang et al., 2012), Mozilla Crash Reporter,[4] and Fedora Analysis Framework.[5] Given a collected crash report, developers can reproduce the crashing scenario and then fix the faulty code. The major part of a collected crash report is a stack trace, which consists of a runtime exception and a function call sequence at the moment of the crash.

Fig. 1 shows a real-world crashing fault, Bug 718 in a widely-used open-source library, Apache Commons Math.[6] The exception type of this crash is `ConvergenceException` and is directly thrown from the function `evaluate()` in a class `ContinuedFraction`. According to the bottom of the stack trace, all functions in Fig. 1 are called by executing a function `inverseCumulativeProbability()` in a class `AbstractIntegerDistribution`.

A typical stack trace can be viewed as a list of $n+1$ frames, from Frame 0 to Frame n (Wu et al., 2014; Chen and Kim, 2015). Frame 0 records the thrown exception of the crash; Frames 1 to n represent the function call sequence. Each frame in the function call sequence is a tuple of a class name, a function name, and a line ID. These records directly indicate the position of a function call. For instance, the caller of the function `regularizedBeta()` at Frame 2 is `regularizedBeta()` at Frame 3; the callee of `regularizedBeta()` at Frame 2 is `evaluate()` at Frame 1.

According to a manually-written patch of Bug 718, the root cause of the crash locates at Line 122 in `ContinuedFraction.java`, which resides out of the stack trace. Thus, developers cannot directly identify where the crashing fault resides and have to carefully review all related code to find out the root cause. As recorded in Bug 718, it finally took 107 days from reporting the crash to finding out the patch. The large time cost and labor effort motivate us to address the problem in this paper, i.e., how to assist developers to locate the root cause. The idea is to automatically predict whether the crashing fault is recorded in the stack trace or not. Before manual crash localization, a developer can utilize our approach to find out a binary result, i.e., the fault is inside the stack trace, or outside. Then, based on the prediction, the developer can focus on related code rather than the whole project.

2.2. Crash localization

It is difficult to localize the root cause of a crash although the function call sequence is recorded in the stack trace. We show the reasons as follows. First, there exist many lines of potentially related code in the recorded functions in a stack trace; for instance, there exist 311 lines of code in all functions that are recorded in the stack trace in Fig. 1. Second, a function call sequence does not

[1] In Java programs, a *function* is usually called a method. In this paper, we keep using the term "function" to avoid the ambiguity between an "approach" and a "method".

[2] Bugzilla, http://bugzilla.org/.

[3] Jira, http://issues.apache.org/jira/.

[4] Mozilla Crash Reporter, http://crash-stats.mozilla.com/.

[5] Fedora Analysis Framework, http://retrace.fedoraproject.org/faf/summary/.

[6] Bug report of Math-718, http://issues.apache.org/jira/browse/MATH-718.

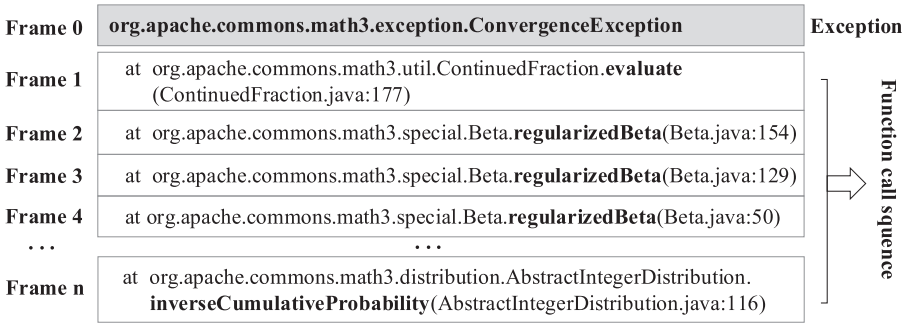

Fig. 1. Stack trace of Bug 718 in the project of Apache Commons Math.

directly link to a crashing fault due to the complexity of program structures (Wu et al., 2016). Third, a crashing fault may associate with a hidden or private function that does not appear in the API reference documentation.

Crash localization is important for program debugging. The goal of crash localization is to help developers find out the root cause of a crash based on the given stack trace. The input of crash localization is the stack trace and the source code while the output is a ranked list of suspicious functions, which may contain the crashing fault. Wu et al. (2014) proposed CrashLocator to synthesize the approximate complete execution traces by expending the function call sequence in the stack trace and then ranking suspicious functions according to their pre-defined suspicious ranking metric.

A related problem is spectrum based fault localization, which localizes faults based on passing and failing execution traces (Jones and Harrold, 2005; Rui et al., 2007; Lucia et al., 2014; Xuan et al., 2017b; Le et al., 2016). In contrast to spectrum based fault localization, crash localization can only leverage the input of the stack trace and the source code, rather than the program execution. This makes accurate crash localization difficult. Wu et al. (2014) showed that their tool CrashLocator, achieves 56.9% of the recall value of crash localization when examining the top-10 recommended functions. Searching for a fault under this recall value is time-consuming.

3. Proposed approach: CraTer

In this section, we present our proposed approach, namely CraTer, in four aspects: the class labeling, the overview of the proposed approach, the feature extraction, and the learning algorithms for imbalance issues.

3.1. Class labeling

The goal of our work is to predict crashing fault residence, i.e., identifying whether a crashing fault resides in the stack trace. We consider a crash whose corresponding crashing fault can be found in the stack trace as the InTrace class: there exists one frame in the stack trace, whose recorded class, function, and line ID are all matched with the faulty code. A crash whose crashing fault does not exist in the stack trace is considered as the OutTrace class. For example in Fig. 1, if the crashing fault locates at Line 154 in the function `regularizedBeta()` in the class `Beta`, then we label the crash as InTrace, because we can find that Frame 2 in stack trace is the position of fault; if the crashing fault is at Line 155 in `regularizedBeta()`, then we can find that none of frames in stack trace cover the fault, then we define the crash as OutTrace.

For an existing crash, we can directly identify whether the crash belongs to either the class InTrace or OutTrace by checking its bug-fixing log. We use these crashes as the training data. For a newly-submitted crash report, we aim to predict its class label. In this way, the original problem of whether the crashing fault resides in the stack trace is transformed into a binary classification problem. Meanwhile, the classes of InTrace and OutTrace are imbalanced: more crashing faults resides outside the stack traces. The imbalanced distribution between InTrace and OutTrace may hurt the performance of the predictive model. The major reason is that a model can hardly characterize the InTrace crashes (i.e., the minority class) and tend to misclassify InTrace crashes into OutTrace crashes (the majority class). It is challenging to build an effective predictive model with imbalanced data.

3.2. Overview

Fig. 2 depicts the overview of our proposed approach CraTer, which consists of two major phases: the *training phase* and the *deployment phase*. For each crash, we first extract 89 features to characterize the crash from its stack trace as well as its source code, then build a predictive model based on machine learning techniques in the training phase; next in the deployment phase, once a new crash report comes, we use the trained model to predict whether the fault resides in a corresponding stack trace.

3.2.1. Training phase

In the training phase, we take the faulty source code (i.e., the source code of a project that contains a fault) and its corresponding stack trace as the input and train a predictive model as the output.

Given the source code and the stack trace, we extract 89 features and its class label to form a feature vector with 89 features and one binary label, i.e., InTrace or OutTrace. Section 3.3 details the process of feature extraction. Based on feature vectors of all known crashes, we train a classifier in machine learning to identify InTrace or OutTrace. Generally, every binary classifier can be used in our approach. In our work, we choose to combine a decision tree algorithm with a SMOTE strategy for imbalanced classification. SMOTE is a well-known technique of imbalanced data processing, which re-constructs a balanced data distribution for the imbalanced data learning problem. The reason for this choice is that the decision tree performs well in our experiment (see Section 5) and its result is human-understandable; meanwhile, the SMOTE strategy is stable in handling imbalanced classes (Han et al., 2011), i.e., the imbalanced distribution of InTrace and OutTrace in our experiment (see Section 4).

3.2.2. Deployment phase

In the deployment phase, we take the model built in the training phase as well as a newly-submitted crash (including the source code and the corresponding stack trace) as input and then output the final prediction result: the crash is InTrace or OutTrace. Given a new crash, we also extract 89 features from both the source code and the stack trace. Then we use the predictive model to predict

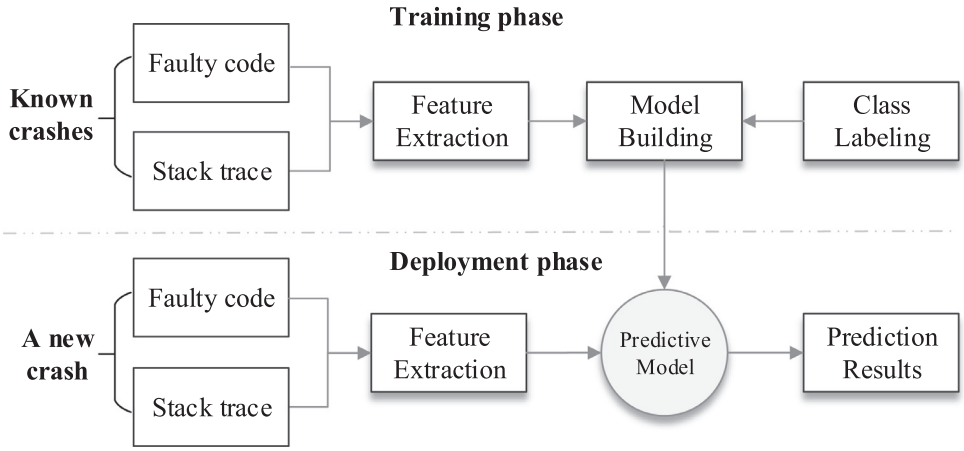

Fig. 2. Overview of CraTer for predicting the residence of the crashing fault.

its class label. This predicted class label could be used as a hint to support developers to assist their manual crash localization. We further elaborate our learning algorithms in Section 3.4.

3.3. Feature extraction

To build the predictive model, we extract 89 features from the given stack trace and the source code. As mentioned in Section 2.1, the function call sequence in the stack trace consists of n frames. Note that functions in some frames may not reside in the source code. For instance, a crashing fault of a third-party library in the source code cannot be localized in Java Development Kit (JDK); but some JDK code may appear in a frame, such as throwing an index-out-of-bounds exception when assigning an incorrect index to an array variable. Meanwhile, hidden or private functions can also bring in the difficulty of localizing the root cause. Given the list of all frames in a stack trace, a sublist of frames can be obtained by filtering out the functions, which are not in the given source code. Developers expect the faulty code resides in somewhere in this sublist of frames. For the sake of simplification, we refer to the first frame and the last frame in this sublist as the *top frame* and the *bottom frame*, respectively. For instance, Frame 1 in Fig. 1 is the top frame while Frame n is the bottom frame. If the sublist of frames contains only one frame, the top frame is identical to the bottom one.

Table 1 shows the detailed list of these 89 features. The 89 features are divided into 5 groups: 11 features related to the stack trace (ST01 to ST11), 23 features extracted from the top frame (CT01 to CT23), 23 features extracted from the bottom frame (CB01 to CB23), 16 features normalized from the top frame (AT01 to AT16), and 16 from the bottom frame (AB01 to AB16).

