Contents lists available at ScienceDirect

Information and Software Technology

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

A comprehensive investigation of the impact of feature selection techniques on crashing fault residence prediction models

Kunsong Zhao [a,b], Zhou Xu [c,a,*], Meng Yan [c,a], Tao Zhang [d], Dan Yang [a], Wei Li [e,f]

[a] School of Big Data and Software Engineering, Chongqing University, Chongqing, China
[b] School of Computer Science, Wuhan University, Wuhan, China
[c] Key Laboratory of Dependable Service Computing in Cyber Physical Society (Chongqing University), Ministry of Education, China
[d] Faculty of Information Technology, Macau University of Science and Technology, Macao, China
[e] School of Artificial Intelligence and Computer Science, Jiangnan University, Wuxi, China
[f] Jiangsu Key Laboratory of Media Design and Software Technology, Wuxi, China

ARTICLE INFO

Keywords:
Crash localization
Stack trace
Feature selection
Empirical study

ABSTRACT

Context: Software crash is a serious form of the software failure, which often occurs during the software development and maintenance process. As the stack trace reported when the software crashes contains a wealth of information about crashes, recent work utilized classification models with the collected features from stack traces and source code to predict whether the fault causing the crash resides in the stack trace. This could speed-up the crash localization task.
Objective: As the quality of features can affect the performance of the constructed classification models, researchers proposed to use feature selection methods to select a representative feature subset to build models by replacing the original features. However, only limited feature selection methods and classification models were taken into consideration for this issue in previous work. In this work, we look into this topic deeply and find out the best feature selection method for crash fault residence prediction task.
Method: We study the performance of 24 feature selection techniques with 21 classification models on a benchmark dataset containing crash instances from 7 real-world software projects. We use 4 indicators to evaluate the performance of these feature selection methods which are applied to the classification models.
Results: The experimental results show that, overall, a probability-based feature selection, called Symmetrical Uncertainty, performs well across the studied classification models and projects. Thus, we recommend such a feature selection method to preprocess the crash instances before constructing classification models to predict the crash fault residence.
Conclusion: This work conducts a large-scale empirical study to investigate the impact of feature selection methods on the performance of classification models for the crashing fault residence prediction task. The results clearly demonstrate that there exist significant performance differences among these feature selection techniques across different classification models and projects.

1. Introduction

As the software functions are continuous abundant, the scale and complexity of the program increases. This leads to that software products inescapably suffer from faults. The fault puts the program into an abnormal state and even terminates the program running status, i.e., program crash. When the crash occurs, the crash reporting system automatically collects the crash reports, such as the stack trace, to analyze the root cause of the crash. Identifying the residence of faults inducing the crash (short for crashing faults) can prompt programmers to inspect the corresponding source code and fix them efficiently, which is a crucial activity for software quality assurance [1].

To facilitate the process of predicting the crashing fault residence, researchers utilized the information of stack trace and source code to find the residence of the crashing faults. A stack trace contains the exception information thrown and a series of the function invocations collected at run-time. The objective of crashing fault residence prediction task is to determine whether the crashing fault resides in the stack trace or not, aiming to save the debugging efforts for programmers [2]. If the crashing fault resides inside the stack trace, programmers only need to focus on the corresponding code position recorded in the stack trace. But if the crashing fault resides outside the stack trace,

* Correspondence to: Sha-Ping-Ba District, School of Big Data and Software Engineering, Chongqing University, Chongqing 401331, China.
E-mail address: zhouxullx@cqu.edu.cn (Z. Xu).

https://doi.org/10.1016/j.infsof.2021.106652
Received 20 December 2020; Received in revised form 5 May 2021; Accepted 24 May 2021
Available online 1 June 2021

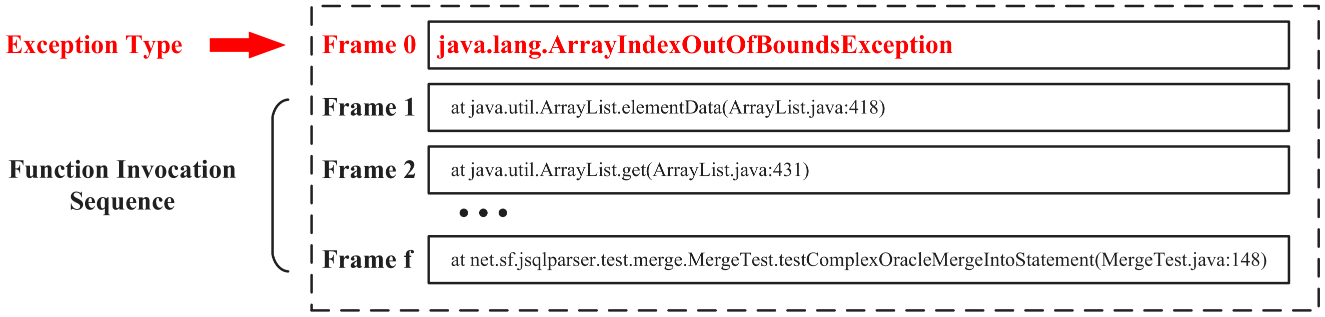

Exception Type
Frame 0
java.lang.ArrayIndexOutOfBoundsException
Function Invocation Sequence
Frame 1
at java.util.ArrayList.elementData(ArrayList.java:418)
Frame 2
at java.util.ArrayList.get(ArrayList.java:431)
Frame f
at net.sf.jsqlparser.test.merge.MergeTest.testComplexOracleMergeIntoStatement(MergeTest.java:148)

Fig. 1. The components of a stack trace.

programmers have to inspect the whole function invocation sequence. This process involves in inspecting a mass of source code with many expensive debugging efforts, which deteriorates the software maintenance efficiency.

Gu et al. [2] collected a benchmark dataset for the crashing fault residence prediction task. More specifically, they proposed a method CraTer, which generated crashes with the program mutation tool and extracted 89 features from the stack trace and faulty code to characterize the crash instances. If the crashing fault information exactly matches one of the recorded information in the stack trace, it is regarded as inside the stack trace, otherwise, it is treated as outside the stack trace. These labeled crash data are used to build classification models to predict whether a new emerged crashing fault resides in the stack trace or not. In this case, the model performance for predicting the crashing fault residence relies on the quality of the crash instance features [3,4]. The useless features from raw crash data negatively impact the performance of the classification models. The feature engineering techniques, such as feature selection, can filter out such features and select a simplified and optimal feature subset. This selected feature subset can facilitate to improve the performance of classification models and save computational cost compared with using all the original features.

However, there is no current research focusing on this topic. To narrow this gap, in this work, we conduct a large-scale empirical study to investigate the impact on 24 feature selection techniques on the performance of 21 classification models for crashing fault residence prediction task. The studied feature selection techniques derive from 4 groups: filter-based feature ranking, filter-based feature subset selection, wrapper-based feature subset selection, and without feature preprocessing. The used classification models belong to seven groups: statistic-based, neural network based, decision tree based, rule-based, nearest neighbor based, support vector machine based, and ensemble-based methods.

We conduct experiments on a publicly available dataset including 7 open-source projects shared by a recent work [2]. We employ 4 indicators to evaluate the performance of these feature selection methods which combine with the classification models for crashing fault residence prediction task. The experimental results indicate that, overall, a probability based feature ranking technique, i.e., the **S**ymmetrical **U**ncertainty (**SU**) method, obtains the best performance across the 21 classification models in terms of 4 indicators, whereas the **C**lustering **V**ariation (**CV**) obtains the worst performance. In addition, 4 feature ranking techniques, i.e., **C**hi-**S**quare (**CS**), **P**robabilistic **S**ignificance (**PS**), **G**ain **R**atio (**GR**), and SU, perform better than other feature selection methods across the 7 projects in terms of 4 indicators.

The main contributions of this paper are highlighted as follows:

- We are the first to conduct such a large-scale empirical study to analyze the performance of 24 feature selection methods with 21 classification models for crashing fault residence prediction task.
- We employ 4 indicators to comprehensively evaluate the performance of these methods on 7 open-source projects, and use a statistic test method to analyze the experimental results from both the classification model level and project level.

The rest of this paper is organized as follows. Section 2 provides the related work of our study. Section 3 describes the preliminaries, i.e., the studied feature selection techniques. The experimental setup is detailed in Section 4, including datasets, performance indicators, the data partition, the used classification models, the experimental design, and the statistic test. We present the experimental results and threats to validity in Section 5 and Section 6, respectively. Finally, we conclude this work in Section 7.

2. Related work

2.1. Stack trace analysis

The stack trace provides the fault information recorded by the system when software crashes, which contains the exception type, the location where the exception occurs, and a list of method calls connected with the corresponding runtime status. As shown in Fig. 1, the stack trace contains multiple frames, in which the first frame (i.e., Frame 0) records the exception type of the crash while other frames record the list of method calls at that time. The second frame (i.e., Frame 1, also named the top frame), is the position where the exception is thrown. The last frame (i.e., Frame f, also named the bottom frame), is the initial position of the function invocation. All frames except for Frame 0 consist of 3 main terms, i.e., the class name, function name, and code line number.

Some previous studies utilized the stack trace information for crash reproduction task which reproduces the crash scenario to assist the software debugging. Chen et al. [5] developed an automatic framework that utilized the collected stack traces and applied a sequence composition approach for crash reproduction. They conducted experiments on 3 open-source projects and the results showed that their method reproduced 22 out of 52 crashes that were useful to reveal the original ones. Xuan et al. [6] proposed an automatic method MuCrash that executed the test case on the given stack traces and selected the ones covered classes in the stack trace for crash reproduction. They conducted experiments on a project with 12 crashes and the preliminary results showed that 7 out of 12 crashes were reproduced by the MuCrash. Nayrolles et al. [7] developed a novel method JCHARMING that used the exception recorded by the stack trace and program slicing to guide the model checking for crash reproduction. They conducted experiments on 7 open-source projects and the results showed that JCHARMING reproduced 85% of bugs. Soltani et al. [8] employed a guided genetic algorithm which utilized the stack trace to guide the research for reducing the search space. They conducted experiments on 3 open-source projects and the results showed that their method could replicate 41 out of 52 crashes in which 32 crashes were useful reproductions.

