The Journal of Systems & Software 170 (2020) 110763

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Imbalanced metric learning for crashing fault residence prediction

Zhou Xu [a,b,c,1], Kunsong Zhao [d,1], Meng Yan [a,b,e,*], Peipei Yuan [f], Ling Xu [a,b], Yan Lei [a,b], Xiaohong Zhang [a,b]

[a] Key Laboratory of Dependable Service Computing in Cyber Physical Society (Chongqing University), Ministry of Education, China
[b] School of Big Data and Software Engineering, Chongqing University, Chongqing, China
[c] College of Computer Science, Chongqing University, Chongqing, China
[d] School of Computer Science, Wuhan University, Wuhan, China
[e] PengCheng Laboratory, China
[f] School of Electronic Information and Communications, Huazhong University of Science and Technology, Wuhan, China

ARTICLE INFO

Article history:
Received 15 April 2020
Received in revised form 23 July 2020
Accepted 24 July 2020
Available online 3 August 2020

Keywords:
Crashing fault residence prediction
Stack trace
Metric learning
Class imbalanced learning

ABSTRACT

As the software crash usually does great harm, locating the fault causing the crash (i.e., the crashing fault) has always been a hot research topic. As the stack trace in the crash reports usually contains abundant information related the crash, it is helpful to find the root cause of the crash. Recently, researchers extracted features of the crash, then constructed the classification model on the features to predict whether the crashing fault resides in the stack trace. This process can accelerate the debugging process and save debugging efforts. In this work, we apply a state-of-the-art metric learning method called IML to crash data for crashing fault residence prediction. This method uses Mahalanobis distance based metric learning to learn high-quality feature representation by reducing the distance between crash instances with the same label and increasing the distance between crash instances with different labels. In addition, this method designs a new loss function that includes four types of losses with different weights to cope with the class imbalanced issue of crash data. The experiments on seven open source software projects show that our IML method performs significantly better than nine sampling based and five ensemble based imbalanced learning methods in terms of three performance indicators.

1. Introduction

Nowadays, software has played a significant role in people's daily work and life. Due to a variety of uncontrollable reasons, programmers will face a lot of software faults during their daily development and maintenance (Mathur, 2007). The serious faults can cause program exceptions and failures unexpectedly, also called that the software crashes. This kind of terrible situation will result in a poor user experience and negatively affect the reputation of the software company. As developing a software that never crashes is impractical, it is critical to find the root cause of the crash, i.e., the residence of the fault causing the crash (crashing fault for short). This process can help the programmers focus on the corresponding code at the location which can greatly promote the efficiency of the software development and improve product quality.

In order to quickly locate the code of the crashing fault during the development process, software usually has crash reporting system to automatically collect and manage the crash report when the crash occurred. The crash report contains a rich source of data about the crash circumstances, such as the crash site environment and crash stack traces (Dhaliwal et al., 2011). The stack trace records the exception type, crash location, and the trajectory of a series of function calls when a program exception is thrown. An empirical study on Mozilla software stated that most of the crashing faults locate in the functions that reside inside the stack trace (Li et al., 2018). If the crashing fault exactly matches the information of a function in the stack trace, that is, the crashing fault resides inside the stack trace, developers only need to focus on the code of the corresponding function; whereas if the crashing fault is in a function that resides outside the stack trace, developers have to check the function call sequence which involves in inspecting massive source code. Therefore, predicting whether the function of the crashing fault resides in the stack trace can promote the crash localization process by helping the developers to quickly determine which part of code need to be reviewed. The prediction results of the crashing fault residence is beneficial to optimize the allocation of debugging resources and save many of inspection efforts. Crashing fault residence prediction is an essential research topic for software quality assurance activities.

[*] Corresponding author at: School of Big Data and Software Engineering, Chongqing University, Chongqing, China.
E-mail address: mengy@cqu.edu.cn (M. Yan).

[1] Zhou Xu and Kunsong Zhao contributed equally to this work.

https://doi.org/10.1016/j.jss.2020.110763
0164-1212/© 2020 Published by Elsevier Inc.

Gu et al. (2019) were the first to study this issue by proposing a method, called CraTer, to automatically detect whether the faulty code of a crash exactly matches the information of a function in the stack trace. More specifically, they simulated the crashes by seeding the fault based on program mutation. For each crash instance, they extracted 89 features from the faulty code and stack traces, and collected the label information of the crashing fault from the bug-fixing logs. They used several traditional classifiers to build the models on the labeled data to predict the crashing fault residence of new-submitted crashes. They treated the crashing fault residence prediction as a typical binary classification learning problem. In general, the performance of classification model heavily relies on the quality of data representation. Thus, transforming the feature space to learn high-quality feature representations has the potential to promote the prediction performance of crashing fault residence. In addition, in the collected data, the number of crash instances that reside inside the stack trace is much fewer than those that reside outside the stack trace, that is, the data have the inherent characteristic of class imbalance. As this characteristic can negatively impact the performance of the classification model, it is critical to alleviate this imbalanced issue for performance improvement. In this work, we propose a new crashing fault residence prediction model based on a state-of-the-art feature engineering method called **I**mbalanced **M**etric **L**earning (**IML**) (Gautheron et al., 2019) to address the above two issues. IML carries out the feature representation learning by applying Mahalanobis distance based metric learning to the crash instance data, aiming to enlarge the distance of crash instances with different labels while shorten the distance of crash instances with the same label. Meanwhile, IML addresses the class imbalanced issue by decomposing the loss function into four parts with different weights according to the labels of a crash instance pair.

To evaluate the effectiveness of our IML method for predicting crashing fault residence, we conduct experiments on benchmark dataset consists of seven open source software projects with 3 performance indicators. Across the seven projects, our IML method achieves average F-measure value of 0.617 for crash instances locating inside the stack trace, average F-measure value of 0.907 for crash instances locating outside the stack trace, and average MCC value of 0.530. The experimental results show that our IML method achieves significantly better performance than 9 sampling based imbalanced learning methods and 5 ensemble based imbalanced learning methods on all 3 performance indicators.

In summary, this paper makes the following main contributions:

(1) We propose to address both the feature representation learning and class imbalanced issues for crashing fault residence prediction with a novel metric learning method called IML. Our IML method uses the metric learning method to learn more discriminative feature representation and utilizes an improved loss function to alleviate the adverse impact of class imbalance.

(2) We comprehensively evaluate our IML method on seven open-source software projects using three performance indicators. The experimental results show the significant superiority of our IML method over 14 baseline methods under comparison.

The remainder of this paper is organized as follows. We introduce the background of software crash and stack trace, and previous studies about stack traces based fault localization in Section 2. We present the technical details of our IML method in Section 3. We illustrate our experimental setup, such as the benchmark dataset, performance indicators, the statistic test method, the process of data partition, and the parameter configuration in Section 4. We report the experimental results in Section 5. We list three types of threats to the validity of our work in Section 6. Finally, we conclude this work in Section 7.

2. Related work

2.1. Software crash and stack traces

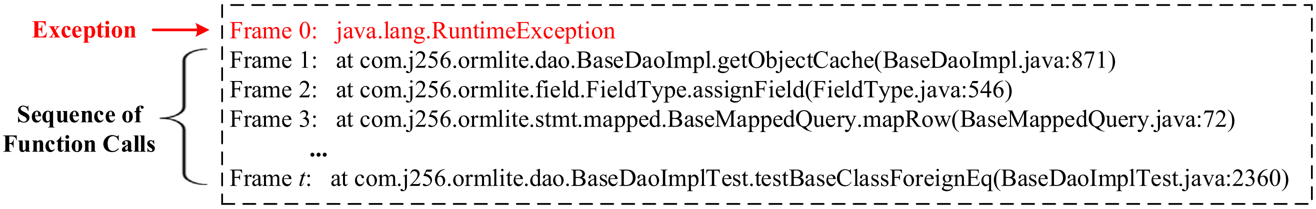
Crash occurs when the software stops functioning properly and exits, which is one of serious manifestations of the software failure. Thus, crashes need to be prioritized for fixing. As the corresponding crash information is helpful for quick defect fixing, many crash reporting systems are used to automatically collect it in which the stack trace is one of the recorded information. The stack trace provides some information related the exception, such as the type of crash exception and the sequence of function calls that lead to the exception being thrown. Such information can be used for crash reproduction (Soltani et al., 2020; Xuan et al., 2015) and crashing faults localization (Gong et al., 2014; Wu et al., 2014).

The stack trace includes multiple (assuming t) frame objects. The first frame (short for Frame 0) records the thrown exception of the crash while each other frame records the information about a function that the code is called. The second frame (short for Frame 1) can be called as the top frame which provides the location of the exception is thrown and the last frame (short for Frame t) records the place of the initial function call. The main terms in the Frame 1 to Frame t include the class name, function name, and line number of the code. Fig. 1 shows an example of stack trace collected by a previous work (Gu et al., 2019) in which the faulty code matches the three elements of the function in Frame 1.

Here, we give an example to illustrate the application scenario of methods for crashing fault residence prediction as our IML method as follows: assume a project team is developing a software system and have collected some historical crash data with labels in the previous development process. One day the system crashed, and the team members got the corresponding stack trace with t frames. Without our method, the team members have to check the sequence of function calls among the stack trace. This process involves in reviewing many lines of code which requires many debugging efforts. With our method, the team members can use the historical labeled crash data to learn the metric representation and then train a classification model to predict whether the new crashing fault locates in the stack trace or not. If yes, the team members only need carefully review the t lines of code in the frame to find the faulty code and then fix it. This prediction result can greatly promote the debugging process.