Features related to the stack trace. We extract features related to the stack trace since we expect these features can reflect the difficulty of handling crashes. An empirical study has explored the usefulness of stack traces during debugging (Schröter et al., 2010); stack traces can be utilized to assist several software tasks, such as crash reproduction (Chen and Kim, 2015), bug-report-oriented fault localization (Wong et al., 2014), and null pointer exception finding (Jiang et al., 2012). The group of Features ST01 to ST11 record the items that characterize the given stack trace, such as the type of the exception (ST01), the number of frames (ST02),[7] and the number of classes in stack trace after removing duplicate ones (ST03). Features ST10 and ST11 are also included in this group, which are extracted based on the source code of the project, i.e., the number of Java files and the number of classes (one Java file may contain two or more classes). Both these features approximately describe the scale of source code of the whole project.

Features from the top frame. The top frame in the stack trace is the location where the unexpected exception is thrown. The empirical study conducted by Schröter et al. (2010) showed the importance of the top frame in stack trace: 40% of faults are fixed in the top frame and close to 88% of bugs are fixed within the top-10 frames. The group of Features CT01 to CT23 is mined from the *top function* and the *top class*, which are short for the function and the class that exist in the top frame, respectively. We mined these features from the source code rather than the stack trace. Features in the top function or the top class characterize the program state when the program crashes. Among these 23 features, Features CT1 to CT6 are designed to characterize the top class, such as the number of local variables, whether the top class is inherited from other classes (a binary feature), and the Lines of Code (LoC) of comments. In addition, we use the next 17 features, i.e., CT07 to CT23, to capture the knowledge from the top function, such as LoC, the number of function calls, and the number of assignments.

Features from the bottom frame. The bottom frame can provide the message of the initial function call. We refer the function and the class in the bottom frame as *bottom function* and the *bottom class*, respectively. In this group, Features CB01 to CB23 are similar to Features CT01 to CT23; these features are based on the bottom frame instead. Given the source code, all the function calls in the frames are directly or indirectly called by the function in the bottom frame. Thus, we capture these 23 features to further characterize the crashing fault.

Features normalized by LoC of the CT features. In 16 features (CT08 to CT23) related to the top function, we normalize the original features by LoC and get Features AT01-AT16. Each of these features calculates the value per line in the top function. For example, AT01 records the number of parameters per line in the top function while AT16 records the number of binary operators.

Features normalized by LoC of the CB features. Features AB01 to AB16 are similar to Features AT1 to AT16, except that these features AB01 to AB16 are based on the bottom frame. For example, AB01 records the number of parameters per line in the bottom function.

3.4. Learning algorithms

In CraTer, predicting whether a crashing fault resides in the stack trace is transformed into a binary classification problem

[7] These frames belong to a subset of the original stack trace without the interference by third-party functions or classes. In our work, we assume that all third-party APIs are fault free although a bad implementation or design of third-party API can cause an unanticipated crash (Kechagia et al., 2015)

Table 1
Detailed list of 89 features in five groups.

Feature		Description	
*Group ST – **features related to the stack trace***			
ST01		Type of the exception in the crash	
ST02		Number of frames of the stack trace	
ST03		Number of classes in the stack trace	
ST04		Number of functions in the stack trace	
ST05		Whether an overloaded function exists in the stack trace	
ST06		Length of the name in the top class	
ST07		Length of the name in the top function	
ST08		Length of the name in the bottom class	
ST09		Length of the name in the bottom function	
ST10		Number of Java files in the project	
ST11		Number of classes in the project	
*Groups CT and CB – **features extracted from the top frame and the bottom frame***			
CT01	CB01	Number of local variables in the top/bottom class	
CT02	CB02	Number of fields in the top/bottom class	
CT03	CB03	Number of functions (except constructor functions) in the top/bottom class	
CT04	CB04	Number of imported packages in the top/bottom class	
CT05	CB05	Whether the top/bottom class is inherited from others	
CT06	CB06	LoC of comments in the top/bottom class	
CT07	CB07	LoC of the top/bottom function	
CT08	CB08	Number of parameters in the top/bottom function	
CT09	CB09	Number of local variables in the top/bottom function	
CT10	CB10	Number of if-statements in the top/bottom function	
CT11	CB11	Number of loops in the top/bottom function	
CT12	CB12	Number of for statements in the top/bottom function	
CT13	CB13	Number of for-each statements in the top/bottom function	
CT14	CB14	Number of while statements in the top/bottom function	
CT15	CB15	Number of do-while statements in the top/bottom function	
CT16	CB16	Number of try blocks in the top/bottom function	
CT17	CB17	Number of catch blocks in the top/bottom function	
CT18	CB18	Number of finally blocks in the top/bottom function	
CT19	CB19	Number of assignment statements in the top/bottom function	
CT20	CB20	Number of function calls in the top/bottom function	
CT21	CB21	Number of return statements in the top/bottom function	
CT22	CB22	Number of unary operators in the top/bottom function	
CT23	CB23	Number of binary operators in the top/bottom function	
*Groups AT and AB – **features normalized by LoC from Groups CT and CB***			
AT01	AB01	CT08 / CT07	CB08 / CB07
AT02	AB02	CT09 / CT07	CB09 / CB07
...	...	...	...
AT16	AB16	CT23 / CT07	CB23 / CB07

based on the 89 extracted features. Generally, any binary classifier can be used, such as the Bayesian Network or the Support Vector Machine (SVM). CraTer uses a decision tree algorithm to predict whether a crash belongs to the InTrace class or OutTrace. *Decision tree* is a family of widely-used classification algorithms, which construct binary trees by evaluating the feature values (Han et al., 2011). In a generated decision tree, each node denotes evaluating a feature and each branch presents the outcome of the evaluation; each leaf is a predicted class. During the development of many novel decision tree algorithms, the criteria of dividing nodes of the decision trees is an important factor of the performance, such as the criteria of using the information gain in ID3 (Quinlan, 1993), the gain ratio in C4.5 (Quinlan, 1993), and the Gini index in CART Breiman et al. (1984). In this paper, we choose a widely-used and robust decision tree algorithm, C4.5, as our classifier.

The numbers of crashes in the InTrace and OutTrace classes are not balanced. As we will see in Section 4, all the projects in our experiment contain fewer crashes in the InTrace class than in the OutTrace class. The imbalanced issue of data distribution may lead to inaccurate classification (He and Garcia, 2009). A typical classifier, such as SVM, assumes that the class distribution in the dataset is balanced. Thus, directly conducting classification without handling the imbalanced issue may lead to unfavorable prediction accuracy. To provide a general and accurate result, CraTer combines the decision tree, C4.5, with the SMOTE strategy to address the imbalanced issue. The *SMOTE* strategy (Chawla et al., 2002) is a typical oversampling technique; it synthesizes the samples of the minority class to balance the class distribution. During the synthesis, SMOTE randomly constructs a new minority instance based on one original minority instance and its corresponding nearest neighbors.

4. Experimental setup

In this section, we introduce the data preparation and the implementation. The data preparation consists of three main steps, as shown in Section 4.1; the implementation details are in Section 4.2.

4.1. Data preparation

Our work aims to build a learning model to predict whether the crashing fault resides in the stack trace. Thus, a number of crashes with known crashing fault locations need to be collected to provide an adequate dataset. However, it takes much effort to collect and reproduce real-world crashes. In existing crash-related works, Chen and Kim (2015) use a dataset of 52 crashes from three projects; Wu et al. (2014) collect a dataset of 160 crashes from eight projects; Gu et al. (2016) select 45 reproducible crashes from a dataset, called Defects4J (Just et al., 2014). In our work, the learning model requires a dataset for its training phase. All the three above datasets cannot be directly used due to the small number of

Table 2
Basic information of the seven projects used in the experiment.

Project	Version	LoC	#Classes	#Test cases	#Mutants	#Killed mutants	#Mutants before selection
Codec	1.10	14,480	84	662	2901	2601	610
Ormlite-Core	5.1	20,024	175	1059	3563	2751	1,303
JSqlParser	0.9.7	32,868	203	489	8757	5636	647
Collections	4.1	61,283	435	16,063	6650	5300	1,350
IO	2.5	26,018	122	1157	3337	2728	686
Jsoup	1.11.1	15,460	137	557	2657	1892	601
Mango	1.5.4	30,208	475	372	5149	1570	733
Total	-	**200,341**	**1,631**	**20,359**	**33,014**	**22,478**	**6,961**
Average	-	**28,620**	**233**	**2,908**	**4,716**	**3,211**	**944**

crashes. Therefore, we used real-world projects with seeded faults to prepare the experimental dataset.

In the evaluation, we use seven widely-studied open-source Java projects. To select these projects, first, we randomly selected several widely-studied and open-source Java projects from prior work (Xuan et al., 2016; Qiu et al., 2016) as well as GitHub projects with a large number of stars. Second, we removed projects that are difficult to be configured in a local machine. The configuration issues mainly related to the building dependency and platforms. Third, we generated mutants for all projects (see Section 4.2) and filtered out the projects with a small number of crashes. The reason is that CraTer learns a predictive model from known data of crashes; the learning process relies on sufficient data. Finally, we have seven projects left: Apache Commons Codec,[8] Ormlite-Core,[9] JSqlParser,[10] Apache Commons Collections,[11] Apache Commons IO,[12] Jsoup,[13] and Mango.[14]

Apache Commons Codec implements many techniques of decoders and encoders, such as Base64, Hex, Phonetic and URLs. *Ormlite-Core* is the core part of Ormlite, which mainly provides a lightweight parser from Java objects to SQL databases. *JSqlParser* parses SQL statements and translates into hierarchical Java classes. *Apache Commons Collections* provides many improvements and functionalities for the Java collections in JDK. *Apache Commons IO* is a library to assist the implementation of Java IO. *Jsoup* is a HTML parser library to manipulate and extract data from real-world HTML files. *Mango* is a fast distributed framework for object relational mapping.

Table 2 shows the details of the seven subject projects. The statistics in this table are collected via SourceMonitor.[15] Column "Version" indicates the version of the project in use in our experiment; Columns "LoC", "# Classes", and "# Test Cases" describe the lines of code without blank lines and comments, the number of classes in the source code without test cases, and the number of test cases executed in each project, respectively. Columns "# Mutants", "# Killed", and "# Mutants before selection" record the number of mutants generated by program mutation, the number of mutants that fails in test execution, and the number of the kept crashes, respectively.

We collected crashes as our dataset based on the following three steps. Details of data preparation are explained as follows. First, we generated seeded (injected) faults for each subject project with program mutation; second, we filtered out the mutants without leading to crashes with four sub-strategies; third, among all the kept crashes in a subject project, we randomly selected 500 crashes to form our dataset. Fig. 3 describes the steps of preparing the dataset.

4.1.1. Seeding faults with program mutation

We utilized program mutation techniques (Zhang et al., 2016), (Gopinath et al., 2016) to seed faults to real-world projects to simulate real crashes. For each of the seven subject projects, we used the PIT tool (see in Section 4.2.1) to generate slightly changed source code (i.e., single-point mutation) with seven default mutation operators. Table 3 shows a detailed list of all mutation operators in use. The column "Mutation operator" represents the name of operators and the column "Description" describes the detailed operation in program mutation. After program mutation, 33,014 mutants are generated for the seven subject projects.

Table 4 shows the top-5 reasons for crashes that are caused by each mutation operator in all the projects. We find that the reasons for crashes change according to the mutation operators. `ArrayIndexOutOfBoundsException` and `NullPointerException` widely exist in crashes by all mutation operators; the negatives invert mutator only generate seven crashes among all projects.