In addition, some previous studies also utilized the stack trace information for crashing fault localization task which finds out the root cause of the crashing faults to assist the developers fix them. Wu et al. [9] proposed a method CrashLocator that generated approximate crash traces based on the stack information and the static analysis for crashing fault localization. They conducted experiments on 3 software projects and the results demonstrated that CrashLocator was superior to the stack-only method. Wong et al. [10] proposed to use the segmentation and the stack trace analysis for locating the fault files. They conducted experiments on 3 projects and the results showed that the two analysis methods were complementary to each other for performance improvement of crashing fault localization. Moreno et al. [11] proposed a method Lobster that combined the similarity between code elements and source code from the collected stack trace and the textual similarity for bug localization. They conducted experiments on 14 projects and the results showed that Lobster was effective in 82% of the cases compared with the Lucene-based method. Wu et al. [12] first conducted an empirical study on the characteristics of the crash-inducing changes and then proposed a method ChangeLocator. This method utilized the features collected from the crash reports and the historical fixed crashes for locating the crash-inducing changes. They conducted experiments on 6 versions of the NetBeans project and the results illustrated that ChangeLocator significantly outperformed the information retrieval based method.

Different from the above studies, Gu et al. [2] proposed a method, called CraTer, to detect whether the crashing fault resided inside the stack trace or not with the collected features of stack traces for assisting the crash localization. Following their study, in this work, we explore how different feature selection methods impact the performance of classification models for crashing fault residence prediction task.

2.2. Empirical studies of feature selection techniques and classification models

Feature selection methods are commonly used in software engineering domain to remove irrelevant or redundant features for performance improvement. Classification models are also widely used to address some learning tasks (such as the software defect prediction task) in software engineering domain. In this subsection, we discuss some previous empirical studies about feature selection methods and classification models for software engineering tasks.

Gao et al. [13] explored 4 filter-based feature selection techniques to eliminate the irrelevant and redundant features for improving the accuracy of software quality classification task. They conducted experiments on one software system with 5 classifiers and the results showed that Kolmogorov–Smirnov obtained the better prediction performance. Muthukumaran et al. [14] investigated 7 feature ranking, 2 feature subset selection, and one wrapper-based techniques with 3 classifiers on 2 software defect datasets to determine whether feature selection techniques could improve the accuracy of bug prediction models. Their results showed that wrapper-based methods obtained the best performance but with higher costs. Gao et al. [15] investigated 7 different feature ranking and 4 feature subset selection techniques with 5 classifiers for defect prediction. They conducted experiments on a large software system and the results showed that the prediction performance will be promoted when over 85% of the features were removed. Wang et al. [16] conducted an empirical study on 17 ensembles of feature ranking techniques. Their experimental results on 16 projects showed that ensembles with very few feature ranking techniques obtained better performance. Wang et al. [17] proposed an ensemble method combining 6 filter-based feature ranking techniques for the defect prediction task. They conducted experiments on 3 software projects and the results demonstrated that their methods performed better than individual filter-based techniques. Xu et al. [18] conducted an empirically study on 14 feature ranking, 2 feature subset selection, 12 wrapper-based, 3 clustering-based, and one feature extraction methods with random forest classifier for the defect prediction task. Their experimental results on 2 datasets showed that feature subset based and wrapper-based methods generally obtained better performance. Stuckman et al. [19] investigated the impact of 5 dimensionality reduction techniques for the software vulnerability prediction task. They conducted experiments on 3 open-source applications and the results demonstrated that dimensionality reduction techniques did not consistently promote the performance in the within-project scenario but could obviously promote the performance in the cross-project scenario. Ghotra et al. [20] investigated the impact of 11 feature ranking, six feature subset selection, 12 wrapper-based techniques, and one method without feature preprocess with 21 classifiers for the defect prediction task. Their experimental results on 2 datasets showed that the correlation-based feature subset selection method with the Best First search strategy was superior to other feature selection techniques. Kondo et al. [21] explored the impact of 8 feature selection techniques on 10 defect prediction models. They conducted experiments on 3 datasets and the results showed that models built on the selected features obtained the better performance than that built on the whole features. Jiarpakdee et al. [22] explored the impact of 12 feature selection techniques on the interpretation of defect models. They conducted experiments on 14 projects and the results demonstrated that most of the selected features were inconsistent among these techniques. Balogun et al. [23] investigated the impact of 4 filter-based feature ranking and 14 feature subset selection techniques with 4 classifiers for the defect prediction task. Their experimental results on 5 datasets showed that the performance of feature selection techniques is discrepant across the studied datasets and used classifiers. Yu et al. [24] employed 3 feature ranking and one feature subset selection techniques to explore the effectiveness of feature selection for the cross-project defect prediction task. They conducted experiments on 2 datasets and the results demonstrated that these studied techniques could improve the prediction performance.

Some researchers explored the impact of classification models on different software engineering tasks. Lessmann et al. [25] investigated the impact of 22 classification models for the defect prediction task. They conducted experiments on 10 projects and the results demonstrated that simple classifiers were sufficient to establish the correlation between static code and software defects. Seiffert et al. [26] explored the impact of class noise and imbalance on 11 classifiers for identifying defective software modules. They conducted experiments on a real-world dataset and the results showed that class noise had more impact on the classification performance than class imbalance. Ghotra et al. [27] explored the impact of 31 classification models for the software defect prediction task. They conducted experiments on 2 datasets and the results showed that there existed significantly performance difference among studied classification models. Tantithamthavorn et al. [28] explored the impact of automated parameter optimization on defect prediction models. Their experimental results on 18 projects showed that random forest and support vector machine were not sensitive to parameters. Chen et al. [29] explored the impact of 48 different models including 36 supervised and 12 unsupervised models on the performance of the security vulnerability prediction models. They conducted experiments on 3 open-source applications and the results demonstrated that some simple unsupervised methods could obtain the competitive performance on both within-project and cross-project scenarios. Aniche et al. [30] investigated the impact of 6 classification models on the performance of software refactoring prediction models. Their experimental results on three datasets showed that random forest classifier obtained the best prediction performance. Chen et al. [31] analyzed the impact of 9 supervised models and a set of unsupervised models on the performance of software defect number prediction task. The experimental results on 7 projects with 24 version showed that two unsupervised models achieved the best performance.

Different from the above studies which mainly focused on the defect or vulnerability prediction tasks, in this work, we investigate the impact of 24 feature selection techniques on the performance of 21 classification models for the crashing fault residence prediction task. To our best knowledge, we are the first to conduct such a large-scale empirical study to explore this topic.

Table 1
The overview of 24 feature selection techniques.

Family	Methods		Abbreviation
Filter-based feature ranking	Statistics-based Techniques	Chi-Square	CS
		Correlation	CR
		Clustering Variation	CV
	Probability-based Techniques	Probabilistic Significance	PS
		Information Gain	IG
		Gain Ratio	GR
		Symmetrical Uncertainty	SU
	Instance-based Techniques	ReliefF	ReF
		ReliefF-Weight	RW
	Classifier-based Techniques	One Rule based selection	ORF
		SVM based Selection	SVMF
Filter-based feature subset selection	Correlation-based Feature Subset Selection	Best First	CorBF
		GreedyStepwise	CorGS
	Consistency-based Feature Subset Selection	Best First	ConBF
		GreedyStepwise	ConGS
Wrapper-based feature subset selection	Nearest Neighbor	Best First	NNBF
		GreedyStepwise	NNGS
	Logistic Regression	Best First	LRBF
		GreedyStepwise	LRGS
	Naive Bayes	Best First	NBBF
		GreedyStepwise	NBGS
	Repeated Incremental Pruning to Produce Error Reduction	Best First	RIPBF
		GreedyStepwise	RIPGS
Without feature selection operation	NONE		NONE

3. Preliminaries

In this section, we briefly describe the used feature selection techniques that consist of 11 filter-based feature ranking techniques, 4 filter-based feature subset selection techniques, and 8 wrapper-based feature subset selection techniques. The main difference between filter-based and wrapper-based feature selection techniques is that the former just uses statistics to measure the importance of each feature towards the class labels while the latter uses a predetermined classification model and a performance indicator to measure the importance of a feature subset [32]. In addition, we treat the **NONE** method that without any feature selection operation as the most basic technique. Thus, we have a total of 24 feature selection techniques as the objects of our study. The overview of these techniques are demonstrated in Table 1.

3.1. Filter-based feature ranking

Filter-based feature ranking techniques first calculate the important score for each feature, and then sort features based on their corresponding scores. The higher score means the stronger correlation between the corresponding feature and class labels.

3.1.1. Statistic-based techniques

- **Chi-S**quare (**CS**) [33] estimates the worth of features with the chi-squared statistic value between each feature and class labels.
- **C**or**R**elation (**CR**) [34] evaluates the worth of features with the Pearson correlation coefficient value between each feature and class labels.
- **C**lustering **V**ariation (**CV**) [35] measures the worth of features with their variation coefficient value between each feature and class labels.

3.1.2. Probability-based techniques

- **P**robabilistic **S**ignificance (**PS**) [36] assigns a significance score to each feature with its capacity to distinguish the difference of class labels.
- **I**nformation **G**ain (**IG**) [37] evaluates the reduction of uncertainty with class labels when given a specific feature. The drawback of IG is that a multivalued feature tends to obtain a higher IG value.
- **G**ain **R**atio (**GR**) [38] alleviates the weakness of IG by penalizing the feature with more values.
- **S**ymmetrical **U**ncertainty (**SU**) [39] alleviates the weakness of IG by evaluating the symmetrical uncertainty between features and class labels.