2.2. Fault localization based on stack traces

The work most relevant to our study is the stack traces based fault (crash or bug) localization. This kind of study resorts the stack trace information to locate the faulty functions (sometimes files). Schröter et al. (2010) conducted an empirical study and showed a strong evidence that the stack traces are important information to support developers from the Eclipse project in debugging. In addition, they found that a crashing fault typically located in one of the top-10 frames of the stack trace. Indi et al. (2016) stated that the Java exception stack traces were helpful for the students to analyze and fix bugs in program. Jiang et al. (2012) proposed a method combining the dynamic information generated from stack traces with the static analysis to locate the faults of null pointer exception. They conducted a case study on two versions of Ant project and showed that their



```
Frame 0:  java.lang.RuntimeException
Frame 1:  at com.j256.ormlite.dao.BaseDaoImpl.getObjectCache(BaseDaoImpl.java:871)
Frame 2:  at com.j256.ormlite.field.FieldType.assignField(FieldType.java:546)
Frame 3:  at com.j256.ormlite.stmt.mapped.BaseMappedQuery.mapRow(BaseMappedQuery.java:72)
          ...
Frame t:  at com.j256.ormlite.dao.BaseDaoImplTest.testBaseClassForeignEq(BaseDaoImplTest.java:2360)
```

Fig. 1. An example of stack trace.

method was effective. Wang et al. (2013) proposed a method called BFFinder to locate defective files with crash correlation groups which were identified by stack traces. Their experiments on Firefox and Eclipse projects showed that BFFinder achieved precision of 100% and 79%, recall of 90% and 65% on the two projects, respectively. Moreno et al. (2014) proposed a static method called Lobster for bug localization using the stack traces based similarity and text retrieval based textual similarity. Their experiments on 14 projects showed that Lobster achieved better performance than Lucene based method. Wu et al. (2014) proposed a method called CrashLocator to locate crashing fault at function level. CrashLocator applied the static analysis methods to the crash stacks for generating approximate crash traces. For the functions in the crash traces, CrashLocator used some factors to calculate the suspicious scores of these functions that cause the crash. Their experiments on three Mozilla products showed that CrashLocator outperformed conventional stack-only methods significantly. Gong et al. (2014) proposed a statistical fault localization framework to locate functions with crashing faults. This framework consisted of instrumentation and static analysis, generation of passing and failing execution traces, and suspiciousness calculation with Ochiai method. Their experiments on two versions of Firefox project showed that their framework can locate more than 63.9% and 52.7% of crashing faults by checking 5% of functions on the two versions, respectively. Wong et al. (2014) proposed a method called BRTracer to identify faulty files with segmentation and stack trace analysis. Their experiments on three projects demonstrated that BRTracer performed significantly better than a representation method called BugLocator. Wu et al. (2018) proposed a method called ChangeLocator to locate the changes that would induce the crashing faults. ChangeLocator combined traditional code characteristics with crash reporting information to train a classification model on the data from the historical fixed crashes, and then calculated the probability of a change inducing the crash. The experiments on six release versions of Netbeans project illustrated that ChangeLocator was superior to an information retrieval-based fault localization method.

Different from the above studies that usually calculated suspicious scores of the functions being fault, our work applies the machine learning method to predict the root cause of a crash, i.e., determining whether the crashing fault matches the class name, function, and line number of the recorded information in the stack trace. From this point of view, our work locates the faulty code at code line level, not the function level. Gu et al. (2019) were the first to study the crashing fault residence prediction by proposing a method called CraTer. However, their work did not perform conversion processing on data for better feature representation. Following their work, Xu et al. (2019c) proposed a cross project model called BDA to predict the crashing fault residence of a project by using the labeled data of another project. However, they did not take the class imbalanced issue into account and the performance of cross project model is usually constrained by the distribution differences across data. Different from the above two studies, our work uses metric learning method to learn high-quality data representation and deals with the class imbalanced issue during the process of feature representation learning. In addition, as we conduct crashing fault residence prediction using the data within the project, the distribution difference issue can be ignored.

3. Method

Imbalanced **M**etric **L**earning (**IML**) (Gautheron et al., 2019) is a novel representation learning that focuses on learning a metric in an imbalanced scenario. Here, we describe the details of IML method for feature representation learning on crash data.

3.1. Notations

Assume that the feature set of the crash instances as $\mathbf{X} = \{x_i\}_{i=1}^{n} \in \mathbb{R}^{n\times m}$ and the corresponding label set as $\mathbf{Y} = \{y_i\}_{i=1}^{n} \in \mathbb{R}^{n\times 1}$, where $x_i = [x_{i1}, x_{i2}, \ldots, x_{im}] \in \mathbb{R}^m$ denotes the ith crash instance, $y_i \in \{-1, 1\}$ denotes the label of x_i, n is the number of crash instances, and m is the feature dimension. $y_i = 1$ indicates that the corresponding crash fault resides inside the stack trace (short for 'InTrace'), whereas $y_i = -1$ indicates that the corresponding crash fault resides outside the stack trace (short for 'OutTrace'). We further define $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$ as the labeled data where $z = (x, y)$ indicates a labeled crash instance. Assume that the data with n labeled crash instances is defined as $S = \{z_i = (x_i, y_i)_{i=1}^{n}\} = S^+ \cup S^-$, where S^+ indicates the set of positive crash instances (i.e., label with 'InTrace') and S^- indicates the set of negative crash instances (i.e., label with 'OutTrace'). For the used crash instance data, the number of positive crash instances $n^+ = |S^+|$ is much smaller than the number of negative crash instances $n^- = |S^-|$, which means that the data are class imbalanced.

3.2. Metric learning

Metric learning is also known as similarity learning. It reconstructs the original data into a more reasonable latent space to measure the similarity between crash instances expecting that crash instances with different labels are less similar while the crash instances with the same label are more similar. Mahalanobis distance is a commonly used distance measure in metric learning. The latent space learned by this distance measure is very suitable for the classification task of kNN model. Mahalanobis distance can be regarded as an improvement version of Euclidean distance by correcting the inconsistent and related problems of various dimensions in Euclidean distance (Maesschalck et al., 2000; Weinberger and Saul, 2009).

The Euclidean distance ($d_E(x, x')$) of two crash instances is defined as

$$d_E(x, x') = \sqrt{(x - x')^\top \mathbf{I}(x - x')} \tag{1}$$

where $\mathbf{I}$ is the identify matrix.

The Mahalanobis distance can be regarded as a modification of Euclidean distance and is defined as

$$d_M(x, x') = \sqrt{(x - x')^\top \mathbf{M}(x - x')} \tag{2}$$

where $\mathbf{M} \in \mathbb{R}^{m\times m}$ is a **P**ositive **S**emi-**D**efinite (**PSD**) matrix which can be decomposed as $\mathbf{M} = \mathbf{L}^\top\mathbf{L}$, where $\mathbf{L} \in \mathbb{R}^{r\times m}$ is a linear projection induced by $\mathbf{M}$ (r is the rank of $\mathbf{M}$). Thus, Eq. (2) can be rewritten as

$$\begin{aligned} d_M(x,x') &= \sqrt{(x-x')^\top\mathbf{L}^\top\mathbf{L}(x-x')} \\ &= \sqrt{(\mathbf{L}x-\mathbf{L}x')^\top(\mathbf{L}x-\mathbf{L}x')} \end{aligned} \tag{3}$$

Compared with Eqs. (1) and (3), the Mahalanobis distance between two crash instances x and x' is equivalent to the Euclidean distance after mapping x and x' with the projection $\mathbf{L}$ into the r-dimensional space, i.e., $\mathbf{L}x$ and $\mathbf{L}x'$.

Mahalanobis distance based metric learning aims to minimize the loss ℓ over all pairs of crash instances in data S that can be expressed as

$$\min_{M\succcurlyeq 0} J = \frac{1}{n^2}\sum_{(z,z')\in S}\ell(\mathbf{M},z,z') + \lambda\|\mathbf{M}-\mathbf{I}\|_F^2 \tag{4}$$

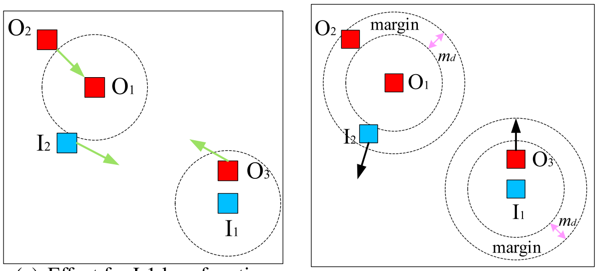
where $\|\mathbf{M}-\mathbf{I}\|_F^2$ is a regularization term under the PSD constraint $M \succcurlyeq 0$ and $\|\cdot\|_F^2$ is the Frobenius norm. This term is able to learn a Mahalanobis metric close to the Euclidean distance while satisfying the best of semantic constraints (Gautheron et al., 2019).

3.3. Imbalanced metric learning

The main disadvantage of the above metric learning formulation is that the loss function treats any pair of crash instances (z,z') equally, that is, it gives the same weight to all pairs of crash instance without considering the corresponding labels y and y'. This treatment is not well suitable for our imbalanced crash data since the interest objects are the positive crash instances with label 'InTrace' which is the minority class. In order to deal with this problem, Gautheron et al. (2019) optimized the above metric learning formulation by decomposing the loss function into multiple parts according to the labels of the two crash instances in a pair. In other words, the idea is to give different weights to the crash instance pairs with the purpose of reducing the negative effect of the class imbalance. More specifically, for all crash instance pairs (z,z') and all $\mathbf{M} \in \mathbb{R}^{m\times m}$, the loss function $\ell(\mathbf{M},z,z')$ is decomposed into the following form

$$\ell = \begin{cases} \mathcal{L}_1 = [d_M^2(x,x')-1]_+ & \text{if } y = y' \\ \mathcal{L}_2 = [1+m_d-d_M^2(x,x')]_+ & \text{if } y \neq y', \end{cases} \tag{5}$$

where $[\cdot]_+$ is the Hinge loss to take the maximum value between 0 and the element inside the brackets, and $m_d \geqslant 0$ is a margin parameter. From the expression of the two decomposed loss functions, we can see that, for the loss function $\mathcal{L}_1$, when the distance between two crash instances of a pair with the same label is larger than 1, it will generate loss. Thus, this loss function aims to narrow the distance of two crash instances with the same label less than 1. For the loss function $\mathcal{L}_2$, when the distance between two crash instances of a pair with different labels is less than $1+m_d$, it will cause loss. Thus, this loss function aims to enlarge the distance of two crash instances with different labels more than 1 plus a margin. Fig. 2 shows the effect of the two loss functions. The three red squares (i.e., O_1, O_2, and O_3) denote three crash instances with label 'OutTrace' and the two blue squares (i.e., I_1 and I_2) denote two crash instances with label 'InTrace'. In Fig. 2(a), for O_1 and I_1, as the distances between $\langle O_1,O_2\rangle$, $\langle O_1,O_3\rangle$, and $\langle I_1,I_2\rangle$ are larger than 1, thus, $\mathcal{L}_1$ loss function will bring them closer together by lessening their distances. In Fig. 2(b), for O_1 and I_1, as the distances between $\langle O_1,I_2\rangle$ and $\langle I_1,O_3\rangle$ are smaller than $1+m_d$, thus, $\mathcal{L}_2$ loss function will separate them further apart by enlarging their distances.