4.1.2. Filtering out mutants without crashes

In this step, we filtered out four kinds of mutants. First, we executed all test cases on each mutant and then discarded the mutants, which can pass all test cases. Therefore, we collected 22,478 killed mutants. Second, several faults may not trigger crashes since assertions in given test cases may be violated before crashes. If an assertion is violated, no crash will be triggered except an assertion failure (i.e., `AssertionFailedError` in Java). Third, when two variables that implement a Java-defined comparable type, a comparison failure (i.e., `ComparisonFailure` in Java) may be thrown if the types are not comparable. Fourth, the crashes that only records test cases are filtered out, because no information about the source code is provided.[16] Hence, we filtered out the above four kinds of mutants and then 6961 crashes are kept in total.

4.1.3. Randomly selecting crashes

In each project, we randomly selected 500 crashes for 10 times; thus, we have 10 datasets for each project. For instance, we can select 10 datasets of 500 crashes from Apache Commons Codec, called $Codec_i$ where $1 \leq i \leq 10$. Then we have in total 70 datasets for all seven projects. The number of 500 crashes is chosen because each project under evaluation has over 600 mutants before selection; then we chose 500 to simplify the calculation.

To obtain a mixed dataset of all projects, we combined datasets with the same index together. Then we get 10 combined datasets,

[8] Apache Commons Codec, http://commons.apache.org/codec/.
[9] Ormlite-Core, http://github.com/j256/ormlite-core.
[10] JSqlParser, http://github.com/JSQLParser/JSqlParser.
[11] Apache Commons Collections, http://commons.apache.org/collections/.
[12] Apache Commons IO, http://commons.apache.org/io/.
[13] Jsoup, http://jsoup.org/.
[14] Mango, http://www.jfaster.org/.
[15] SourceMonitor, http://www.campwoodsw.com/.

[16] A stack trace that only records test cases is mainly caused by the implementation of test class inheritance. Once the parent test class is involved in the crash, no source code is directly recorded in the stack trace.

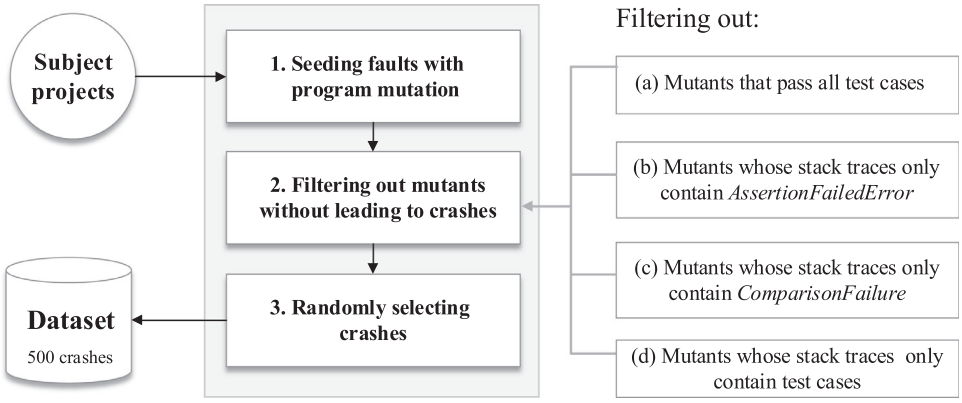

Fig. 3. Three steps of data preparation in the experimental setup.

Table 3
Seven mutation operators in use.

Mutation operator	Description
Conditional boundary mutator	Adding or removing the boundary in relational operators
Increment mutator	Replacing between ++ and −− or between += and −=
Negatives invert mutator	Inverting negation of integer and floating point numbers
Math mutator	Replacing one arithmetic operator to another arithmetic operator
Conditional negating mutator	Inverting negation of relational operators
Return value mutator	Mutating the return value of a function call
Void function call mutator	Removing a void function call

Table 4
Top-5 reasons for crashes that are caused by each mutation operator.

Mutation operator	Top-5 reasons (the count of each reason) †
Conditional boundary mutator	ArrayIndexOutOfBounds (395), IndexOutOfBounds (66), IllegalArgument (63), SQL (16), NullPointer (14)
Increment mutator	ArrayIndexOutOfBounds (62), StringIndexOutOfBounds (34), NegativeArraySize (7), IndexOutOfBounds (5), Decoder (4)
Negatives invert mutator	ArrayIndexOutOfBounds (5), Runtime (1), IndexOutOfBounds (1)
Math mutator	ArrayIndexOutOfBounds (314), JSQLParser (146), IndexOutOfBounds (68), StringIndexOutOfBounds (64), TokenMgrError (54)
Conditional negating mutator	NullPointer (721), IllegalArgument (342), ArrayIndexOutOfBounds (240), SQL (165), JSQLParser (130)
Return value mutator	NullPointer (675), Runtime (461), IllegalArgument (117), Initialization (67), SQL (56)
Void function call mutator	NullPointer (80), IllegalState (37), IO (31), SQL (15), TokenMgrError (11)

† We omit the word "Exception" for sake of space, e.g., NullPointer is short for NullPointerException.

each of which has 3500 crashes from seven projects. We refer a combined dataset to $Combined_i$,

$$Combined_i = Codec_i \cup Ormlite_i \cup JSqlParser_i \cup Collections_i \cup IO_i \cup Jsoup_i \cup Mango_i$$

where i denotes the ith randomly sampling and $1 \le i \le 10$. According to the class labeling in Section 3.1, we label these crashes into the InTrace class and the OutTrace class for training or evaluating the predictive model.

Furthermore, to ensure the consistency of distribution of the two classes (InTrace and OutTrace) before and after randomly selection, we employ the proportional random sampling to maintain the original distribution of InTrace and OutTrace when sampling 500 crashes from the whole project. That is, given one project, if the sampling size is 500, the number of crashes in InTrace or OutTrace does not change during each sampling. Fig. 4 presents the distribution of crashes in both classes in each project after randomly selection. We notice that the distribution of classes is imbalanced: the crashes in OutTrace are more than those in InTrace.

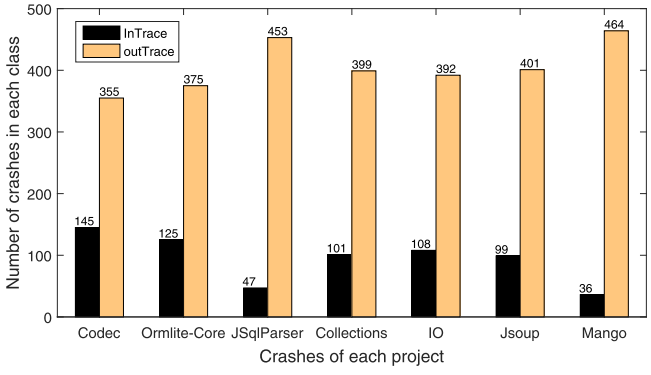

Fig. 4. Distribution of InTrace and OutTrace crashes of 500 crashes in each project.

4.2. Implementation

We implemented our approach in Java and Python: Java is used in program mutation and feature extraction while Python is used to filter out the invalid mutants.[17]

[17] The dataset and the prototype of CraTer are publicly available, http://cstar.whu.edu.cn/p/crater/.

4.2.1. Program mutation

We chose PIT[18] as our mutation tool in data preparation. PIT is one of the most robust and efficient tools in mutation testing (Delahaye and du Bousquet, 2013). Given a subject project, PIT can mutate one point in the original project using pre-defined mutation operators. All seven mutation operators in Table 3 are default operators in PIT. Note that existing work (Zhang et al., 2013; Moon et al., 2014) has also used program mutation to mimic real-world program faults.

4.2.2. Feature extraction

Feature extraction in CraTer is implemented through static program analysis using Spoon.[19] Spoon (Pawlak et al., 2016) is a Java library, which supports program analysis and transformation. Before extracting features, we properly configured and compiled each subject project since Spoon requires compilable source code as the input.

4.2.3. Machine learning

Machine learning algorithms in CraTer are implemented using Weka.[20] Weka, developed by Hall et al. (2009), is a collection of machine learning and data mining algorithms. Techniques of feature selection and imbalanced class processing methods are also integrated into Weka.

All experiments are run on a PC with an Intel Core i7 3.60GHz CPU and 8 GByte memory.

5. Experimental results

We first present four widely-used metrics to evaluate our predictive model in Section 5.1; then we propose four research questions in Section 5.2; finally, experimental results are given in Section 5.3.

5.1. Evaluation metrics

We use precision, recall, F-measure, and accuracy to evaluate CraTer. These four evaluation metrics are standard metrics to evaluate the prediction performance (He and Garcia, 2009; Han et al., 2011), and are widely-used in recent work of software maintenance (Wang et al., 2014; Xia et al., 2015; Li et al., 2016). Given a class X, i.e., InTrace or OutTrace, we define the evaluation metrics based on True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN) as follows,

$TP(X)$: # of crashes in X that are predicted as X;
$FP(X)$: # of crashes not in X that are predicted as X;
$TN(X)$: # of crashes not in X that are not predicted as X;
$FN(X)$: # of crashes in X that are not predicted as X.

The detailed metrics are given as follows. Among these metrics, Precision reflects the ratio of truly predicted positive samples in all samples predicted as positive, while recall represents the ratio of truly predicted positive samples in all true positive samples. F-measure is the trade-off metric between the precision and the recall, i.e., a high precision (or recall) might result in a low recall (or precision). Accuracy calculate the ratio of truly predicted samples in all samples.

$$Precision(X) = \frac{TP(X)}{TP(X) + FP(X)}$$

$$Recall(X) = \frac{TP(X)}{TP(X) + FN(X)}$$

$$F\text{-}measure(X) = \frac{2 \times Precision(X) \times Recall(X)}{Precision(X) + Recall(X)}$$

$$Accuracy(X) = \frac{TP(X) + TN(X)}{TP(X) + TN(X) + FP(X) + FN(X)}$$

We performed ten-fold cross validation to evaluate the prediction performance of CraTer. Ten-fold cross validation is a widely-used evaluation method in data mining. This method randomly splits the original dataset into 10 equal-size folds. In each time, one fold is selected as the dataset in the deployment phase and the other nine folds are as the dataset in the training phase. After 10-time evaluation, we got 10 results and use their average as the final result.

5.2. Research question

We empirically evaluated our proposed approach, namely CraTer, by answering four Research Questions (RQs). These RQs examine the effectiveness of the proposed approach, the imbalanced data processing strategies, the impactful features, and the efficiency as follows.

RQ1. **How effective is our approach in predicting whether a crashing fault resides in stack traces or not?**

We evaluated the effectiveness to show whether our approach can be used in practice. Four evaluation metrics are examined on crashes from seven subject projects.

RQ2. **Can imbalanced data processing strategies improve the prediction results?**

In our approach, we combined an imbalanced data processing strategy, i.e., SMOTE, with the decision tree algorithm (C4.5) to address the imbalance issue of crash data. Thus, we compared the effectiveness of the SMOTE strategy with that of other imbalanced data processing strategies.

RQ3. **Which features are more impactful on the prediction results?**

Our approach is conducted based on 89 extracted features. This experiment can find out dominant features, i.e., the features that have more impact on the prediction.

RQ4. **How efficient is our approach in the prediction?**

We calculated the time cost in millisecond and the manual effort in terms of lines of code.