3.1.3. Instance-based techniques

- **ReliefF** (**ReF**) [40] randomly selects an instance and its multiple nearest neighbors from its same and different class labels respectively in the training set, and then updates the correlation score of each feature with the comparison of this instance and its neighbors.
- **R**eliefF-**W**eight (**RW**) is a parameter tuning of the ReF, which weights nearest neighbors with their distance to the selected instance.

3.1.4. Classifier-based techniques

- **O**ne **R**ule based **F**eature selection (**ORF**) [41] gets a set of candidate rules by generating one rule for each feature and then select the rule with the least error ratio.
- **S**upport **V**ector **M**achine based **F**eature selection (**SVMF**) [42] ranks the features according to the square of the weight assigned by the SVM classifier.

3.2. Filter-based feature subset selection

Filter-based feature subset selection techniques aim to evaluate the feature subset selected from the original feature set rather than evaluate each feature individually.

- **Cor**relation-based feature subset selection (**Cor**) [43] uses the correlation measure to select a feature subset in which features have the low relevance among each other but high relevance to class labels.

Table 2
The statistic information of the used 7 Projects.

Project	# LOC	# Crashes	# InTrace	# OutTrace	Ratio
Codec	14,480	610	177	433	0.41
Collections	61,283	1,350	273	1,077	0.25
IO	26,018	686	149	537	0.28
Jsoup	15,460	601	120	481	0.25
JSqlParser	32,868	647	61	586	0.10
Mango	30,208	733	53	680	0.08
Ormlite	20,024	1,303	326	977	0.33

- **Con**sistency-based feature subset selection (**Con**) [44] introduces the consistency measure to select a minimal feature subset whose consistency equals to the original feature set.

To generate the feature subset with the correlation-based and consistency-based methods, we employ two kinds of search strategies in this work, which are described as follows.

- **B**est **F**irst (**BF**) is a heuristic search algorithm, which obtains a feature subset with the hillclimbing and backtracking greedily. BF has three types of strategies including forward search began with a set without features, backword search began with a set contains all features, and bidirectional search began with any one position in the feature set.
- **G**reedy **S**tepwise (**GS**) generates a feature subset with a greedy forward or backward search and stops until a feature is added or deleted that causes the performance deterioration.

Combining the above two filter-based feature subset selection techniques and the two search strategies, we have the following four techniques, including Cor with BF, Con with BF, Cor with GS, and Con with GS, which are short for **CorBF**, **ConBF**, **CorGS**, and **ConGS**, individually.

3.3. Wrapper-based feature subset selection

Wrapper-based feature subset selection techniques generate a feature subset based on predetermined classification models and evaluation indicators, aiming to select a feature subset that achieves the best performance. In this work, we apply four classification models including **N**earest **N**eighbor (**NN**), **L**ogistic **R**egression (**LR**), **N**aive **B**ayes (**NB**), and **R**epeated **I**ncremental **P**runing to produce error reduction (**RIP**) following the previous study [18,20]. The 4 classification models are briefly described in Section 4.4. In addition, we employ the **A**rea **U**nder the receiver operating characteristic **C**urve (**AUC**) as the evaluation indicator which is described in Section 4.2. We also use the aforementioned 2 search strategies for the selection process. Combining the four classifiers and 2 search strategies, we have a total of 8 techniques, including NN with BF, NN with GS, LR with BF, LR with GS, NB with BF, NB with GS, RIP with BF, and RIP with GS, which are short for **NNBF**, **NNGS**, **LRBF**, **LRGS**, **NBBF**, **NBGS**, **RIPBF**, and **RIPGS**, individually.

4. Experiment setup

4.1. Studied corpora

In this work, we conduct experiments on a publicly available benchmark dataset [2] consisting of 7 Java projects, i.e., **Codec**, Apache Commons **Collections** (**Collections**), Apache Commons **IO** (**IO**), **Jsoup**, **JSqlParser**, **Mango**, and **Ormlite**-Core (**Ormlite**). Table 2 demonstrates the statistic information of these projects including the Lines of Code (# **LOC**), the number of the crash instances (# **Crashes**), the number of crash instances inside the stack trace (# **InTrace**) and outside the stack trace (# **OutTrace**), and the ratio of # InTrace to # OutTrace (**Ratio**). The main steps for collecting the data are briefly described as follows:

Step 1: Crash generation. The testing tool PIT is applied to mimic the real-world crashes in which 7 mutators are selected for mutation generation. The mutators include conditionals boundary, increments, invert negatives, math, negate conditionals, return values, and void method calls. Then, 4 rules (i.e., the mutation passes all test cases and the exception stack traces only contains the AssertionFailedError, ComparisonFailure, or test cases) are utilized to filter the mutations that do not induce the crashes. The remaining ones are the crashes used in this work.

Step 2: Feature extraction. After generating crashes, 89 features are extracted to characterize each crash with a static program analyzer Spoon [45]. These features come from 5 categories: features **R**elated to the **S**tack **T**race (**RST**), features **R**elated to the **T**op **F**rame (**RTF**) and **B**ottom **F**rame (**RBF**), and features **N**ormalized by LOC from RTF (**NRTF**) and RBF (**NRBF**). Table 3 presents the brief description of these features.

Step 3: Crash labeling. As mentioned in Section 2, there are 3 main elements in one frame, i.e., the class name, function name, and code line number. If the information of a crashing fault exactly matches the 3 elements of one frame in the stack trace, this crash is regarded as inside the stack trace and the crash instance is labeled as 'InTrace', otherwise, the crash instance is labeled as 'OutTrace'. The label information can be collected from the bug-fixing logs [2].

The corpora can be easy to extend to other projects as long as the features of the crash instances and the corresponding labels are collected.

4.2. Evaluation indicators

In this work, we follow the previous related studies [1,3] to employ F-measure, **M**atthews **C**orrelation **C**oefficient (**MCC**), and **A**rea **U**nder the receiver operating characteristic **C**urve (**AUC**) as indicators to evaluate the performance of the combination methods of feature selection techniques with classification models for crashing fault residence prediction task. We introduce these indictors respectively.

There are four basic components for the outputs of the classification models, i.e., **T**rue **P**ositive (**TP**), **F**alse **P**ositive (**FP**), **F**alse **N**egative (**FN**), and **T**rue **N**egative(**TN**). The definitions of these components are presented in Table 4 where '#' denotes the specific number. Note that the four items are for the crash instances with label 'InTrace'.

F-measure is the weighted harmonic average between Precision and Recall where Precision = TP/(TP+FP) and Recall =TP/(TP+FN). The first indicator used is F-measure for crash instances inside the stack trace (short for F_{IT}) which is formulized as follows:

$$F_{IT} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{1}$$

Similarly, for the crash instances with label 'OutTrace', there also has four components, i.e., TP′, FP′, FN′, and TN′ which are equal to TN, FN, FP, and TP, respectively. The second indicator, i.e., F-measure for crash instances outside the stack trace (short for F_{OT}), is formulized as follows:

$$F_{OT} = \frac{2 \times \text{Precision}' \times \text{Recall}'}{\text{Precision}' + \text{Recall}'} \tag{2}$$

where $\text{Precision}' = TP'/(TP' + FP')$ and $\text{Recall}' = TP'/(TP' + FN')$.

The third indicator used is MCC, a Pearson correlation coefficient based comprehensive evaluation. MCC for crash instances with label 'InTrace' (MCC_{IT}) is formulized as follows:

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

Due to the relationship between the four components for the crash instances with label 'InTrace' and 'OutTrace', MCC_{IT} is equal to MCC_{OT}.

Table 3
The brief description of 89 features.

Feature	Description	Feature	Description
RST	Features related to the stack trace	RTF (and RBF)	Features related to the top frame and the bottom frame
RST01	Type of the exception in the crash	RTF01 (RBF01)	Number of local variables in the top/bottom class
RST02	Number of frames of the stack trace	RTF02 (RBF02)	Number of fields in the top/bottom class
RST03	Number of classes in the stack trace	RTF03 (RBF03)	Number of functions (except constructor functions) in the top/bottom class
RST04	Number of functions in the stack trace	RTF04 (RBF04)	Number of imported packages in the top/bottom class
RST05	Whether an overloaded function exists in the stack trace	RTF05 (RBF05)	Whether the top/bottom class is inherited from others
RST06	Length of the name in the top class	RTF06 (RBF06)	LOC of comments in the top/bottom class
RST07	Length of the name in the top function	RTF07 (RBF07)	LOC of the top/bottom function
RST08	Length of the name in the bottom class	RTF08 (RBF08)	Number of parameters in the top/bottom function
RST09	Length of the name in the bottom function	RTF09 (RBF09)	Number of local variables in the top/bottom function
RST10	Number of Java files in the project	RTF10 (RBF10)	Number of if-statements in the top/bottom function
RST11	Number of classes in the project	RTF11 (RBF11)	Number of loops in the top/bottom function
NRTF (and NRBF)	Features normalized by LOC from RTF and RBF	RTF12 (RBF12)	Number of for statements in the top/bottom function
NRTF01 (NRBF01)	RTF08/RTF07 (RBF08/RBF07)	RTF13 (RBF13)	Number of for-each statements in the top/bottom function
NRTF02 (NRBF02)	RTF09/RTF07 (RBF09/RBF07)	RTF14 (RBF14)	Number of while statements in the top/bottom function
NRTF03 (NRBF03)	RTF10/RTF07 (RBF10/RBF07)	RTF15 (RBF15)	Number of do-while statements in the top/bottom function
NRTF04 (NRBF04)	RTF11/RTF07 (RBF11/RBF07)	RTF16 (RBF16)	Number of try blocks in the top/bottom function
NRTF05 (NRBF05)	RTF12/RTF07 (RBF12/RBF07)	RTF17 (RBF17)	Number of catch blocks in the top/bottom function
NRTF06 (NRBF06)	RTF13/RTF07 (RBF13/RBF07)	RTF18 (RBF18)	Number of finally blocks in the top/bottom function
NRTF07 (NRBF07)	RTF14/RTF07 (RBF14/RBF07)	RTF19 (RBF19)	Number of assignment statements in the top/bottom function
NRTF08 (NRBF08)	RTF15/RTF07 (RBF15/RBF07)	RTF20 (RBF20)	Number of function calls in the top/bottom function
…	…	RTF21 (RBF21)	Number of return statements in the top/bottom function
NRTF15 (NRBF15)	RTF22/RTF07 (RBF22/RBF07)	RTF22 (RBF22)	Number of unary operators in the top/bottom function
NRTF16 (NRBF16)	RTF23/RTF07 (RBF23/RBF07)	RTF23 (RBF23)	Number of binary operators in the top/bottom function

Table 4
The definition of four basic components.