(a) Effect for L1 loss function (b) Effect for L2 loss function

Fig. 2. An example of the effect of the two loss functions.

By considering different losses for the crash instance pairs under district scenarios, the Mahalanobis distance based imbalanced metric learning is formalized as the following convex expression

$$\begin{aligned} \min_{M\succcurlyeq 0} J = & \frac{a}{4|Same^+|}\sum_{(z,z')\in Same^+}\mathcal{L}_1 + \frac{a}{4|Same^-|}\sum_{(z,z')\in Same^-}\mathcal{L}_1 \\ & + \frac{1-a}{4|Diff^+|}\sum_{(z,z')\in Diff^+}\mathcal{L}_2 + \frac{1-a}{4|Diff^-|}\sum_{(z,z')\in Diff^-}\mathcal{L}_2 \\ & + \lambda\|\mathbf{M}-\mathbf{I}\|_F^2 \end{aligned} \tag{6}$$

where the four terms $Same^+$, $Same^-$, $Diff^+$, and $Diff^-$ are defined as the subsets $S^+\times S^+$, $S^-\times S^-$, $S^+\times S^-$, and $S^-\times S^+$ respectively, and the parameter $a \in [0,1]$ which is used to balance the effect of bringing crash instance pairs with the same label closer and keeping crash instance pairs with different labels apart. The decomposition by dividing the original loss function of metric learning into different parts can treat the losses caused by the crash instance pairs from the above four sets differently.

The difference between Eq. (4) of original metric learning and Eq. (6) of the imbalanced metric learning is that the former one gives all crash instance pairs from the 4 subsets (i.e., $Same^+$, $Same^-$, $Diff^+$, and $Diff^-$) the same weigh $\frac{1}{n^2}$, while the latter one given the pairs diverse weights. In our crash instance data, the number of pairs in $Same^+$ ($|Same^+|$) and $Same^-$ ($|Same^-|$) are much smaller than that in $Diff^+$ ($|Diff^+|$). Intuitively, the pairs in the subsets $Same^+$ and $Diff^+$ should have larger impact on the loss function compared with the pairs in other two subsets since the first elements in such pairs are the positive crash instances, i.e., the interest minority class. Thus, such kind of losses should be given higher weights. For this purpose, IML method employs a simple strategy to assign the weight to the loss function according to the number of pairs in the corresponding subset. This weighting strategy enables to treat the four types of loss functions under the four subsets differently, which makes the learned metric more robust to imbalanced data.

As the number of the crash instance pairs in the subsets $Same^+$, $Same^-$, $Diff^+$, and $Diff^-$ are n^+n^+, n^-n^-, n^+n^-, and n^-n^+, respectively, it is quite inefficient to consider all pairs in these subsets for the metric learning especially when the data consist of a large number of crash instances. There are two strategies to reduce the pair space: one is to randomly select the pairs for a crash instance (Zadeh et al., 2016), another is to carefully select the pairs for a crash instance based on the nearest neighbor rule (Lu et al., 2014). In terms of the former one, this strategy may generate pairs in only one subset for some crash instances since the data are usually imbalanced. For example, for a negative crash instance, it may randomly select multiple instances with the same label since they occupy the majority of the data. In terms of the latter one, for each crash instance, this strategy selects its multiple (such as k) neighborhood with the same as

Table 1
Statistic information of the seven projects.

Projects	Version	# Mutants	# Killed mutants	# Crashing faults	% InTrace
Codec (http://commons.apache.org/codec/)	1.10	2901	2601	610	29.0%
Collections (http://commons.apache.org/collections/)	4.1	6650	5300	1350	20.2%
IO (http://commons.apache.org/io/)	2.5	3337	2728	686	21.7%
Jsoup (http://jsoup.org/)	1.11.1	2657	1892	601	20.0%
JSqlParser (http://github.com/JSQLParser/JSqlParser)	0.9.7	8757	5636	647	9.4%
Mango (http://www.jfaster.org/)	1.5.4	5149	1570	733	7.2%
Ormlite-Core (http://github.com/j256/ormlite-core)	5.1	3563	2751	1303	25.0%

well as different labels to form the pairs. Thus, each positive crash instance will generate k pairs in both subsets $Same^+$ and $Diff^+$, and each negative crash instance will generate k pairs in both subsets $Same^-$ and $Diff^-$. From this point of view, this strategy can well adapt to the imbalanced data. Thus, in this work, our IML method employs the second strategy to reduce the pair space and take the imbalance property of the data into account.

4. Experiment setup

In this section, we first detail the used benchmark dataset including the basic information of the dataset and the corresponding collection process, then we describe the used performance indicators, the statistic test method, the used classification model, the parameter configuration, and the data partition process.

4.1. Benchmark dataset

To evaluate the performance of our IML method for crashing fault residence prediction task, we conduct experiments on a publicly available benchmark dataset shared by a previous study (Gu et al., 2019). This dataset consists of seven open source Java projects, i.e., Apache Commons **Codec**, Apache Commons **Collections**, Apache Commons **IO**, **Jsoup**, **JSqlParser**, **Mango**, and **Ormlite-Core**. Codec provides solutions of common encoders and decoders. Collections adds many powerful data structures that speeds up the development of most important Java applications. IO is a tool library that assists the development of IO functionality. Jsoup is a Java HTML parser that can directly parse the URL address and HTML text content. JSqlParser is a plug-in that parses SQL statements and translates them into a hierarchy of Java classes. Mango is a high-performance distributed framework for object relational mapping. Ormlite-Core is a package that provides the core functionality for the Java database connectivity and Android packages.

Table 1 presents the basic information of these projects, including the version number, the number of mutants produced by program mutation (# Mutants), the number of mutants without passing the test execution based on the test cases (# Killed mutants), the number of remaining crash instances after removing useless mutants from the killed ones(# Crashes), and the percentage of crash instances inside the stack trace (% InTrace). Note that the crashing faults are the killed mutants that are reserved after removing useless ones based on some predesignated rules. The rules consist of four types, i.e., the mutants with all test cases passed and whose stack traces only include AssertionFailedError, ComparisonFailure, and test cases (Gu et al., 2019).

The data were collected through 3 steps, including generating crashing faults, extracting features, and labeling crashing fault residence. The brief description of each step is presented as follows:

4.1.1. Generating crashing faults

To obtain the crashing faults, Gu et al. (2019) first seeded faults into projects with the mutation testing tool PIT system to simulate the crashing faults following previous studies (Zhang et al., 2013; Moon et al., 2014). They chose seven default mutation operators (including conditionals boundary mutator, increments mutator, invert negatives mutator, math mutator, negate conditionals mutator, return values mutator, and void method call mutator) to generate the program mutation. Then, they followed four types of rules to discard some mutants without causing crashes and the remaining ones are the crash instances used in this work.

4.1.2. Extracting features

The features of the crashing faults were extracted with a Java compiler called Spoon (Pawlak et al., 2016) for static program analysis and transformation. Gu et al. (2019) extracted five types of features including a total of 89 ones. These features are extracted from the **S**tack **T**race (short for **ST** group), the source code in the **T**op **F**rame and **B**ottom **F**rame (short for **TF** group and **BF** group, respectively), and the **N**ormalized ones for TF group and BF group (short for **NTF** group and **NBF** group, respectively.) The features extracted from the stack trace have the potential to characterize the difficulty of dealing with the crash. The features extracted from the top frame can represent the state of the program when it crashes. The features extracted from the bottom frame can reflect some information of the initial function call. The brief description of these features are presented in Table 2.

4.1.3. Labeling crashing fault residence

The residence labels of the crashing faults are based on the three main elements recorded in the frame, i.e., class name, function name, and line number. The crashing fault is labeled as 'InTrace' if its code information is the same as the above three elements in one of frames in the stack trace, which means that this crashing fault resides inside the stack trace. Otherwise, the crash fault is labeled as 'OutTrace'. The label collection is done by checking the bug-fixing logs.

4.2. Performance indicators

As predicting the residence of crashing faults is a binary classification task, we employ some commonly-used indicators, including F-measure and Matthew Correlation Coefficient (MCC) in the field of information retrieval to evaluate the performance of our IML method. To calculate these indicators, we first define four basic terms for positive crash instances as follows: **T**rue **P**ositive (**TP**) denotes the number of crashing faults with label 'InTrace' that are predicted as 'InTrace'; **F**alse **N**egative (**FN**) denotes the number of crashing faults with label 'InTrace' that are predicted as 'OutTrace'; **T**rue **N**egative (**TN**) denotes the number of crashing faults with label 'OutTrace' that are predicted as 'OutTrace'; **F**alse **P**ositive (**FP**) denotes the number of crashing faults with label 'OutTrace' that are predicted as 'InTrace'.

Based on the above terms, we can get two commonly-used indicators that are used to calculate F-measure for positive crash

Table 2
Brief descriptions of extracted 89 features for crashing faults.