5.3. Results

In this section, we present and analyze the results of four RQs in our experiment.

5.3.1. RQ1. How effective is our approach in predicting whether a crashing fault resides in stack traces or not?

We first used 10 combined datasets from seven subject projects as overall datasets, i.e., $Combined_1$ to $Combined_{10}$ defined in Section 4.1.3, to evaluate the effectiveness of CraTer. Then the average results of 10 datasets are calculated as the overall evaluation results. As mentioned in Section 3.4, CraTer combines a decision tree classifier (C4.5) with the SMOTE strategy. In the experiment, we used five other classifiers to conduct the comparison, including RandomForest, i.e., an ensemble classifier of multiple decision trees, BayesNet, i.e., a network classifier of multiple Bayesian nodes, SMO, i.e., a sequential minimal optimization classifier, KStar, i.e., a lazy learning classifier, and SVM (Support Vector Machine). All these classifiers are also combined with the same strategy to eliminate the risk of imbalanced data, i.e., SMOTE.

For parameters of classifiers, we followed the guide document of Weka. In C4.5, a decision tree is built with the confidence factor of 0.25 within 3 folds; in RandomForest, the maximum number of decision trees is set to 100; in SMO, the complexity is set to 1.0

[18] PIT, http://pitest.org/.
[19] Spoon, http://spoon.gforge.inria.fr/.
[20] Weka, http://www.cs.waikato.ac.nz/ml/weka/.

Table 5
Ten-fold cross validation on all the crashes.

Classifier	InTrace			OutTrace			Accuracy
	Precision	Recall	F-measure	Precision	Recall	F-measure	
C4.5	**0.818**	**0.792**	**0.805**	**0.952**	**0.959**	**0.955**	**0.927**
RandomForest	0.786	0.706	0.744	0.933	0.955	0.944	0.908
BayesNet	0.452	0.754	0.565	0.932	0.787	0.853	0.781
SMO	0.559	0.528	0.543	0.891	0.903	0.897	0.832
KStar	0.663	0.661	0.662	0.921	0.922	0.921	0.873
SVM	0.660	0.501	0.569	0.890	0.940	0.914	0.857

and the calibrator is the logistic regression; in SVM, the cache size is set to 40 and the loss is set to 0.1; in SMOTE, the number of nearest neighbors is set to 5. For each combined dataset $Combined_i$ ($1 \le i \le 10$), we conducted ten-fold cross validation and recorded the results. Then we calculated the average result of 10 datasets, i.e., $Combined_1$ to $Combined_{10}$.

Table 5 presents the average results on 10 combined datasets. As shown in the table, all classifiers except BayesNet can achieve the accuracy over 0.80. Specially, C4.5 achieves the best accuracy among these six classifiers under evaluation: it can reach the highest accuracy of 0.927 and also achieve the highest Precision, Recall, and F-measure values for both classes. In addition, RandomForest preforms slightly worse than C4.5.

For each individual project, we also performed ten-fold cross validation on one crash dataset (e.g., $Codec_i$ in the project Codec) and calculate the average results for its 10 datasets (e.g., $Codec_1$ to $Codec_{10}$). Table 6 shows the average experimental results of each project. COD, ORM, JSQ, COL, IO, JSO, and MAN denote the projects of Apache Commons Codec, Ormlite-Core, JSqlParser, Apache Commons Collections, Apache Commons IO, Jsoup, and Mango, respectively.

As shown in Table 6, no classifier can completely beat all the others for all projects. C4.5 performs well among the six classifiers under evaluation in all the projects. We can observe several facts as follows. First, in Codec, Ormlite-Core, and Collections, C4.5 can get the highest values in all seven metrics (i.e., the precision, recall, F-measure for both classes, and the accuracy). Second, C4.5 can reach the highest accuracy in the first six projects (Codec, Ormlite-Core, JSqlParser, Collections, IO, and Jsoup); one exception is the accuracy of Mango: C4.5 reaches 0.954, which is extremely close to the highest value, i.e., 0.960 by RandomForest and SMO. Third, in seven metrics of each project, C4.5 can at least get three highest values, except Mango. Fourth, C4.5 can reach the most balanced results for both InTrace and OutTrace classes; meanwhile, none of its metric values is under 0.6.

Results in Tables 5 and 6 also surprise us and break the inertial thinking in machine learning that a simple classifier, such as C4.5 may not outperform complex ones, such as RandomForest. In our experiment, we have carefully tuned the setup parameters of RandomForest and other classifiers. For instance, in RandomForest, we gradually tuned major parameters according to the parameter ranges, e.g., increasing 50 each time for the maximum number of decision trees. As shown in the above results, well-tuned classifiers, such as RandomForest, cannot achieve better performance than C4.5.

Answer to RQ 1. Our approach is effective in predicting whether a crashing fault resides in the stack trace or not. Among six classifiers under evaluation, C4.5 performs the best.

5.3.2. RQ2. Can imbalanced data processing strategies improve the prediction results?

As mentioned in Section 4.1, the distribution of crashes in InTrace and OutTrace classes is imbalanced: crashes in the InTrace class are fewer than those in OutTrace. We empirically evaluated different imbalanced data processing strategies for overcoming the imbalanced classification issue.

To compare with the combination of C4.5 and the SMOTE strategy, we replaced the SMOTE strategy with no strategy (called *NoStrategy* for short), cost-sensitive learning, and resampling. NoStrategy means we do not apply any strategy for imbalanced data processing and directly use a classifier to train a model; cost-sensitive learning (Elkan, 2001) handles the imbalanced issue by assigning different costs to misclassified data, which are characterized with the cost matrix; resampling (He and Garcia, 2009) is a simple and direct sampling method, which randomly selects samples from the original dataset to construct a new balanced dataset.

Fig. 5 demonstrates the impact of different strategies of processing imbalanced data on the dataset. For the OutTrace class, the four strategies, including NoStrategy, achieve similar results, close to 1.0. For the InTrace class, SMOTE reaches better recall and F-measure values than NoStrategy while NoStrategy can get higher precision values. For the accuracy, all four strategies also get similar results close to 1.0. Additionally, the two strategies of cost-sensitive learning and resampling perform slightly worse than SMOTE and NoStrategy.

In most projects, the SMOTE strategy and NoStrategy preforms better than the cost-sensitive learning and resampling. To further study the influence of the strategies on results, we used the Wilcoxon signed-rank test to compare the results between NoStrategy and the SMOTE strategy in the seven projects. The Wilcoxon signed-rank test (Wohlin et al., 2012) is a non-parametric statistical hypothesis test, which is used to assess whether there exists a significant difference between two independent samples.

As mentioned above, for each dataset of one project, we conducted ten-fold cross validation and evaluated the effectiveness for ten times. Then, in each time of evaluation, we recorded the result of evaluation metrics as a 7-dimension vector μ,

$$\begin{aligned}\mu &= <\mu_1, \mu_2, \mu_3, \mu_4, \mu_5, \mu_6, \mu_7> \\ &= <Precision(\text{InTrace}), \quad Recall(\text{InTrace}), \quad F\text{-}measure(\text{InTrace}), \\ &\quad Precision(\text{OutTrace}), Recall(\text{OutTrace}), F\text{-}measure(\text{OutTrace}), \\ &\quad Accuracy>\end{aligned}$$

where an element μ_i of the vector μ denotes the value of ith evaluation metric and $1 \le i \le 7$. For instance, $i = 1$ denotes the value of the precision of the class InTrace.

Given the ith metric, we define one value in the evaluation as $\alpha_{i,j,k,l}$, i.e., the value of the ith metric on the kth dataset in jth project with the lth time of evaluation during ten-fold cross validation. Then for the ith metric, we have in total $7 \times 10 \times 10 = 700$ values since there are 7 projects, each project has 10 crash datasets, and each dataset is evaluated for 10 times due to ten-fold cross validation. Therefore, for each metric μ_i, 700 result pairs can be formed to evaluate the difference between the results with or without the SMOTE strategy.

Based on the 700 result pairs of each μ_i, we conducted the Wilcoxon signed-rank test and got the p-value for each μ_i: 1.0592e-27, 4.6484e-17, 8.3603e-11, 1.4685e-10, 1.6200e-11,

Table 6
Ten-fold cross validation on crashes from each project.

Project	Classifier	InTrace			OutTrace			Accuracy
		Precision	Recall	F-measure	Precision	Recall	F-measure	
COD	C4.5	**0.761**	**0.812**	**0.785**	**0.921**	**0.895**	**0.908**	**0.871**
	RandomForest	0.720	0.752	0.736	0.897	0.881	0.889	0.843
	BayesNet	0.564	0.707	0.627	0.866	0.777	0.819	0.757
	SMO	0.566	0.776	0.654	0.892	0.756	0.818	0.762
	KStar	0.664	0.701	0.681	0.875	0.855	0.865	0.810
	SVM	0.553	0.668	0.604	0.852	0.779	0.813	0.747
ORM	C4.5	**0.878**	**0.881**	**0.879**	**0.960**	**0.959**	**0.960**	**0.939**
	RandomForest	0.774	0.762	0.768	0.921	0.926	0.923	0.885
	BayesNet	0.613	0.814	0.699	0.930	0.827	0.876	0.824
	SMO	0.623	0.683	0.651	0.891	0.861	0.875	0.816
	KStar	0.706	0.687	0.696	0.897	0.904	0.900	0.850
	SVM	0.656	0.565	0.604	0.861	0.898	0.879	0.815
JSQ	C4.5	0.831	0.685	**0.750**	0.968	0.985	**0.976**	**0.957**
	RandomForest	0.829	0.660	0.734	0.965	0.986	0.976	0.955
	BayesNet	0.295	**0.800**	0.430	**0.975**	0.800	0.879	0.800
	SMO	0.813	0.681	0.741	0.967	0.984	0.975	0.955
	KStar	0.711	0.679	0.693	0.967	0.971	0.969	0.944
	SVM	**0.848**	0.436	0.575	0.944	**0.992**	0.967	0.940
COL	C4.5	**0.804**	**0.741**	**0.771**	**0.936**	**0.954**	**0.945**	**0.911**
	RandomForest	0.762	0.611	0.677	0.906	0.951	0.928	0.882
	BayesNet	0.543	0.553	0.547	0.886	0.880	0.883	0.814
	SMO	0.581	0.608	0.593	0.900	0.889	0.894	0.832
	KStar	0.574	0.573	0.573	0.892	0.892	0.892	0.827
	SVM	0.626	0.349	0.447	0.852	0.947	0.897	0.826
IO	C4.5	0.813	0.768	**0.789**	**0.937**	0.951	**0.944**	**0.911**
	RandomForest	**0.820**	0.757	0.787	0.935	0.954	**0.944**	**0.911**
	BayesNet	0.665	0.711	0.687	0.919	0.901	0.910	0.860
	SMO	0.734	**0.769**	0.750	0.935	0.923	0.929	0.890
	KStar	0.759	0.744	0.751	0.930	0.934	0.932	0.893
	SVM	0.815	0.670	0.735	0.913	**0.958**	0.935	0.896
JSO	C4.5	0.643	**0.660**	**0.650**	**0.916**	0.909	**0.912**	**0.860**
	RandomForest	**0.657**	0.587	0.619	0.901	**0.924**	**0.912**	0.857
	BayesNet	0.424	0.621	0.503	0.894	0.791	0.839	0.757
	SMO	0.607	0.565	0.584	0.894	0.910	0.902	0.841
	KStar	0.563	0.577	0.569	0.895	0.889	0.892	0.827
	SVM	0.545	0.371	0.440	0.856	0.924	0.889	0.814
MAN	C4.5	0.717	0.608	0.658	0.970	0.981	0.976	0.954
	RandomForest	0.808	0.586	**0.678**	0.969	0.989	**0.979**	**0.960**
	BayesNet	0.185	0.750	0.295	**0.974**	0.736	0.837	0.737
	SMO	0.761	**0.658**	0.703	**0.974**	0.983	0.978	**0.960**
	KStar	0.720	0.628	0.669	0.971	0.980	0.976	0.955
	SVM	**0.871**	0.422	0.567	0.957	**0.995**	0.976	0.954

8.3603e-08, and 1.2479e-04. That is, consider 0.005 as a threshold of the significant difference, using SMOTE or not leads to significant differences for all seven metrics.