Component	Definition
True Positive (TP)	# crash instances with label InTrace that are predicted as InTrace
False Positive (FP)	# crash instances with label OutTrace but are predicted as InTrace
False Negative (FN)	# crash instances with label InTrace but are predicted as OutTrace
True Negative (TN)	# crash instances with label OutTrace that are predicted as OutTrace

The last indicator is AUC which is the area under the receiver operating characteristic curve. The x-axis and y-axis of the curve for the crash instances with label 'InTrace' are $\frac{FP}{FP+TN}$ and $\frac{TP}{TP+FN}$, respectively. As the horizontal and vertical coordinates of the curve for the crash instances with label 'InTrace' correspond to the vertical and horizontal coordinates of the curve for the crash instances with label 'OutTrace', respectively. Thus, AUC_{IT} is equal to AUC_{OT}.

Among the 4 indicators, F_{IT}, F_{OT}, and AUC range from 0 to 1, and MCC ranges from −1 to 1. F-measure=0, MCC=-1, and AUC=0 imply that all the focused crash instances are predicted as the incorrect labels, while F-measure=1, MCC=1, and AUC=1 mean that all crash instances are predicted as the correct labels. The model with larger indicator value indicates that it obtains the better performance. These indicators are widely used in previous studies for software engineering tasks [1,2,27,46–48].

4.3. Data partition

In this work, the stratified sampling technique is applied to construct the training set and test set. More specifically, we select half of the crash instances with label 'InTrace' and 'OutTrace' as training set and the remainder as the test set. This partition technique ensures that both the training set and test set have the same ratio of the two kinds of crash instances as the original set. We repeat this segmentation process 100 times to mitigate the random biases. In this work, we first divide the data into training set and test set with the stratified sampling and use the z-score based normalization method to preprocess the two sets separately, and then individually apply the 24 feature selection techniques to the training set to find out the indexes of the representative features.

4.4. Classification models

To evaluate the performance of the investigative feature selection techniques for predicting crashing faults residence, in this work, we employ a total of 21 classifiers to construct our classification models. These models derive from 7 families including 3 statistic-based, 2 neural network based, 7 decision tree based, 5 rule-based, 2 nearest neighbor based, one support vector machine based, and one ensemble-based classifiers. Table 5 demonstrates the overview of these classifiers. The detailed introduction of these models can be found in [49,50]. Here, we briefly describe these classification models as follows:

4.4.1. Statistic-based classification models

Statistic-based method has an explicit underlying probability model that can calculate the probability an instance belonging to each class [51].

- **N**aive **B**ayes (**NB**) holds the assumption that all features are independent among each other when given a specific class.
- **B**ayesian **N**etwork (**BN**) characterizes the dependencies between features based on a directed acyclic graph and depicts the joint probability between features with a conditional probability table, which can express the conditional independence among features effectively.
- **L**ogistic **R**egression (**LR**) is a generalized linear model which introduces the logistic function for classification.

4.4.2. Neural network based classification models

Neural network based method consists of a set of units belonging to different types of layers that are connected with distinct weights. The network adjusts the weights to make the label prediction of the input instances correct during the learning process [52].

Table 5
The overview of 21 classifiers.

Family	Classifiers	Abbreviation
Statistic-based classification models	Naive Bayes	NB
	Bayesian Network	BN
	Logistic Regression	LR
Neural network based classification models	Radial Basis Function	RBF
	Multi-Layer Perceptron	MLP
Decision tree based classification models	Logistic Model Tree	LMT
	Classification and Regression Tree	CART
	J48	J48
	Alternating Decision Tree	ADT
	Decision Stump	DS
	Naive Bayes/Decision-Tree	NDT
	Random Tree	RT
Rule-based classification models	Repeated Incremental Pruning to Produce Error Reduction	RIP
	One Rule based Classifier	ORC
	Decision Table	DT
	Partial Decision Tree	PDT
	Ripple Down Rule	RDR
Nearest neighbor based classification models	Nearest Neighbor	NN
	KStar	KS
Support vector machine based classification models	Voted Perceptron	VP
Ensemble-based classification models	Random Forest	RF

- **R**adial **B**asis **F**unction (**RBF**) is a feedforward network with one hidden layer and then outputs the linear combination of hidden units.
- **M**ulti-**L**ayer **P**erceptron (**MLP**) trains models with forward and backward propagation algorithms, which contains three types of layers, i.e., the input layer, hidden layer, and output layer.

4.4.3. *Decision tree based classification models*

Decision tree based method is a tree structure in which each internal node denotes a feature to be classified, each branch denotes possible value for an internal node, and each leaf node denotes a class label. The classification of an instances is determined based on its feature values [53].

- **L**ogistic **M**odel **T**ree (**LMT**) uses LR to select relevant features to construct a decision tree.
- **C**lassification **A**nd **R**egression **T**ree (**CART**) applies the gini index to split features for decision tree construction.
- **J48** applies the information gain ratio to split features for decision tree construction.
- **Al**ternating **D**ecision **T**ree (**ADT**) consists of two types of nodes, i.e., prediction node and decision node. ADT repeats the following rule for constructing the tree: a prediction node generates multiple decision nodes and a decision node generates two prediction nodes until all the nodes follow this rule.
- **D**ecision **S**tump (**DS**) generates a one-level decision tree for classification.
- **N**aive Bayes/**D**ecision-**T**ree (**NDT**) [54] generates a classification model by combining the NB and decision tree, which is cost insensitive.
- **R**andom **T**ree (**RT**) generates the decision tree by randomly selecting features.

4.4.4. *Rule-based classification models*

Rule-based method makes the class decision by using various rules, such as ‘IF-THEN’ rules [55]. The merit of this type of model is its comprehensibility.

- **R**epeated **I**ncremental **P**runing to produce error reduction (**RIP**) constructs a rule set based classification model, which ranks the classes with their frequencies and applies the IG for adding rules.
- **O**ne **R**ule based **C**lassifier (**ORC**) generates a classification rule with the values from a specific feature.
- **D**ecision **T**able (**DT**) produces a table in which each row corresponds to each input combination (i.e., the rule) while each column represents each feature. DT comprehensively considers the feature combinations.
- **P**artial **D**ecision **T**ree (**PDT**) applies the separate-and-conquer technique to generate a decision tree list where each decision tree is built by partial C4.5.
- **R**ipple **D**own **R**ule (**RDR**) uses a two-decision tree for classification and the final class is determined by the majority class in leaf nodes.

4.4.5. *Nearest neighbor based classification models*

Nearest neighbor based method is a family of lazy learner which requires less computation time during the training phase but more computation time during the test phase. The learning process is based on the neighbors of the test instance. More specifically, for a test instance, this kind of method makes a decision of its class label by calculating the distance or similarity between it and its neighbors [56].

- **N**earest **N**eighbour (**NN**) determines the class to which a specific instance belongs by the majority class of its multiple nearest neighbors.
- **K**Star (**KS**) is a variant of NN that introduces an entropic measure for computing the distance between instances.

4.4.6. *Support vector machine based classification models*

Support Vector Machine (SVM) based method applies a nonlinear operation to map the original data into a higher-dimensional feature space, and then uses a hyperplane to separate the instances with different labels in the new space [57].

- **V**oted **P**erceptron (**VP**) makes full use of the classification vectors for performance improvement. A vector producing more correct predictions means that it has a higher weight.

4.4.7. *Ensemble-based classification models*

Ensemble-based method is a family of composite model that combines the results of multiple base learners to determine the label of an instance, such as using the majority voting mechanism [58]. In general, this kind of method tends to be more effective than the individual learner.

- **R**andom **F**orest (**RF**) generates an ensemble model with the decision tree as the basic model. Each instance is sampled randomly for training different decision trees.

4.5. Parameter configuration

For the feature selection techniques that need to specify the number of selected features, as previous studies [59,60] which suggested that the model built on 15% of the original features can obtain the same even better performance than on the original ones, we set the number of remaining features to 89 × 15% = 14. Note that the test set reserves the same features as the simplified training set. After obtaining the processed data, we build the classification model on the simplified training set and evaluate the performance on the simplified test set. Fig. 2 demonstrates the overview of our experiment process in this work.

4.6. Statistic test

Some previous studies [18,27,61,62] applied the Scott–Knott test to conduct significance analysis for performance results, which generates the discrepant ranking results with the hierarchical clustering technique [63]. However, the Scott–Knott test contains some restrictions: the analyzed data needs to follow the normal distribution and this test is insensitive to the groups with the negligible effect size. To overcome the above drawbacks, Tantithamthavorn et al. [61] proposed an extended version of the Scott–Knott test, i.e., **S**cott-**K**nott **E**ffect **S**ize **D**ifference (**SKESD**), which disposes the inputs with the log-transforming and amends the groups with the negligible Cohens delta effect size. In this work, we employ a two-phase SKESD test to analyze the significant differences of the results for these explored methods that combine the feature selection techniques with classification models. The process of the SKESD test is demonstrated in Fig. 3. In the first phase, SKESD receives the 100 indicator values of each feature selection method on each project or classification model and outputs their ranking results individually. In the second phase, SKESD receives the ranking results produced by previous phase and outputs the final ranking. The two-phase SKESD test considers the overall ranking value for each feature selection technique among all projects or classification models, aiming to obtain an appropriate global ranking value. The lower ranking value of a feature selection technique means that it achieves better performance. SKESD incorporates the effect size analysis, which merges the statistically different groups that have a negligible effect size into one group [61].

