Empirical Software Engineering (2023) 28:49
https://doi.org/10.1007/s10664-023-10294-y

The impact of class imbalance techniques on crashing fault residence prediction models

Kunsong Zhao[1,2] · Zhou Xu[2,3] · Meng Yan[2,3] · Tao Zhang[4] · Lei Xue[5] · Ming Fan[6] · Jacky Keung[7]

Accepted: 16 January 2023

Abstract
Software crashes occur when the software program is executed wrongly or interrupted compulsively, which negatively impacts on user experience. Since the stack traces offer the exception-related information about software crashes, researchers used features collected from the stack trace to automatically identify whether the fault residence where the crash occurred is in the stack trace, aiming at accelerating the process of crash localization. A recent work conducted the first large-scale empirical study, which investigated the impact of feature selection methods on the performance of classification models for this task. However, the crash data have the intrinsic class imbalance characteristic, i.e., there exists a large difference between the number of crash instances inside and outside the stack trace, which is ignored by the previous work. To fill this gap, in this work, we conduct a large-scale empirical study to explore how different imbalanced learning techniques impact the performance of crashing fault residence prediction models on a benchmark dataset comprising seven software projects with four evaluation indicators. Our experimental results demonstrate that two imbalanced variants of the bagging classifier perform better than other compared techniques in both the normal and cross-project settings, and can constantly generate excellent prediction performance even though the imbalance level changes.

Keywords Crash localization · Stack trace · Imbalanced learning · Empirical study

1 Introduction

The flourishment of the Internet has accelerated the process of human development. In particular, the emergence of software products pushes this process into a new era. As the irreplaceable part of people's everyday life, software products ineluctably bring faults

Communicated by: Mehdi Mirakhorli

✉ Zhou Xu
zhouxullx@cqu.edu.cn
✉ Meng Yan
mengy@cqu.edu.cn

Extended author information available on the last page of the article.

Published online: 22 February 2023

during the development with many uncertain factors (Mathur 2013). This kind of awful conditions can lead to negative impacts on user experience. When the faults occur, the software program will be executed incorrectly and even forced to break off, which is also called software crashes. Once software crashes, the built-in monitoring system automatically records related information, such as the stack trace, which can be used to analyze where the crashes come from Dhaliwal et al. (2011). Ascertaining the residence of faults giving rise to the crash (short for crashing faults) can accelerate programmers to pay attention to the corresponding source code and fix them emphatically, which becomes an important activity for software quality assurance (Xu et al. 2020).

To locate the residence of crashing faults more quickly during the development and maintenance process, researchers took advantages of the stack trace information. These information have been shown to be favourable for programmers when debugging (Schroter et al. 2010). The stack trace offers the exception-related information when the software crash happens, including the exception type, the root where this exception thrown, and the trajectory of function invocation. Each stack trace comprises multiple frames in which the exception type is recorded in the first frame (i.e., top frame) and the function calls are traced until the last frame (i.e., bottom frame). As a previous empirical study indicated that most of the crashing faults reside in the stack traces (Li et al. 2018), the objective of locating the residence of crashing faults lies in predicting whether the crashing faults reside in the stack trace or not. This can be treated as a traditional binary classification task that is typically modeled by the classification techniques in machine learning. If the crashing fault resides in the stack trace, practitioners just need to concentrate on the corresponding code snippets that the stack trace points, which can greatly economize on programmers' efforts. Otherwise, programmers need to spend a mass of efforts on inspecting the function invocation tracks involving a crowd of lines of code, which is inefficient within the limited resource during the maintenance period.

A recent study (Gu et al. 2019) released a benchmark dataset to assist the crashing faults residence prediction task. Specifically, they simulated the real-world crashes by virtue of the program mutation tool and characterized each crash instance with 89 features extracted from the stack trace and faulty code. Also, they labeled each crash instance based on the following matching rules: if the crashing fault residence exactly matches with the information piece (i.e., the class name, the method name, and the code line number) in one of the frame in the stack trace, this crash instance is deemed to be inside the stack trace; otherwise, outside the stack trace (Gu et al. 2019). In addition, they proposed a simple model to analyze the collected dataset. Following their work, researchers developed many approaches to improve the performance for this task from different perspectives, such as feature selection (Zhao et al. 2021a), metric learning (Xu et al. 2020), and even the extension to the cross-project scenario (Xu et al. 2019b). Zhao et al. (2021b) comprehensively analyze the impact of 24 feature selection methods on the performance of 21 classification models for the crashing faults residence prediction task. This work mainly focused on exploring how the feature selection methods impact the performance of crashing faults residence prediction models from two aspects, i.e., classification model level and project level. However, the crash data are always inherently imbalanced, i.e., the crash instances in the stack trace (i.e., minority instances) are fewer than those outside the stack trace (i.e., majority instances) (Gu et al. 2019), which is ignored by their work. The class imbalance issues are existing in many software engineering tasks, such as software defect prediction (Xu et al. 2019a; Kamei et al. 2016; Li et al. 2020) and technical debt detection (Ren et al. 2019; Wang et al. 2020), which

negatively impact the performance of models. Thus, how to address the class imbalance issues is always a hot research topic (Tantithamthavorn et al. 2018; Song et al. 2018).

1.1 Motivation

A common phenomenon encountered in real world is that crash residence data comprises only a few positive instances (i.e., inside the stack trace) and a large number of negative ones (i.e., outside the stack trace), that is the class imbalance issue. For example, the dataset used in this work holds different ratios between positive and negative instances (as shown in Section 3.1). When building predictive models from imbalanced data, classical techniques assume the balanced class distributions or equal misclassification costs (He and Garcia 2009; Song et al. 2018) and thus obtain poor performance in identifying rare classes. Although plenty of imbalanced learning techniques have been developed to enhance the classification performance, the impact of different imbalance techniques on crashing fault residence prediction has never been investigated.

Thus, we believe the uncertainty of employing imbalanced learning techniques for crashing fault residence prediction task lie in the following motivations. First, the choice of various imbalanced learning techniques for this task is not well explored and understood. Second, different imbalance levels of crash data and their influences of distinct predictive performance are undiscovered. Third, there is a lack of a unified experimental process for comparing the predictive results of these techniques strictly. Fourth, there exist biases among the used performance evaluation indicators.

As a result, we systematically explore the research questions below:

- RQ1: What are the impacts of different sampling-based imbalanced learning techniques on the performance of crashing fault residence prediction models?
- RQ2: How different ensemble-based imbalanced learning techniques impact the performance of crashing fault residence prediction models?
- RQ3: How different cost-sensitive-based imbalanced learning techniques impact the performance of crashing fault residence prediction models?
- RQ4: How different imbalanced learning techniques impact the performance of crashing fault residence prediction models in cross-project scenario?
- RQ5: How does imbalance level of the crash data impact the performance of imbalanced learning techniques?

1.2 Our Work

To fill this gap, we conduct a large-scale empirical study to investigate how different imbalanced learning techniques impact the performance of crashing fault residence prediction models. More specifically, we explored 19 sampling-based, 15 ensemble-based, and 8 cost-sensitive-based imbalanced learning techniques. The studied sampling-based techniques belong to four groups: under-sampling based, over-sampling based, combinative based techniques, and the method without any imbalanced treatment. The investigated ensemble-based techniques derived from four groups: commonly-used, under-sampling based, over-sampling based, and compatible techniques.

The experiments are conducted on 7 open-source Java projects released by a previous work (Gu et al. 2019). To evaluate the performance of these imbalanced learning techniques

on the classification models for crashing fault residence prediction task, we employ four evaluation indicators, including F-measure for crash instances in the stack trace (i.e., F_{IT}), F-measure for crash instances outside the stack trace (i.e., F_{OT}), **M**atthews **C**orrelation **C**oefficient (**MCC**), and **A**rea **U**nder the receiver operating characteristic **C**urve (**AUC**). We employ the advanced **S**cott-**K**nott **E**ffect **S**ize **D**ifference (**SKESD**) test proposed by Tantithamthavorn et al. (2016) to assess the experimental results. The results reveal that, overall, the **DT** (**D**ecision **T**ree) classifier without any imbalanced process obtains better prediction performance among all studied sampling-based imbalanced learning techniques in terms of four performace indicators. Two imbalanced variants of bagging classifier, including **OBag** (**O**ver-sampling with **Bag**ging) and **UBag** (**U**nder-sampling with **Bag**ging), achieve better prediction performance among all the investigated ensemble-based imbalanced learning techniques, whereas two techniques **AsymB** (**Asym**metric adaptive **B**oost) and **CSDT** (**C**ost-**S**ensitive **D**ecision **T**ree) perform better than other cost-sensitive-based techniques. In addition, experimental results also reveal that UBag and OBag are more suitable for crashing fault residence prediction task in the context of the cross-project scenario. Further investigations demonstrate the imbalance level impacts on the predictive performance of imbalanced learning techniques, in which the UBag technique always obtains better performance even though the imbalance level changes.

1.3 Contributions

The contributions of this paper are summarized as follows:

- To the best of our knowledge, we are the first to empirically analyze the impact of imbalanced learning techniques on the performance of crashing fault residence prediction models on such a large scale, because we evaluate the performance of 42 techniques on a benchmark consisting of 7 projects in terms of four indicators and yield $(19 \times 7 + 15 + 8) \times 7 \times 4 = 4{,}368$ results.
- We comprehensively analyze these prediction results in different perspectives by means of heatmaps and boxplots, and statistically assess the experimental results using the ranking values derived by the SKESD test.
- We generate some practical findings for developers and researchers. Our findings suggest that practitioners can choose the imbalanced variants of bagging technique (including OBag and UBag) to build their predictive models when the performance is the major concern because these two techniques show excellent results and are more stable even though the imbalance level changes.
- Our Python code and related materials are shared in our online repository at https://github.com/sepine/EMSE-2022 to facilitate replication or extension of the work by the software engineering community.

1.4 Paper Organization

The remainder of this paper is organized as follows. Section 2 provides the preliminaries of the studied imbalanced learning techniques. Sections 3 and 4 describe the experimental setup and analyze the experimental results, respectively. We discuss the implications, error analysis, and threats to validity in Section 5 followed by the related work in Section 6. Finally, we draw the conclusion in Section 7.

2 Preliminaries

In this section, we briefly introduce the studied imbalanced learning techniques, including 19 sampling-based imbalanced learning techniques (containing one method that only uses the original crash instances, short for **NONE**), 15 ensemble-based imbalanced learning techniques, and 8 cost-sensitive-based techniques. Accordingly, we have totally 42 techniques used for our study. Table 1 depicts the overview of these techniques and the corresponding abbreviations.

2.1 Sampling-based Techniques

Sampling-based imbalanced learning techniques aim to alter the number of majority (i.e., negative) or minority (i.e., positive) instances to deal with the imbalance issue by means of removing a certain number of majority instances to obtain the balanced dataset (i.e., under-sampling), adding a certain number of minority instances to rebalance the dataset (i.e., over-sampling), or the combinations of these two techniques.

2.1.1 Under-sampling Based Techniques

- **C**luster **C**entroid**s** (**CCs**) (Leisch 2006) first uses the k-means algorithm to group all the negative instances into n clusters and then negative instances in each cluster are replaced by the correponding cluster centroid. n is affected by the number of positive instances.
- **R**andom **U**nder-**S**ampling (**RUS**) (He and Garcia 2009) randomly removes some negative instances to cater to the data balance.
- **I**nstance **H**ardness **T**hreshold (**IHT**) (Smith et al. 2014) first employs one classifier on the instances and then removes the instances whose classification probibilities are less than a specific threshold.
- **N**ear**M**iss (**NM**) (Mani and Zhang 2003) selects the negative instances by the heuristic rule, that is, choosing n negative instances from the nearest neighbors of positive instances with the shortest average distance.
- **T**ome**kL**ink**s** (**TkLs**) (Tomek et al. 1976b) removes the negative instances according to the TomekLinks. Specially, assume i_1 and i_2 are the positive and negative instances individually, for a given distance function $dis(,)$ which means the nearest neighbor relationship between two instances, we have the following relations: $dis(i_1, i_3) < dis(i_1, i_2)$ or $dis(i_2, i_3) < dis(i_1, i_2)$, where i_3 is a specific instance. If there does not exist such an instance i_3, then we remove the negative instance i_2.
- **E**dited **N**earest **N**eighbors (**ENN**) (Wilson 1972) adjusts the instances by using the nearest neighbor algorithm. ENN cleans the negative instances which are close to the decision boundary.
- **Rep**eated **E**dited **N**earest **N**eighbors (**RepENN**) (Tomek et al. 1976a) is an extended version of ENN, which repeats the ENN methods several times.
- **A**ll**KNN** (**AKNN**) (Tomek et al. 1976a) is also an improved version of ENN, which repeats the original ENN techniques several times. The difference between RepENN and AKNN is that the number of the nearest neighbor keeps increasing in each iteration when using the AKNN methods.
- **O**ne **S**ided **S**election (**OSS**) (Kubat et al. 1997) removes the negative instances based on the one-sided selection by the TomekLinks measurement (Tomek et al. 1976b).

Table 1 The overview of 42 studied techniques with their abbreviations

Family	Techniques		Abbreviations
Sampling-based Techniques	Under-sampling Techniques	Cluster Centroids	CCs
		Random Under-Sampling	RUS
		Instance Hardness Threshold	IHT
		NearMiss	NM
		TomekLinks	TkLs
		Edited Nearest Neighbors	ENN
		Repeated Edited Nearest Neighbors	RepENN
		AllKNN	AKNN
		One Sided Selection	OSS
		Condensed Nearest Neighbor	CNN
		Neighborhood Cleaning Rule	NCR
	Over-sampling Techniques	Adaptive Synthetic Sampling	AdaS
		Random Over-Sampling	ROS
		The Symthetic Minority Over-sampling	SMO
		Borderline SMO	BSMO
		Support Vector Machine SMO	SSMO
	Combinative techniques	SMO with ENN	SMOENN
		SMO with TomekLinks	SMOTk
	Other	NONE	NONE
Ensemble-based Techniques	Commonly-used Techniques	Bagging	Bag
		Balanced Bagging	BalBag
		Adaptive Boost	AdaB
	Under-sampling based Techniques	Self-Paced Ensemble	SPE
		Balance Cascade	BalCas
		Balanced Random Forest	BalRF
		Easy Ensemble	EasyE
		Random Under-Sampling with Boost	RUSB
		Under Bagging	UBag
	Over-sampling based Techniques	Over-Sampling with Adaptive Boost	OverB
		SMO with Adaptive Boost	SMOB
		Over-Sampling with Bagging	OBag
		SMO with Bagging	SMOBag
	Compatible Techniques	Compatible Adaptive Boost	ComAdB
		Compatible Bagging	ComBag
Cost-sensitive-based Techniques	Adaptive Cost-sensitive boost		AdaC
	Cost-sensitive AdaUBoost		AdaUB
	Asymmetric Adaptive Boost		AsymB
	Cost-Sensitive Neural Network		CSNN
	Cost-Sensitive Logistic Regression		CSLR
	Cost-Sensitive Decision Tree		CSDT
	Cost-Sensitive Random Forest		CSRF
	Cost-Sensitive Support Vector Machine		CSSVM

- **C**ondensed **N**earest **N**eighbor (**CNN**) (Hart 1968) cleans the instances using the following rules: (1) treating all the positive instances as one set S_1; (2) randomly selecting one negative instance and adding into S_1, while the other negative instances are treated as another set S_2; (3) training a nearest neighbor classfier on S_2 and obtaining the classification results; (4) adding the mis-classification instances into S_1; (5) repeating the above procedure until there donot exist any instances that need to be added into S_1. Finally, we retain the set S_1 and discard S_2.
- **N**eighborhood **C**leaning **R**ule (**NCR**) (Laurikkala 2001) resorts the ENN and k-nearest neighbor techniques for cleaning instances.