Answer to RQ 2. In our experiment, the SMOTE strategy achieves the most stable results. The imbalanced data processing strategies can obtain accurate prediction results, comparing with no strategy.

5.3.3. RQ3. Which features are more impactful on the prediction results?

In CraTer, we propose five groups of features from the stack trace and the source code. We explore which features are more impactful on the prediction results. We used Pearson correlation coefficient (Egghe and Leydesdorff, 2009; Zhang et al., 2018) to find out the correlation between a feature and the predicted class. *Pearson correlation coefficient* is one of commonly-used relativity coefficients, which measures the relationship between two random variables.

Let y and x denote the predicted class and one feature. Given a dataset of m crashes, the value of Pearson correlation coefficient between the class y and the feature x is defined as follows,

$$Pearson(x, y) = \frac{\sum_{i=1}^{m}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{m}(x_i - \overline{x})^2}\sqrt{\sum_{i=1}^{m}(y_i - \overline{y})^2}}$$

where x_i and y_i are the values on the ith crash of the feature x and the class, $\overline{x}$ and $\overline{y}$ are the average values of x and y of m crashes, respectively. Pearson correlation coefficient ranges from -1 to 1. The absolute value of the coefficient indicates the strength of the correlation. The coefficient is 0 if the feature has no correlation with the predicted label while 1 or -1 indicate that the feature has positively or negatively strongest correlation with the predicted label. In our work, values of binary features, such as InTrace or OutTrace of the class label, are treated as 1 and -1 to adjust to the calculation of correlation.

Table 7 lists the top-10 dominant features according to the absolute values of Pearson correlation coefficients of features. The top-10 dominant features of each project are determined based on the average ranking list of 10 crash datasets, where the ranking list in each dataset is calculated via the absolute value of the Pearson correlation coefficient.

As shown in Table 7, two features, AT14 and AT16, have frequent occurrences in the top-10 list and each feature appears for six times. Another feature AT06 also appears for five times. Recall the definition of features in Table 1, these dominant features are the number of for-each statements per line in the top function (AT06), the number of return statements per line in the top function (AT14), and the number of binary operators per line in the top function (AT16). All these three features (AT06, AT14, and

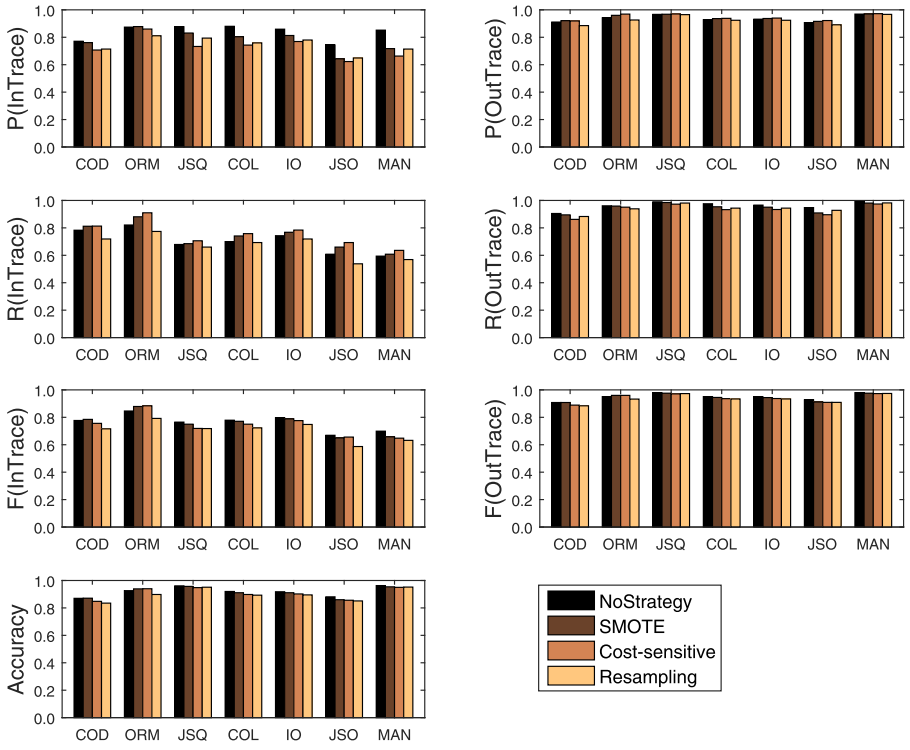

Fig. 5. Prediction using four strategies of imbalanced data processing. For the sake of space, P, R, and F denote the precision, the recall, and the F-measure, respectively.

Table 7
Lists of top-10 dominant features for each project.

Rank	COD	ORM	JSQ	COL	IO	JSO	MAN
1	CT21	**AT16**	**AT16**	ST01	CT13	**AT06**	**AT16**
2	**AT06**	**AT14**	**AT14**	**AT06**	**AT06**	AT15	**AT14**
3	CT13	CT21	AT03	AB06	AB06	ST07	AT01
4	CB21	AT12	CT04	AB04	CB13	**AT16**	AB13
5	CT22	**AT06**	AB10	AT03	**AT16**	**AT14**	CT04
6	CT14	CT23	CT10	CT13	**AT14**	CB08	CT05
7	CT23	CT19	AB09	**AT16**	AT04	CB22	CB06
8	CT10	CT08	CT09	CB13	AB04	CB19	AT03
9	CB06	CT07	CT23	**AT14**	CB22	CB10	CB22
10	CB02	CT22	CT11	AB02	CB04	CB23	CB13

AT16) strongly relate to the program control flow. Note that AT16 relates to both the control flow (e.g., a *logic-and* operator &&) or the data flow (e.g., an *algorithmic-add* operator +). We can intuitively conclude that features related to the control flow can highly influence the result of our predictive model.

We counted the number of occurrences of different groups of features based on Table 7: Fig. 6(a) shows the absolute number of each group while Fig. 6(b) shows the ratio between the number of features in the top-10 list (including duplicate features in different projects) and the number of features defined in the group. For instance, 1.500 of AT in Fig. 6(b) indicates the ratio between the number of recorded AT features in the list and the total number of features, i.e., $24/16 = 1.500$.

As shown in Fig. 6(a), features from AT group have the most occurrence (i.e., 24) for all features in the top-10 list. The other four groups have the occurrences of 2 at least. As shown in Fig. 6(b), features from AT group have the highest ratio (i.e., 1.500) of all features in the top-10 list; ratio of features from CT group is 0.913, i.e., the second rank. The above occurrences and percentages reflect the importance of different groups of features: features in CT and AT groups better impact the prediction than those in the other groups.

We have also checked whether the top-10 dominant features in Table 7 can represent the whole set of 89 features. Table 8 shows the comparison between the whole set of 89 features and a subset of the top-10 dominant features. In this comparison, we present two groups of results for each project: one group uses the whole set of 89 features based on the combination of C4.5 with SMOTE while the other uses the top-10 dominant features in Table 7 instead.

As shown in Table 8, the prediction with the whole set of 89 features achieves higher results in six out of seven projects than that with the subset of the top-10 dominant features. One exception project is Collections, the prediction with the subset of the top-10 features performs slightly better. We followed Section 5.3.2 to conduct the Wilcoxon signed-rank test. The results are significantly different in two sets of features if we consider 0.005 as the threshold of significance. However, we can also observe that several results are similar, especially the accuracy; that is, in some case, there exists no practical difference whether we use the feature selection technique or not. We can observe that in most projects, using the subset of top-10 dominant features may lose several features, which can predict the residence of the faulty code.

To further study the impact of 89 features, we examined whether the feature selection technique works for our approach based on our observation of the dominant features. Feature selection, also known as feature subset selection, aims to improve the prediction results via removing redundant and irrelevant features (Han et al., 2011). A feature selection technique can output a subset of features of the original feature set. We used Chi-Square (χ^2) as our major feature selection method because it performs well in the empirically evaluation (Guyon and Elisseeff, 2003; Xuan et al., 2017a). In addition, we also used Information Gain (Wang and Lochovsky, 2004) and ReliefF (Kononenko, 1994) as another two feature selection methods.

Fig. 7 shows the empirical results of applying Chi-Square, Information Gain, and ReliefF. Similar to the hypothesis test in RQ2, we

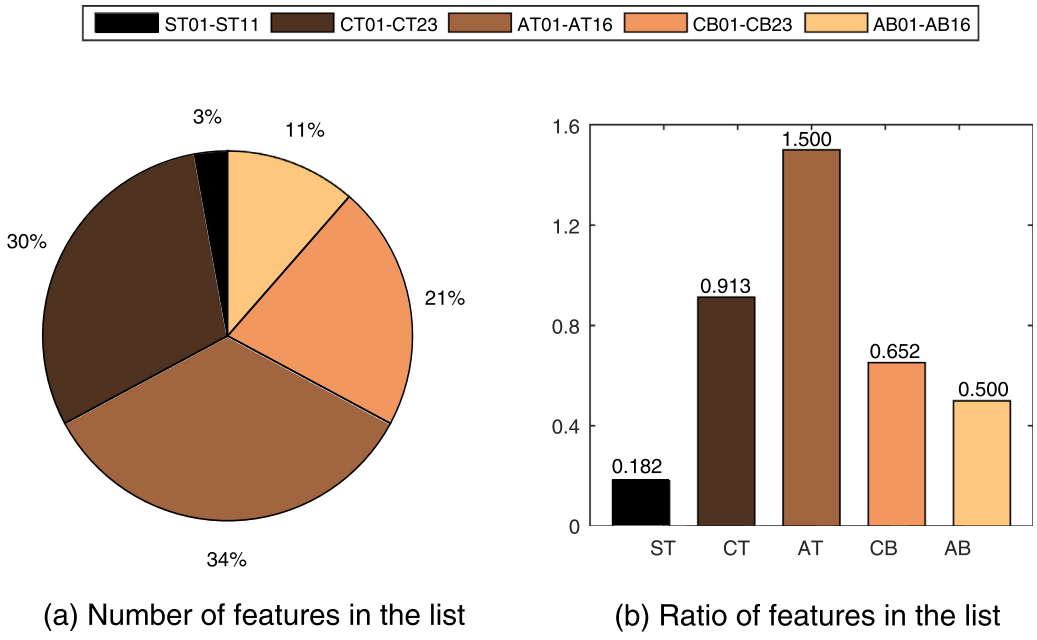

Fig. 6. Distribution of different groups of features in the top-10 list .