In this work, we mainly use WEKA 3.8.4 to implement these feature selection techniques and classification models, and conduct all experiments on Windows 10 (Memory: 16 GB, Processor: Intel(R) Core(TM) i7-4790 CPU @ 3.60 GHz × 4). We have released the code scripts and benchmark dataset on the GitHub at https://github.com/sepine/IST-2021 for reproducing our experiments.

5. Experiment results

As we apply these studied feature selection techniques on 7 projects with 21 classification models, in this section, we analyze the results from two perspectives: the first one is to investigate the impact the feature selection techniques on each classification model, and the second one is to investigate the impact of feature selection techniques on each project, i.e., from the classification model level and project level, respectively.

5.1. RQ1: How different feature selection techniques impact the performance of each classification model across all 7 projects?

Methods: Since this question investigates the performance impact of different feature selection techniques on each classification model, to answer this question, we analyze the obtained results at the classifier level. More specifically, in terms of each indicator, we report its average values of each feature selection technique on each classification model across all 7 projects. In addition, we conduct SKESD test on these results and report the ranking values with each indicator individually.

Results: We have a total of 24 × 21 × 100 × 7 = 352,800 values and 24 × 21 = 504 average values in terms of each indicator for this research question. In this work, we use the heat map to report the average results. Fig. 4 depicts the average value of each feature selection technique for each classifier across all projects in terms of 4 indicators individually. The cell with darker color means that it obtains the better indicator value. Fig. 5 demonstrates the corresponding SKESD ranking value of each feature selection technique on each classifier. The cell with darker color means that it obtains the worse ranking value. From the two figures, we can observe that the SU (Symmetrical Uncertainty) method performs best on the used indicators overall. More detailed findings are as follows:

First, in terms of F_{IT}, from Fig. 4(a), we can observe that, 6 classification models (including CART, ADT, NDT, DT, KS, and RF) achieve the better performance on most of feature selection techniques, while 4 classification models (including NB, RBF, DS, and VP) achieve the worse performance on most of feature selection techniques. In addition, one feature selection technique (i.e., CV (Clustering Variation)) always achieves the worse performance on nearly all classification models. From Fig. 5(a), the number of SKESD ranking for 9 classification models (including BN, LR, RBF, MLP, ADT, RT, DT, VP, and RF) is equal to or more than 10. This means that the performance results on these 9 classifiers have large differences across these feature selection techniques. In addition, 3 feature selection techniques (including PS (Probabilistic Significance), GR (Gain Ratio), and SU (Symmetrical Uncertainty)) belong to the top-3 SKESD ranking across most of the classification models in which SU (Symmetrical Uncertainty) appears in the top-3 SKESD ranking for nearly 86% of the studied classification models.

Second, in terms of F_{OT}, from Fig. 4(b), we can observe that, 5 classification models (including CART, ADT, NDT, DT, and RF) achieve the better performance on most of feature selection techniques, while one classification model (i.e., NB) achieves the worse performance on all feature selection techniques. In addition, 2 feature selection methods (including CV (Clustering Variation) and SVMF (SVM based Selection)) always achieves the worse performance on nearly all classification models. From Fig. 5(b), the number of SKESD ranking for 7 classification models (including NB, BN, LR, NN, KS, VP, and RF) is equal to or more than 10. This means that the performance results on these 7 classifiers have large differences across these feature selection techniques. In addition, 4 feature selection techniques (including CS (Chi-Square), PS (Probabilistic Significance), GR (Gain Ratio), and SU (Symmetrical Uncertainty)) belong to the top-3 SKESD ranking across most of the classification models in which SU (Symmetrical Uncertainty) appears in the top-3 SKESD ranking for nearly 81% of the studied classification models.

Third, in terms of MCC, from Fig. 4(c), we can observe that, 5 classification models (including CART, ADT, NDT, DT, and RF) achieve the better performance on most of feature selection techniques, while 4 classification models (including NB, RBF, DS, and VP) achieve the worse performance on most of feature selection techniques. In addition, one feature selection method (i.e., CV (Clustering Variation)) always achieves the worse performance on nearly all classification models. From Fig. 5(c), the number of SKESD ranking for most of classification models (except for LMT, CART, J48, DS, NDT, PDT, and RF) is equal to or more than 10. This means that the performance results on

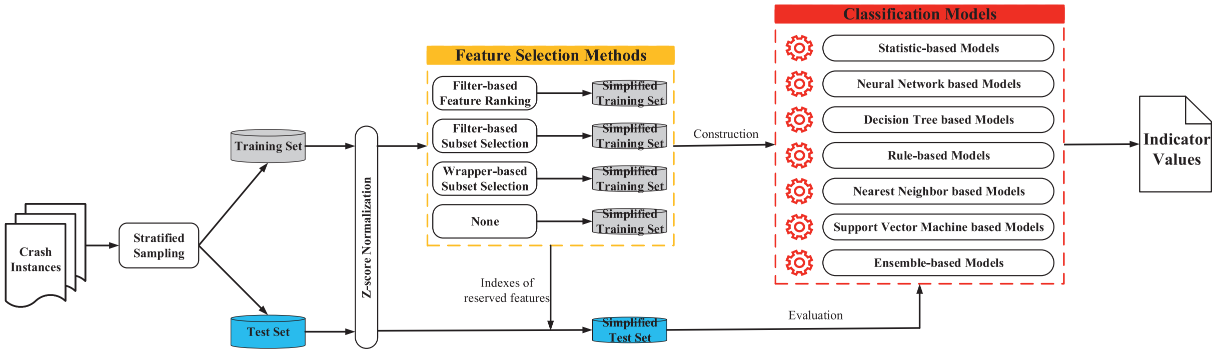

Crash Instances
Stratified Sampling
Training Set
Test Set
Z-score Normalization
Feature Selection Methods
Filter-based Feature Ranking
Filter-based Subset Selection
Wrapper-based Subset Selection
None
Simplified Training Set
Indexes of reserved features
Simplified Test Set
Construction
Classification Models
Statistic-based Models
Neural Network based Models
Decision Tree based Models
Rule-based Models
Nearest Neighbor based Models
Support Vector Machine based Models
Ensemble-based Models
Evaluation
Indicator Values

Fig. 2. The overview of our experimentation.

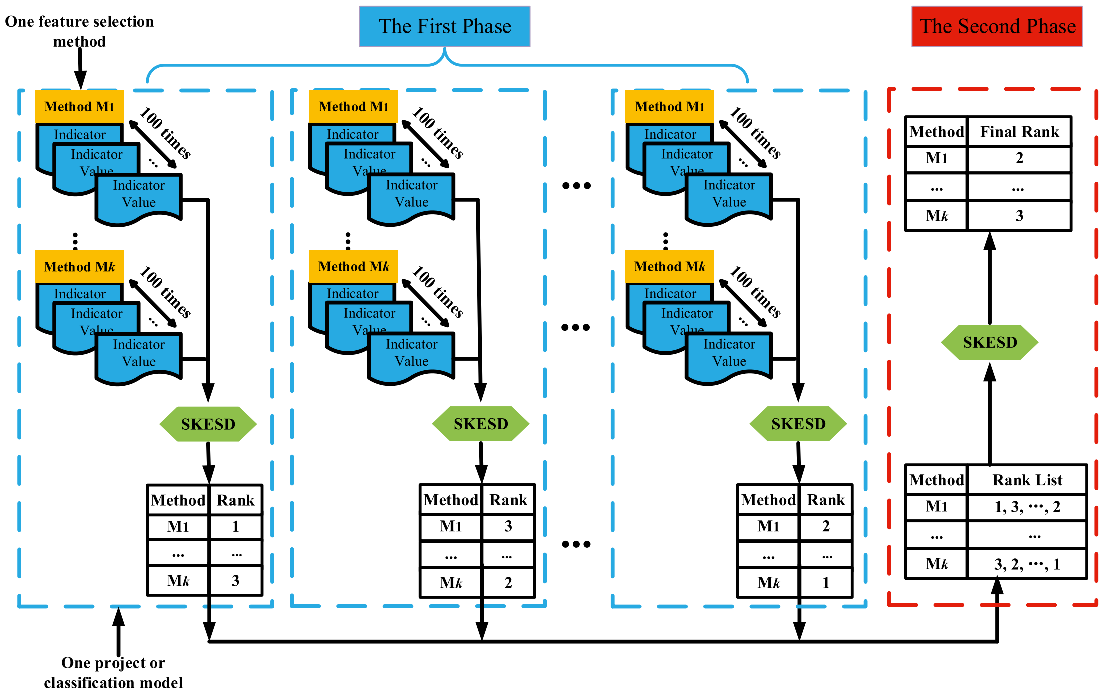


Fig. 3. The framework of SKESD test.

most of classifiers have large differences across these feature selection techniques. In addition, 3 feature selection techniques (including PS (Probabilistic Significance), GR (Gain Ratio), and SU (Symmetrical Uncertainty)) belong to the top-3 SKESD ranking across most of the classification models in which GR appears in the top-3 SKESD ranking for nearly 62% of the studied classification models.