2.1.2 Over-sampling Based Techniques

- **Ada**ptive **S**ynthetic sampling (**AdaS**) (He et al. 2008) generates the positive instances according to the weighted distribution of these instances.
- **R**andom **O**ver-**S**ampling (**ROS**) (He and Garcia 2009) randomly adds replicated positive instances into the original data until the same number of positive and negative instances.
- The **S**ymthetic **M**inority **O**ver-sampling (**SMO**) (Chawla et al. 2002) generates the artificial instances based on the original set of positive ones. Specifically, for each positive instance i, SMO first calculates its k nearest neighbors and then randomly selects several instances from its nearest neighbors according to a pre-defined sampling rate associated with the imbalanced ratio of instances. Finally, SMO randomly synthesizes the new instances based on the instance i and the selected nearest neighbors.
- **B**orderline **SMO** (**BSMO**) (Han et al. 2005) is a variant of SMO, which generates the new instances according to the positive instances located on the boundary.
- **S**upport vector machine **SMO** (**SSMO**) (Nguyen et al. 2011) is an improved version of SMO, which incorporates the support vector machine method to check instances used for synthesizing new ones.

2.1.3 The Combinative Techniques

- **SMO** with **ENN** (**SMOENN**) (Batista et al. 2004) copes with the imbalance issue by combining the over-sampling based SMO technique with under-sampling based ENN method.
- **SMO** with **T**ome**k**Links (**SMOTk**) (Batista et al. 2003) adopts SMO method to generate new instances meanwhile deletes the negative ones based on the TomekLinks algorithm (Tomek et al. 1976b).

2.2 Ensemble-based Techniques

Ensemble-based imbalanced learning techniques first train a group of individual classifiers that derived from the existing learning algorithms and then incorporate these classifiers by means of some specific strategies. The aim is to deal with the imbalance issue for obtaining the advanced classification performance compared with the individual classifier. In this work, we focus on the following five types of the ensemble techniques: commonly-used, under-sampling based, over-sampling based, re-weighting based, and compatible techniques.

2.2.1 Commonly-used Techniques

- **Bag**ging (**Bag**) (Breiman 1996) randomly selects several instance subsets by means of putting back instances in each sampling process. For each subset, Bag trains a weak classifier and then integrates such basic classifiers to produce a stronger classifier.
- **Bal**anced **Bag**ging (**BalBag**) (Louppe and Geurts 2012) introduces the additional balancing strategy into the bagging classifier.
- **Ada**ptive **B**oost (**AdaB**) (Freund and Schapire 1997) adjusts the weights of instances that are inaccurately predicted in each step of the boosting algorithm.

2.2.2 Under-sampling Based Techniques

– **S**elf-**P**aced **E**nsemble (**SPE**) (Liu et al. 2020) utilizes a self-paced factor to execute the self-paced harmonize under-sampling process according to the hardness distribution of negative instances.
– **Bal**anced **Cas**cade (**BalCas**) (Liu et al. 2008) sequentially trains the classfication model and the negative instances that are correctly predicted in each training iteration are deleted.
– **Bal**anced **R**andom **F**orest (**BalRF**) (Chen et al. 2004) randomly under-samples instances by a balanced random forest model to make the whole dataset rebalanced.
– **EasyE**nsemble (**EasyE**) (Liu et al. 2008) first samples several subsets from the negative instances and then treats each subset and all the positive instances as a whole to train a base classifier. Finally, EasyE combines the results of each classifier as the final output.
– **R**andom **U**nder-**S**ampling with **B**oost (**RUSB**) (Seiffert et al. 2009) adopts the RUS technique during each process of the boosting algorithm.
– **U**nder **Bag**ging (**UBag**) (Chen et al. 2004) incorporates the RUS technique into the bagging classifier.

2.2.3 Over-sampling Based Techniques

– **Over**-sampling with adaptive **B**oost (**OverB**) (Chawla et al. 2003) combines the ROS method and boosting process to relieve the imbalance issue.
– **SMO** with adaptive **B**oost (**SMOB**) (Chawla et al. 2003) introduces the SMO technique into the boosting process to alleviate the imbalance issue.
– **O**ver-sampling with **Bag**ging (**OBag**) (Maclin and Opitz 1997) integrates the ROS approach into the bagging procedure.
– **SMO** with **Bag**ging (**SMOBag**) (Wang and Yao 2009) first employs the SMO technique to generate more comprehensively positive instances and then trains the bagging classifier based on the newly-produced instances.

2.2.4 Compatible Techniques

– **Com**patible **Ad**aptive Boost (**ComAdB**) (Freund and Schapire 1997) ameliorates the AdaB method to cater to the imbalanced-ensemble style in which the weights of inaccurately predicted instances are modified.
– **Com**patible **Bag**ging (**ComBag**) (Ho 1998; Louppe and Geurts 2012) modifies the bagging approach in the way of randomizing the construction of base classfiers. The aim is to decrease the variance of the classifiers.

2.3 Cost-sensitive-based Techniques

Cost-sensitive-based imbalanced learning techniques aim to change the weights of crash instances from different classes based on the number of data in each class.

- **Ada**ptive **C**ost-sensitive boost (**AdaC**) (Fan et al. 1999) modifies the AdaB algorithm via reducing the misclassification cost by means of increasing weights of incorrectly classified instances and decreasing weights of correctly classified ones.
- Cost-sensitive **AdaUB**oost (**AdaUB**) (Shawe-Taylor and Karakoulas 1999) is a variant of the AdaB technique, which introduces the step of data preprocessing to optimize the unequal loss. The corresponding aim is to lessen the misclassification cost.
- **Asym**metric adaptive **B**oost (**AsymB**) (Viola and Jones 2001) improves the AdaB method by means of changing the data distribution via the asymmetric misclassification cost during the boosting process step.
- **C**ost-**S**ensitive **N**eural **N**etwork (**CSNN**) employs the forward neural network to classify the crash instances in which different classes are given distinct class weights (Abadi et al. 2016).
- **C**ost-**S**ensitive **L**ogistic **R**egression (**CSLR**) adopts the logistic regression as the basic classifier meanwhile relieving the imbalanced issue by adjusting the instance weights of different classes (Yu et al. 2011).
- **C**ost-**S**ensitive **D**ecision **T**ree (**CSDT**) employs the decision tree as the basic classifier meanwhile adjusting weights of instances in different classes to deal with the class imbalanced issue (Loh 2011).
- **C**ost-**S**ensitive **R**andom **F**orest (**CSRF**) alters the instance weights of different classes and utilizes the random forest for classification (Breiman 2001).
- **C**ost-**S**ensitive **S**upport **V**ector **M**achine (**CSSVM**) takes class imbalanced issue into account by changing the weights of instances in different classes and uses the support vector machine for classification (Platt et al. 1999).

3 Experimental Setup

3.1 Data Preparation

As our goal is to conduct a large-scale empirical study to investigate how different imbalanced learning techniques impact the performance of the crash fault residence prediction models, in this work, we adopt a publicly available dataset released by a recent study (Gu et al. 2019) as our benchmark dataset. It contains 7 Java projects: **Codec**, Apache Commons **Collections**, Apache Commons **IO**, **Jsoup**, **JSqlParser**, **Mango**, and **Ormlite**-Core. The crash instances in these projects are derived from the real-world crashes simulated by the PIT[1] tool and the simulated crash instances are filtered based on the pre-defined rules to retain the useful instances. Then, the static program analyzer Spoon (Pawlak et al. 2016) is employed to extract 89 features that are used to represent each crash instance. The detailed information of the 89 features are showed in Table 2, which consists of the following 5 categories: **F**eatures **R**elated to the **S**tack **T**race (**FRST**), the **T**op **F**rame (**FRTF**), the **B**ottom **F**rame (**FRBF**), and features **N**ormalized by LOC (lines of code) from **FRTF** (**NFRTF**)

[1] http://pitest.org

and **FRBF** (**NFRBF**). As for the instance labels, if the residence of a crash instance exactly matches the elements of one frame in the stack trace, i.e., the class name, function name, and code line number, this crash instance is considered as inside the stack trace and labeled as '**InTrace**'; otherwise, '**OutTrace**'. All these information associated with labels can be obtained from the bug-fixing logs (Gu et al. 2019). The detailed statistic information of the

Table 2 The detailed information of 89 features (Gu et al. 2019)

Feature	Description
FRST	Features related to the stack trace
FRST01	Type of the exception in the crash
FRST02	Number of frames of the stack trace
FRST03	Number of classes in the stack trace
FRST04	Number of functions in the stack trace
FRST05	Whether an overloaded function exists in the stack trace
FRST06	Length of the name in the top class
FRST07	Length of the name in the top function
FRST08	Length of the name in the bottom class
FRST09	Length of the name in the bottom function
FRST10	Number of Java files in the project
FRST11	Number of classes in the project
FRTF (and FRBF)	Features related to the top frame and the bottom frame
FRTF01 (FRBF01)	Number of local variables in the top/bottom class
FRTF02 (FRBF02)	Number of fields in the top/bottom class
FRTF03 (FRBF03)	Number of functions (except constructor functions) in the top/bottom class
FRTF04 (FRBF04)	Number of imported packages in the top/bottom class
FRTF05 (FRBF05)	Whether the top/bottom class is inherited from others
FRTF06 (FRBF06)	LOC of comments in the top/bottom class
FRTF07 (FRBF07)	LOC of the top/bottom function
FRTF08 (FRBF08)	Number of parameters in the top/bottom function
FRTF09 (FRBF09)	Number of local variables in the top/bottom function
FRTF10 (FRBF10)	Number of if-statements in the top/bottom function
FRTF11 (FRBF11)	Number of loops in the top/bottom function
FRTF12 (FRBF12)	Number of for statements in the top/bottom function
FRTF13 (FRBF13)	Number of for-each statements in the top/bottom function
FRTF14 (FRBF14)	Number of while statements in the top/bottom function
FRTF15 (FRBF15)	Number of do-while statements in the top/bottom function
FRTF16 (FRBF16)	Number of try blocks in the top/bottom function
FRTF17 (FRBF17)	Number of catch blocks in the top/bottom function
FRTF18 (FRBF18)	Number of finally blocks in the top/bottom function
FRTF19 (FRBF19)	Number of assignment statements in the top/bottom function
FRTF20 (FRBF20)	Number of function calls in the top/bottom function
FRTF21 (FRBF21)	Number of return statements in the top/bottom function
FRTF22 (FRBF22)	Number of unary operators in the top/bottom function
FRTF23 (FRBF23)	Number of binary operators in the top/bottom function

Table 2 (continued)

Feature	Description
NFRTF (and NFRBF)	Features normalized by LOC from FRTF and FRBF
NFRTF01 (NFRBF01)	FRTF08/FRTF07 (FRBF08/FRBF07)
NFRTF02 (NFRBF02)	FRTF09/FRTF07 (FRBF09/FRBF07)
NFRTF03 (NFRBF03)	FRTF10/FRTF07 (FRBF10/FRBF07)
NFRTF04 (NFRBF04)	FRTF11/FRTF07 (FRBF11/FRBF07)
NFRTF05 (NFRBF05)	FRTF12/FRTF07 (FRBF12/FRBF07)
NFRTF06 (NFRBF06)	FRTF13/FRTF07 (FRBF13/FRBF07)
NFRTF07 (NFRBF07)	FRTF14/FRTF07 (FRBF14/FRBF07)
...	...
NFRTF14 (NFRBF14)	FRTF21/FRTF07 (FRBF21/FRBF07)
NFRTF15 (NFRBF15)	FRTF22/FRTF07 (FRBF22/FRBF07)
NFRTF16 (NFRBF16)	FRTF23/FRTF07 (FRBF23/FRBF07)

7 projects is displayed in Table 3, where # LOC means the lines of code and # Instances represents the number of crash instances. # InTrace and # OutTrace means the number of crash instances inside the stack trace and outside the stack trace, individually. IR signifies the imbalance ratio that is defined as the the number of negative crash instances (i.e., OutTrace) divided by positive ones (i.e., InTrace).

Furthermore, for each used project, we adopt the stratified sampling method to separate all the crash instances into a training set and a test set. Specifically, we first split the crash instances according to their labels into two subsets S_p (with label F_{IT}) and S_n (with label F_{OT}). Then, we randomly single out half of the crash instances from the two subsets S_p and S_n respectively, and put them together to form the training set. The remainder in S_p and S_n are comprised as the test set. This strategy ensures the same proportion of classes among the training set, test set and original data. After that, we also use the z-score technique to normalize the training set and the test set separately. Specifically, we first fit the training set into the z-score method to obtain the mapping rule and then transform the training set and test set according to this rule. For the sampling-based techniques, we first apply them to the training set to make it rebalanced and then use the rebalanced training set to construct the classification model. Finally, we use the trained classification model on the test set for prediction. For the ensemble-based techniques, the normalized training set is directly applied to building the ensemble classifiers and the corresponding test set is used to evaluate

Table 3 The detailed statistic information of the 7 Java projects

Project	# LOC	# Instances	# InTrace	# OutTrace	IR
Codec	14,480	610	177	433	2.45
Collections	61,283	1,350	273	1,077	3.95
IO	26,018	686	149	537	3.60
Jsoup	15,460	601	120	481	4.01
JSqlParser	32,868	647	61	586	9.61
Mango	30,208	733	53	680	12.83
Ormlite-Core	20,024	1,303	326	977	3.00

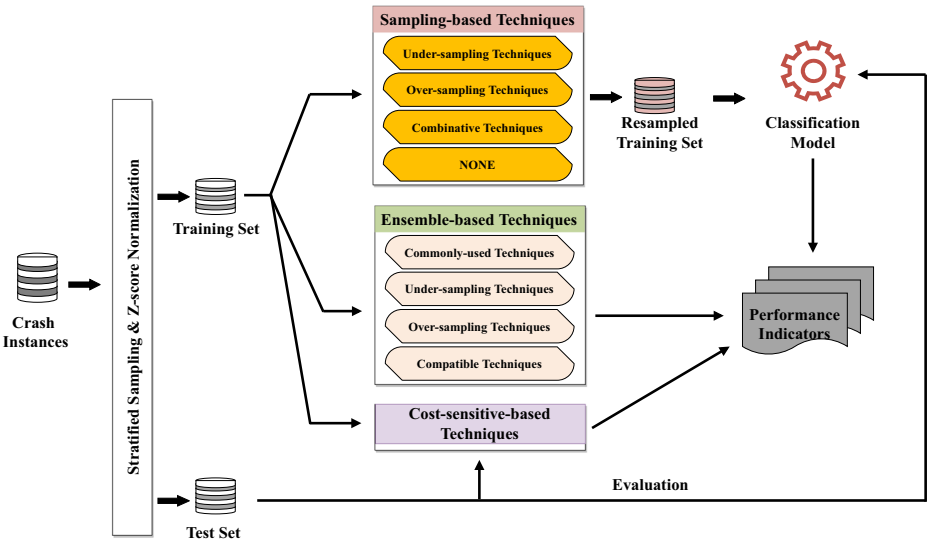


Fig. 1 An overall framework of our experiment process

the performance of these ensemble models. Figure 1 elaborates the overall framework of our experiment process. For each project, we employ the stratified sampling to obtain the training set and test set. Because this splitting process is random, we repeat it 50 times for each project to alleviate the deviation. Thus, we have 50 experimental results for each technique on each project in term of each indicator.

3.2 Evaluation Indicators

Following the previous studies (Xu et al. 2020; Zhao et al. 2021a, b), we apply the F-measure, **M**atthews **C**orrelation **C**oefficient (**MCC**), and **A**rea **U**nder the receiver operating characteristic **C**urve (**AUC**) as indicators to evaluating the performance of crashing fault residence prediction models. As our prediction task is a traditional binary classification task, in general, the evaluation indicators are derived from the confusion matrix in Table 4, where each part indicates the possibly predicted counts produced by a binary classifier. More specifically, TP denotes the number of truely positive crash instances that are predicted as positive. FN denotes the number of truely positive crash instances that are predicted as negative. FP denotes the number of truely negative crash instances that are predicted as positive and TN denotes the number of truely negative crash instances that are predicted as negative.

Table 4 The Confusion matrix

	Predicted as Positive	Predicted as Negative
Truely Positive	TP	FN
Truely Negative	FP	TN

When we treat the postive instances as those of inside the stack trace (i.e., InTrace), the F_{IT} can be calculated by the following formula:

$$F_{IT} = \frac{2 \times \text{TP}}{2 \times \text{TP} + \text{FN} + \text{FP}} \tag{1}$$

where TP, FN, and FP are counted when crash instances that reside in the stack trace are regared as positive ones. Similarly, when the crash instances located outside the stack trace are deemed as positive instances, we can also define the F_{OT} by the following equation:

$$F_{OT} = \frac{2 \times \text{TP}'}{2 \times \text{TP}' + \text{FN}' + \text{FP}'} \tag{2}$$

where TP′, FN′, and FP′ are counted when crash instances that reside outside the stack trace are considered as positive ones.