Features	Brief description	
ST group	Features related to the stack trace (ST)	
ST01	Type of the exception in the crash	
ST02	Number of frames of the stack trace	
ST03	Number of classes of the stack trace	
ST04	Number of functions of the stack trace	
ST05	Whether an overloaded function exists in 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	
TF group (BF group)	Features related to the top frame (TF) and bottom frame (BF)	
TF01 (BF01)	Class in the top (bottom) frame	Number of local variables
TF02 (BF02)		Number of filed number
TF03 (BF03)		Function (except constructor one) number
TF04 (BF04)		Imported packages number
TF05 (BF05)		Whether the class is inherited from others
TF06 (BF06)		Lines of code of comments
TF07 (BF07)	Function in the top (bottom) frame	Lines of code
TF08 (BF08)		Number of parameters
TF09 (BF09)		Number of local variable
TF10 (BF10)		Number of if-statements
TF11 (BF11)		Number of loops
TF12 (BF12)		Number of for statements
TF13 (BF13)		Number of for-each statement
TF14 (BF14)		Number of while statements
TF15 (BF15)		Number of do-while statements
TF16 (BF16)		Number of try blocks
TF17 (BF17)		Number of catch block
TF18 (BF18)		Number of finally blocks
TF19 (BF19)		Number of assignment statements
TF20 (BF20)		Number of function calls
TF21 (BF21)		Number of return statements
TF22 (BF22)		Number of unary operators
TF23 (BF23)		Number of binary operators
NTF group (NBF group)	Normalizing features in TF group (NTF) and BF group (NBF) by lines of code	
NTF01 (NBF01)	TF08/TF07 (BF08/BF07)	
NTF02 (NBF02)	TF09/TF07 (BF09/BF07)	
...	...	
NTF15 (NBF15)	TF22/TF07 (BF22/BF07)	
NTF16 (NBF16)	TF23/TF07 (BF23/BF07)	

instances (short for F_P). The first one is Precision for positive crash instances (short for P_P) which is defined as the ratio of the number of crash instances with label 'InTrace' that are correctly predicted to the number of crash instances that are predicted as 'InTrace', i.e., $P_P = \frac{TP}{TP+FP}$. The second one is Recall for positive crash instances (short for R_P) which is defined as the ratio of the number of crash instances with label 'InTrace' that are correctly predicted to the number of crash instances with label 'InTrace', i.e., $R_P = \frac{TP}{TP+FN}$. Then, F_P is the harmonic mean of P_P and R_P as

$$F_P = \frac{2 \times P_P \times R_P}{P_P + R_P} \tag{7}$$

Similarly, let define four basic terms for negative crash instance as TP′, FN′, TN′, and FP′, we can get Precision and Recall for negative crash instances (short for P_N and R_N) as $P_N = \frac{TP'}{TP'+FP'}$ and $R_N = \frac{TP'}{TP'+FN'}$. Then, F-measure for negative crash instances (short for F_N) is defined as

$$F_N = \frac{2 \times P_N \times R_N}{P_N + R_N} \tag{8}$$

MCC essentially measures the correlation coefficient between the actual result and the predicted result by the classification model. It comprehensively considers the four basic terms TP, TN, FP, and FN which is deemed as a relatively balanced indicator for imbalanced data. MCC for positive crash instances (short for MCC_P) is defined as

$$MCC_P = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \tag{9}$$

Similarly, MCC for negative crash instances (short for MCC_N) is defined as

$$MCC_N = \frac{TP' \times TN' - FP' \times FN'}{\sqrt{(TP'+FP')(TP'+FN')(TN'+FP')(TN'+FN')}} \tag{10}$$

According to the definition of the above four basic terms, we can find that $TP = TN'$, $TN = TP'$, $FP = FN'$, and $FN = FP'$. Thus, $MCC_P = MCC_N$. This means that MCC can comprehensively evaluate the overall performance of the method on both labels.

The value range of MCC is between −1 and 1. $MCC = 1$ means that the predicted results of the crash instances are completely consistent with the actual ones while $MCC = -1$ means that the predicted results are completely inconsistent with the actual ones. $MCC = 0$ means that the performance is equal to random prediction.

As a result, we total use three indicators. i.e., F_P, F_N, and MCC for performance evaluation which are widely used in previous studies (Gu et al., 2019; Song et al., 2018; Yao and Shepperd, 2020; Xu et al., 2019b; Li et al., 2020) for software engineering tasks.

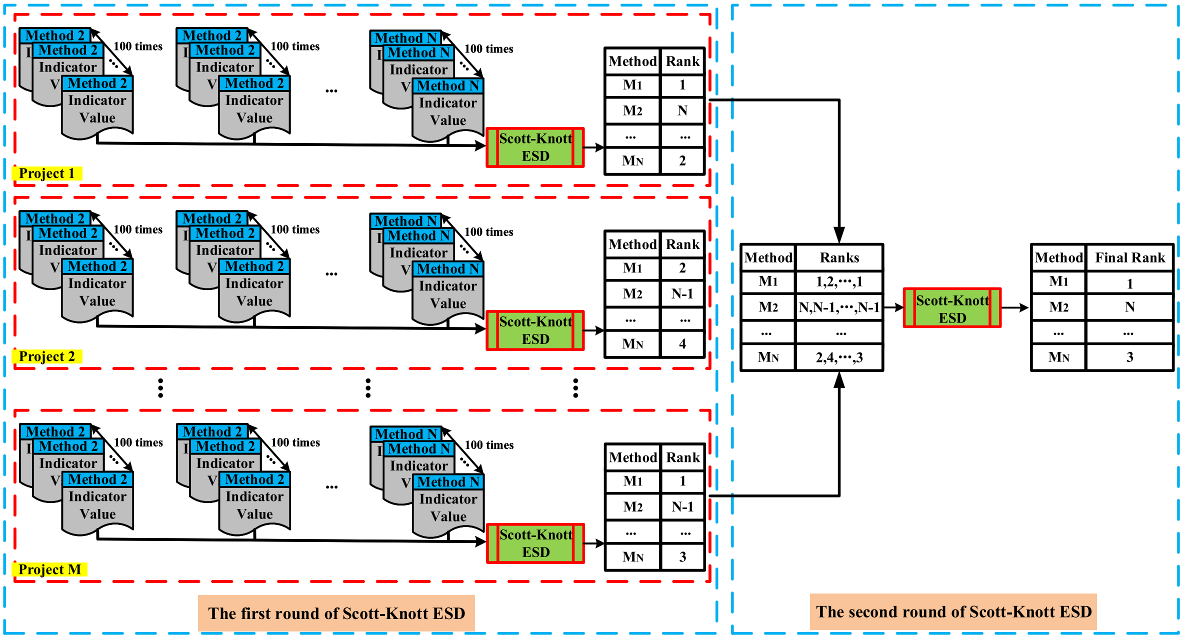


Fig. 3. The framework of Scott-Knott ESD.

4.3. Statistic test

To conduct a significant analysis for the performance differences between our IML method and the comparative methods, in this work, we use a state-of-the-art statistical test method called **S**cott-**K**nott **E**ffect **S**ize **D**ifference (**SKESD**) (Tantithamthavorn et al., 2017). Scott-Knott test employs hierarchical clustering algorithm to divide the methods with significant performance differences into different groups (Ghotra et al., 2015; Xu et al., 2016). SKESD is an improved version of Scott-Knott test via normality correction and effect size correction to relieve its two limitations. More specifically, SKESD amends the normality requirement of performance results by applying log-transforming (Menzies et al., 2006) to preprocess them, and merges the groups with negligible effect size into one group by using Cohen's delta (Cohen, 2013) to quantify the effect size. Compared with the commonly-used Friedman test with Nemenyi test, Scott-Knott test has the advantage to cluster the multiple methods into completely non-overlapping groups with significantly differences. In this work, we run the Scott-Knott ESD two rounds to analyze the results as shown in Fig. 3: in the first round, the inputs of SKESD are the 100 different indicator values of each method on each project and the corresponding outputs are the ranks of the methods on each project; in the second round, the inputs of SKESD are the outputs in the previous round and the corresponding outputs are the final ranks for the methods over all projects. The method with lower rank means that it achieves better performance.

4.4. Classification model

After obtaining the new feature representation of the crash data via our IML method, a classification model is needed to conduct the prediction task. As the Mahalanobis distance based imbalanced metric learning is designed to improve the accuracy of the kNN classifier rule in the latent space with the optimization goal to make the k nearest neighbor samples always belong to the same class, in this work, we choose kNN as our basic learner which is a simple instance-based classifier. To determine which label a crash instance in the test set belongs to, kNN first finds the k neighbors closest to the test instance from all the crash instances in the training set; then it counts the label most of the k neighbors belong to; finally, kNN assigns this label to the test instance as the prediction result.

4.5. Parameter configuration

For our IML method, there are three parameters that are needed to tune, i.e., the margin parameter m_d, regularization parameter λ, and the balanced parameter a. In this work, we empirically set m_d as {1, 10, 100, 1000, 10000}, λ as {0, 0.01, 0.1, 1, 10}, and $a \in [0, 1]$ with a step of 0.05 following the previous work (Gautheron et al., 2019). We use grid search to find the optimal combination of the three parameters on the training set and validation set, and then apply the optimal combination to the test set. For kNN classifier, we empirically set k as 3 to predict the residence of the crashing fault in the test set.

4.6. Data partition

To investigate the effectiveness of our IML method for predicting the crashing fault residence, we use a two-round stratified sampling method to divide the data into three parts as the training set, validation set, and test set. More specifically, in the first round, we randomly select half of the crash instances with label 'InTrace' and 'OutTrace' as the test set; in the second round, for the remaining half of crash instances, we also randomly select half of the crash instances with label 'InTrace' and 'OutTrace' as the training set, and the last remaining crash instances as the validation set. The stratified sampling ensures the ratios of two types of crash instances in the three sets are the same as that in the original data. To reduce the randomness bias of the data partition, we repeat the data partition process 100 times to neutralize fluctuations in performance and report the corresponding average value and standard deviation for each indicator. Note that the stratified sampling method and multiple runnings for reducing the variability of the random division are commonly used in previous studies in the field of software engineering (Wang et al., 2016; Ryu et al., 2016; Xu et al., 2019a).

Table 3
F_P values of our IML method and the sampling based methods on each project.