Fourth, in terms of AUC, from Fig. 4(d), we can observe that, one classification model (i.e., RF) always achieves the better performance on nearly all feature selection techniques, while one classification model (i.e., DS) always achieves the worse performance on nearly all feature selection techniques. In addition, one feature selection technique (i.e., CV (Clustering Variation)) achieves the worse performance on all classification models. From Fig. 5(d), the number of SKESD ranking for most of classification models (except for LR, RBF, CART, J48, ADT, DS, RIP, ORC, PDT, and RDR) is equal to or more than 10. This means that the performance results on most of classifiers have large differences across these feature selection techniques. In addition, 3 feature selection techniques (including PS (Probabilistic Significance), GR (Gain Ratio), and SU (Symmetrical Uncertainty)) belong to the top-3 SKESD ranking across most of the classification models in which SU (Symmetrical Uncertainty) appears in the top-3 SKESD ranking for nearly 81% of the studied classification models.

Fifth, from Fig. 4(a)–(d), we observe that most feature selection techniques, except for several methods such as CR (Correlation), CV (Clustering Variation), ReF (ReliefF), and RW (ReliefF-Weight), can achieve similar or even better overall performance across all projects compared with the NONE method that using the original features without any feature selection process on most classification models. This implies that the crash data simplified by most feature selection techniques can maintain and improve the performance of most classification models with fewer features.

Answer: To sum up, probability-based feature ranking technique SU (Symmetrical Uncertainty) outperforms other feature selection techniques, appearing in the top-3 SKESD ranking for 86%, 81%, and 81% of the studied models in terms of F_{IT}, F_{OT}, and AUC, respectively. GR (Gain Ratio) outperforms other feature selection methods, appearing in the top-3 SKESD ranking for 62% of the studied classification models in terms of MCC. Meanwhile, statistic-based feature ranking technique CV (Clustering Variation) obtains the worst rank across all classification models in terms of all 4 indicators.

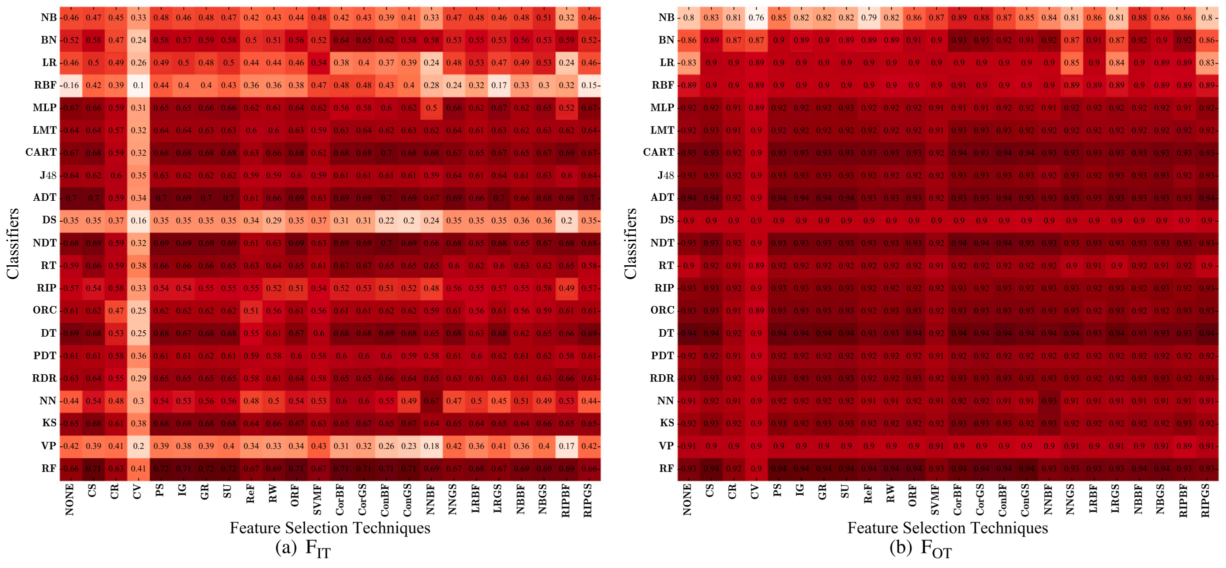
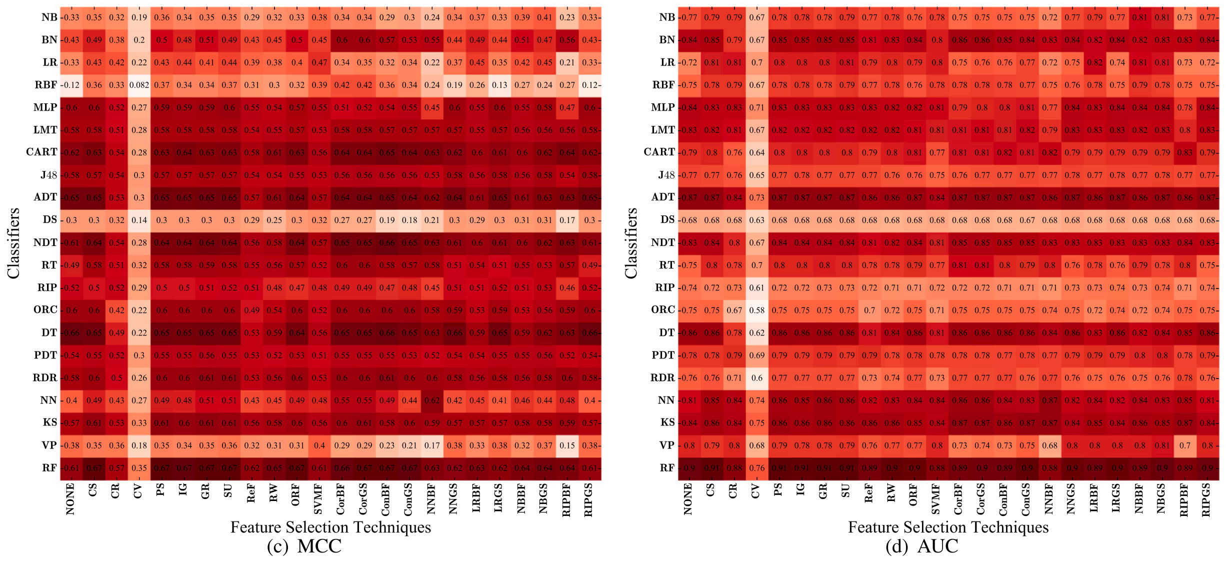



Fig. 4. Mean value of each feature selection technique for each classifier across all projects in terms of the 4 indicators.

5.2. *RQ2: What are the impacts of different feature selection techniques on the performance of each project across all 21 classification models?*

Methods: Since this question investigates the performance impact of different feature selection techniques on each project, to answer this question, we analyze the obtained results at the project level. More specifically, in terms of each indicator, we report its average values of each feature selection technique on each project across all 21 classification models. In addition, we conduct SKESD test on these results and report the ranking values with each indicator individually.

Results: We have a total of 24 × 7 = 168 average results in terms of each indicator for this research question. Fig. 6 depicts the average value of each feature selection technique for each project across all classification models in terms of 4 indicators individually. Fig. 7 demonstrates the corresponding SKESD ranking value of each feature selection technique on each project. From the two figures, we can observe that the methods CS (Chi-Square), PS (Probabilistic Significance), GR (Gain Ratio), and SU (Symmetrical Uncertainty) perform the best on the used indicators overall. More detailed findings are as follows:

First, in terms of F_{IT}, from Fig. 6(a), we can observe that, most of feature selection techniques achieve the better performance on 2 projects (including JSqlParser and Ormlite), while most of feature selection techniques achieve worse performance on 3 projects (including Codec, Jsoup, and Mango). In addition, one feature selection method (i.e., CV (Clustering Variation)) achieves the worse performance on all projects. From Fig. 7(a), the number of SKESD ranking for one project (i.e., Mango) is equal to or more than 10. This means that the performance results on this project have big differences across these feature selection techniques. In addition, 6 feature selection techniques (including CS (Chi-Square), PS (Probabilistic Significance), IG (Information Gain), GR (Gain Ratio), SU (Symmetrical Uncertainty), and SVMF (SVM based Selection)) belong to the top-3 SKESD ranking across more than half of projects in which PS (Probabilistic Significance) appears in the top-3 SKESD ranking on all studied projects.

Second, in terms of F_{OT}, from Fig. 6(b), we can observe that, most of feature selection techniques achieve the better performance on 2 projects (including JSqlParser and Mango), while most of feature selection techniques achieve worse performance on another 2 projects (including Codec and Jsoup). In addition, one feature selection technique (i.e., CV (Clustering Variation)) achieves the worse performance on all projects. From Fig. 7(b), the number of SKESD ranking for 3 projects (including Collections, Mango, and Ormlite) is equal to or more than 10. This means that the performance results on the 3 projects have