From the above evaluation indicators, we can observe that F-measure ignores the item 'TN', which causes the information loss to a certain degree. Thus, we introduce a comprehensive evaluation indicator MCC that is more suitable for assessing the performance of the model trained on the dataset with inherently imbalanced characteristic. MCC is defined as the following equation:

$$\text{MCC} = \frac{\text{TP} \times \text{TN} - \text{FP} \times \text{FN}}{\sqrt{(\text{TP} + \text{FP})(\text{TP} + \text{FN})(\text{TN} + \text{FP})(\text{TN} + \text{FN})}} \tag{3}$$

When considering the positive instances as the crash instances inside and outside the stack trace separately, we can acquire the same MCC value (Zhao et al. 2021b).

In addition to the above-mentioned indicators, we also utilize the AUC to evaluate the prediction performance. In the curve, the x-aixs represents the false positive rate $\left(\frac{\text{FP}}{\text{FP+TN}}\right)$ and the y-axis represents the true positive rate $\left(\frac{\text{TP}}{\text{TP+FN}}\right)$. The AUC value is defined as the area enclosed by the curve and the two coordinates. Obviously, treating the two classes of crash instances separately as positive ones simply means swapping the x-axis and y-axis. Thus, the AUC value is the same under the above two situations.

F_{IT}, F_{OT}, and AUC are in the range of 0 to 1, and MCC is in the range of -1 to 1. Among these indicators, the larger value means better prediction performance. Previous studies have widely used these indicators for model evaluation towards research topics in software engineering (Xu et al. 2020; Zhao et al. 2021a, b; Song et al. 2018; Fan et al., 2018; Fang et al. 2020).

3.3 Classification Models

Since the ensemble-based techniques contain the built-in function for classification, we can employ them to construct the crash fault residence prediction model directly. However, as the sampling-based techniques only focus on modifying the number of positive and negative instances to be equal, to build the prediction model, we employ 7 traditional classification models to this end, including **D**ecision **T**ree (**DT**) (Loh 2011), **R**andom **F**orest (**RF**) (Breiman 2001), **L**ogistic **R**egression (**LR**) (Yu et al. 2011), **S**upport **V**ector **M**achine (**SVM**) (Platt et al. 1999), **M**ulti-**L**ayer **P**erceptron (**MLP**) (Hinton 1990), **N**earest **N**eighbor (**NN**) (Cover and Hart 1967), and **R**epeated **i**ncremental **p**runing to **p**roduce **e**rror **r**eduction (**Ripper**) (Fürnkranz 1999). Below, we briefly present these classification models.

– **DT** adopts the tree-based structure to build the decision process, which consists of a root node, several internal nodes that represent the judgement condition for spliting

features and several leaf nodes that means the decision results. We use the gini index (Lerman and Yitzhaki 1984) as spliting condition for producing decision tree.
- **RF** is an extended version of the bagging classifier, which constructs the bagging ensemble based on the decision tree model.
- **LR** is a classic linear classification model that uses the logistic function.
- **SVM** uses a non-linear mapping function to convert the original crash data into a new latent space and then introduces a hyperplane to produce robust classification results.
- **MLP** is a type of forward artificial neural network, which consists of the input layer, hidden layer and output layer, and also integrates the activation function to deal with the non-linear classification problem.
- **NN** predicts the class label of each instance by means of the majority class label among its several nearest neighbors.
- **Ripper** is a rule-based classification model that generates a set of rules by the information gain criteria and then ranks the classes according to their frequencies.

3.4 Statistic Test

To investigate the significant difference of the experimental results for crashing faults residence prediction task, we employ the advanced **S**cott-**K**nott **E**ffect **S**ize **D**ifference (**SKESD**) test proposed by Tantithamthavorn et al. (2016). This statistic test method adopts the hierarchical clustering approach to produce the distinct ranking groups. It is worth mentioning that this method modifies the drawback of the original Scott-Knott test by log-transforming the inputs and perfecting the discrepant groups with inappreciable Cohens delta effect size into one group. Concretely, the double-round SKESD test is used for significant differences analysis. In the first round, we take as input the 50 indicator values of each imbalance learning technique on each studied project to SKESD and obtain their ranking results on project-level, respectively. In the second round, we combine the ranking results of each imbalance learning technique among all studied projects and take as input them to SKESD again, and then the overall ranking results are output as the final ranks. By the double-round SKESD test, we can generate a comprehensively global ranking list for each imbalance learning method among all projects. Note that the lower ranking value on each method implies better prediction performance compared with other baselines.

4 Results and Analysis

4.1 RQ1: What are the Impacts of Different Sampling-based Imbalanced Learning Techniques on the Performance of Crashing Fault Residence Prediction Models?

The main goal of this question is to explore how the sampling-based imbalanced learning techniques affect the performance of classification models for crashing fault residence prediction task. For this purpose, we report the results of each sampling-based technique on each classification model in terms of each indicator among all 7 studied projects. Besides, we apply the SKESD test to these classification results and analyze the ranks of these sampling-based techniques in terms of each evaluation indicator.

Accordingly, we have totally 19 (sampling-based techniques) $\times$ 7 (classification models) $\times$ 50 (experiment repeats) $\times$ 7 (projects) = 46,550 values and 19 $\times$ 7 = 133 average values for each indicator. To make the results more intuitive, we report the average results using the heat map. Figure 2 demonstrates the average values of each sampling-based imbalanced

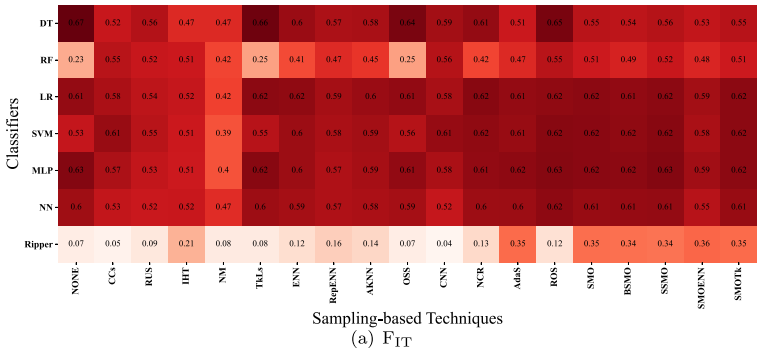


Fig. 2 Average value of each sampling-based technique for each classifier among all projects in terms of all four indicators

learning technique on each classification model among all 7 projects with each indicator, respectively. The darker color among the cells means better performance. Also, the corresponding SKESD ranking results are showed in Fig. 3. The lighter color in each row means that it obtains higher ranks (i.e., poor performance) among the sampling-based techniques when adopting the same classifier. From these two figures, we can draw the following findings:

First, in terms of F_{IT} from Fig. 2(a), we can observe that, two classification models (including LR and MLP) obtain better prediction performance on most of the sampling-based techniques, while one classifier (i.e., Ripper) achieves the worst prediction performance among nearly all sampling-based techniques. In addition, the SMO technique and its variants (including SSMO, SMOTk, and BSMO) always obtain better prediction performance on nearly all classfication models, while one under-sampling based technique (i.e., NM) achieves the worst prediction performance on 5 out of 7 classification models (except for LR and Ripper). Interestingly, the combination of NONE with DT classifier acquires the best F_{IT} value of 0.67. From Fig. 3(a), we can observe that, the largest ranking number for the 7 classifiers is around 10, which signifies that the prediction performance on these classification models have significant differences among all studied sampling-based imbalanced learning techniques. In addition, three over-sampling based techniques (including ROS, SMO, and BSMO) and one combinative technique (i.e., SMOTk) belong to the top-3 SKESD ranking groups among most of the classifiers, in which ROS belongs to top-1 or top-2 ranking groups among the classifiers (except for Ripper) and appears in the top-3 SKESD ranking groups for nearly 86% of the classification models.

Second, in terms of F_{OT} from Fig. 2(b), we can observe that, two classification models (including LR and SVM) obtain better prediction performance on most of the sampling-based techniques, while two classifiers (including DT and RF) achieve worse prediction performance on most of sampling-based techniques. In addition, two under-sampling based techniques (including TkLs and OSS) and the NONE technique always obtains better prediction performance on nearly all classification models, while one under-sampling based technique (i.e., NM) achieves the worst prediction performance on 5 out of 7 classfication models (except for DT and Ripper). Moreover, six combinations (including NONE with DT, NONE with SVM, TkLs with DT, TkLs with SVM, OSS with SVM, and ROS with DT) obtain the best F_{OT} value of 0.92. From Fig. 3(b), we can observe that, the largest ranking number for the 7 classifiers is equal or more than 10, which signifies that the prediction performance on these classification models have significant differences among all studied sampling-based imbalanced learning techniques. In addition, the NONE method belongs to the top-3 SKESD ranking groups among all the classifiers. In particular, it appears in the top-1 SKESD group for nearly 86% of the classification models.

Third, in terms of MCC from Fig. 2(c), we can observe that, two classification models (including LR and MLP) obtain better prediction performance on most of the sampling-based techniques, while one classifier (i.e., Ripper) achieves the worst prediction performance among all sampling-based techniques. In addition, two over-sampling based techniques (including ROS and SSMO) always obtain better prediction performance on nearly all classification models, while one under-sampling based technique (i.e., NM) achieves the worst prediction performance on all classification models. Interestingly, the NONE method with DT classifier acquires the best MCC value of 0.6. From Fig. 3(c), we can observe that, the largest ranking number for the 7 classifiers is around 10, which signifies that the prediction performance on these classification models have significant differences among all studied sampling-based imbalanced learning techniques. In addition,

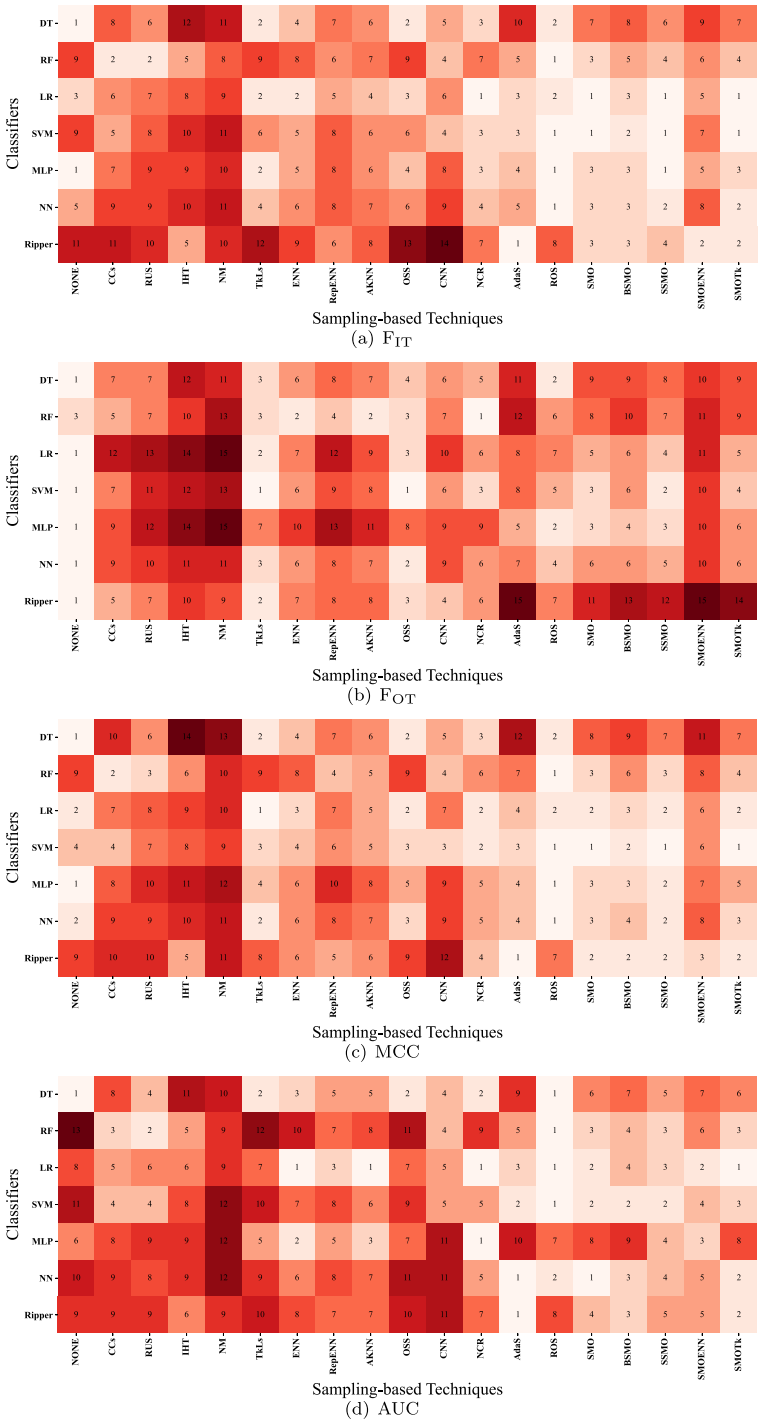


Fig. 3 The rank of SKESD test of each sampling-based technique for each classifier among all projects in terms of all four indicators

three over-sampling based techniques (including ROS, SMO, and SSMO) belong to the top-3 SKESD ranking groups among most of the classifiers in which ROS appears in the top-3 SKESD ranking groups for nearly 86% of the classification models.

Fourth, in terms of AUC from Fig. 2(d), we can observe that, one classification model (i.e., LR) obtains better prediction performance on most of sampling-based techniques, while another classifier (i.e., Ripper) achieves the worst prediction performance among all studied sampling-based techniques. In addition, one over-sampling based technique (i.e., SMO) and one combinative technique (i.e., SMOTk) always obtain better prediction performance on nearly all classification models, while one under-sampling based technique (i.e., NM) achieves worse prediction performance on 6 out of 7 classification models (except for RF). Moreover, seven combinations (including NONE with DT, TkLs with DT, ENN with MLP, OSS with DT, NCR with MLP, ROS with SVM, and ROS with DT) obtain the best AUC value of 0.79. From Fig. 3(d), we can observe that, the largest ranking number for the 7 classifiers is around 10, which signifies that the prediction performance on these classification models have significant differences among all studied sampling-based imbalanced learning techniques. In addition, one over-sampling based technique (i.e., ROS) and one combinative technique (i.e., SMOTk) belong to the top-3 SKESD ranking groups among most of the classifiers, in which ROS appears in the top-3 SKESD ranking groups for nearly 71% of the classification models.

Fifth, from Fig. 2(a)-(d), we can find that, one under-sampling based technique (i.e., NM) always achieves the worst prediction performance among all four indicators, while two over-sampling based techniques (including ROS and SMO) and the NONE method achieve better prediction performance among all four indicators. Particularly, the NONE method with DT classifier obtains the best prediction performance among all four indicators, while the classifier Ripper seems an inadaptable choice for crashing fault residence prediction task because it always acquires the worst performance among nearly all sampling-based techniques.

It is worth mentioning that a recent study (Gu et al. 2019) showed that only adopting the DT classifier achieved better performance on 5 out of 7 projects for crashing fault residence prediction task in terms of F_{IT} and F_{OT} compared with incorporating SMOTE strategy into DT. This conclusion demonstrates the correctness of our experimental results to some extent.

Answer to RQ1

DT classifier combining the NONE method without any treatment to balance the crash instances produces the best prediction performance under the imbalanced dataset, which indicates that sampling-based techniques do not always promote the performance of crashing fault residence prediction models.

4.2 RQ2: How Different Ensemble-based Imbalanced Learning Techniques Impact the Performance of Crashing Fault Residence Prediction Models?