Project	Original	ROS	RUS	SMO	SMOTOM	SMOENN	BLSMO	SVMSMO	ADASYN	IML
Codec	0.510 ± (0.05)	0.563 ± (0.05)	0.516 ± (0.04)	0.557 ± (0.05)	0.551 ± (0.05)	0.501 ± (0.04)	0.552 ± (0.05)	0.552 ± (0.04)	0.562 ± (0.04)	**0.585** ± (0.06)
Collections	0.469 ± (0.04)	0.507 ± (0.03)	0.462 ± (0.03)	0.499 ± (0.03)	0.497 ± (0.03)	0.461 ± (0.03)	0.499 ± (0.03)	0.504 ± (0.03)	0.495 ± (0.03)	**0.599** ± (0.05)
IO	0.639 ± (0.05)	0.653 ± (0.04)	0.589 ± (0.05)	0.639 ± (0.04)	0.640 ± (0.04)	0.574 ± (0.04)	0.633 ± (0.05)	0.638 ± (0.05)	0.637 ± (0.04)	**0.697** ± (0.05)
Jsoup	0.314 ± (0.08)	**0.446** ± (0.05)	0.381 ± (0.04)	0.443 ± (0.05)	0.441 ± (0.04)	0.415 ± (0.04)	0.433 ± (0.05)	0.429 ± (0.05)	0.441 ± (0.04)	0.439 ± (0.07)
JSsqlParser	0.710 ± (0.05)	0.640 ± (0.06)	0.504 ± (0.10)	0.634 ± (0.07)	0.633 ± (0.06)	0.546 ± (0.07)	0.635 ± (0.07)	0.645 ± (0.07)	0.615 ± (0.07)	**0.724** ± (0.06)
Mango	0.490 ± (0.12)	0.501 ± (0.09)	0.302 ± (0.08)	0.483 ± (0.08)	0.483 ± (0.08)	0.390 ± (0.07)	0.504 ± (0.10)	0.511 ± (0.10)	0.478 ± (0.08)	**0.588** ± (0.10)
Ormlite-Core	0.596 ± (0.04)	0.630 ± (0.03)	0.574 ± (0.03)	0.625 ± (0.03)	0.626 ± (0.03)	0.573 ± (0.02)	0.622 ± (0.03)	0.626 ± (0.03)	0.625 ± (0.03)	**0.690** ± (0.04)
Average	0.533 ± (0.12)	0.563 ± (0.08)	0.475 ± (0.10)	0.554 ± (0.08)	0.553 ± (0.08)	0.494 ± (0.07)	0.554 ± (0.07)	0.558 ± (0.08)	0.550 ± (0.07)	**0.617** ± (0.09)

Table 4
F_N values of our IML method and the sampling based methods on each project.

Project	Original	ROS	RUS	SMO	SMOTOM	SMOENN	BLSMO	SVMSMO	ADASYN	IML
Codec	0.814 ± (0.02)	0.769 ± (0.03)	0.684 ± (0.05)	0.760 ± (0.03)	0.759 ± (0.03)	0.605 ± (0.06)	0.756 ± (0.03)	0.772 ± (0.03)	0.752 ± (0.03)	**0.827** ± (0.02)
Collections	0.883 ± (0.01)	0.831 ± (0.02)	0.758 ± (0.03)	0.816 ± (0.02)	0.815 ± (0.02)	0.731 ± (0.04)	0.822 ± (0.02)	0.834 ± (0.02)	0.806 ± (0.02)	**0.904** ± (0.01)
IO	0.906 ± (0.01)	0.881 ± (0.02)	0.841 ± (0.03)	0.871 ± (0.02)	0.871 ± (0.02)	0.817 ± (0.03)	0.871 ± (0.02)	0.878 ± (0.02)	0.863 ± (0.02)	**0.916** ± (0.02)
Jsoup	**0.873** ± (0.01)	0.824 ± (0.02)	0.731 ± (0.05)	0.794 ± (0.03)	0.791 ± (0.03)	0.682 ± (0.06)	0.797 ± (0.03)	0.817 ± (0.03)	0.788 ± (0.03)	0.861 ± (0.02)
JSsqlParser	**0.975** ± (0.00)	0.958 ± (0.01)	0.900 ± (0.05)	0.954 ± (0.02)	0.953 ± (0.01)	0.924 ± (0.03)	0.956 ± (0.02)	0.958 ± (0.01)	0.949 ± (0.02)	0.974 ± (0.01)
Mango	0.970 ± (0.01)	0.954 ± (0.01)	0.843 ± (0.07)	0.945 ± (0.01)	0.945 ± (0.02)	0.907 ± (0.03)	0.953 ± (0.01)	0.957 ± (0.01)	0.943 ± (0.01)	**0.971** ± (0.01)
Ormlite-Core	0.870 ± (0.01)	0.839 ± (0.02)	0.780 ± (0.03)	0.832 ± (0.02)	0.833 ± (0.02)	0.753 ± (0.03)	0.831 ± (0.02)	0.841 ± (0.02)	0.825 ± (0.02)	**0.896** ± (0.02)
Average	0.899 ± (0.05)	0.865 ± (0.07)	0.791 ± (0.07)	0.853 ± (0.07)	0.852 ± (0.07)	0.774 ± (0.11)	0.855 ± (0.07)	0.865 ± (0.07)	0.847 ± (0.07)	**0.907** ± (0.05)

Table 5
MCC values of our IML method and the sampling based methods on each project.

Project	Original	ROS	RUS	SMO	SMOTOM	SMOENN	BLSMO	SVMSMO	ADASYN	IML
Codec	0.329 ± (0.06)	0.354 ± (0.07)	0.261 ± (0.06)	0.342 ± (0.07)	0.333 ± (0.07)	0.225 ± (0.07)	0.334 ± (0.08)	0.34 ± (0.07)	0.347 ± (0.07)	**0.415** ± (0.07)
Collections	0.359 ± (0.04)	0.362 ± (0.04)	0.296 ± (0.04)	0.349 ± (0.04)	0.348 ± (0.04)	0.297 ± (0.04)	0.351 ± (0.04)	0.359 ± (0.04)	0.345 ± (0.04)	**0.507** ± (0.06)
IO	0.549 ± (0.06)	0.549 ± (0.06)	0.461 ± (0.07)	0.530 ± (0.06)	0.532 ± (0.06)	0.444 ± (0.06)	0.522 ± (0.07)	0.529 ± (0.06)	0.528 ± (0.06)	**0.615** ± (0.06)
Jsoup	0.214 ± (0.08)	0.286 ± (0.07)	0.177 ± (0.06)	0.275 ± (0.07)	0.272 ± (0.06)	0.227 ± (0.06)	0.263 ± (0.07)	0.263 ± (0.07)	0.272 ± (0.06)	**0.305** ± (0.09)
JSsqlParser	0.708 ± (0.05)	0.605 ± (0.07)	0.476 ± (0.10)	0.601 ± (0.07)	0.599 ± (0.07)	0.514 ± (0.07)	0.603 ± (0.08)	0.612 ± (0.07)	0.580 ± (0.07)	**0.710** ± (0.06)
Mango	0.499 ± (0.11)	0.464 ± (0.10)	0.268 ± (0.09)	0.448 ± (0.09)	0.448 ± (0.09)	0.363 ± (0.08)	0.469 ± (0.10)	0.476 ± (0.11)	0.444 ± (0.09)	**0.571** ± (0.10)
Ormlite-Core	0.468 ± (0.05)	0.492 ± (0.04)	0.409 ± (0.05)	0.485 ± (0.04)	0.487 ± (0.04)	0.414 ± (0.04)	0.481 ± (0.04)	0.486 ± (0.05)	0.486 ± (0.04)	**0.587** ± (0.05)
Average	0.447 ± (0.15)	0.445 ± (0.11)	0.335 ± (0.11)	0.433 ± (0.18)	0.431 ± (0.11)	0.355 ± (0.10)	0.432 ± (0.11)	0.438 ± (0.11)	0.429 ± (0.10)	**0.530** ± (0.13)

5. Evaluation results

5.1. RQ1: Does our IML method perform better than sampling based imbalanced learning methods?

Motivation: As mentioned above, IML method is a metric learning method to deal with imbalanced data. Since sampling methods are widely used for imbalanced learning, this question is proposed to investigate whether our IML method is more effective to achieve better prediction performance for crashing fault residence than sampling methods.

Methods: To answer this question, we choose nine comparative sampling methods. These methods are only applied to the training set. For a fair comparison, all these comparative methods and our IML method are conducted on the same training set and test set. These baseline methods are briefly described as follows:

- **ROS**: **R**andom **O**ver-**S**ampling randomly duplicates crash instances from the minority class (i.e., the crashing instances with label 'InTrace').
- **RUS**: **R**andom **U**nder-**S**ampling randomly removes crash instances from the majority class (i.e., the crashing instances with label 'OutTrace').
- **SMO**: The **S**ynthetic **M**inority **O**versampling technique synthesizes new crash instances for minority class based on interpolation.
- **SMOTOM**: This method uses **SMO** for over-sampling and **TOM**ek links for data cleaning.
- **SMOENN**: This method uses **SMO** for over-sampling and **E**dited **N**earest **N**eighbours for data cleaning.
- **BLSMO**: This method uses **B**order**L**ine **SMO** algorithm to generate new synthetic crash instances for minority class.
- **SVMSMO**: This method uses a **SVM** algorithm to detect the crash instances that are used to generate new synthetic crash instances by **SMO** for the minority class.
- **ADASYN**: This method performs over-sampling using **ADA**ptive **SYN**thetic sampling approach for the minority class.
- **Original**: Original method means that we do not use metric learning to preprocess the training set and test set. We treat Original as the most basic method for comparison.

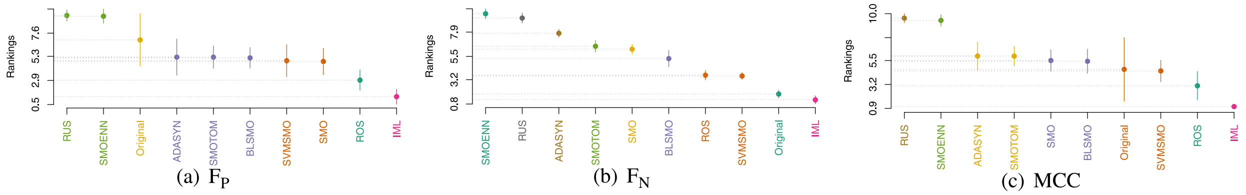
Results: Tables 3, 4, and 5 report the indicator values and corresponding standard deviations (the value in the bracket denotes the standard deviation) of our IML method and the nine sampling based imbalanced learning methods in terms of F_P, F_N, and MCC, respectively. From these tables, we have the following observations:

First, in terms of F_P, our IML method achieves better performance than the nine comparative sampling based imbalanced learning methods on 6 out of 7 projects and gets the best average F_P values across all projects. Compared with the average F_P values of the nine baseline methods, IML achieves improvements between 9.6% (for ROS) and 29.9% (for RUS) with an average improvement of 15.2%.

Second, in terms of F_N, our IML method achieves better performance than the nine comparative sampling based imbalanced learning methods on 5 out of 7 projects and gets the best average F_N values across all projects. Compared with the average F_N values of the nine baseline methods, IML achieves improvements between 0.9% (for Original) and 17.2% (for SMOENN) with an average improvement of 7.6%.