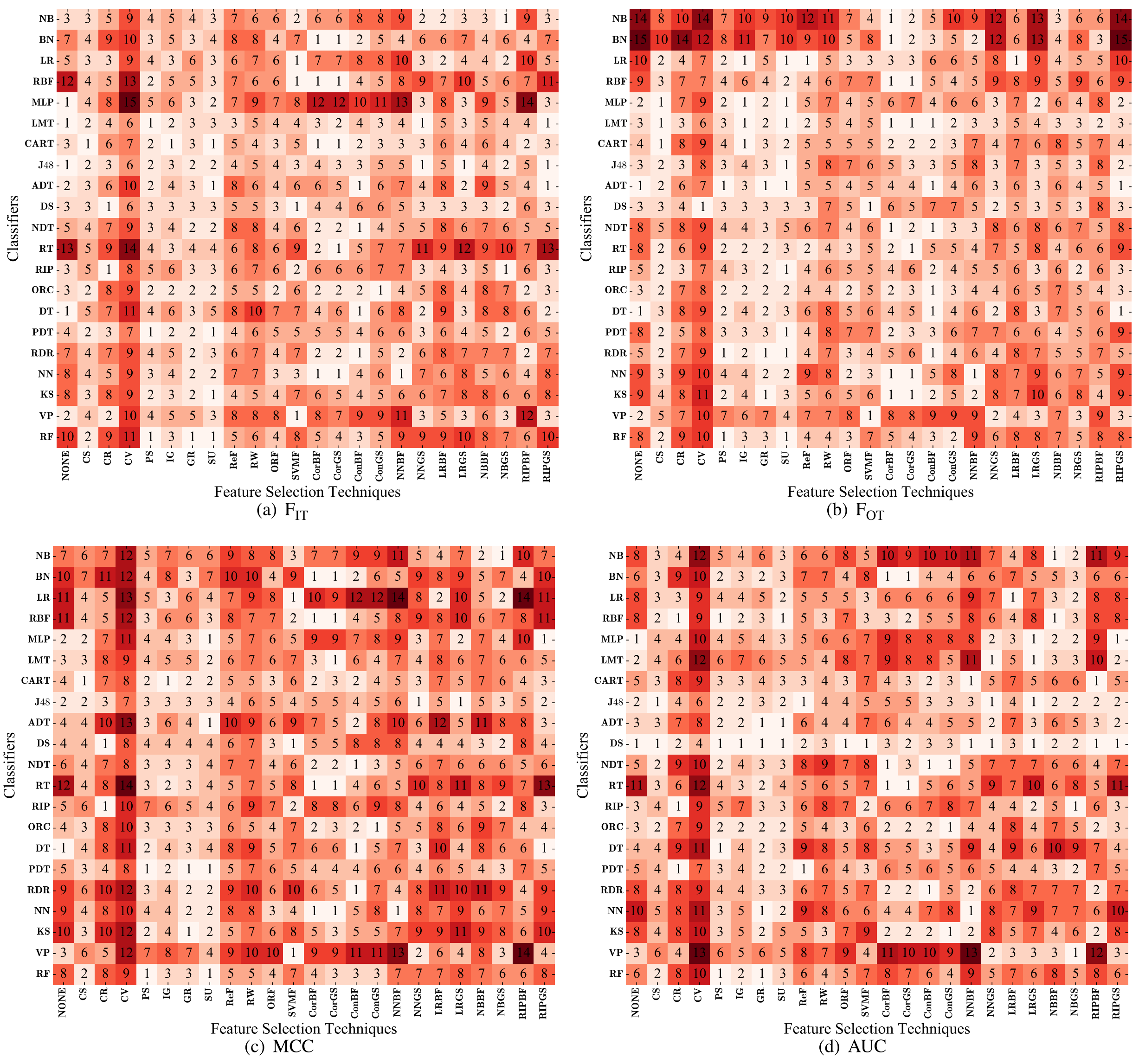


Fig. 5. SKESD ranking of each feature selection technique on each classifier across all projects in terms of the 4 indicators.

big differences across these feature selection techniques. In addition, 7 feature selection techniques (including CS (Chi-Square), PS (Probabilistic Significance), GR (Gain Ratio), SU (Symmetrical Uncertainty), CorBF (Correlation-based Feature Subset Selection with Best First), CorGS (Correlation-based Feature Subset Selection with GreedyStepwise), and ConBF (Consistency-based Feature Subset Selection with Best First)) belong to the top-3 SKESD ranking across more than half of projects, in which all the techniques appear in the top-3 SKESD ranking on nearly 71% of all studied projects.

Third, in terms of MCC, from Fig. 6(c), we can observe that, most of feature selection techniques achieve the better performance on 2 projects (including JSqlParser and Ormlite), while most of feature selection techniques achieve worse performance on another 2 projects (including Codec and Jsoup). In addition, one feature selection technique (i.e., CV (Clustering Variation)) achieves the worse performance on all projects. From Fig. 7(c), the number of SKESD ranking for most of projects (except for Codec and JSqlParser) is equal to or more than 10. This means that the performance results on these projects have big differences across these feature selection techniques. In addition, 6 feature selection techniques (including CS (Chi-Square), PS (Probabilistic Significance), IG (Information Gain), GR (Gain Ratio), SU (Symmetrical Uncertainty), and SVMF (SVM based Selection)) belong to the top-3 SKESD ranking across more than half of projects in which SU (Symmetrical Uncertainty) appears in the top-3 SKESD ranking on all studied projects.

Fourth, in terms of AUC, from Fig. 6(d), we can observe that, most of feature selection techniques achieve the better performance on 2 projects (including JSqlParser and Ormlite), while most of feature selection techniques achieve worse performance on another 2 projects (including Codec and Jsoup). In addition, one feature selection technique (i.e., CV (Clustering Variation)) achieves the worse performance on all projects. From Fig. 7(d), the number of SKESD ranking for 2 projects (including Collections and JSqlParser) is equal to or more than 10. This means that the performance results on the 2 projects have big differences across these feature selection techniques. In addition, 6 feature selection methods (including CS (Chi-Square), PS (Probabilistic Significance), IG (Information Gain), GR (Gain Ratio), SU (Symmetrical Uncertainty), and ReF (ReliefF)) belong to the top-3 SKESD ranking across more than half of projects in which CS (Chi-Square) and SU (Symmetrical Uncertainty) appear in the top-3 SKESD ranking on nearly 86% of all studied projects.

Fifth, from Fig. 6(a)–(d), we observe that most feature selection techniques, except for several methods such as CR (Correlation) and

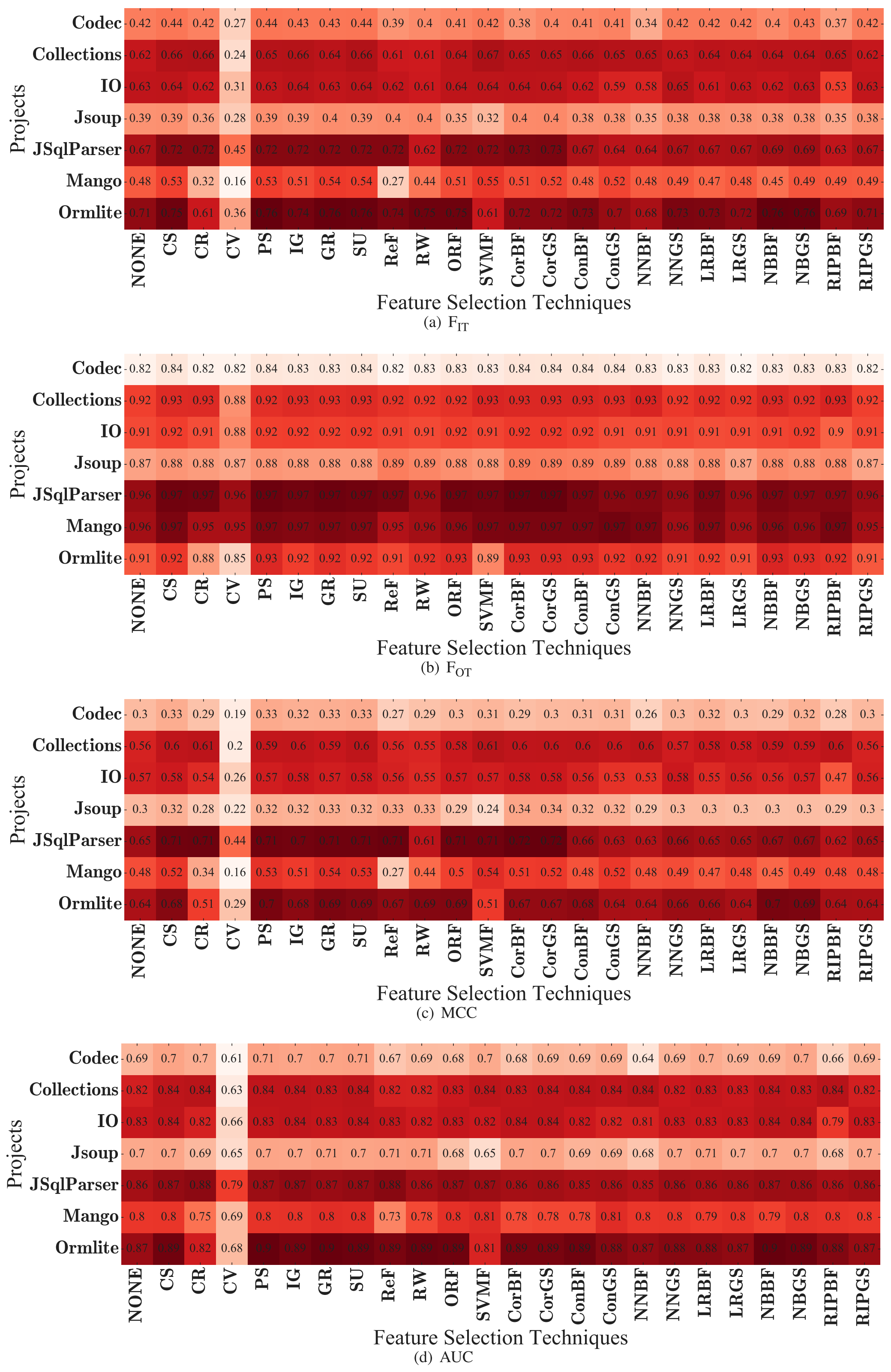
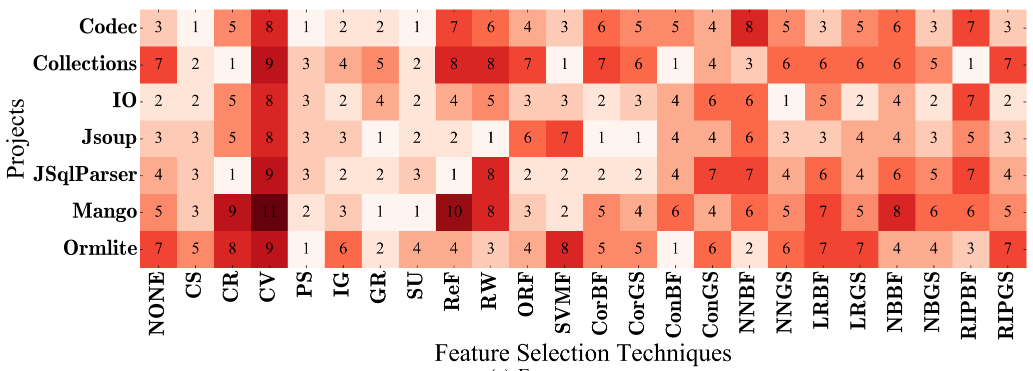