The main aim of this question is to explore how the ensemble-based imbalanced learning techniques affect the performance of models. For this purpose, we report the results of each ensemble-based technique in terms of each indicator among all 7 studied projects. We choose DT as the base classifier because it obtains better prediction result as shown in Section 4.1. We also treat DT as the basic method to explore whether ensemble-based

Table 5 The average F_{IT} value of each ensemble-based technique

Method	Codec	Collections	IO	Jsoup	JSqlParser	Mango	Ormlite
SPE	0.714	0.790	0.770	0.616	0.688	0.607	0.862
BalCas	0.719	0.784	0.765	0.599	0.647	0.597	0.871
BalRF	0.666	0.662	0.735	0.529	0.522	0.366	0.753
EasyE	0.710	0.776	0.762	0.590	0.612	0.514	0.857
RUSB	0.513	0.579	0.640	0.426	0.569	0.498	0.698
UBag	0.710	0.776	0.762	0.590	0.612	0.514	0.857
OverB	0.555	0.726	0.748	0.552	0.683	0.702	0.834
SMOB	0.634	0.666	0.704	0.510	0.501	0.502	0.760
OBag	0.715	0.798	0.763	0.564	0.664	0.734	0.862
SMOBag	0.709	0.641	0.771	0.592	0.703	0.623	0.806
ComAdB	0.529	0.726	0.743	0.517	0.708	0.665	0.835
ComBag	0.723	0.798	0.749	0.590	0.735	0.625	0.843
Bag	0.707	0.789	0.734	0.564	0.716	0.595	0.837
BalBag	0.684	0.757	0.737	0.551	0.589	0.481	0.847
AdaB	0.545	0.717	0.742	0.526	0.714	0.661	0.833
DT	0.663	0.749	0.704	0.531	0.708	0.541	0.812

techniques improve the prediction performance. In addition, we also adopt the SKESD test to these results and analyze the ranks of these ensemble-based techniques in terms of each evaluation indicator.

Accordingly, we have totally 16 (15 ensemble-based techniques and the basic DT) $\times$ 50 (experiment repeats) $\times$ 7 (projects) = 5,600 values and 16 $\times$ 7 = 112 average values for each

Table 6 The average F_{OT} value of each ensemble-based technique

Method	Codec	Collections	IO	Jsoup	JSqlParser	Mango	Ormlite
SPE	0.881	0.947	0.937	0.884	0.964	0.957	0.957
BalCas	0.877	0.942	0.933	0.873	0.954	0.950	0.958
BalRF	0.822	0.886	0.911	0.813	0.913	0.876	0.899
EasyE	0.871	0.939	0.927	0.871	0.946	0.939	0.953
RUSB	0.819	0.892	0.905	0.855	0.947	0.951	0.900
UBag	0.871	0.939	0.927	0.871	0.946	0.939	0.953
OverB	0.856	0.938	0.937	0.909	0.973	0.980	0.948
SMOB	0.819	0.894	0.903	0.805	0.884	0.916	0.899
OBag	0.889	0.951	0.940	0.912	0.972	0.982	0.958
SMOBag	0.857	0.840	0.933	0.891	0.970	0.958	0.926
ComAdB	0.855	0.940	0.937	0.907	0.975	0.978	0.947
ComBag	0.894	0.953	0.936	0.912	0.967	0.976	0.952
Bag	0.890	0.950	0.934	0.908	0.967	0.971	0.951
BalBag	0.860	0.933	0.921	0.865	0.937	0.931	0.949
AdaB	0.859	0.937	0.937	0.907	0.975	0.978	0.947
DT	0.867	0.936	0.922	0.879	0.966	0.957	0.939

Table 7 The average MCC value of each ensemble-based technique

Method	Codec	Collections	IO	Jsoup	JSqlParser	Mango	Ormlite
SPE	0.600	0.739	0.710	0.516	0.659	0.591	0.821
BalCas	0.603	0.728	0.702	0.492	0.615	0.577	0.831
BalRF	0.513	0.574	0.660	0.398	0.496	0.357	0.668
EasyE	0.590	0.718	0.694	0.479	0.580	0.493	0.813
RUSB	0.348	0.476	0.556	0.296	0.532	0.464	0.604
UBag	0.590	0.718	0.694	0.479	0.580	0.493	0.813
OverB	0.437	0.673	0.694	0.480	0.676	0.701	0.784
SMOB	0.468	0.576	0.622	0.370	0.461	0.484	0.678
OBag	0.612	0.752	0.709	0.494	0.662	0.729	0.824
SMOBag	0.582	0.547	0.708	0.488	0.677	0.608	0.740
ComAdB	0.418	0.677	0.689	0.450	0.702	0.663	0.785
ComBag	0.625	0.756	0.691	0.513	0.718	0.622	0.800
Bag	0.607	0.744	0.678	0.486	0.697	0.588	0.792
BalBag	0.554	0.694	0.663	0.428	0.553	0.460	0.797
AdaB	0.438	0.663	0.689	0.456	0.707	0.659	0.783
DT	0.538	0.687	0.630	0.417	0.684	0.515	0.754

indicator. According to the 50 random data splits for each project, we report the average results of each compared technique on each project in terms of each indicator in Tables 5, 6, 7 and 8, respectively. Similarly, we also report the average results of 50 random splits in the following sections. To make the results more intuitive, we report the average results using the box plot. Figure 4 demonstrates the indicator values of each ensemble-based imbalanced

Table 8 The average AUC value of each ensemble-based technique

Method	Codec	Collections	IO	Jsoup	JSqlParser	Mango	Ormlite
SPE	0.800	0.865	0.849	0.783	0.847	0.852	0.904
BalCas	0.807	0.874	0.852	0.768	0.837	0.844	0.915
BalRF	0.776	0.828	0.859	0.739	0.836	0.790	0.863
EasyE	0.804	0.870	0.863	0.767	0.843	0.821	0.907
RUSB	0.665	0.737	0.767	0.645	0.782	0.755	0.800
UBag	0.804	0.870	0.863	0.767	0.843	0.821	0.907
OverB	0.691	0.809	0.823	0.707	0.789	0.796	0.881
SMOB	0.746	0.817	0.824	0.715	0.784	0.809	0.864
OBag	0.800	0.863	0.834	0.714	0.777	0.820	0.901
SMOBag	0.806	0.802	0.857	0.752	0.826	0.820	0.887
ComAdB	0.678	0.806	0.818	0.687	0.804	0.776	0.882
ComBag	0.803	0.856	0.827	0.733	0.828	0.755	0.886
Bag	0.791	0.852	0.815	0.717	0.819	0.744	0.880
BalBag	0.783	0.859	0.842	0.735	0.828	0.808	0.901
AdaB	0.687	0.802	0.817	0.691	0.801	0.774	0.881
DT	0.764	0.841	0.804	0.706	0.827	0.746	0.873

learning technique among all 7 projects with each indicator, respectively. The red lines inside the boxes represent the mean indicator values. Besides, the corresponding SKESD ranking results are showed in Fig. 5. Different colors mean that these techniques belong to distinct groups with significant differences. From these four tables and two figures, we can draw the following observations:

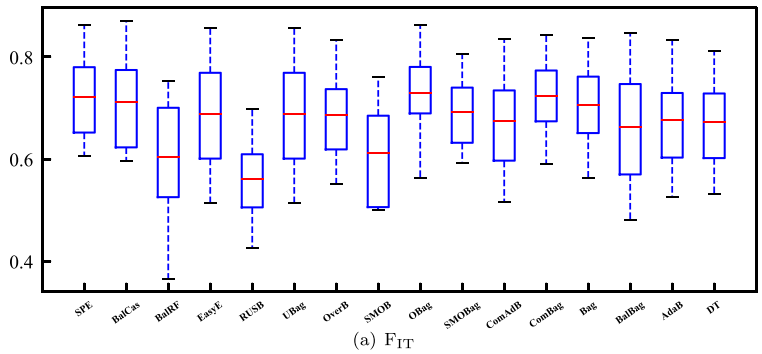
First, in terms of F_{IT} from Table 5 and Fig. 4(a), we can observe that, three ensemble-based techniques (including OBag, ComBag, and SPE) achieve better prediction performance, i.e., the average F_{IT} value of 0.729, 0.723, and 0.721 respectively, among all 16 studied baseline techniques, while one under-sampling based technique (i.e., RUSB) obtains the worst prediction performance with the average F_{IT} value of 0.560. Compared with DT, 11 out of 15 ensemble-based techniques obtain better performance with an average improvement by 4.0%. In addition, from Fig. 5(a), we can see that four ensemble techniques (including OBag, SPE, ComBag, and BalCas) belong to the top SKESD ranking group in which OBag ranks the first and has significant differences compared with other baselines in terms of F_{IT}.

Second, in terms of F_{OT} from Table 6 and Fig. 4(b), we can observe that, two ensemble-based techniques (including OBag and ComBag) achieve better prediction performance, i.e., the average F_{OT} value of 0.943 and 0.941 respectively, among all 16 studied baseline techniques, while one under-sampling based technique (i.e., BalRF) and one over-sampling based technique (i.e., SMOB) obtain the worst prediction performance with the average F_{OT} value of 0.874. Compared with DT, 8 out of 15 ensemble-based techniques obtain better performance with an average improvement by 1.2%. In addition, from Fig. 5(b), we can see that one over-sampling based ensemble technique (i.e., OBag) belongs to the top SKESD ranking group and has significant differences compared with other baselines in terms of F_{OT}.

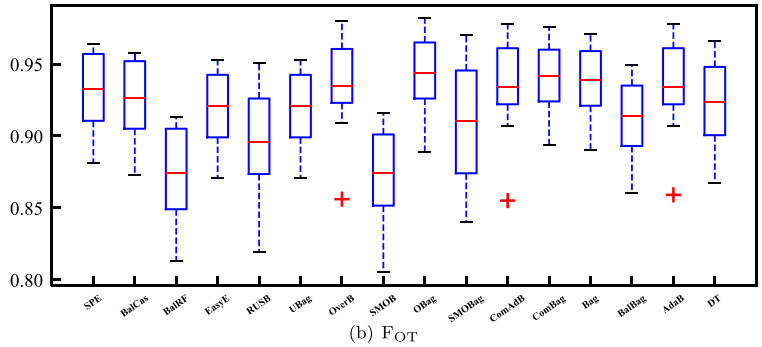
Third, in terms of MCC from Table 7 and Fig. 4(c), we can observe that, two ensemble-based techniques (including OBag and ComBag) achieve better prediction performance, i.e., the average MCC value of 0.683 and 0.675 respectively, among all 16 studied baseline techniques, while one under-sampling based technique (i.e., RUSB) obtains the worst prediction performance with the average MCC value of 0.468. Compared with DT, 11 out of 15 ensemble-based techniques obtain better performance with an average improvement by 6.6%. In addition, from Fig. 5(c), we can see that two ensemble techniques (including OBag and ComBag) belong to the top SKESD ranking group in which OBag ranks the first and has significant differences compared with other baselines in terms of MCC.

Fourth, in terms of AUC from Table 8 and Fig. 4(d), we can observe that, three under-sampling based ensemble techniques (including SPE, BalCas, and UBag) achieve better prediction performance, i.e., the average AUC value of 0.843, 0.842, and 0.839 respectively, among all 16 studied baseline techniques, while two ensemble-based technique (including ComAdB and AdaB) obtain the worst prediction performance with the average AUC value of 0.779. Compared with DT, 10 out of 15 ensemble-based techniques obtain better performance with an average improvement by 4.0%. In addition, from Fig. 5(d), we can see that four ensemble-based techniques (including UBag, EasyE, BalCas, and SPE) belong to the top SKESD ranking group and have significant differences compared with other baselines in terms of AUC.

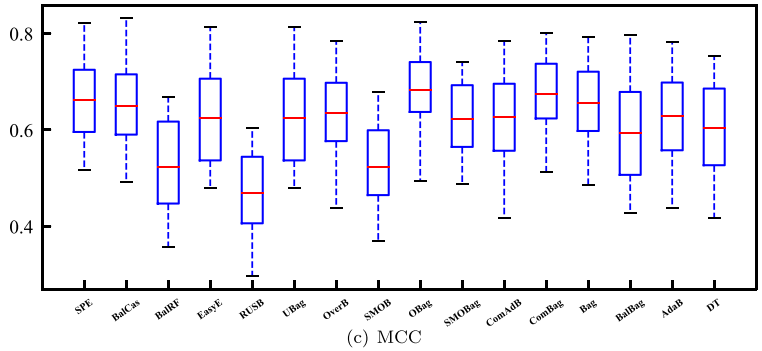
Fifth, by analyzing Tables 5-8, Figs. 4(a)-(d), and 5(a)-(d), we can find that, the over-sampling based ensemble technique OBag achieves the best prediction performance and ranks the first in the SKESD ranking groups in terms of three indicators except for AUC,



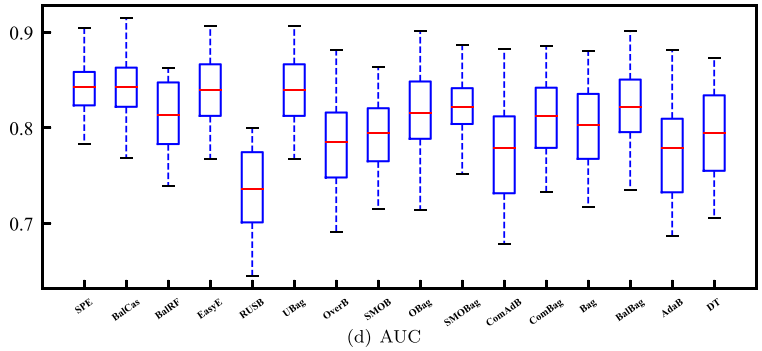
(a) F_{IT}



(b) F_{OT}



(c) MCC



(d) AUC

Fig. 4 Box plot of average value of each ensemble-based technique among all projects in terms of all four indicators

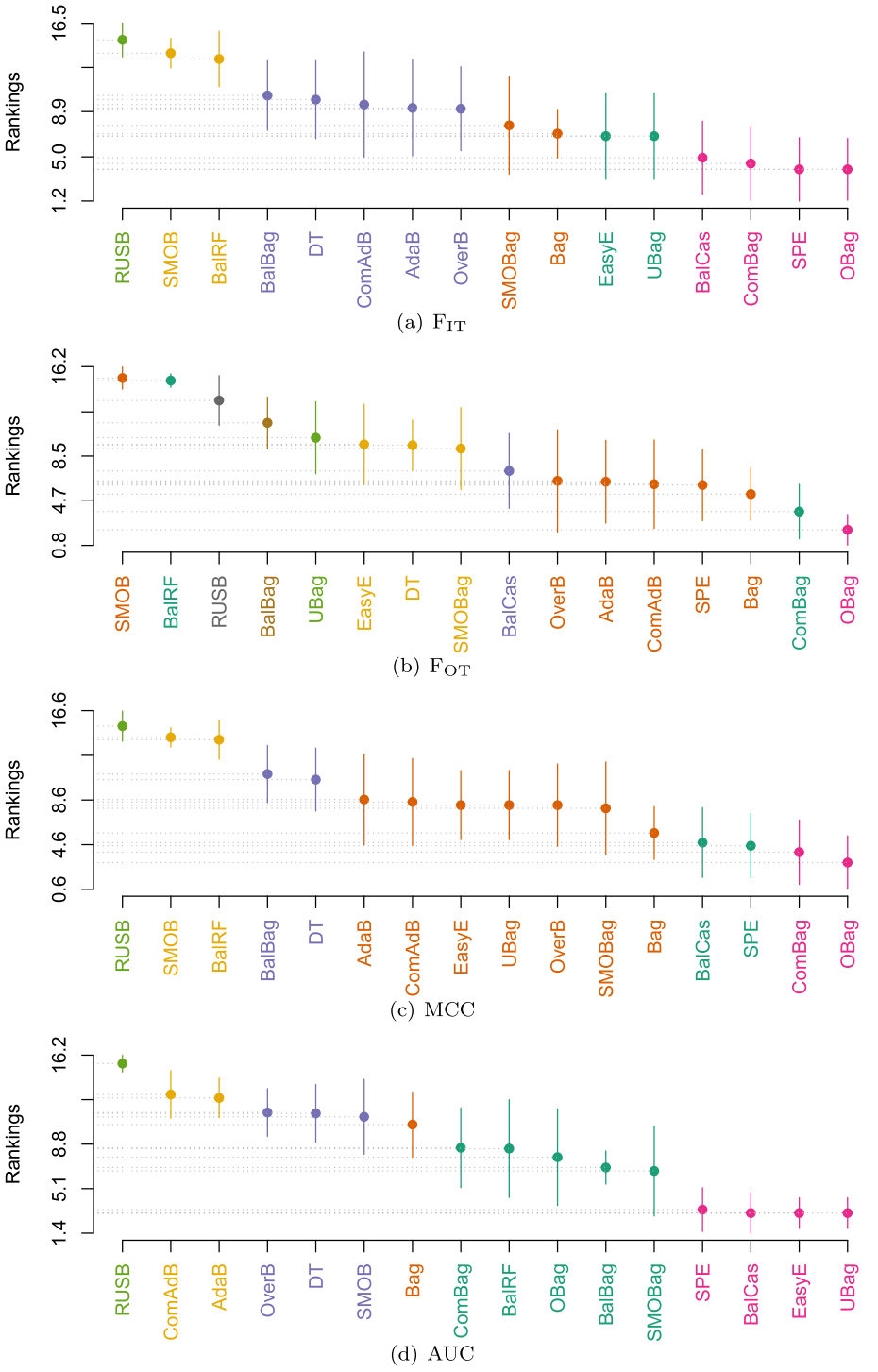


Fig. 5 SKESD rank of each ensemble-based technique across all projects in terms of all four indicators

Table 9 The average F_{IT} value of each cost-sensitive-based technique

Method	Codec	Collections	IO	Jsoup	JSqlParser	Mango	Ormlite
AdaC	0.586	0.574	0.725	0.540	0.328	0.197	0.793
AdaUB	0.533	0.717	0.739	0.514	0.690	0.688	0.826
AsymB	0.543	0.716	0.739	0.525	0.731	0.650	0.829
CSNN	0.500	0.589	0.723	0.365	0.734	0.495	0.675
CSLR	0.627	0.568	0.702	0.550	0.624	0.590	0.668
CSDT	0.646	0.741	0.706	0.539	0.531	0.631	0.829
CSRF	0.565	0.580	0.691	0.448	0.624	0.384	0.640
CSSVM	0.637	0.615	0.738	0.577	0.612	0.499	0.695

while the under-sampling based ensemble technique RUSB constantly obtains worse prediction performance among all four indicators. In addition, another under-sampling based ensemble technique UBag obtains the best prediction performance when treating the AUC as evaluation indicator.