Table 8
Comparison between the whole set of 89 features and a subset of the top-10 dominant features on crashes from each project.

Project	Features	InTrace			OutTrace			Accuracy
		Precision	Recall	F-measure	Precision	Recall	F-measure	
COD	All 89	0.761	0.812	0.785	0.921	0.895	0.908	0.871
	Top-10	0.571	0.733	0.642	0.877	0.774	0.822	0.762
ORM	All 89	0.878	0.881	0.879	0.960	0.959	0.960	0.939
	Top-10	0.624	0.662	0.642	0.885	0.867	0.876	0.816
JSQ	All 89	0.831	0.685	0.750	0.968	0.985	0.976	0.957
	Top-10	0.819	0.679	0.741	0.967	0.984	0.975	0.955
COL	All 89	0.804	0.741	0.771	0.936	0.954	0.945	0.911
	Top-10	0.811	0.752	0.780	0.939	0.955	0.947	0.914
IO	All 89	0.813	0.768	0.789	0.937	0.951	0.944	0.911
	Top-10	0.684	0.781	0.730	0.937	0.901	0.919	0.875
JSO	All 89	0.643	0.660	0.650	0.916	0.909	0.912	0.860
	Top-10	0.538	0.604	0.569	0.899	0.871	0.885	0.818
MAN	All 89	0.717	0.608	0.658	0.970	0.981	0.976	0.954
	Top-10	0.488	0.444	0.460	0.957	0.962	0.960	0.925
p-value		2.1095e-43	8.9694e-25	4.9942e-60	2.9540e-34	2.9722e-53	4.9942e-60	1.6165e-60

also conducted the Wilcoxon signed-rank test to explore the influence between using a feature selection method (i.e., Chi-Square) or not. The *p*-value of seven evaluation metrics are 0.1699, 0.6977, 0.1061, 0.6998, 0.1361, 0.1061, and 0.1689, respectively. This result indicates that using feature selection cannot lead to significant differences, comparing with the prediction with no feature selection.

Answer to RQ 3. According to Pearson correlation coefficient, AT06, AT14, and AT16 are most impactful on the prediction results among seven projects. Furthermore, the results obtained using three typical feature selection methods are not better than those without feature selection.

5.3.4. RQ4. How efficient is our approach in the prediction?

We show the time cost and the manual effort to investigate the efficiency of our approach.

Given a newly-submitted crash, CarTer first extracts 89 features from its stack trace and source code, and then predicts either InTrace or OutTrace by the built predictive model. Let t_e denote the average time cost of feature extraction for one crash. Hence, the time cost of the prediction on a newly-submitted crash consists of the time of its feature extraction t_e^d and the time of predicting its label t_p^d. In the experiment, the process of ten-fold cross validation consists of 10 rounds of training phases and deployment phases (see Section 3.2). In one training phase, let T_e^t and T_m^t be the total time cost of feature extraction and model building, respectively; in one deployment phase, let T_e^d and T_p^d be the total time cost of feature extraction and prediction, respectively. Then the time cost of one round in ten-fold cross validation is as follows,

$$T = (T_e^t + T_m^t) + (T_e^d + T_p^d) \quad ,$$

and the time cost of the prediction on one newly-submitted crash is as follows,

$$t = t_e^d + t_p^d \quad ,$$

where T is the average time cost of one round and t^d is the average time cost of the prediction on a newly-submitted crash.

Table 9 present the average time cost of one round in ten-fold cross validation and the average time cost of prediction on one newly-submitted crash. As mentioned in Section 5.1, in each round of ten-fold cross validation, we used 450 crashes in the training phase and 50 crashes in the deployment phase; all values in Table 9 are the average.

As shown in Table 9, the time cost t_e^d of feature extraction for one crash varies in the range of 315 to 2509 milliseconds; the

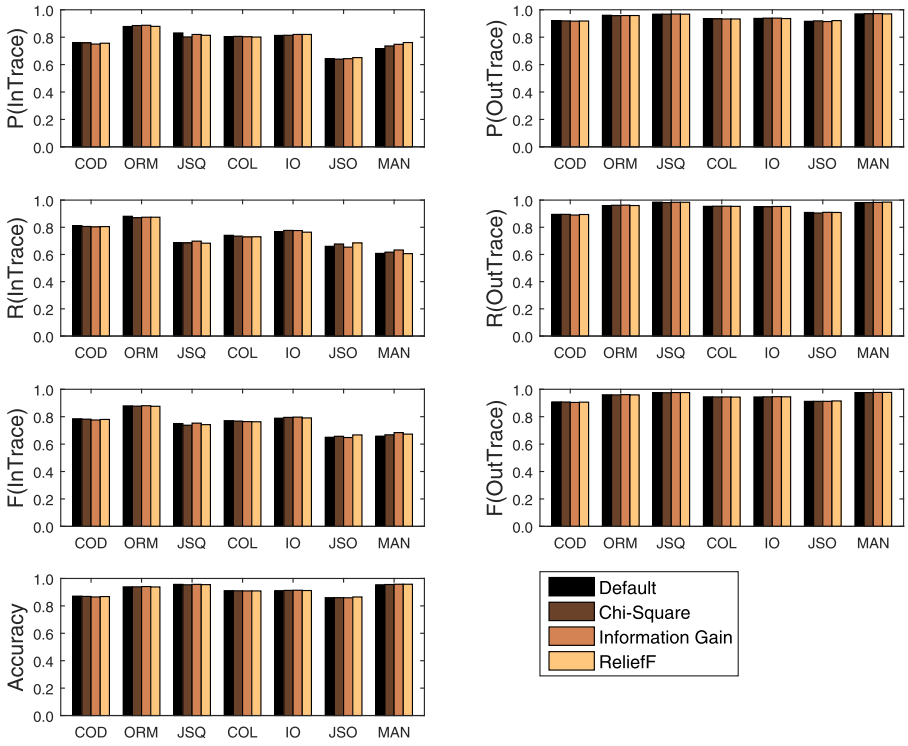

Fig. 7. Results with three feature selection methods. For the sake of space, P, R, and F denote the precision, the recall, and the F-measure, respectively.

Table 9
Time cost in each round of CraTer (in millisecond).

Project	Training phase per round †		Deployment phase per round †		Deployment phase per crash ‡	
	T_e^t	T_m^t	T_e^d	T_p^d	t_e^d	t_p^d
Codec	141,795	79.5	15,755	0.03	315	0.0006
Ormlite-Core	222,525	43.3	24,725	0.02	495	0.0004
JSqlParser	1,128,870	14.2	125,430	0.02	2509	0.0004
Collections	460,260	37.6	51,140	0.02	1023	0.0004
IO	186,030	38.2	20,670	0.04	413	0.0008
Jsoup	175,815	36.2	19,535	0.03	391	0.0006
Mango	250,830	12.5	27,870	0.02	557	0.0004
Average	366,589	37.4	40,732	0.03	815	0.0005

† The time cost of one training phase and one deployment phase of one round in ten-fold cross validation. ‡ The time cost of one crash in the deployment phase.

time cost t_p^d of prediction for one crash is less than 1 millisecond. The total time cost of a newly-submitted crash is 815 milliseconds. Therefore, we consider this less than 1 second time per crash is acceptable.

We estimated the manual effort in terms of lines of code to better understand the benefit of our proposed approach. We define the following four kinds of effort: E_A, E_B, E_C, and E_D, each of which calculates the Lines of Code (LoC) in different situations, respectively,

- E_A - LoC when reviewing all **functions** that are recorded in the stack trace.
- E_B - LoC when reviewing all **functions** that are recorded in the stack trace from Frame 0 until the faulty code is found.
- E_C - LoC when reviewing all **lines** that are recorded in the stack trace.
- E_D - LoC when reviewing all **lines** that are recorded in the stack trace from Frame 0 until the faulty code is found.

Fig. 8 presents an example of a stack trace from the project Jsoup. The crashing fault is at Line 19 of the constructor `ConstrainableInputStream()`. There are three functions that are recorded in the stack trace: `isTrue()`, `ConstrainableInputStream()`, and `parseInputStream()` with 4, 6, and 65 lines of code, respectively. In practice, a developer usually reviews all functions that are recorded in the stack trace from Frame 0 until the faulty code is found; that is, the manual effort equals to E_B.

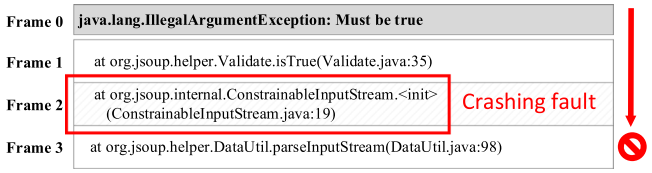

Fig. 8. Example of a stack trace by a crashing fault in the project Jsoup. The faulty code is at Frame 2.

For E_A, we assume that a developer reviews all functions that are recorded in the stack trace, i.e., $E_A = 4 + 6 + 65 = 75$. For E_B, we assume that a developer reviews all functions that are recorded in the stack trace from Frame 0 until the faulty code is found. In this case, $E_B = 4 + 3 = 7$ because Line 19 is the third line in the function `ConstrainableInputStream()`. For E_C, we assume that a developer reviews all lines that are recorded in the stack trace, i.e., $E_C = 3$. For E_D, we assume that a developer reviews all lines that are recorded in the stack trace from Frame 0 until the faulty code is found. $E_D = 2$, i.e., Line 35 in the file `Validate.java` and Line 19 in `ConstrainableInputStream.java`.

If a newly-submitted crash is predicted as InTrace by CraTer, a developer only needs to focus on the specific lines in the stack trace, i.e. E_D. Comparing with the manual effort E_B in practice, the saved effort is defined as follows,

$$E_{saved} = \frac{E_B - E_D}{E_B} \times 100\%$$

Table 10
Manual effort for each project (in LoC).

Project	E_A	E_B	E_C	E_D	P_{saved}
Codec	8,665.4	2,594.5	258.8	114.5	95.6%
Ormlite-Core	12,539.6	735.3	475.3	102.6	86.0%
JSqlParser	4,374.5	36.7	199.0	31.9	13.1%
Collections	2,697.6	574.5	115.9	72.9	87.2%
IO	1,913.9	538.5	128.8	80.8	85.0%
Jsoup	4,574.5	409.5	309.0	76.3	81.4%
Mango	1,060.1	39.8	117.7	21.4	46.2%
Average	5,117.9	704.1	229.2	71.5	70.6%

If a newly-submitted crash is predicted as OutTrace by CraTer, we do not help a developer reduce any effort in manual crash localization. Then the developer has to review all functions in the stack trace until the faulty line is found; this makes the manual effort by the developer equal to E_B.

Table 10 shows all the manual efforts for crashes that are correctly predicted as InTrace by CraTer. For crashes that are predicted as OutTrace, we do not change the process of manual crash localization; then the saved manual effort is zero. As shown in Table 10, for Projects Codec, Collections, IO, Jsoup, and OrmLite, the saved efforts in percentage are over 80%. For Project JSqlParser, the saved effort in percentage is 13.1%; the major reason is that the faulty code is near the top of the stack trace in many crashes of JSqlParser. In average, for crashes that are predicted as InTrace, CraTer can save 70.6% of manual efforts, comparing with manual crash localization.

Answer to RQ 4. CraTer is efficient. It can quickly predict for a newly-submitted crash in average 815 milliseconds; meanwhile, CraTer can save 70% of manual efforts in average.