Third, in terms of MCC, our IML method achieves better performance than the nine comparative sampling based imbalanced learning methods on all 7 projects and gets the best average MCC values across all projects. Compared with the average MCC values of the nine baseline methods, IML achieves improvements between 18.6% (for Original) and 58.2% (for RUS) with an average improvement of 28.6%.

Fourth, Fig. 4 visualizes the corresponding statistical test results by SKESD for our IML method and the nine sampling based imbalanced learning methods. The methods with significant differences are drawn with different colors. The figure shows that our IML method ranks the first and has significant differences in performance toward all baseline methods in terms of all three indicators.



(a) F_P (b) F_N (c) MCC

Fig. 4. Results of Scott-Knott ESD test for our IML method and nine sampling methods.

Table 6
F_P values of our IML method and the ensemble based methods on each project.

Project	Bagging	BalBag	BalRF	RUSBoost	EasyEnse	IML
Codec	0.505 ± (0.06)	0.528 ± (0.04)	0.581 ± (0.04)	0.433 ± (0.10)	**0.613** ± (0.07)	0.585 ± (0.06)
Collections	0.465 ± (0.04)	0.490 ± (0.03)	0.581 ± (0.04)	0.516 ± (0.07)	0.584 ± (0.04)	**0.599** ± (0.05)
IO	0.639 ± (0.05)	0.616 ± (0.05)	0.663 ± (0.05)	0.645 ± (0.08)	0.658 ± (0.09)	**0.697** ± (0.05)
Jsoup	0.312 ± (0.08)	0.388 ± (0.04)	0.459 ± (0.04)	0.378 ± (0.09)	**0.487** ± (0.05)	0.439 ± (0.07)
JSsqlParser	0.709 ± (0.05)	0.505 ± (0.09)	0.470 ± (0.07)	0.534 ± (0.17)	0.476 ± (0.07)	**0.724** ± (0.06)
Mango	0.490 ± (0.12)	0.334 ± (0.06)	0.278 ± (0.05)	0.467 ± (0.15)	0.319 ± (0.09)	**0.588** ± (0.10)
Ormlite-Core	0.594 ± (0.04)	0.598 ± (0.03)	0.680 ± (0.02)	0.655 ± (0.07)	**0.767** ± (0.08)	0.690 ± (0.04)
Average	0.531 ± (0.12)	0.494 ± (0.10)	0.530 ± (0.13)	0.518 ± (0.10)	0.558 ± (0.13)	**0.617** ± (0.09)

Table 7
F_N values of our IML method and the ensemble based methods on each project.

Project	Bagging	BalBag	BalRF	RUSBoost	EasyEnse	IML
Codec	0.818 ± (0.02)	0.730 ± (0.04)	0.762 ± (0.03)	0.806 ± (0.03)	0.793 ± (0.04)	**0.827** ± (0.02)
Collections	0.886 ± (0.01)	0.795 ± (0.02)	0.845 ± (0.02)	0.888 ± (0.02)	0.838 ± (0.03)	**0.904** ± (0.01)
IO	0.907 ± (0.02)	0.864 ± (0.03)	0.878 ± (0.02)	0.912 ± (0.02)	0.872 ± (0.06)	**0.916** ± (0.02)
Jsoup	**0.874** ± (0.01)	0.763 ± (0.04)	0.757 ± (0.03)	0.868 ± (0.02)	0.778 ± (0.06)	0.861 ± (0.02)
JSsqlParser	**0.974** ± (0.01)	0.905 ± (0.04)	0.888 ± (0.04)	0.957 ± (0.01)	0.892 ± (0.04)	**0.974** ± (0.01)
Mango	0.968 ± (0.01)	0.879 ± (0.04)	0.809 ± (0.06)	0.959 ± (0.02)	0.844 ± (0.06)	**0.971** ± (0.01)
Ormlite-Core	0.876 ± (0.01)	0.804 ± (0.02)	0.858 ± (0.02)	0.894 ± (0.02)	**0.905** ± (0.04)	0.896 ± (0.02)
Average	0.900 ± (0.05)	0.820 ± (0.06)	0.828 ± (0.05)	0.898 ± (0.05)	0.846 ± (0.04)	**0.907** ± (0.05)

Table 8
MCC values of our IML method and the ensemble based methods on each project.

Project	Bagging	BalBag	BalRF	RUSBoost	EasyEnse	IML
Codec	0.332 ± (0.07)	0.292 ± (0.07)	0.378 ± (0.07)	0.260 ± (0.10)	**0.437** ± (0.10)	0.415 ± (0.07)
Collections	0.363 ± (0.05)	0.338 ± (0.04)	0.466 ± (0.05)	0.412 ± (0.08)	0.473 ± (0.06)	**0.507** ± (0.06)
IO	0.550 ± (0.06)	0.498 ± (0.07)	0.564 ± (0.06)	0.568 ± (0.09)	0.558 ± (0.12)	**0.615** ± (0.06)
Jsoup	0.215 ± (0.08)	0.193 ± (0.06)	0.297 ± (0.06)	0.263 ± (0.09)	**0.339** ± (0.08)	0.305 ± (0.09)
JSsqlParser	0.702 ± (0.06)	0.475 ± (0.09)	0.445 ± (0.06)	0.510 ± (0.16)	0.449 ± (0.07)	**0.710** ± (0.06)
Mango	0.483 ± (0.12)	0.304 ± (0.08)	0.256 ± (0.07)	0.441 ± (0.15)	0.298 ± (0.10)	**0.571** ± (0.10)
Ormlite-Core	0.475 ± (0.05)	0.445 ± (0.05)	0.565 ± (0.04)	0.556 ± (0.08)	**0.689** ± (0.10)	0.587 ± (0.05)
Average	0.446 ± (0.15)	0.364 ± (0.10)	0.424 ± (0.11)	0.430 ± (0.12)	0.463 ± (0.12)	**0.530** ± (0.13)

Answer: Our IML method is more effective to achieve significantly better prediction performance for crashing fault residence than sampling methods.

5.2. RQ2: Is our IML method superior to ensemble based imbalanced learning methods?

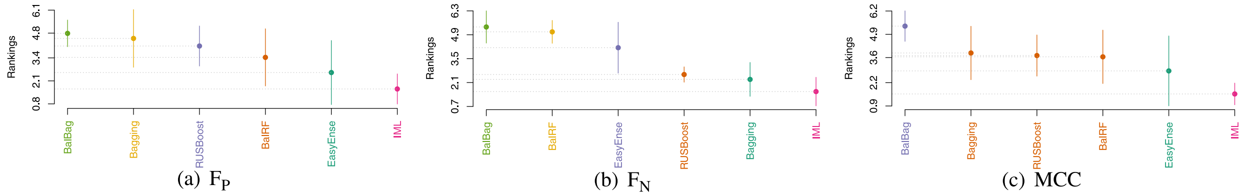
Motivation: Ensemble learning creates multiple base models in which each model is trained to solve the same problem, then improves overall performance on imbalanced data by integrating the outputs of these base models. This question is proposed to investigate whether our IML method performs better than ensemble methods.

Methods: To answer this question, we choose five comparative ensemble methods. For a fair comparison, all these comparative methods and our IML method are also conducted on the same training set and test set. The brief descriptions of the five baseline methods are presented as follows:

- **Bagging**: Bagging resamples crashing instances to generate multiple subsets and trains a model on each subset. The result is the average of the outputs of these models.
- **BalBag**: **Bal**anced**Bag**ging classifier is a variant Bagging which includes an additional step to balance the training set by using RUS.
- **BalRF**: **Bal**anced**R**andom**F**orest classifier uses RUS in each bootstrap sample for balance.
- **RUSBoost**: RUSBoost classifier uses RUS for the sample at each iteration of the boosting algorithm to alleviate the class imbalanced issue.
- **EasyEnse**: **EasyEnse**mble classifier creates an ensemble of different sets by iteratively selecting a random crashing instance subset.

Results: Tables 6, 7, and 8 report the indicator values and corresponding standard deviations of our IML method and the five ensemble based imbalanced learning methods in terms of F_P, F_N, and MCC, respectively. From these tables, we have the following observations:

First, in terms of F_P, our IML method achieves better performance than the five comparative ensemble based imbalanced learning methods on 4 out of 7 projects and gets the best average F_P values across all projects. Compared with the average F_P values



(a) F_P (b) F_N (c) MCC

Fig. 5. Results of Scott-Knott ESD test for our IML method and five ensemble methods.

Table 9
F_P values of our IML method with different classifiers on each project.

Project	IML_NB	IML_DT	IML_SVM	IML_LR	IML_RF	IML_NN	IML_3NN	IML_5NN	IML_7NN	IML_9NN
Codec	0.481 ± (0.09)	0.462 ± (0.08)	0.154 ± (0.19)	0.476 ± (0.07)	0.255 ± (0.12)	0.568 ± (0.07)	**0.585** ± (0.06)	0.553 ± (0.07)	0.529 ± (0.08)	0.496 ± (0.09)
Collections	0.325 ± (0.23)	0.512 ± (0.07)	0.219 ± (0.23)	0.508 ± (0.05)	0.228 ± (0.12)	0.597 ± (0.05)	**0.599** ± (0.05)	0.588 ± (0.05)	0.581 ± (0.06)	0.565 ± (0.07)
IO	0.459 ± (0.29)	0.591 ± (0.08)	0.450 ± (0.29)	0.667 ± (0.05)	0.571 ± (0.11)	0.679 ± (0.05)	**0.697** ± (0.05)	0.691 ± (0.05)	0.684 ± (0.06)	0.674 ± (0.06)
Jsoup	0.351 ± (0.11)	0.354 ± (0.08)	0.109 ± (0.15)	0.413 ± (0.07)	0.218 ± (0.12)	**0.442** ± (0.07)	0.439 ± (0.07)	0.430 ± (0.08)	0.421 ± (0.09)	0.409 ± (0.10)
Jsqlparser	0.388 ± (0.26)	0.570 ± (0.17)	0.481 ± (0.31)	0.687 ± (0.06)	0.617 ± (0.16)	0.696 ± (0.07)	**0.724** ± (0.06)	0.722 ± (0.06)	0.718 ± (0.07)	0.711 ± (0.09)
Mango	0.254 ± (0.22)	0.384 ± (0.15)	0.257 ± (0.27)	0.537 ± (0.11)	0.376 ± (0.17)	0.584 ± (0.10)	**0.588** ± (0.10)	0.572 ± (0.11)	0.550 ± (0.12)	0.511 ± (0.16)
Ormlite-Core	0.559 ± (0.10)	0.596 ± (0.07)	0.340 ± (0.29)	0.604 ± (0.04)	0.423 ± (0.13)	**0.702** ± (0.05)	0.690 ± (0.04)	0.682 ± (0.04)	0.669 ± (0.05)	0.656 ± (0.05)
Average	0.402 ± (0.10)	0.496 ± (0.09)	0.287 ± (0.13)	0.556 ± (0.09)	0.384 ± (0.15)	0.610 ± (0.09)	**0.617** ± (0.09)	0.605 ± (0.09)	0.593 ± (0.10)	0.575 ± (0.10)

Table 10
F_N values of our IML method with different classifiers on each project.