Answer to RQ2

The over-sampling based ensemble technique OBag (with F_{IT}, F_{OT}, and MCC indicators) and the under-sampling based ensemble technique UBag (with AUC indicator) achieve better prediction performance under the imbalance dataset, which indicates that the imbalanced variants of the bagging classifier improve the performance of the crashing fault residence prediction model.

4.3 RQ3: How Different Cost-sensitive-based Imbalanced Learning Techniques Impact the Performance of Crashing Fault Residence Prediction Models?

The main aim of this question is to explore how the cost-sensitive-based imbalanced learning techniques affect the performance of models. For this purpose, we report the results of each cost-sensitive-based technique in terms of each indicator among all 7 studied projects. In addition, we also adopt the SKESD test to these results and analyze the ranks of these cost-sensitive-based techniques in terms of each evaluation indicator.

Table 10 The average F_{OT} value of each cost-sensitive-based technique

Method	Codec	Collections	IO	Jsoup	JSqlParser	Mango	Ormlite
AdaC	0.848	0.812	0.913	0.855	0.000	0.000	0.920
AdaUB	0.856	0.937	0.935	0.903	0.974	0.980	0.945
AsymB	0.857	0.937	0.935	0.907	0.975	0.973	0.947
CSNN	0.835	0.916	0.930	0.886	0.976	0.970	0.905
CSLR	0.808	0.847	0.904	0.854	0.952	0.959	0.866
CSDT	0.835	0.930	0.920	0.876	0.936	0.965	0.940
CSRF	0.749	0.858	0.911	0.800	0.951	0.898	0.848
CSSVM	0.806	0.878	0.918	0.875	0.957	0.945	0.879

Table 11 The average MCC value of each cost-sensitive-based technique

Method	Codec	Collections	IO	Jsoup	JSqlParser	Mango	Ormlite
AdaC	0.444	0.455	0.648	0.415	0.215	0.078	0.723
AdaUB	0.421	0.664	0.683	0.437	0.683	0.689	0.776
AsymB	0.431	0.662	0.684	0.458	0.721	0.646	0.779
CSNN	0.352	0.524	0.658	0.287	0.720	0.505	0.587
CSLR	0.453	0.447	0.613	0.424	0.588	0.565	0.547
CSDT	0.495	0.674	0.631	0.425	0.494	0.607	0.772
CSRF	0.353	0.462	0.605	0.284	0.594	0.359	0.508
CSSVM	0.466	0.509	0.662	0.462	0.576	0.467	0.586

Accordingly, we have totally 8 (cost-sensitive-based techniques) $\times$ 50 (experiment repeats) $\times$ 7 (projects) = 2,800 values and 8 $\times$ 7 = 56 average values for each indicator. We report the average results of each compared technique on each project in terms of each indicator in Tables 9, 10, 11 and 12, respectively. To make the results more intuitive, we report the average results using the box plot. Figure 6 demonstrates the indicator values of each cost-sensitive-based imbalanced learning technique among all 7 projects with each indicator, respectively. The red lines inside the boxes represent the mean indicator values. Besides, the corresponding SKESD ranking results are showed in Fig. 7. Different colors mean that these techniques belong to distinct groups with significant differences. From these four tables and two figures, we can draw the following observations:

First, in terms of F_{IT} from Table 9 and Fig. 6(a), we can observe that, two cost-sensitive-based techniques (including AsymB and AdaUB) achieve better prediction performance, i.e., the average F_{IT} value of 0.676 and 0.672 respectively, among all 8 studied cost-sensitive-based techniques, while one technique (i.e., AdaC) obtains the worst prediction performance with the average F_{IT} value of 0.535. In addition, from Fig. 7(a), we can see that four cost-sensitive-based techniques (including AsymB, CSDT, AdaUB, and CSSVM) belong to the top SKESD ranking group in which AsymB ranks the first and has significant differences compared with other baselines in terms of F_{IT}.

Second, in terms of F_{OT} from Table 10 and Fig. 6(b), we can observe that, two cost-sensitive-based techniques (including AsymB and AdaUB) achieve better prediction performance, i.e., the average F_{OT} value of 0.933, among all 8 studied cost-sensitive-based techniques, while one technique (i.e., AdaC) obtains the worst prediction performance

Table 12 The average AUC value of each cost-sensitive-based technique

Method	Codec	Collections	IO	Jsoup	JSqlParser	Mango	Ormlite
AdaC	0.710	0.764	0.839	0.728	0.626	0.542	0.881
AdaUB	0.680	0.802	0.818	0.686	0.790	0.787	0.876
AsymB	0.686	0.802	0.817	0.691	0.814	0.771	0.878
CSNN	0.657	0.724	0.810	0.610	0.819	0.684	0.773
CSLR	0.743	0.759	0.828	0.741	0.832	0.829	0.795
CSDT	0.755	0.846	0.811	0.716	0.750	0.820	0.891
CSRF	0.691	0.761	0.808	0.663	0.827	0.767	0.778
CSSVM	0.752	0.783	0.849	0.751	0.792	0.792	0.815

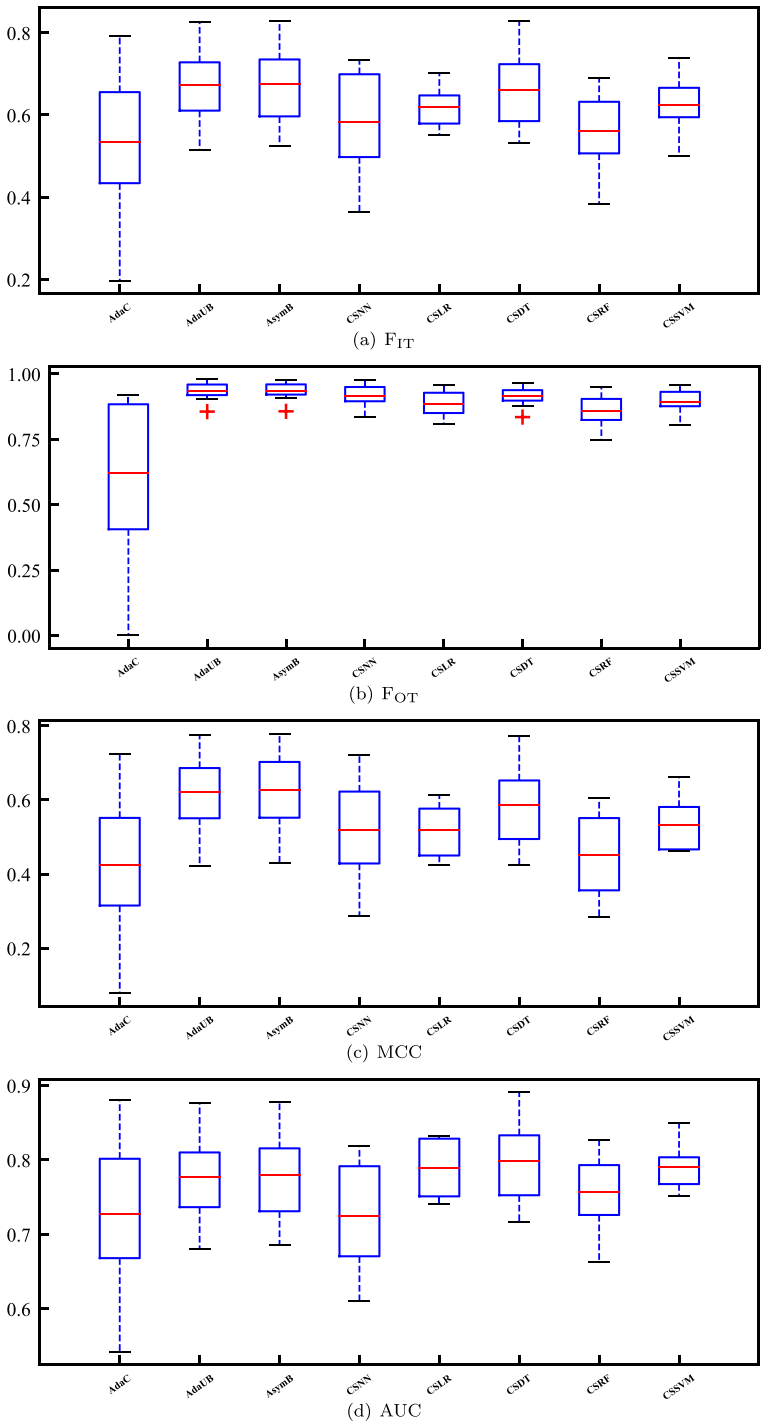


Fig. 6 Box plot of average value of each cost-sensitive-based technique among all projects in terms of all four indicators

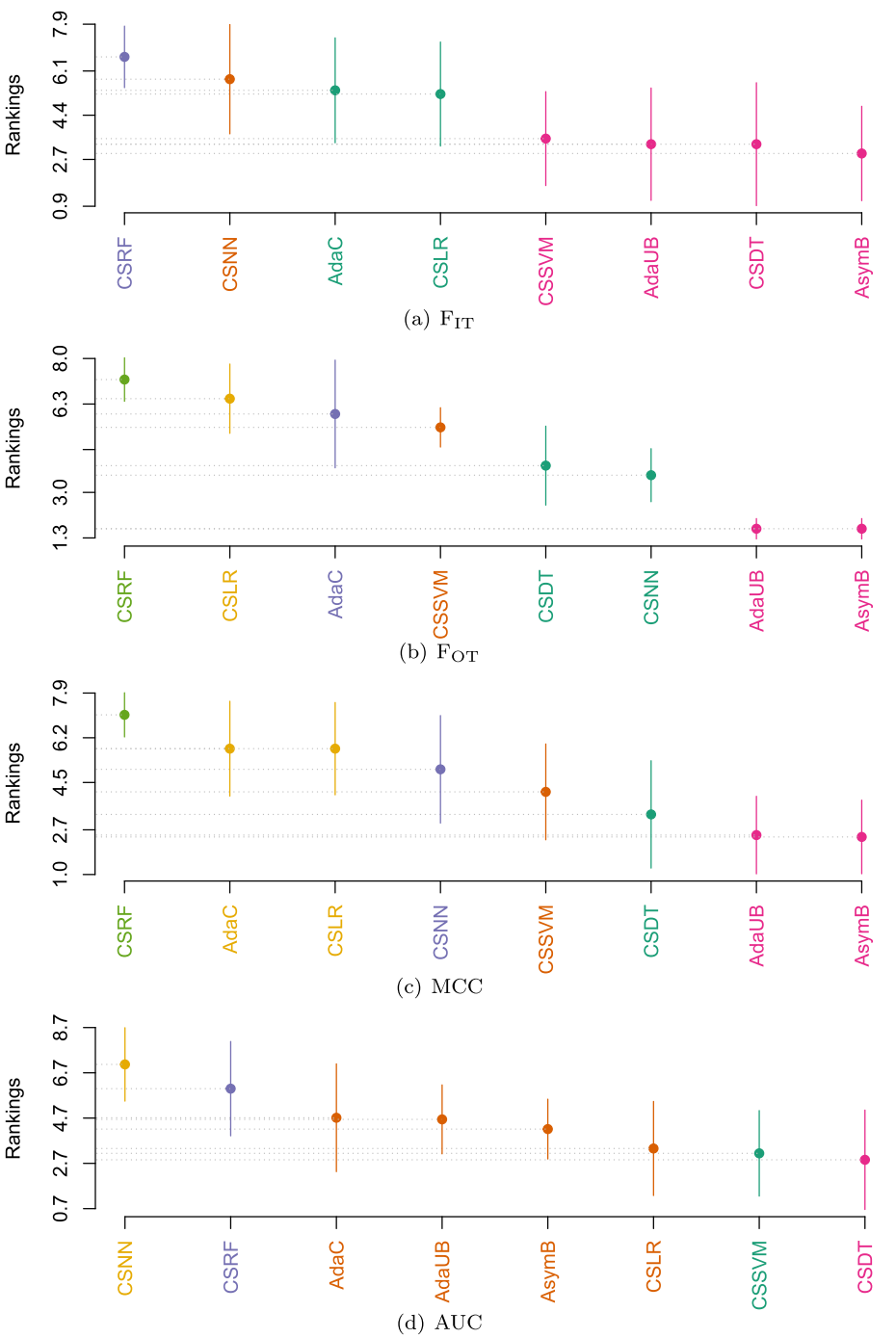


Fig. 7 SKESD rank of each cost-sensitive-based technique across all projects in terms of all four indicators

with the average F_{OT} value of 0.621. In addition, from Fig. 7(b), we can see that two cost-sensitive-based techniques (including AsymB and AdaUB) belong to the top SKESD ranking group in which AsymB ranks the first and has significant differences compared with other baselines in terms of F_{OT}.

Third, in terms of MCC from Table 11 and Fig. 6(c), we can observe that, two cost-sensitive-based techniques (including AsymB and AdaUB) achieve better prediction performance, i.e., the average MCC value of 0.626 and 0.622 respectively, among all 8 studied cost-sensitive-based techniques, while one technique (i.e., AdaC) obtains the worst prediction performance with the average MCC value of 0.425. In addition, from Fig. 7(c), we can see that two cost-sensitive-based techniques (including AsymB and AdaUB) belong to the top SKESD ranking group in which AsymB ranks the first and has significant differences compared with other baselines in terms of MCC.

Fourth, in terms of AUC from Table 12 and Fig. 6(d), we can observe that, three cost-sensitive-based techniques (including CSDT, CSSVM, and CSLR) achieve better prediction performance, i.e., the average AUC value of 0.798, 0.791, and 0.790 respectively, among all 8 studied cost-sensitive-based techniques, while one technique (i.e., CSNN) obtains the worst prediction performance with the average AUC value of 0.725. In addition, from Fig. 7(d), we can see that one cost-sensitive-based technique (i.e., CSDT) belongs to the top SKESD ranking group and has significant differences compared with other baselines in terms of AUC.

Fifth, by analyzing Tables 9–12, Figs. 6(a)-(d), and 7(a)-(d), we can find that, the cost-sensitive-based technique AsymB achieves the best prediction performance and ranks the first in the SKESD ranking groups in terms of three indicators except for AUC, while another technique AdaC constantly obtains the worst prediction performance. In addition, one technique CSDT obtains the best prediction performance when treating the AUC as evaluation indicator.

Answer to RQ3

The cost-sensitive-based technique AsymB achieves better prediction performance under the imbalance dataset in terms of F_{IT}, F_{OT}, and MCC indicators. In addition, when choosing the AUC as evaluation indicator, the cost-sensitive-based classifier, such as CSDT, can be the best choice.