6. Threats to validity

In this section, we present three major threats to the validity of our work: the construct validity, the internal validity, and the external validity.

6.1. Construct validity

In the experiment, seven real-world projects are selected as subject projects, which are further seeded faults with program mutation. A threat is that all seven projects are written in Java and we have not considered other programming languages, such as C/C++ or Python in our experiment. Such selection may hurt the generality of our experiment. Another construct validity is that our selected subject projects are all built with Maven.[21] The configuration with Maven may ease the complex process of feature extraction; meanwhile, only handling Maven based projects may also involve the bias of experiment construction. Projects in our experiment range from 14K to 61K lines of code. The generality of smaller or larger projects can be viewed as a threat to the validity. Further experiments can be conducted to address this issue.

6.2. Internal validity

In this work, we extracted 89 features from the stack trace and the source code. The features are selected based on our programming experience. It is possible that some other choices of features may better characterize the addressed prediction problem. The algorithms, such C4.5 and the SMOTE strategy, are empirically evaluated and shown to be effective. Further investigation of better algorithms could help to improve the prediction results. Considering the randomness in data preparation such as random selection of 500 crashes and evaluation such as ten-fold cross validation, there exists another threat to the internal validity for the replication of experiments.

6.3. External validity

Our experiments are conducted on the crashes, which are generated via seeded faults based on program mutation. Existing empirical studies by Namin and Kakarla (2011) and Ali et al. (2009) have shown that there is no significant impact on the results of fault localization when using the mutants to mimic the real-world bugs. However, these seeded faults may still result in the risk of unrepresentative crashes. The main reason is that real-world bugs are usually caused by multiple and complex logical faults while crashes in our experiment are all caused by one single seeded fault. The steps of seeding faults to real-world projects can be viewed as a trade-off between the requirements of solving real-world problems and the lack of available real-world crashes.

In machine learning, classifiers may yield different results depending on the parameter settings; it is infeasible to check all possibilities of parameters. In our work, we tuned the classifier parameters based on the guide document of Weka. There exists a threat that a particular parameter setting may lead to a different result in the comparison of classifiers. Meanwhile, crashes in our work are inadequate to build an extremely precise classifier. Collecting more real-world crashes may lead to better explanation to our current result.

7. Related work

We describe two categories of related work in this section, i.e., crash localization and crash reproduction.

7.1. Crash localization

Crash localization aims to map a stack trace onto its root cause; in practice, it aims to identify a faulty function that causes the crash. As mentioned in Section 2.2, Wu et al. (2014) and Gong et al. (2014) have proposed two automatic approaches to recover the links between the crashes and their root-cause functions based on a recommendation list. To support the localization of crashes, many software companies deploy crash reporting systems to gather user-submitted crashes and then extract a crash report for similar crashes. Kim et al. (2011) propose to build a crash graph to cluster similar crashes and to reduce the cost of dealing with duplicate crashes. Dang et al. (2012) design a re-bucketing technique to enhance the existing crash clustering system in Microsoft to support the duplicate detection by refining clusters. Kechagia et al. (2015) employ software telemetry data from Android applications to study the association between crashes and API deficiencies.

Two related techniques in debugging are spectrum based fault localization and information-retrieval based bug location. Spectrum based *fault localization* aims to find out the faulty code based on the execution of given test cases (Jones and Harrold, 2005; Rui et al., 2007; Lucia et al., 2014; Le et al., 2016). The spectrum is a matrix of collected numbers of passing or failing test cases for each program entity; all candidate program entities are ranked based on the pre-designed likelihood metric. Recent work by Le and Lo (2013) analyzes the empirical results of whether a fault localization technique can correctly identify the root cause via a predictive model with 50 extracted features; these features are expected to potentially relate to the effectiveness of fault localization. Their model shows that it is feasible to predict the effectiveness of fault localization. Information-retrieval based *bug location* aims to map a

[21] Maven, http://maven.apache.org/.

bug report onto its related source code file (Zhou et al., 2012; Xia et al., 2014; Le et al., 2015). The bug report and the source code are converted into the problem of information retrieval and the source code file with high similarity is recommended to developers.

Jiang et al. (2012) have proposed an automatic approach to identify the null-pointer exceptions based on the combination of stack traces, the static slicing, and spectrum based fault localization. Their task of identifying the null-pointer exceptions can be viewed as a type of software crash in our work. However, their work (Jiang et al., 2012) falls in the category of fault localization, which can leverage the execution of pre-defined test cases to capture the program behavior; in the contrast, in our paper, test cases are unavailable since only a stack trace exists in a submitted crash.

Different from fault localization or bug location, our work does not have the assistance from the input of test cases or bug reports. In the prediction on whether the crashing fault resides in the stack trace, neither test cases nor bug reports are available. In this paper, instead of directly mapping a crash to a function in crash localization (Wu et al., 2014; Gong et al., 2014), we predict whether the crashing code resides in the lines of the stack trace.

7.2. Crash reproduction

Crash reproduction is to automatically generate a test case to trigger a given stack trace (Rößler et al., 2013; Chen and Kim, 2015). ReCore (Rößler et al., 2013) is a typical post-failure crash reproduction technique. ReCore only uses the stack trace and the core dump when a crash occurs. Star (Chen and Kim, 2015) and MuCrash (Xuan et al., 2015) are two stack-trace-based approaches for crash reproduction. Star utilizes the symbolic execution technique to identify the precondition of a crash while MuCrash applies the program mutation technique on existing test cases to trigger a given crash. These two approaches can automate the process of crash reproduction and reduce the manual effort, but both approaches are limited by the combination explosion problem.

A recent work, EvoCrash (Soltani et al., 2016; 2017) employs a genetic algorithm to transform the test generation problem into a search-based problem. During each evolution process of test cases, the fitness function of EvoCrash can narrow down the distance between generated test cases and target test cases.

One of the most related work to our paper is by Gu et al. (2016). This work has modeled the difficulty of crash reproduction with 23 features. The difficulty of crash reproduction is heuristically defined and is evaluated on 45 crashes. In contrast to that work, first, our paper is to predict the linkage between the crashing fault and the stack trace; second, in our paper, five groups of 89 features are extracted to characterize the behavior of the stack trace and the source code; third, our paper conducts detailed empirical evaluation on multiple sampling of crashes from seven projects.

8. Conclusion and future work

To assist manual crash localization by developers, we propose an automatic approach, namely CraTer, to predict crashing fault residence; that is, predicting whether the crashing fault resides in the stack trace or not. This approach can help developers filter out unnecessary statements and prioritize the debugging effort via scheduling crashes. In CraTer, we first extract features from both source code and stack traces. Second, we build a stable and effective model by combining a decision tree algorithm with the SMOTE strategy to process the imbalanced distribution of training data. Third, given a new crash, the trained model is used to predict whether the crashing fault resides in the stack trace. Experiments show that our approach is effective, comparing with other algorithms and strategies under evaluation.

In future work, we plan to design and extract a large number of features to enhance the prediction performance. We would like to visualize the extracted features via syntax highlighting and an interface of pattern searching to help debuggers speed up the current crash localization. Bug reports can be leveraged to assist crash localization. We plan to enhance CraTer with the support of data from bug tracking systems, such as Bugzilla. This may help to reveal the nature of crashes. As mentioned in Section 4.1, configuration issues hurt the scale of the dataset under evaluation. Thus, we plan to try new ways to automatically configure projects in local machines to enlarge potential datasets. A future goal is to conduct large datasets with real-world and large-scale projects and to evaluate the effectiveness and efficiency of our proposed approach.

To improve the performance of CraTer, we also plan to invite developers to evaluate CraTer in daily development; developers can judge the usability and reliability according to their knowledge and experience. Furthermore, it is useful to design a plug-in inside Java IDEs, e.g., Eclipse, to give direct recommendation to developers.

Acknowledgements

The work is supported by the National Key R&D Program of China under Grant No. 2018YFB1003901, the National Natural Science Foundation of China under Grant Nos. 61872273, 61502345, and 61572375, the Young Elite Scientists Sponsorship Program by CAST under Grant No. 2015QNRC001, and the Technological Innovation Projects of Hubei Province under Grant No. 2017AAA125.

References

Ali, S., Andrews, J.H., Dhandapani, T., Wang, W., 2009. Evaluating the accuracy of fault localization techniques. In: 24th IEEE/ACM International Conference on Automated Software Engineering, ASE 2009, Auckland, New Zealand, November 16–20, 2009, pp. 76–87. doi:10.1109/ASE.2009.89.

Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J., 1984. Classification and regression trees. Wadsworth Press.

Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P., 2002. SMOTE: Synthetic minority over-sampling technique. J. Artif. Intell. Res. (JAIR) 16, 321–357. doi:10.1613/jair.953.

Chen, N., Kim, S., 2015. STAR: Stack trace based automatic crash reproduction via symbolic execution. IEEE Trans. Software Eng. 41 (2), 198–220. doi:10.1109/TSE.2014.2363469.

Dang, Y., Wu, R., Zhang, H., Zhang, D., Nobel, P., 2012. Rebucket: A method for clustering duplicate crash reports based on call stack similarity. In: 34th International Conference on Software Engineering, ICSE 2012, June 2–9, 2012, Zurich, Switzerland, pp. 1084–1093. doi:10.1109/ICSE.2012.6227111.

Delahaye, M., du Bousquet, L., 2013. A comparison of mutation analysis tools for java. In: 2013 13th International Conference on Quality Software, Najing, China, July 29–30, 2013, pp. 187–195. doi:10.1109/QSIC.2013.47.

Egghe, L., Leydesdorff, L., 2009. The relation between pearson's correlation coefficient r and salton's cosine measure. JASIST 60 (5), 1027–1036. doi:10.1002/asi.21009.

Elkan, C., 2001. The foundations of cost-sensitive learning. In: Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4–10, 2001, pp. 973–978.

Gong, L., Zhang, H., Seo, H., Kim, S., 2014. Locating crashing faults based on crash stack traces. CoRR abs/1404.4100.

Gopinath, R., Alipour, M.A., Ahmed, I., Jensen, C., Groce, A., 2016. On the limits of mutation reduction strategies. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14–22, 2016, pp. 511–522. doi:10.1145/2884781.2884787.

Gu, Y., Xuan, J., Qian, T., 2016. Automatic reproducible crash detection. In: International Conference on Software Analysis, Testing and Evolution, SATE 2016, November 3–4, 2016, pp. 48–53. doi:10.1109/SATE.2016.15.

Guyon, I., Elisseeff, A., 2003. An introduction to variable and feature selection. J. Mach. Learn. Res. 3, 1157–1182.

Hall, M.A., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H., 2009. The WEKA data mining software: an update. SIGKDD Explorations 11 (1), 10–18. doi:10.1145/1656274.1656278.

Han, J., Kamber, M., Pei, J., 2011. Data mining: Concepts and techniques, 3rd edition. Morgan Kaufmann.

He, H., Garcia, E.A., 2009. Learning from imbalanced data. IEEE Trans. Knowl. Data Eng. 21 (9), 1263–1284. doi:10.1109/TKDE.2008.239.

Jiang, S., Li, W., Li, H., Zhang, Y., Zhang, H., Liu, Y., 2012. Fault localization for null pointer exception based on stack trace and program slicing. In: 2012 12th International Conference on Quality Software, Xi'an, Shaanxi, China, August 27–29, 2012, pp. 9–12. doi:10.1109/QSIC.2012.36.