Project	IML_NB	IML_DT	IML_SVM	IML_LR	IML_RF	IML_NN	IML_3NN	IML_5NN	IML_7NN	IML_9NN
Codec	0.744 ± (0.09)	0.797 ± (0.03)	**0.830** ± (0.01)	0.809 ± (0.02)	0.825 ± (0.01)	0.825 ± (0.02)	0.827 ± (0.02)	0.814 ± (0.03)	0.806 ± (0.03)	0.798 ± (0.03)
Collections	0.816 ± (0.10)	0.880 ± (0.02)	0.898 ± (0.01)	0.893 ± (0.01)	0.895 ± (0.01)	0.898 ± (0.01)	**0.904** ± (0.01)	**0.904** ± (0.01)	**0.904** ± (0.01)	0.903 ± (0.01)
IO	0.890 ± (0.03)	0.889 ± (0.02)	0.907 ± (0.02)	0.910 ± (0.01)	0.910 ± (0.01)	0.911 ± (0.02)	**0.916** ± (0.02)	0.914 ± (0.02)	0.913 ± (0.02)	0.910 ± (0.02)
Jsoup	0.828 ± (0.07)	0.846 ± (0.03)	**0.886** ± (0.01)	0.867 ± (0.02)	0.883 ± (0.01)	0.862 ± (0.02)	0.861 ± (0.02)	0.856 ± (0.02)	0.853 ± (0.03)	0.851 ± (0.03)
Jsqlparser	0.938 ± (0.04)	0.943 ± (0.08)	0.967 ± (0.01)	0.966 ± (0.01)	0.968 ± (0.01)	0.970 ± (0.01)	**0.974** ± (0.01)	**0.974** ± (0.01)	0.973 ± (0.01)	0.972 ± (0.01)
Mango	0.943 ± (0.06)	0.935 ± (0.09)	0.967 ± (0.01)	0.966 ± (0.01)	0.968 ± (0.01)	0.970 ± (0.01)	**0.971** ± (0.01)	0.970 ± (0.01)	0.968 ± (0.01)	0.965 ± (0.01)
Ormlite-Core	0.788 ± (0.07)	0.867 ± (0.03)	0.880 ± (0.02)	0.879 ± (0.01)	0.882 ± (0.01)	**0.898** ± (0.02)	0.896 ± (0.02)	0.893 ± (0.02)	0.891 ± (0.02)	0.889 ± (0.02)
Average	0.850 ± (0.07)	0.880 ± (0.05)	0.905 ± (0.05)	0.899 ± (0.05)	0.904 ± (0.05)	0.905 ± (0.05)	**0.907** ± (0.05)	0.904 ± (0.05)	0.901 ± (0.06)	0.898 ± (0.06)

Table 11
MCC values of our IML method with different classifiers on each project.

Project	IML_NB	IML_DT	IML_SVM	IML_LR	IML_RF	IML_NN	IML_3NN	IML_5NN	IML_7NN	IML_9NN
Codec	0.250 ± (0.09)	0.268 ± (0.09)	0.115 ± (0.14)	0.293 ± (0.08)	0.178 ± (0.10)	0.397 ± (0.08)	**0.415** ± (0.07)	0.372 ± (0.08)	0.343 ± (0.09)	0.307 ± (0.09)
Collections	0.221 ± (0.17)	0.394 ± (0.09)	0.222 ± (0.21)	0.413 ± (0.06)	0.254 ± (0.10)	0.497 ± (0.06)	**0.507** ± (0.06)	0.500 ± (0.06)	0.495 ± (0.06)	0.480 ± (0.06)
IO	0.392 ± (0.26)	0.485 ± (0.10)	0.421 ± (0.26)	0.580 ± (0.06)	0.514 ± (0.10)	0.593 ± (0.07)	**0.615** ± (0.06)	0.609 ± (0.06)	0.601 ± (0.07)	0.590 ± (0.07)
Jsoup	0.213 ± (0.09)	0.210 ± (0.09)	0.088 ± (0.12)	0.287 ± (0.08)	0.175 ± (0.10)	**0.307** ± (0.08)	0.305 ± (0.09)	0.296 ± (0.08)	0.289 ± (0.09)	0.280 ± (0.09)
Jsqlparser	0.378 ± (0.26)	0.535 ± (0.18)	0.490 ± (0.31)	0.659 ± (0.07)	0.618 ± (0.15)	0.671 ± (0.08)	**0.710** ± (0.06)	0.708 ± (0.07)	0.703 ± (0.07)	0.696 ± (0.09)
Mango	0.268 ± (0.24)	0.353 ± (0.16)	0.280 ± (0.27)	0.513 ± (0.11)	0.407 ± (0.16)	0.564 ± (0.11)	**0.571** ± (0.10)	0.557 ± (0.11)	0.536 ± (0.12)	0.502 ± (0.14)
Ormlite-Core	0.396 ± (0.10)	0.467 ± (0.09)	0.310 ± (0.25)	0.489 ± (0.05)	0.397 ± (0.09)	0.601 ± (0.06)	**0.587** ± (0.05)	0.577 ± (0.06)	0.564 ± (0.06)	0.549 ± (0.06)
Average	0.303 ± (0.08)	0.387 ± (0.11)	0.275 ± (0.14)	0.462 ± (0.13)	0.363 ± (0.16)	0.519 ± (0.12)	**0.530** ± (0.13)	0.517 ± (0.13)	0.504 ± (0.13)	0.486 ± (0.14)

of the five baseline methods, our IML method achieves improvements between 10.6% (for EasyEnse) and 24.9% (for BalBag) with an average improvement of 17.4%.

Second, in terms of F_N, our IML method achieves better performance than the five comparative ensemble based imbalanced learning methods on 5 out of 7 projects and gets the best average F_N values across all projects. Compared with the average F_N values of the five baseline methods, our IML method achieves improvements between 0.8% (for Bagging) and 10.6% (for BalBag) with an average improvement of 5.8%.

Third, in terms of MCC, our IML method achieves better performance than the five comparative ensemble based imbalanced learning methods on 4 out of 7 projects and gets the best average MCC values across all projects. Compared with the average MCC values of the five baseline methods, our IML method achieves improvements between 14.5% (for EasyEnse) and 45.6% (for BalBag) with an average improvement of 25.4%.

Fourth, Fig. 5 visualizes the corresponding statistical test results by SKESD for our IML method and the five ensemble based imbalanced learning methods. The figure shows that our IML method always ranks the first and has significant performance differences toward all five baseline methods in terms of all three indicators.

Answer: Our IML method performs significantly better than ensemble methods for crashing fault residence prediction task.

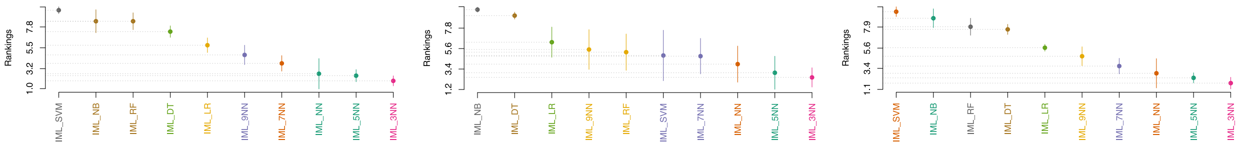
5.3. RQ3: How does our IML work with different classifiers?

Motivation: As the metric learning method is customized for *k*NN model for performance improvement, in this work, we use the *k*NN classifier as the basic classification model. This question is proposed to investigate how different classifiers impact the performance of our IML method. In addition, as we set our basic classifier with parameter $k = 3$, this research question is also designed to investigate how the *k*NN classifier with different parameters work with our IML method.

Methods: To answer this research question, we select additional five typical classifiers, including Naive Bayes (NB), Decision Tree (DT), Support Vector Machine (SVM), Logistic Regression (LR), and Random Forest (RF) for comparison. Besides, to discuss the performance of different *k*NN classifiers with distinct parameter settings, we empirically select additional four settings, i.e., $k = 1, 5, 7, 9$ for comparison.

Results: Tables 9, 10, and 11 report the indicator values and corresponding standard deviations of our IML method with different classifiers in terms of F_P, F_N, and MCC, respectively. From these tables, we can observe that:

First, in terms of F_P, our IML method using *k*NN classifier with $k = 3$ achieves better performance than other nine comparative classifiers on 5 out of 7 projects and gets the best average F_P values across all projects. Compared with the average F_P values of the nine baseline methods, our IML method using *k*NN classifier with $k = 3$ achieves improvements between 1.1% (for using *k*NN



(a) F_P (b) F_N (c) MCC

Fig. 6. Results of Scott-Knott ESD test for our IML method with different classifiers.

classifier with $k = 1$) and 115.0% (for using SVM) with an average improvement of 31.0%.

Second, in terms of F_N, our IML method using kNN classifier with $k = 3$ achieves better performance than other nine comparative classifiers on 4 out of 7 projects and gets the best average F_N values across all projects. Compared with the average F_N values of the nine baseline methods, our IML method using kNN classifier with $k = 3$ achieves improvements between 0.2% (for using SVM) and 6.7% (for using NB) with an average improvement of 1.5%.

Third, in terms of MCC, our IML method using kNN classifier with $k = 3$ achieves better performance than other nine comparative classifiers on 6 out of 7 projects and gets the best average MCC values across all projects. Compared with the average MCC values of the nine baseline methods, our IML method using kNN classifier with $k = 3$ achieves improvements between 2.1% (for using kNN with $k = 1$) and 92.7% (for using SVM) with an average improvement of 31.6%.