4.4 RQ4: How Different Imbalanced Learning Techniques Impact the Performance of Crashing Fault Residence Prediction Models in Cross-project Scenario?

As the labeled data are not always available for newly developed project in real world, practitioners always resort the history labeled data to build the classification model to meet the requirements. The main goal of this question is to investigate the prediction performance of different imbalanced learning techniques under cross-project scenario for crashing fault residence prediction models. According to the analysis in RQ1, RQ2, and RQ3, we have found that (1) the DT classifier without any imbalance process obtains better prediction performance among all studied sampling-based techniques; (2) the ensemble-based techniques, such as OBag and UBag, perform better than other comparative methods; and (3) the cost-sensitive-based techniques, such as AsymB and CSDT, obtain better prediction performance than other cost-sensitive-based methods. Based on the above observations, we select

Table 13 The average F_{IT} value of each baseline technique in cross-project scenario

Project	OBag	UBag	AsymB	CSDT	DT
Codec	0.279	0.356	0.204	0.356	0.369
Collections	0.100	0.396	0.189	0.141	0.183
IO	0.705	0.724	0.448	0.681	0.530
Jsoup	0.147	0.385	0.203	0.289	0.232
JSqlParser	0.000	0.006	0.042	0.000	0.015
Mango	0.153	0.257	0.342	0.213	0.177
Ormlite-Core	0.739	0.851	0.564	0.640	0.479
Average	0.303	0.425	0.285	0.331	0.284

Table 14 The average F_{OT} value of each baseline technique in cross-project scenario

Project	OBag	UBag	AsymB	CSDT	DT
Codec	0.849	0.845	0.832	0.802	0.802
Collections	0.890	0.876	0.882	0.751	0.777
IO	0.939	0.936	0.906	0.912	0.829
Jsoup	0.893	0.884	0.874	0.850	0.737
JSqlParser	0.950	0.882	0.832	0.936	0.483
Mango	0.939	0.885	0.907	0.862	0.839
Ormlite-Core	0.931	0.950	0.901	0.888	0.785
Average	0.913	0.894	0.876	0.857	0.750

Table 15 The average MCC value of each baseline technique in cross-project scenario

Project	OBag	UBag	AsymB	CSDT	DT
Codec	0.307	0.306	0.192	0.186	0.194
Collections	0.164	0.287	0.157	−0.105	−0.038
IO	0.688	0.675	0.456	0.594	0.380
Jsoup	0.193	0.292	0.134	0.150	−0.014
JSqlParser	−0.001	−0.088	−0.096	−0.050	−0.354
Mango	0.105	0.200	0.287	0.142	0.094
Ormlite−Core	0.692	0.801	0.531	0.530	0.276
Average	0.307	0.353	0.237	0.207	0.077

Table 16 The average AUC value of each baseline technique in cross-project scenario

Project	OBag	UBag	AsymB	CSDT	DT
Codec	0.578	0.599	0.547	0.577	0.583
Collections	0.524	0.623	0.543	0.443	0.479
IO	0.775	0.799	0.646	0.796	0.714
Jsoup	0.537	0.618	0.542	0.565	0.492
JSqlParser	0.500	0.443	0.431	0.486	0.203
Mango	0.546	0.654	0.684	0.614	0.579
Ormlite-Core	0.801	0.903	0.698	0.755	0.650
Average	0.609	0.663	0.584	0.605	0.529

these five superior techniques as baselines to investigate how these techniques perform for crashing fault residence prediction task under cross-project scenario.

To meet the cross-project setting, in this question, we separately treat each project as the target project in turn and the remainder six projects are merged to form the candidate source project. We train each imbalanced learning model on the source project and test it on the target project. We repeat this experiment 10 times to reduce the deviation and report the results in Tables 13, 14, 15 and 16 in terms of four indicators, respectively. In addition, we employ the SKESD test to these results and analyze the ranks of the five techniques in terms of each evaluation indicator. From these four tables and the figure, we can draw the following findings:

First, in terms of F_{IT} from Table 13, we can find that, UBag achieves better average F_{IT} value on 4 out of 7 cross-project scenarios compared with other 4 baseline methods. The average F_{IT} by UBag obtains performance improvements by 40.3%, 49.1%, 28.4%, and 49.6% compared with OBag, AsymB, CSDT, and DT, respectively. UBag obtains the best average F_{IT} value of 0.425 and achieves an average improvement by 41.9%. In terms of F_{OT} from Table 14, we can find that, OBag achieves better average F_{OT} value among all 7 cross-project scenarios compared with other 4 baseline methods. The average F_{OT} by OBag obtains performance improvements by 2.1%, 4.2%, 6.5%, and 21.7% compared with UBag, AsymB, CSDT, and DT, respectively. OBag obtains the best average F_{OT} value of 0.913 and achieves an average improvement by 8.7%. In terms of MCC from Table 15, we can find that, UBag achieves better average MCC value on 3 out of 7 cross-project scenarios compared with other 4 baseline methods. The average MCC by UBag obtains performance improvements by 15.0%, 48.9%, 70.5%, and 358.4% compared with OBag, AsymB, CSDT, and DT, respectively. UBag obtains the best average MCC value of 0.353 and achieves an average improvement by 123.2%. In terms of AUC from Table 16, we can find that, UBag achieves better average AUC value on 5 out of 7 cross-project scenarios compared with other 4 baseline methods. The average AUC by UBag obtains performance improvements by 8.9%, 13.5%, 9.6%, and 25.3% compared with OBag, AsymB, CSDT, and DT, respectively. UBag obtains the best average AUC value of 0.663 and achieves an average improvement by 14.3%.

Second, the ensemble-based techniques, such as UBag and OBag, always obtain better prediction performance even under the cross-project setting, while the basic DT classifier constantly achieves the worst prediction performance. According to Fig. 8, we can find that, UBag ranks the first and has significant differences compared with other baselines in terms of F_{IT}, MCC, and AUC, while OBag ranks the first and performs better than others in terms of F_{OT}. In addition, when treating the project JSqlParser as the target project, all the five models obtain worse prediction performance. It implies that the project data itself has a certain impact on the performance of crashing fault residence prediction models as shown in previous work (Zhao et al. 2021b).

Answer to RQ4

Two ensemble-based imbalanced learning techniques UBag (with F_{IT}, MCC, and AUC indicators) and OBag (with F_{OT} indicator) achieve better prediction performance for crashing fault residence prediction task under the cross-project scenario.

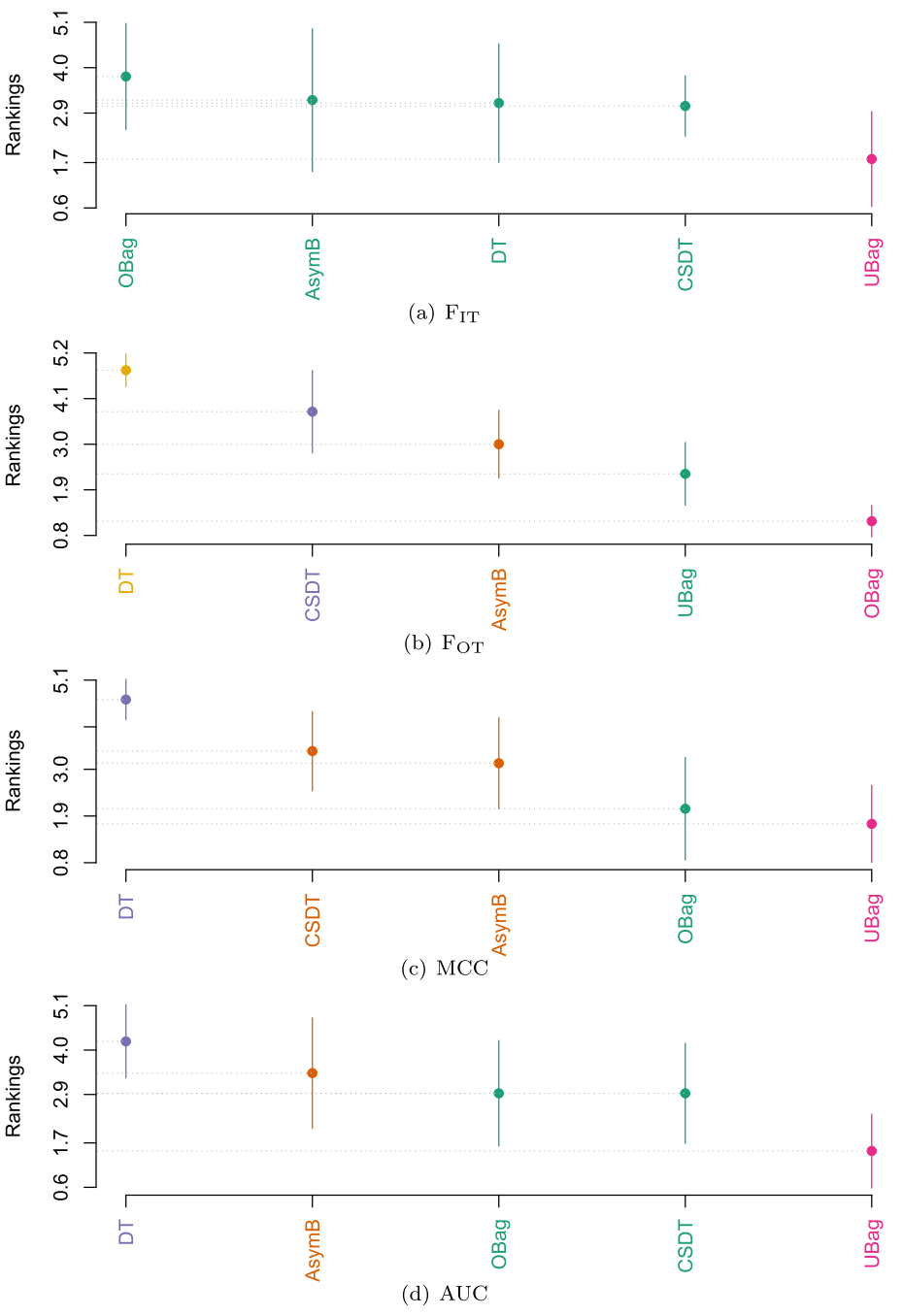


Fig. 8 SKESD rank of the five imbalanced learning techniques in cross-project scenario in terms of all four indicators

4.5 RQ5: How Does Imbalance Level of the Crash Data Impact the Performance of Imbalanced Learning Techniques?

Since the imbalance rate varies among different projects as shown in Table 3, in this research question, we explore the impact caused by the imbalance rate (i.e., imbalance level) of the crash data on different imbalanced learning techniques.

Based on the above observation from RQ1-RQ4, we also select the five superior techniques (including OBag, UBag, AsymB, CSDT, and DT) as baselines to investigate how does the imbalance level of the crash data impact the performance of these techniques for crashing fault residence prediction task in this question. We first treat all the 7 projects as a whole and manually choose ten different IR settings (from 1 to 10 with the increment of 1) to split this whole dataset. More specifically, we randomly select 200 crash instances with label F_{IT} and randomly select a certain amount of crash instances with label F_{OT} to meet distinct IR settings. For example, we randomly select 1000 crash instances with label F_{OT} under the IR of 5 and these 1200 crash instances are treating as the training set and the remainder is as the test set. This process is repeated 10 times to reduce biases.

To facilitate the understanding, we plot the line chart of IR. More specifically, we take the ordered imbalance level, i.e., IR, as the x-axis and the corresponding average prediction results for each technique as y-axis. Figure 9 demonstrates the line charts in terms of four indicators. From this figure, we can draw the following findings:

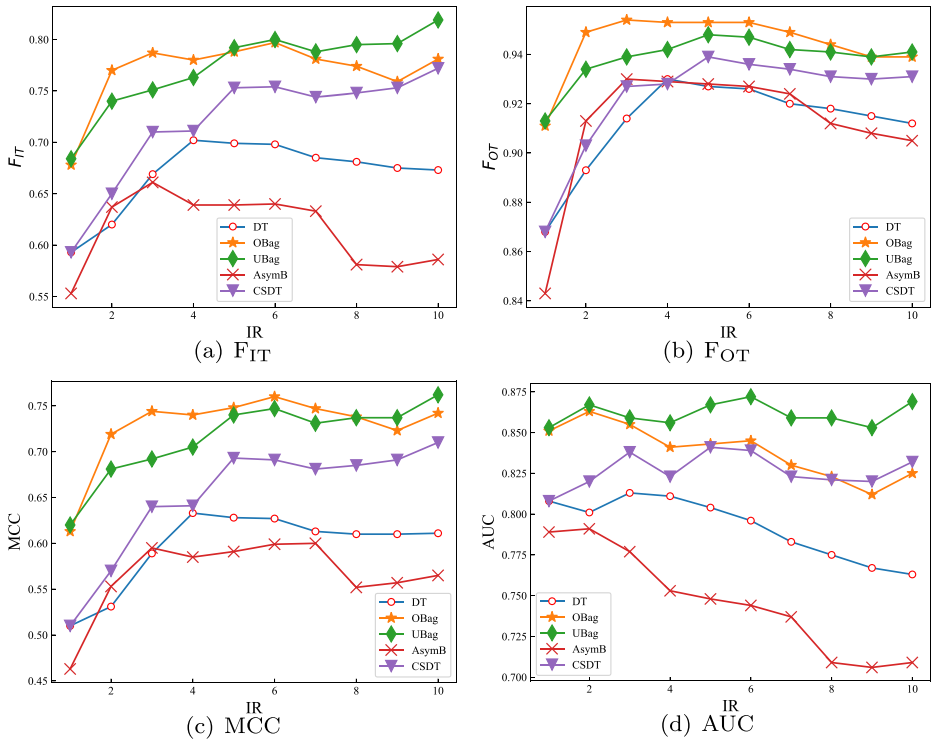


Fig. 9 Line chart of average value of the five imbalanced learning techniques under different imbalance levels in terms of all four indicators

Obviously, the prediction performance of each imbalanced learning technique varies among different IR settings. Two variants of the bagging technique (including UBag and OBag) always obtain the best prediction performance compared with other baselines in terms of four indicators, in which UBag can still achieve excellent results, i.e., F_{IT} of 0.819, F_{OT} of 0.941, MCC of 0.762, and AUC of 0.869, even under the extremely imbalance level (i.e., IR = 10). These two techniques seem to be insensitive to the imbalance level because they suffer from relatively small fluctuations as shown in Fig. 9. According to the definition of IR, the larger IR value means more negative crash instances. When applying the under-sampling technique, the redundant negative crash instances are removed randomly but the positive ones remain unchanged. Although this operation alters the proportion of the positive crash instances, the number of truly positive ones stay the same. As a result, the confusion matrix occurs a smaller variation, especially the term TP, which brings a slight impact on the performance indicators. On the other hand, cost-sensitive-based technique AsymB and the DT classifier are seriously affected by different imbalance levels. They always obtain relatively better performance with lower imbalance levels (i.e., IR < 4). As the growth of the imbalance level, their predictive performance drops. In addition, the performance of CSDT constantly increases according to the rise of IR in terms of F_{IT}, F_{OT}, and MCC, while CSDT performs stable in terms of AUC.

Answer to RQ5

The performance of the five imbalanced learning techniques for crash fault residence prediction task is affected by different imbalance levels, in which the UBag technique obtains better and relatively stable performance even though the imbalance level changes.

5 Discussion

5.1 Implications

Our empirical study reveals the following practical guidelines when building crashing fault residence prediction models using the class imbalance techniques.

Implications for Developers First, since most of the crashing faults located in stack traces, developers can mainly focus on analyzing the information stored in the stack trace and treat them as a guide to improve code quality. Second, based on our findings from experiments, developers can identify more crashing faults using suitable class imbalancing techniques (such as OBag and UBag). Then, developers can spend fewer efforts to recognize the corresponding source code and fix such bugs. Third, we found that although the prevalence of class imbalanced issues, the proportion of crash instances located inside the stack traces is still not negligible. Because of the potential impacts introduced by the different proportions between distinct classes, the handling of crash instances with inherent imbalanced issues cannot be ignored by developers. According to our experiments, developers can choose some applicable class imbalancing techniques (such as UBag and OBag) to deal with such issues. Fourth, when the history data of the newly developed project is not available, developers can train effective cross-project models based on the data from other projects because such models are more useful in practice. Under such scenario, the under-sampling-based

bagging technique UBag can be an optional solution building predictive models to identify crashing faults.