Jones, J.A., Harrold, M.J., 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In: 20th IEEE/ACM International Conference on Automated Software Engineering ASE 2005, November 7–11, 2005, Long Beach, CA, USA, pp. 273–282. doi:10.1145/1101908.1101949.

Just, R., Jalali, D., Ernst, M.D., 2014. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: International Symposium on Software Testing and Analysis, ISSTA 2014, San Jose, CA, USA - July 21, - 26, 2014, pp. 437–440. doi:10.1145/2610384.2628055.

Kechagia, M., Mitropoulos, D., Spinellis, D., 2015. Charting the API minefield using software telemetry data. Emp. Softw. Eng. 20 (6), 1785–1830. doi:10.1007/s10664-014-9343-7.

Kim, S., Zimmermann, T., Nagappan, N., 2011. Crash graphs: An aggregated view of multiple crashes to improve crash triage. In: Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27–30 2011, pp. 486–493. doi:10.1109/DSN.2011.5958261.

Kononenko, I., 1994. Estimating attributes: Analysis and extensions of RELIEF. In: European Conference on Machine Learning, ECML 1994, Catania, Italy, April 6–8, 1994, pp. 171–182. doi:10.1007/3-540-57868-4_57.

Le, T.B., Lo, D., 2013. Will fault localization work for these failures? an automated approach to predict effectiveness of fault localization tools. In: 2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22–28, 2013, pp. 310–319. doi:10.1109/ICSM.2013.42.

Le, T.B., Lo, D., Le Goues, C., Grunske, L., 2016. A learning-to-rank based fault localization approach using likely invariants. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18–20, 2016, pp. 177–188. doi:10.1145/2931037.2931049.

Le, T.B., Oentaryo, R.J., Lo, D., 2015. Information retrieval and spectrum based bug localization: better together. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015, pp. 579–590. doi:10.1145/2786805.2786880.

Li, D., Li, L., Kim, D., Bissyandé, T.F., Lo, D., Traon, Y.L., 2016. Watch out for this commit! a study of influential software changes. CoRR abs/1606.03266.

Li, Y., Ying, S., Jia, X., Xu, Y., Zhao, L., Cheng, G., Wang, B., Xuan, J., 2018. EH-recommender: Recommending exception handling strategies based on program context. In: 23rd IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2018, Melbourne, Austrilia, December 12–14, 2018, to appear.

Lucia, Lo, D., Xia, X., 2014. Fusion fault localizers. In: ACM/IEEE International Conference on Automated Software Engineering, ASE 2014, Vasteras, Sweden - September 15, - 19, 2014, pp. 127–138. doi:10.1145/2642937.2642983.

Moon, S., Kim, Y., Kim, M., Yoo, S., 2014. Ask the mutants: Mutating faulty programs for fault localization. In: Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA, pp. 153–162. doi:10.1109/ICST.2014.28.

Namin, A.S., Kakarla, S., 2011. The use of mutation in testing experiments and its sensitivity to external threats. In: Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17–21, 2011, pp. 342–352. doi:10.1145/2001420.2001461.

Oliveira, J., Borges, D., Silva, T., Cacho, N., Castor, F., 2018. Do android developers neglect error handling? a maintenance-centric study on the relationship between android abstractions and uncaught exceptions. Journal of Systems and Software 136, 1–18. doi:10.1016/j.jss.2017.10.032.

Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., Seinturier, L., 2016. SPOON: A library for implementing analyses and transformations of java source code. Softw. Pract. Exper. 46 (9), 1155–1179. doi:10.1002/spe.2346.

Qiu, D., Li, B., Leung, H., 2016. Understanding the API usage in java. Inform. Softw. Technol. 73, 81–100. doi:10.1016/j.infsof.2016.01.011.

Quinlan, J.R., 1993. C4.5: Programs for machine learning. Morgan Kaufmann.

Rößler, J., Zeller, A., Fraser, G., Zamfir, C., Candea, G., 2013. Reconstructing core dumps. In: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18–22, 2013, pp. 114–123. doi:10.1109/ICST.2013.18.

Rui, A., Zoeteweij, P., Gemund, A.J.C.V., 2007. On the accuracy of spectrum-based fault localization. In: Testing: Academic and Industrial Conference Practice and Research Techniques - Mutation, 2007. Taicpart-Mutation, pp. 89–98.

Schröter, A., Bettenburg, N., Premraj, R., 2010. Do stack traces help developers fix bugs? In: Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2–3, 2010, Proceedings, pp. 118–121. doi:10.1109/MSR.2010.5463280.

Soltani, M., Panichella, A., van Deursen, A., 2016. Evolutionary testing for crash reproduction. In: Proceedings of the 9th International Workshop on Search-Based Software Testing, SBST@ICSE 2016, Austin, Texas, USA, May 14–22, 2016, pp. 1–4. doi:10.1145/2897010.2897015.

Soltani, M., Panichella, A., van Deursen, A., 2017. A guided genetic algorithm for automated crash reproduction. In: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20–28, 2017, pp. 209–220. doi:10.1109/ICSE.2017.27.

Theisen, C., Herzig, K., Morrison, P., Murphy, B., Williams, L.A., 2015. Approximating attack surfaces with stack traces. In: 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16–24, 2015, Volume 2, pp. 199–208. doi:10.1109/ICSE.2015.148.

Wang, G., Lochovsky, F.H., 2004. Feature selection with conditional mutual information maximin in text categorization. In: Proceedings of the 2004 ACM CIKM International Conference on Information and Knowledge Management, Washington, DC, USA, November 8–13, 2004, pp. 342–349. doi:10.1145/1031171.1031241.

Wang, S., Lo, D., Vasilescu, B., Serebrenik, A., 2014. Entagrec: An enhanced tag recommendation system for software information sites. In: 30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29, - October 3, 2014, pp. 291–300. doi:10.1109/ICSME.2014.51.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., 2012. Experimentation in software engineering. Springer doi:10.1007/978-3-642-29044-2.

Wong, C., Xiong, Y., Zhang, H., Hao, D., Zhang, L., Mei, H., 2014. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: 30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29, - October 3, 2014, pp. 181–190. doi:10.1109/ICSME.2014.40.

Wu, R., Xiao, X., Cheung, S., Zhang, H., Zhang, C., 2016. Casper: an efficient approach to call trace collection. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20, - 22, 2016, pp. 678–690. doi:10.1145/2837614.2837619.

Wu, R., Zhang, H., Cheung, S., Kim, S., 2014. Crashlocator: locating crashing faults based on crash stacks. In: International Symposium on Software Testing and Analysis, ISSTA 2014, San Jose, CA, USA - July 21, - 26, 2014, pp. 204–214. doi:10.1145/2610384.2610386.

Xia, X., Lo, D., Shihab, E., Wang, X., Zhou, B., 2015. Automatic, high accuracy prediction of reopened bugs. Autom. Softw. Eng. 22 (1), 75–109. doi:10.1007/s10515-014-0162-2.

Xia, X., Lo, D., Wang, X., Zhang, C., Wang, X., 2014. Cross-language bug localization. In: 22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2–3, 2014, pp. 275–278. doi:10.1145/2597008.2597788.

Xuan, J., Cornu, B., Martinez, M., Baudry, B., Seinturier, L., Monperrus, M., 2016. B-Refactoring: automatic test code refactoring to improve dynamic analysis. Inform. Softw. Technol. 76, 65–80. doi:10.1016/j.infsof.2016.04.016.

Xuan, J., Jiang, H., Zhang, H., Ren, Z., 2017. Developer recommendation on bug commenting: a ranking approach for the developer crowd. SCIENCE CHINA Inform. Sci. 60 (7), 072105:1–072105:18.

Xuan, J., Martinez, M., Demarco, F., Clement, M., Marcote, S.R.L., Durieux, T., Berre, D.L., Monperrus, M., 2017. Nopol: automatic repair of conditional statement bugs in java programs. IEEE Trans. Softw. Eng. 43 (1), 34–55. doi:10.1109/TSE.2016.2560811.

Xuan, J., Xie, X., Monperrus, M., 2015. Crash reproduction via test case mutation: let existing test cases help. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015, pp. 910–913. doi:10.1145/2786805.2803206.

Zhang, J., Wang, Z., Zhang, L., Hao, D., Zang, L., Cheng, S., Zhang, L., 2016. Predictive mutation testing. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18–20, 2016, pp. 342–353. doi:10.1145/2931037.2931038.

Zhang, L., Zhang, L., Khurshid, S., 2013. Injecting mechanical faults to localize developer faults for evolving software. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26–31, 2013, pp. 765–784. doi:10.1145/2509136.2509551.

Zhang, X., Chen, Y., Gu, Y., Zou, W., Xie, X., Jia, X., Xuan, J., 2018. How do multiple pull requests change the same code: A study of competing pull requests in GitHub. In: 34th IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23–29, 2018, to appear.

Zhou, J., Zhang, H., Lo, D., 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In: 34th International Conference on Software Engineering, ICSE 2012, June 2–9, 2012, Zurich, Switzerland, pp. 14–24. doi:10.1109/ICSE.2012.6227210.

Yongfeng Gu is a PhD candidate at the School of Computer Science, Wuhan University, China. He received the BSc degree in 2015 from the School of Computer Science and Information Engineering, Hubei University. His research interests include software testing and debugging, mining software repositories, and software performance prediction.

Jifeng Xuan is a professor at the School of Computer Science, Wuhan University, China. He received the BSc degree and the PhD degree from Dalian University of Technology, China. He was previously a postdoctoral researcher at the INRIA Lille Nord Europe, France. His research interests include software testing and debugging, software data analysis, and search based software engineering. He is a member of the ACM, IEEE, and CCF.

Hongyu Zhang is an associate professor with The University of Newcastle, Australia. Previously, he was a lead researcher at Microsoft Research Asia and an associate professor at Tsinghua University, China. He received his PhD degree from National University of Singapore in 2003. His research is in the area of Software Engineering, in particular, software analytics, testing, maintenance, metrics, and reuse. He has published more than 120 research papers in reputable international journals and conferences. He received two ACM Distinguished Paper awards. He has also served as a program committee member for many software engineering conferences. More information about him can be found at: https://sites.google.com/site/hongyujohn/.

Lanxin Zhang received the BSc degree in 2018 from Wuhan University, China. He will start his master study in Carnegie Mellon University. His research interests include software testing and debugging.

Qingna Fan received the BSc degree in software engineering and the MSc degree in computer science and technology, from Dalian University of Technology, China. She was an automation engineer in Intel Semiconductor Dalian Ltd and an AI researcher in HY Cross-Domain. Her research interests include artificial intelligence and software optimization.

Xiaoyuan Xie received the BSc and MPhil degrees in computer science from Southeast University, China in 2005 and 2007, respectively, and received the PhD degree in Computer Science from Swinburne University of Technology, Australia in 2012. She is currently a professor at the School of Computer Science, Wuhan University, China. Her research interests include software analysis, testing, debugging, and search-based software engineering.

Tieyun Qian received the PhD degree in computer science from Huazhong University of Science and Technology in 2006. She is a professor at Wuhan University, China. Her research interests include web mining, data management, etc. She has published over 40 papers on leading journals and conferences like ACL, EMNLP, COLING, SIGIR, CIKM, and INS. She is a member of ACL, ACM, IEEE, and CCF.