Fourth, Fig. 6 visualizes the corresponding statistical test results by SKESD for our IML method with different classifiers. The figure shows that our IML method always ranks the first and has significant performance differences toward all other nine comparative classifiers in terms of all three indicators.

$\boxed{\textbf{Answer}}$: kNN classifier with $k = 3$ is a good choice for our IML method to achieve the better average performance for predicting crashing fault residence.

6. Threats to validity

6.1. External validity

The generalization of the experimental results threatens the external validity of our work. As the projects in our benchmark dataset are all developed in Java language, future replication experiments are needed to verify whether our conclusion still holds when applying our IML method to projects that developed in other languages. Although the crash instances in the dataset are generated by caused by artificial mutation, previous studies showed that the faults by mutation can be used as a substitute for the real-world faults in empirical assessment (Andrews et al., 2005; Namin and Kakarla, 2011; Just et al., 2014). Thus, the results derived from the artificial simulated crashing faults can basically generalize to the real ones.

6.2. Internal validity

The possible faults in the implementation of methods threaten the internal validity of our work. To minimize this kind of validity, we implement our IML method by modifying the source code shared by the original authors to adapt to our crashing fault residence prediction task. Besides, we make full use of the third-party libraries in the scikit-learn to implement some off-the-shelf imbalanced learning methods as our comparative methods aiming to avoid the potential mistakes in the implementation process by ourselves.

6.3. Construct validity

The suitability of the used evaluation indicators and statistic test methods threatens the construct validity of our work. We use F-measure and MCC to evaluate the effectiveness of our IML method for the crashing fault residence prediction task. F-measure and MCC are comprehensive performance indicators which are more suitable than the ones like precision and recall. In addition, we use the state-of-the-art statistic test called SKESD for significance analysis. This statistic test generates non-overlapping groups with significant differences for multiple methods, which is better than the traditional statistic test like Friedman test with Nemenyi test.

7. Conclusion

In this work, we propose a crashing fault residence prediction model based on a novel metric learning method called IML. This method learns high-quality feature representation with a Mahalanobis distance based metric learning method and tackles the class imbalanced issue by assigning different weights to the loss functions of crash instance pairs according to their labels. The experiments on seven open source software projects show that our IML method achieves significantly better prediction performance for crashing fault residence task when compared with some sampling and ensemble based imbalanced learning methods.

In the future, we will adapt our IML method to cross project prediction scenario. In addition, we plan to investigate the effectiveness of our method on more projects and real crashes.

CRediT authorship contribution statement

Zhou Xu: Writing - original draft, Methodology, Data curation. **Kunsong Zhao:** Methodology, Software, Visualization. **Meng Yan:** Supervision. **Peipei Yuan:** Conceptualization, Writing - review & editing. **Ling Xu:** Formal analysis. **Yan Lei:** Writing - review & editing. **Xiaohong Zhang:** Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported in part by the National Key Research and Development Project (No. 2018YFB2101200), China Postdoctoral Science Foundation (No. 2020M673137), the Fundamental Research Funds for the Central Universities (No. 2020CDJQY-A021), and the Chongqing Technology Innovation and Application Development Project (No. cstc2019jscx-dxwtBX0012).

References

Andrews, J.H., Briand, L.C., Labiche, Y., 2005. Is mutation an appropriate tool for testing experiments? In: Proceedings of the 27th International Conference on Software Engineering (ICSE). pp. 402–411.

Cohen, J., 2013. Statistical Power Analysis for the Behavioral Sciences. Academic press.

Dhaliwal, T., Khomh, F., Zou, Y., 2011. Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox. In: Proceedings of the IEEE 27th International Conference on Software Maintenance (ICSM). pp. 333–342.

Gautheron, L., Habrard, A., Morvant, E., Sebban, M., 2019. Metric learning from imbalanced data. In: Proceedings of the 31st IEEE International Conference on Tools with Artificial Intelligence. pp. 923–930.

Ghotra, B., McIntosh, S., Hassan, A.E., 2015. Revisiting the impact of classification techniques on the performance of defect prediction models. In: Proceedings of the 37th International Conference on Software Engineering (ICSE). IEEE Press, pp. 789–800.

Gong, L., Zhang, H., Seo, H., Kim, S., 2014. Locating crashing faults based on crash stack traces. arXiv preprint arXiv:1404.4100.

Gu, Y., Xuan, J., Zhang, H., Zhang, L., Fan, Q., Xie, X., Qian, T., 2019. Does the fault reside in a stack trace? Assisting crash localization by predicting crashing fault residence. J. Syst. Softw. (JSS) 148, 88–104.

Indi, T.S., Yalagi, P.S., Nirgude, M.A., 2016. Use of java exception stack trace to improve bug fixing skills of intermediate java learners. In: International Conference on Learning and Teaching in Computing and Engineering. pp. 194–198.

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: Proceedings of the 12th International Conference on Quality Software (ICQS). pp. 9–12.

Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G., 2014. Are mutants a valid substitute for real faults in software testing? In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE). pp. 654–665.

Li, N., Shepperd, M., Guo, Y., 2020. A systematic review of unsupervised learning techniques for software defect prediction. Inf. Softw. Technol. (IST) 106287.

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: Proceedings of the 23rd International Conference on Engineering of Complex Computer Systems. IEEE, pp. 104–114.

Lu, J., Zhou, X., Tan, Y.P., Shang, Y., Jie, Z., 2014. Neighborhood repulsed metric learning for kinship verification. IEEE Trans. Pattern Anal. Mach. Intell. (TPAMI) 36 (2), 331–345.

Maesschalck, R.D., Jouan-Rimbaud, D., Massart, D.L., 2000. The mahalanobis distance. Chemometr. Intell. Lab. Syst. 50 (1), 1–18.

Mathur, A.P., 2007. Foundations of Software Testing.

Menzies, T., Greenwald, J., Frank, A., 2006. Data mining static code attributes to learn defect predictors. IEEE Trans. Softw. Eng. (TSE) 33 (1), 2–13.

Moon, S., Kim, Y., Kim, M., Yoo, S., 2014. Ask the mutants: Mutating faulty programs for fault localization. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation. IEEE, pp. 153–162.

Moreno, L., Treadway, J.J., Marcus, A., Shen, W., 2014. On the use of stack traces to improve text retrieval-based bug localization. In: Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME). pp. 151–160.

Namin, A.S., Kakarla, S., 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). pp. 342–352.

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. Exp. 46 (9), 1155–1179.

Ryu, D., Choi, O., Baik, J., 2016. Value-cognitive boosting with a support vector machine for cross-project defect prediction. Empir. Softw. Eng. (EMSE) 21 (1), 43–71.

Schröter, A., Bettenburg, N., Premraj, R., 2010. Do stack traces help developers fix bugs? In: Proceedings of the 7th International Conference on Mining Software Repositories (MSR). IEEE, pp. 118–121.

Soltani, M., Derakhshanfar, P., Devroey, X., Deursen, A.v., 2020. A benchmark-based evaluation of search-based crash reproduction. Empir. Softw. Eng. (EMSE) 25, 96–138.

Song, Q., Guo, Y., Shepperd, M., 2018. A comprehensive investigation of the role of imbalanced learning for software defect prediction. IEEE Trans. Softw. Eng. (TSE) 45 (12), 1253–1269.

Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K., 2017. An empirical comparison of model validation techniques for defect prediction models. IEEE Trans. Softw. Eng. (TSE) 43 (1), 1–18.

Wang, S., Khomh, F., Zou, Y., 2013. Improving bug localization using correlations in crash reports. In: Proceedings of the 10th Working Conference on Mining Software Repositories (MSR). pp. 247–256.

Wang, T., Zhang, Z., Jing, X., Zhang, L., 2016. Multiple kernel ensemble learning for software defect prediction. Autom. Softw. Eng. (ASE) 23 (4), 569–590.

Weinberger, K.Q., Saul, L.K., 2009. Distance metric learning for large margin nearest neighbor classification. J. Mach. Learn. Res. 10, 207–244.

Wong, C.-P., Xiong, Y., Zhang, H., Hao, D., Zhang, L., Mei, H., 2014. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME). pp. 181–190.

Wu, R., Wen, M., Cheung, S.C., Zhang, H., 2018. Changelocator: locate crash-inducing changes based on crash reports. Empir. Softw. Eng. (EMSE) 23 (5), 2866–2900.

Wu, R., Zhang, H., Cheung, S.-C., Kim, S., 2014. CrashLocator: Locating crashing faults based on crash stacks. In: Proceedings of the 23rd International Symposium on Software Testing and Analysis (ISSTA). pp. 204–214.

Xu, Z., Li, S., Xu, J., Liu, J., Luo, X., Zhang, Y., Zhang, T., Keung, J., Tang, Y., 2019a. LDFR: Learning deep feature representation for software defect prediction. J. Syst. Softw. (JSS) 158, 110402.

Xu, Z., Liu, J., Yang, Z., An, G., Jia, X., 2016. The impact of feature selection on defect prediction performance: An empirical comparison. In: Proceedings of the 27th International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp. 309–320.

Xu, Z., Zhang, T., Zhang, Y., Tang, Y., Liu, J., Luo, X., Keung, J., Cui, X., 2019b. Identifying crashing fault residence based on cross project model. In: Proceedings of the 30th International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp. 183–194.

Xu, Z., Zhang, T., Zhang, Y., Tang, Y., Luo, X., Keung, J., Cui, X., 2019c. Identifying crashing fault residence based on cross project model. In: Proceedings of the 30th International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp. 183–194.

Xuan, J., Xie, X., Monperrus, M., 2015. Crash reproduction via test case mutation: let existing test cases help. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (FSE). pp. 910–913.

Yao, J., Shepperd, M., 2020. Assessing software defection prediction performance: why using the Matthews correlation coefficient matters. In: Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering (EASE). pp. 120–129.

Zadeh, P.H., Hosseini, R., Sra, S., 2016. Geometric mean metric learning. In: International Conference on Machine Learning.

Zhang, L., Zhang, L., Khurshid, S., 2013. Injecting mechanical faults to localize developer faults for evolving software. ACM SIGPLAN Not. 48 (10), 765–784.