Implications for Researchers First, the over-sampling-based bagging technique OBag is advantageous when researchers expect to improve the ability of crashing fault residence prediction models (such as F_{IT}, F_{OT}, and MCC), while the under-sampling-based bagging technique UBag is beneficial for the same purpose in term of AUC. The ensemble techniques indeed increase the ability to identify crashing faults and are more stable under the changes of imbalance levels of crash data. Thus, we recommend adopt ensemble techniques to build the crashing fault residence prediction model when predictive performance is the major concern. Second, simply altering the number or changing the weights of crash instances should be avoided when developing crashing fault residence prediction models. Such imbalanced learning techniques (such as DT, AsymB, and CSDT) are sensitive to the imbalance levels and impact the stability and predictive results of the models. Third, we explore the classification performance of imbalanced techniques that achieve better results in the same project under cross-project scenarios. As our paper is the first large-scale empirical study on this problem, further works are needed on exploring the impacts of different techniques on cross-project scenarios.

5.2 Error Analysis

To further analyze our experimental results, we demonstrate the actual confusion matrix of three representative techniques (including OBag and UBag, AsymB) in Table 17. Note that we treat crash instances with label F_{IT} as positives. We choose these three techniques for example because they show better prediction performance for identifying crashing faults as shown in Section 4. From this table, we can find that the cost-sensitive-based technique (i.e., AsymB) produces larger false negatives across nearly all 7 projects due to the weight changes of different classes. Although AsymB holds smaller false negatives, it will prevent quality maintenance practitioners from identifying actual residence of crashing faults because finding the crashes is more important for them in reality. Although OBag and UBag have closer false negatives, OBag shows smaller false postives across all projects. This is because OBag inherently uses the over-sampling technique that generates more positive instances based on the original positive instances and thus the implicit information from original positive instances is fully utilized. In addition, taking the extremely imbalanced project Mongo (IR = 12.83) as an example, we can see that all the three techniques achieve

Table 17 The actual confusion matrix of three representative techniques

Project	UBag				OBag				AsymB			
	TP	FP	TN	FN	TP	FP	TN	FN	TP	FP	TN	FN
Codec	70	37	180	19	69	22	195	20	44	23	194	45
Collections	108	55	484	29	100	19	520	37	82	16	523	55
IO	66	34	235	9	65	22	247	10	50	10	259	25
Jsoup	36	34	207	24	26	10	231	34	29	11	230	31
JSqlParser	25	22	271	6	18	4	289	13	18	3	290	13
Mongo	20	21	319	7	18	2	338	9	16	2	338	11
Ormlite-Core	144	21	468	19	141	11	478	22	130	17	472	33

lower false positive rates, which implies that the imbalance learning techniques are helpful for developers to identify more crashing faults.

5.3 Threats to Validity

Threats to Internal Validity This kind of threats mainly focuses on the potential faults in our coding implementation. To dispose such threats, in this work, we make full use of the off-the-shelf implementation provided by the third-part libraries, including scikit-learn, imbalanced-learn, and imbalanced-ensemble, with the default parameter settings, to implement the studied 19 sampling-based and 18 ensemble-based imbalanced learning techniques.

Threats to External Validity This kind of threats mainly lies in the representative of the used benchmark dataset used in our work. To relieve such threats, we adopt seven publicly-available Java projects released by the pervious study. These projects come from different Java application scenarios and with diverse imbalance levels, which ensures the generalization of our experimental results to some extent. These open-source projects also allow other researchers to verify our experimental results or to do further analytical invesitigation.

Threats to Construct Validity This kind of threats mainly concentrates on the reasonability of the used performance evaluation indicators and the statistical test method. To deal with such threats, we utilize four measurement indicators, including F_{IT}, F_{OT}, MCC, and AUC, to assess the performance of the studied class imbalance techniques. In addition, we employ the state-of-the-art SKESD test to conduct the significant difference analysis on experiment results. The above selections can make our experiments more comprehensive and rigorous.

Threats to Conclusion Validity This kind of threats mainly fastens on the data analysis process. To deal with such threats, we adopt the heatmaps to elaborate our results instead of tables due to the large-scale indicator values for each technique on each project in RQ1. For RQ2 and RQ3, we elaborate tables and the corresponding boxplots to detailedly and vividly display the comparison results. We also employ the SKESD test that has the advantage of holding the rank of each compared technique among all projects to express our results, which makes the results more rigorous and intuitive.

6 Related Work

6.1 Stack Trace Analysis

Since our work resorts the stack trace information for data collection and analysis, we first present the related studies based on the stack trace. The starting point lies in Schroter et al. (2010), which conducted experiment analysis on Eclipse project and empirically indicated that the stack trace information was very helpful to programming practitioners when debugging. Subsequently, researchers mainly focused on using such information for reproducing crashes (such as STAR (Chen and Kim 2014) and MuCrash (Xuan et al. 2015)) and assisting developers for locating crashing faults (such as CrashLocator (Wu et al. 2014), BugLocator (Wong et al. 2014), and Lobster (Moreno et al. 2014)).

Nayrolles et al. (2017) alleviated the requirement of code instrumentation and the content access of heap data. A hybrid technique called JCHARMING was proposed, which

employed the stack trace producing when program exception occurred to assist the model checking tool for replicating the crashes. Their experiments adopting 30 bugs showed that JCHARMING replicated 80% of bugs with the shorter average time. Soltani et al. (2017) modified environmental dependency constraints and path explosion in previous studies when repreducing crashes. They proposed EvoCrash, an evolutionary search based method that incorporated the novel guided genetic algorithm and also proposed the new fitness function to replicate crashes. Their experiments on 50 real-world crashes in different software versions demonstrated that EvoCrash reproduced 82% of crashes in which 89% crashes were helpful when debugging. Soltani et al. (2020) developed the JCrashPack, a scalable benchmark for replicating crashes. They collected 200 crashes from seven actively maintained Java projects and employed the EvoCrash to evaluate the results. Their experiments depicted that EvoCrash repreduced 43.5% of crashes. The analysis on failed reproductions illustrated that the search-based crash replicating methods were more effective on real-wrold crashes and the *NullPointerExceptions* could be replicated more easily.

Gong et al. (2014) developed a statistical fault localization approach that comprised three parts: instrumentation and static analysis, passing and failing execution traces collection, and locating crashing faults by distance re-weighting and test coverage adjustment heuristics. Their experiments showed that this approach could successfully locate 63.9% and 52.7% of crashing faults on Firefox 3.6 and Firefox 4.0 individually by only inspecting 5% of functions. Wu et al. (2018) developed ChangeLocator to locate crash-inducing changes at the commit level. This method first extracted 10 features to specify each crash-inducing change and then constructed a classification model with logistic regression classifier. Their experiments on six versions of NetBeans demonstrated that ChangeLocator significantly performed better in terms of the mean reciprocal rank and the mean average precision indicators.

Recently, some researchers focused on identifying whether the crashing faults located in the stack trace or not (Gu et al. 2019; Xu et al. 2019b; Zhao et al. 2021b). This topic started from Gu et al. (2019) who developed a benchmark dataset based on 89 features extracted from the stack trace and fauly code for crashing fault residence prediction task but their model only obtained inferior performance. Following this work, Xu et al. (2020) proposed an imbalanced metric learning method IML, which decomposed all the crash instances into four parts and adopted the Mahalanobis distance to enlarge the distance between crash instances with discrepant labels and lessen those with the same label. Their experiments on seven Java projects showed that IML performed better than 16 baseline methods. Zhao et al. (2021a) proposed a new method ConDF that applied the consistency based feature selection technique to refine features and employed a simplified version of deep forest on the reduced features to construct classification model for crashing fault residence prediction. Their experimenst on seven open-source Java projects illustrated that ConDF outperformed 17 comparative methods in terms of three indicators.

In this work, we focus on the crashing fault residence prediction task. Different from the above studies, our main aim is to explore how different imbalanced learning techniques impact the performance of crashing fault residence prediction models.

6.2 Investigations of Class Imbalance Techniques

There were some comprehensive investigations about class imbalance techniques on the impacts of prediction models in software engineering. He and Garcia (2009) explored the research developments about imbalanced learning consisting of sampling techniques, cost-sensitive learning techniques, kernel-based learning techniques, and active learning

techniques. They also described the widely-used evaluation indicators under imbalance datasets and specified some challenges and potential research directions for imbalanced learning. Branco et al. (2016) investigated the related challenges when suffering data imbalance issues. They categoried the existing techniques dealing with imbalanced issues into four groups: data preprocessing, special-purpose learning techniques, prediction post-processing techniques, and hybrid techniques. In software engineering field, previous studies (Wang and Yao 2013; Tan et al. 2015; Agrawal and Menzies 2018; Bennin et al. 2019) investigated the impact of class imbalance issues on model performance but only with small-scale datasets. To alleviate this shortcoming, Song et al. (2018) conducted a comprehensive investigation on the characteristics and impacts of class imbalance issues in the software prediction task. Specifically, they empirically evaluated 27 datasets, seven classification models, seven types of software matrics, 17 imbalanced learning techniques, and their interations in terms of three assessment indicators. Their experimental results showed that most software prediction datasets are with low and medium levels of imbalance and the selection of imbalanced learning techniques is crucial, in particular, powerful imbalaced learning techniques and sensitive classification models are more suitable. In addition, Tantithamthavorn et al. (2018) empirically explored the impact of class rebalancing methods on the performance evaluation and interpretation of defect prediction models. More specifically, they investigated the performance of five class rebalancing methods on seven classifiers with 101 imbalanced datasets and 10 evaluation indicators. Their results illustrated that the optimized synthetic minority over-sampling technique and under-sampling technique could acquire better AUC and recall values, but these two techniques should be avoided when understanding the defect prediction models.

In this work, we concentrate on the impacts of class imbalance techniques. Different from the above studies, we focus on investigating the impact of 42 imbalanced learning techniques on the model performance for crashing fault residence prediction task. To the best of our knowledge, we are the first to conduct such a large-scale empirical study to explore this research topic.

6.3 Software Fault Prediction

Software fault prediction aims at identifying defective software components or modules, which is a hotspot in software engineering and quality assurance. Nam et al. (2013) proposed the TCA+ technique that made the feature distribution between source and target projects similar for defect prediction under the cross-project scenario. The experimental results demonstrated the superiority of TCA+. Jing et al. (2015) first proposed the unified metric representation to process the defect data from source and target projects, and then used canonical correlation analysis to build the classification model for fault prediction under the cross-company scenario. Their results showed the effectiveness of the proposed method for this task. Li et al. (2020) empirically analyze how the parameter optimization of transfer learning techniques impact the performance of cross-project fault prediction models. The experimental results give some useful insights about designing new defect prediction models. Different from these traditional fault prediction task, researchers proposed the just-in-time (JIT) defect prediction that can timely detect bug-inducing changes (Kamei et al. 2012). Kamei et al. (2016) conducted an empirical study to explore how different JIT models perform under the cross-project scenario. They found that the data should be carefully chosen to train the cross-project JIT prediction models. McIntosh and Kamei (2017) explored whether the change-level properties stay the same with the evolution of systems.

They trained JIT models based on six groups of change-level features and the results illustrated that these features fluctuated as systems evolve. Cabral et al. (2019) investigated the impact of class imbalance issues on the performance of JIT fault prediction models and then proposed a novel method for this purpose. Their results demonstrated the effectiveness of such method. Catolino et al. (2017, 2019) took the first attempt to develop a JIT fault prediction model in the context of Andriod mobile apps. They extracted six code change features to build the predictive model and the results showed that Naive Bayes obtains the best performance compared with other traditional classifiers. Followed by their work, researchers further developed new JIT fault prediction models for Andriod mobile apps in different aspects, such as feature learning (Zhao et al. 2021c) and cross-app settings (Xu et al. 2021).

Different from the above studies, in this work, we focus on the crashing fault residence prediction task. We empirically explore how 42 different imbalanced learning techniques impact the performance of crashing fault residence prediction models from different perspectives.

7 Conclusion

Predicting the residence of crashing faults in the stack trace accurately can prioritize the testing efforts to accelerate the software maintenance process. In this work, we conduct a large-scale empirical study to investigate how different imbalanced learning techniques (including 19 sampling-based, 15 ensemble-based, and 8 cost-sensitive-based techniques) impact the model performance for crashing fault residence prediction task using four performance indicators on seven open-source Java projects. We employ the SKESD test to analyze our experiment results. In addition, we explore the impact of imbalance level of the crash data on the performance of crashing fault residence prediction models. The main findings are listed as follows:

- For the sampling-based imbalance learning techniques, the DT (Decision Tree) classifier without any imbalance treatment performs better than other sampling-based techniques, which implies that the sampling-based techniques do not always work when dealing with the class imbalance issues for the crashing fault residence prediction task. This finding is consistent with the previous studies (Gu et al. 2019; Zhao et al. 2021b) which also demonstrated that the decision tree based model existed superiority on this task.
- For the ensemble-based imbalance learning techniques, one variant of the bagging classifier, i.e., OBag (Over-sampling with Bagging) achieves better performance in terms of F_{IT}, F_{OT}, and MCC indicators, whereas another variant of the bagging classifier, i.e., UBag (Under-sampling with Bagging) obtains better prediction performance with the AUC indicator. This implies that the imbalanced variants of the bagging classifier are suitable for the crashing fault residence prediction task.
- For the cost-sensitive-based imbalance learning techniques, one technique AsymB (Asymmetric adaptive Boost) achieves better performance in terms of F_{IT}, F_{OT}, and MCC indicators, whereas another technique CSDT (Cost-Sensitive Decision Tree) obtains better prediction performance with the AUC indicator.
- The ensemble-based imbalanced techniques, such as UBag (Under-sampling with Bagging) and OBag (Over-sampling with Bagging), perform better than other sampling-based or cost-sensitive-based imbalanced techniques for crashing fault residence prediction in the within-project scenario. Thus, we recommend that developers can

adopt ensemble-based techniques to build their predictive models under within-project settings.
– Two imbalanced variants of the bagging technique, including UBag (Under-sampling with Bagging) and OBag (Over-sampling with Bagging), still perform better for crashing fault residence prediction task under cross-project scenario settings. These two techniques stably outperforms other comparative methods under different imbalance levels. Thus, we recommend using them to construct the crashing fault residence prediction model when the imbalance level of the crash data is unknown.

In the future, we plan to collect more real-world crash data and explore crashes derived from other programming languages, such as Python, to enrich our experiments. We will revisit our results in other specific scenarios and explore other evaluation models and performance indicators. In addition, we will explore the impacts of the investigated techniques in practice.

Acknowledgements This work was supported in part by the National Key Research and Development Project (No. 2021YFB1714200), the Open Foundation of Key Laboratory of Dependable Service Computing in Cyber Physical Society, Ministry of Education of China (No. Grant CPSDSC202004), the National Natural Science Foundation of China (No. 62002034, 62002306, and 62272377), the Fundamental Research Funds for the Central Universities (No. 2022CDJDX-005, xxj022019001, and xzy012020009), the Natural Science Foundation of Chongqing (No. cstc2021jcyj-msxmX0538), the Macao Science and Technology Development Fund under Grant (0047/2020/A1 and 0014/2022/A), the CCF-NOFOCUS kunpeng Fund, and the Young Talent Fund of Association for Science and Technology in Shaanxi, China.

Data Availability The datasets generated during and/or analysed during the current study are available in the GitHub repository, https://github.com/sepine/EMSE-2022.

Declarations

Conflict of Interests The authors have no conflict of interest.

References

Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M et al (2016) {TensorFlow}: A system for {Large-Scale} machine learning. In: Proceedings of the 12th USENIX symposium on operating systems design and implementation (OSDI), pp 265–283

Agrawal A, Menzies T (2018) Is "better data" better than "better data miners"? In: Proceedings of 40th IEEE/ACM international conference on software engineering (ICSE). IEEE, pp 1050–1061

Batista GE, Bazzan AL, Monard MC et al (2003) Balancing training data for automated annotation of keywords: a case study. In: WOB, pp 10–18

Batista GE, Prati RC, Monard MC (2004) A study of the behavior of several methods for balancing machine learning training data. ACM SIGKDD Explor Newsl 6(1):20–29

Bennin KE, Keung JW, Monden A (2019) On the relative value of data resampling approaches for software defect prediction. Empir Softw Eng (EMSE) 24(2):602–636

Branco P, Torgo L, Ribeiro RP (2016) A survey of predictive modeling on imbalanced domains. ACM Comput Surv (CSUR) 49(2):1–50

Breiman L (1996) Bagging predictors. Mach Learn 24(2):123–140

Breiman L (2001) Random forests. Mach Learn 45(1):5–32

Cabral GG, Minku LL, Shihab E, Mujahid S (2019) Class imbalance evolution and verification latency in just-in-time software defect prediction. In: Proceedings of the IEEE/ACM 41st international conference on software engineering (ICSE). IEEE, pp 666–676

Catolino G (2017) Just-in-time bug prediction in mobile applications: the domain matters! In: Proceedings of the IEEE/ACM 4th international conference on mobile software engineering and systems (MOBILESoft). IEEE, pp 201–202

Catolino G, Di Nucci D, Ferrucci F (2019) Cross-project just-in-time bug prediction for mobile apps: An empirical assessment. In: Proceedings of the IEEE/ACM 6th international conference on mobile software engineering and systems (MOBILESoft). IEEE, pp 99–110
Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP (2002) SMOTE: synthetic minority over-sampling technique. J Artif Intell Res 16:321–357
Chawla NV, Lazarevic A, Hall LO, Bowyer KW (2003) SMOTEBoost: Improving prediction of the minority class in boosting. In: European conference on principles of data mining and knowledge discovery. Springer, pp 107–119
Chen C, Liaw A, Breiman L et al (2004) Using random forest to learn imbalanced data. Univ Calif Berkeley 110(1-12):24
Chen N, Kim S (2014) Star: Stack trace based automatic crash reproduction via symbolic execution. IEEE Trans Softw Engi (TSE) 41(2):198–220
Cover T, Hart P (1967) Nearest neighbor pattern classification. IEEE Trans Inf Theory 13(1):21–27
Dhaliwal T, Khomh F, Zou Y (2011) Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox. In: Proceedings of the 27th IEEE international conference on software maintenance (ICSM). IEEE, pp 333–342
Fan W, Stolfo SJ, Zhang J, Chan PK (1999) Adacost: misclassification cost-sensitive boosting. In: ICML, vol 99. Citeseer, pp 97–105
Fan Y, Xia X, Lo D, Hassan AE (2018) Chaff from the wheat: Characterizing and determining valid bug reports. IEEE Trans Softw Eng (TSE) 46(5):495–525
Fang C, Liu Z, Shi Y, Huang J, Shi Q (2020) Functional code clone detection with syntax and semantics fusion learning. In: Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis (ISSTA), pp 516–527
Freund Y, Schapire RE (1997) A decision-theoretic generalization of on-line learning and an application to boosting. J Comput Syst Sci 55(1):119–139
Fürnkranz J (1999) Separate-and-conquer rule learning. Artif Intell Rev 13(1):3–54
Gong L, Zhang H, Seo H, Kim S (2014) Locating crashing faults based on crash stack traces. arXiv:14044100
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
Han H, Wang WY, Mao BH (2005) Borderline-SMOTE: a new over-sampling method in imbalanced data sets learning. In: International conference on intelligent computing. Springer, pp 878–887
Hart P (1968) The condensed nearest neighbor rule (corresp.) IEEE Trans Inf Theory 14(3):515–516
He H, Garcia EA (2009) Learning from imbalanced data. IEEE Trans Knowl Data Eng (TKDE) 21(9):1263–1284
He H, Bai Y, Garcia EA, Li S (2008) ADASYN: Adaptive synthetic sampling approach for imbalanced learning. In: 2008 IEEE international joint conference on neural networks (IEEE world congress on computational intelligence), IEEE, pp 1322–1328
Hinton GE (1990) Connectionist learning procedures. In: Machine learning. Elsevier, pp 555–610
Ho TK (1998) The random subspace method for constructing decision forests. IEEE Trans Pattern Anal Mach Intell 20(8):832–844
Jing X, Wu F, Dong X, Qi F, Xu B (2015) Heterogeneous cross-company defect prediction by unified metric representation and CCA-based transfer learning. In: Proceedings of the 10th joint meeting on foundations of software engineering (FSE), pp 496–507
Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N (2012) A large-scale empirical study of just-in-time quality assurance. IEEE Trans Softw Eng (TSE) 39(6):757–773
Kamei Y, Fukushima T, McIntosh S, Yamashita K, Ubayashi N, Hassan AE (2016) Studying just-in-time defect prediction using cross-project models. Empir Softw Eng (EMSE) 21(5):2072–2106
Kubat M, Matwin S et al (1997) Addressing the curse of imbalanced training sets: one-sided selection. In: ICML, vol 97. Citeseer, pp 179–186
Laurikkala J (2001) Improving identification of difficult small classes by balancing class distribution. In: Conference on artificial intelligence in medicine in Europe. Springer, pp 63–66
Leisch F (2006) A toolbox for K-centroids cluster analysis. Comput Stat Data Anal 51(2):526–544
Lerman RI, Yitzhaki S (1984) A note on the calculation and interpretation of the Gini index. Econ Lett 15(3-4):363–368
Li K, Xiang Z, Chen T, Wang S, Tan KC (2020) Understanding the automated parameter optimization on transfer learning for cross-project defect prediction: an empirical study. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering (ICSE), pp 566–577
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 (ICECCS). IEEE, pp 104–114

Liu XY, Wu J, Zhou ZH (2008) Exploratory undersampling for class-imbalance learning. IEEE Trans Syst, Man, Cybern Part B (Cybernetics) 39(2):539–550
Liu Z, Cao W, Gao Z, Bian J, Chen H, Chang Y, Liu TY (2020) Self-paced ensemble for highly imbalanced massive data classification. In: Proceedings of 36th IEEE international conference on data engineering (ICDE). IEEE, pp 841–852
Loh WY (2011) Classification and regression trees. Wiley Interdiscip Rev Data Min Knowl Discov 1(1):14–23
Louppe G, Geurts P (2012) Ensembles on random patches. In: Joint European conference on machine learning and knowledge discovery in databases. Springer, pp 346–361
Maclin R, Opitz D (1997) An empirical evaluation of bagging and boosting. AAAI/IAAI 1997:546–551
Mani I, Zhang I (2003). In: Proceedings of workshop on learning from imbalanced datasets, ICML United States, vol 126
Mathur AP (2013) Foundations of software testing, 2/e. Pearson Education India
McIntosh S, Kamei Y (2017) Are fix-inducing changes a moving target? A longitudinal case study of just-in-time defect prediction. IEEE Trans Softw Eng (TSE) 44(5):412–428
Moreno L, Treadway JJ, Marcus A, Shen W (2014) On the use of stack traces to improve text retrieval-based bug localization. In: Proceedings of 30th IEEE international conference on software maintenance and evolution (ICSME). IEEE, pp 151–160
Nam J, Pan SJ, Kim S (2013) Transfer defect learning. In: Proceedings of the 35th international conference on software engineering (ICSE). IEEE, pp 382–391
Nayrolles M, Hamou-Lhadj A, Tahar S, Larsson A (2017) A bug reproduction approach based on directed model checking and crash traces. J Softw Evol Process (JSEP) 29(3):e1789
Nguyen HM, Cooper EW, Kamei K (2011) Borderline over-sampling for imbalanced data classification. Int J Knowl Eng Soft Data Paradigms 3(1):4–21
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 Experience 46(9):1155–1179
Platt J et al (1999) Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. Adv Large Margin Classifiers 10(3):61–74
Ren X, Xing Z, Xia X, Lo D, Wang X, Grundy J (2019) Neural network-based detection of self-admitted technical debt: From performance to explainability. ACM Trans Softw Eng Methodol (TOSEM) 28(3):1–45
Schroter A, Schröter A, Bettenburg N, Premraj R (2010) Do stack traces help developers fix bugs? In: Proceedings of 7th IEEE working conference on mining software repositories (MSR). IEEE, pp 118–121
Seiffert C, Khoshgoftaar TM, Van Hulse J, Napolitano A (2009) RUSBoost: A hybrid approach to alleviating class imbalance. IEEE Trans Syst Man Cybern-Part A Syst Hum 40(1):185–197
Shawe-Taylor GKJ, Karakoulas G (1999) Optimizing classifiers for imbalanced training sets. Adv Neural Inf Process Syst 11(11):253
Smith MR, Martinez T, Giraud-Carrier C (2014) An instance level analysis of data complexity. Mach Learn 95(2):225–256
Soltani M, Panichella A, Van Deursen A (2017) A guided genetic algorithm for automated crash reproduction. In: Proceedings of 39th IEEE/ACM international conference on software engineering (ICSE). IEEE, pp 209–220
Soltani M, Derakhshanfar P, Devroey X, Van Deursen A (2020) A benchmark-based evaluation of search-based crash reproduction. Empir Softw Eng (EMSE) 25(1):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
Tan M, Tan L, Dara S, Mayeux C (2015) Online defect prediction for imbalanced data. In: Proceedings of 37th IEEE international conference on software engineering (ICSE), vol 2. IEEE, pp 99–108
Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2016) An empirical comparison of model validation techniques for defect prediction models. IEEE Trans Softw Eng (TSE) 43(1):1–18
Tantithamthavorn C, Hassan AE, Matsumoto K (2018) The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. IEEE Trans Softw Eng (TSE) 46(11):1200–1219
Tomek I et al (1976a) An experiment with the edited nearest-neighbor rule
Tomek I et al (1976b) Two modifications of CNN
Viola P, Jones M (2001) Fast and robust classification using asymmetric adaboost and a detector cascade. Adv Neural Inf Process Syst 14
Wang S, Yao X (2009) Diversity analysis on imbalanced data sets by using ensemble models. In: 2009 IEEE symposium on computational intelligence and data mining. IEEE, pp 324–331

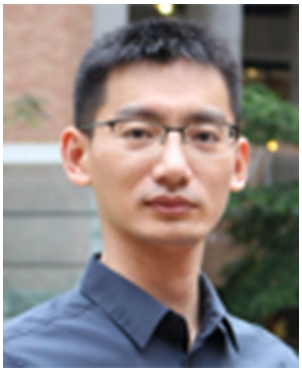
Wang S, Yao X (2013) Using class imbalance learning for software defect prediction. IEEE Trans Reliab 62(2):434–443
Wang X, Liu J, Li L, Chen X, Liu X, Wu H (2020) Detecting and explaining self-admitted technical debts with attention-based neural networks. In: Proceedings of the 35th IEEE/ACM international conference on automated software engineering (ASE), pp 871–882
Wilson DL (1972) Asymptotic properties of nearest neighbor rules using edited data. IEEE Trans Syst Man Cybern (3):408–421
Wong CP, 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 30th IEEE international conference on software maintenance and evolution (ICSME). IEEE, pp 181–190
Wu R, Zhang H, Cheung SC, Kim S (2014) Crashlocator: Locating crashing faults based on crash stacks. In: Proceedings of the 23th international symposium on software testing and analysis (ISSTA), pp 204–214
Wu R, Wen M, Cheung SC, Zhang H (2018) Changelocator: locate crash-inducing changes based on crash reports. Empir Softw Eng (EMSE) 23(5):2866–2900
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, 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 30th IEEE international symposium on software reliability engineering (ISSRE). IEEE, pp 183–194
Xu Z, Zhao K, Yan M, Yuan P, Xu L, Lei Y, Zhang X (2020) Imbalanced metric learning for crashing fault residence prediction. J Syst Softw (JSS) 170:110763
Xu Z, Zhao K, Zhang T, Fu C, Yan M, Xie Z, Zhang X, Catolino G (2021) Effort-aware just-in-time bug prediction for mobile apps via cross-triplet deep feature embedding. IEEE Trans Reliab 71(1):204–220
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, pp 910–913
Yu HF, Huang FL, Lin CJ (2011) Dual coordinate descent methods for logistic regression and maximum entropy models. Mach Learn 85(1-2):41–75
Zhao K, Liu J, Xu Z, Li L, Yan M, Yu J, Zhou Y (2021a) Predicting crash fault residence via simplified deep forest based on a reduced feature set. In: Proceedings of 29th IEEE/ACM international conference on program comprehension (ICPC). IEEE, pp 242–252
Zhao K, Xu Z, Yan M, Zhang T, Yang D, Li W (2021b) A comprehensive investigation of the impact of feature selection techniques on crashing fault residence prediction models. Information and Software Technology (IST) p 106652
Zhao K, Xu Z, Zhang T, Tang Y, Yan M (2021c) Simplified deep forest model based just-in-time defect prediction for android mobile apps. IEEE Trans Reliab 70(2):848–859

Kunsong Zhao is currently a PhD student at Department of Computing, The Hong Kong Polytechnic University. His research interests lie in blockchain and smart contract, program analysis, software engineering, and deep learning.

Zhou Xu is an assistant professor in the School of Big Data and Software Engineering at Chongqing University, China. He received two Ph.D. degrees from Wuhan University (Wuhan, China) and The Hong Kong Polytechnic University (Hong Kong, China) in 2019 and 2021, respectively. His research interests include software defect prediction, empirical software engineering, feature engineering, and data mining.

Meng Yan is now a Research Professor at the School of Big Data & Software Engineering, Chongqing University, China. He received Ph.D. degree in June 2017 from Chongqing University, China. His current research focuses on how to improve developers' productivity, how to improve software quality, and how to reduce the effort during software development by analyzing rich software repository data.

Tao Zhang received the BS degree in automation, the MEng degree in software engineering from Northeastern University, China, and the PhD degree in computer science from the University of Seoul, South Korea. After that, he spent one year with the Hong Kong Polytechnic University as a postdoctoral research fellow. Currently, he is an associate professor with the School of Computer Science and Engineering, Macau University of Science and Technology (MUST). Before joining MUST, he was the faculty member of Harbin Engineering University and Nanjing University of Posts and Telecommunications, China. He published more than 70 high-quality papers at renowned software engineering and security journals and concerences such as TSE, TIFS, TDSC, TR, ICSE, etc. His current research interests include AI for software engineering and mobile software security. He is a senior member of IEEE and ACM.



Lei Xue is an associate professor in School of Cyber Science and Technology at Sun Yat-Sen University, and before he was a research assistant professor at The Hong Kong Polytechnic University (PolyU). He also received from his PH.D. degree from PolyU. He is widely interested in designing and implementing efficient and practical security systems, with a particular focus on mobile and IoT system security, program analysis, and automotive security.

Ming Fan received the B.S. and Ph.D. degrees in computer science and technology from Xi'an Jiaotong University, China, in 2013 and 2019, respectively, and the Ph.D. degree in computing from The Hong Kong Polytechnic University. He is currently an Associate Professor with the School of Cyber Science and Engineering, Xi'an Jiaotong University. His research interests include Android malware analysis and AI security.

Jacky Keung received the BSc (Hons) degree in computer science from the University of Sydney, and the PhD degree in software engineering from the University of New South Wales, Australia. He is currently an Associate Professor at the Department of Computer Science, City University of Hong Kong. He has authored papers in prestigious journals, including IEEE Transactions on Software Engineering, Empirical Software Engineering, and many other leading journals and conferences. His research interests include software engineering, artificial intelligence, machine learning and blockchain technologies. He is a Senior Member of the IEEE, and Chairman IEEE Hong Kong Section Computer Society Chapter

Affiliations

Kunsong Zhao[1,2] · Zhou Xu[2,3] · Meng Yan[2,3] · Tao Zhang[4] · Lei Xue[5] · Ming Fan[6] · Jacky Keung[7]

Kunsong Zhao
kszhao@whu.edu.cn

Tao Zhang
tazhang@must.edu.mo

Lei Xue
xuelei3@mail.sysu.edu.cn

Ming Fan
mingfan@xjtu.edu.cn

Jacky Keung
Jacky.Keung@cityu.edu.hk

1 Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China

2 Key Laboratory of Dependable Service Computing in Cyber Physical Society (Chongqing University), Ministry of Education, Chongqing, China

3 School of Big Data and Software Engineering, Chongqing University, Chongqing, China

4 School of Computer Science and Engineering, Macau University of Science and Technology, Macao SAR, China

5 School of Cyber Science and Technology, Sun Yat-Sen University, Shenzhen, China

6 School of Cyber Science and Engineering, Xi'an Jiaotong University, Xi'an, China

7 Department of Computer Science, City University of Hong Kong, Hong Kong, China