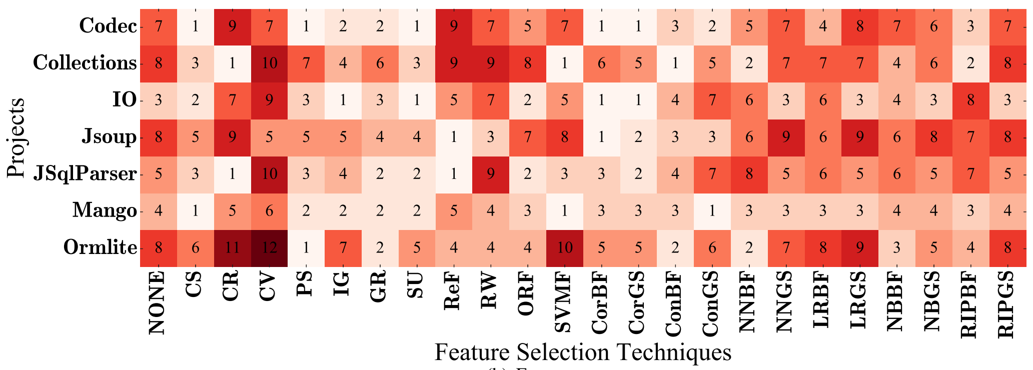
Fig. 6. Mean value of each feature selection technique on each project across all classification models in terms of the 4 indicators.

CV (Clustering Variation), can achieve similar or even better overall performance across all classification models compared with the NONE method that using the original features without any feature selection process on most projects. This implies the effectiveness of most feature selection techniques across all classification models when applied to most projects for the crashing fault residence prediction task.

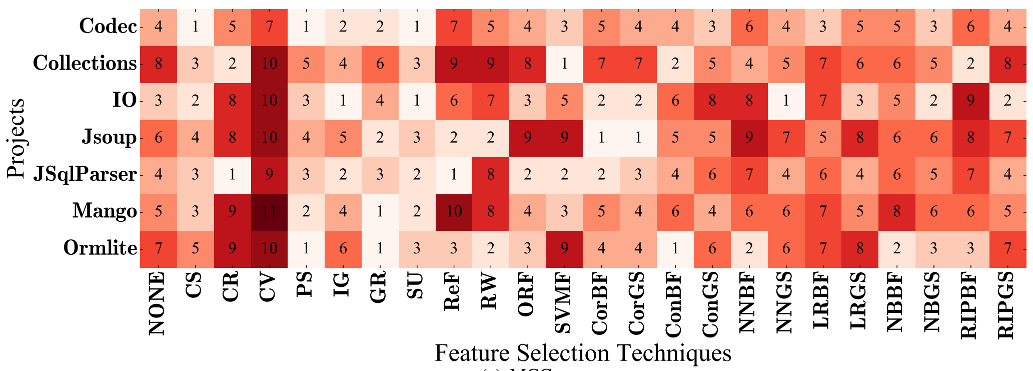
Answer: In summary, 4 feature ranking techniques, i.e., CS (Chi-Square), PS (Probabilistic Significance), GR (Gain Ratio), and SU (Symmetrical Uncertainty), obtain the outstanding performance, in which CS (Chi-Square) appears in the top-3 SKESD ranking on 86%, 71%, 71%, and 86% of all studied projects in terms of 4 indicators, respectively. PS (Probabilistic Significance) appears in the top-3 SKESD ranking



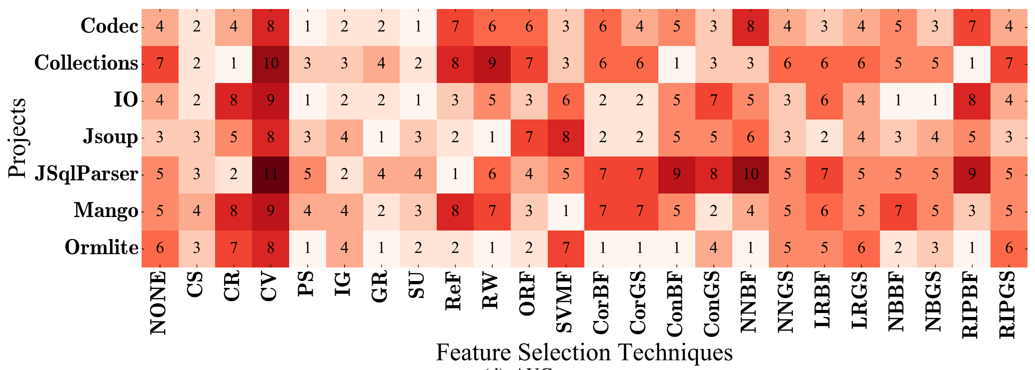
(a) F_{IT}



(b) F_{OT}



(c) MCC



(d) AUC

Fig. 7. SKESD ranking of each feature selection technique on each project across all classification models in terms of the 4 indicators.

on 100%, 71%, 71%, and 71% of all studied projects in terms of 4 indicators respectively. GR (Gain Ratio) appears in the top-3 SKESD ranking on 71%, 71%, 71%, and 71% of all studied projects in terms of 4 indicators respectively. SU (Symmetrical Uncertainty) appears in the top-3 SKESD ranking on 86%, 71%, 100%, and 86% of all studied projects in terms of 4 indicators respectively. Meanwhile, statistic-based feature ranking technique CV (Clustering Variation) nearly obtains the worse ranking across all studied projects in terms of all 4 indicators.

6. Threats to validity

6.1. Internal validity

Threats to internal validity come from the implementation mistakes during our experiments. To relieve these threats, in this work, we take full use of the off-the-shelf implementation provided by third-party libraries to implement the studied 24 feature selection techniques and 21 classification models to avoid the potential mistakes.

6.2. External validity

Threats to external validity lie in the generalization of our study results. In this work, we conduct experiments on a benchmark dataset including 7 open-source projects. Since the crash data in these projects generated by the program mutation tool to mimic the real-world ones, which cannot completely substitute the actual crashes. We will collect real crash data and conduct experiments on them to enhance the generalization of our work. In addition, the applied feature selection techniques come from 4 families and the classification models come from 7 families, which enables the diversity and representativeness of the research objects. This helps to improve the generalization ability of the results. Another threat to external validity is that, in this work, we do not consider the class imbalance issue of the data which may affect our experimental results. We plan to do a special study that focuses on the impact of different class imbalance methods on the performance of crashing fault residence prediction model.

6.3. Construct validity

Threats to construct validity focus on the reasonability of the used performance indicators and the statistic test method. In this work, we apply 4 indicators to comprehensively evaluate the performance for crashing fault residence prediction, including F_{IT}, F_{OT}, MCC, and AUC. We also employ the state-of-the-art SKESD test to analyze the significant differences of these feature selection techniques. These can make the analysis of our experimental results more rigorous.

7. Conclusion

Predicting whether the crashing fault resides in the stack trace can assist the crash localization process by reducing the corresponding search space and prioritizing the testing efforts. In this work, we conduct a large-scale empirical study to explore how the 24 feature selection techniques impact on the performance of 21 classification models for the crashing fault residence prediction task with 4 evaluation indicators on a benchmark dataset including 7 open-source software projects. In addition, we analyze the experimental results with SKESD test on both the classification model level and project level. Our findings are that

- On classification model level, a probability-based feature selection method (i.e., SU (Symmetrical Uncertainty)) performs better than other feature selection methods, appearing in the top-3 SKESD ranking for more than 80% of the studied classification models in terms of 3 indicators (i.e., F_{IT}, F_{OT}, and AUC).
- On project level, one statistic-based feature selection method (i.e., CS (Chi-Square)) and three probability-based feature selection methods (i.e., PS (Probabilistic Significance), GR (Gain Ratio), and SU (Symmetrical Uncertainty)) achieve better performance than other feature selection methods, appearing in the top-3 SKESD ranking for more than 70% of the studied project in terms of all 4 indicators.
- CV (Clustering Variation, a statistic-based feature selection method) is not sensitive to the studied classification models and projects as it always achieves the worse performance at both the classification model level and the project level.
- Two decision tree based classification models (i.e., ADT and DT) and one ensemble-based classification models (i.e., RF) present the superiority compared with other models across all 4 indicators. This finding is consistent with Gu et al.'s [2] which also showed that a decision tree based classifier performs the best.
- The impact of feature selection techniques varies across classification models and projects. These feature selection techniques are divided into several groups with significant differences on some studied classification models or projects, whereas there is a clear significant differences with many groups among these methods when applied to some other studied classification models or projects.
- Not all feature selection methods help improve the performance for the crashing fault residence prediction task. For example, the NONE method outperforms the CV method in most case on both classification model level and project level. Thus, researchers and practitioners should carefully select the appropriate feature selection methods to preprocess the crash instance data for performance improvements.

This work is just a preliminary exploration to find out the most appropriate feature selection techniques for the crashing fault residence prediction task. In the future, we plan to expand the experiments to other feature selection techniques, such as hybrid based methods and feature extraction methods, on the data of real crashes.

CRediT authorship contribution statement

Kunsong Zhao: Writing - original draft, Methodology, Software, Data curation. **Zhou Xu:** Supervision, Project administration. **Meng Yan:** Formal analysis, Visualization. **Tao Zhang:** Conceptualization, Writing - review & editing. **Dan Yang:** Funding acquisition. **Wei Li:** Writing - review & editing.

Declaration of competing interest

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

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (No. 62002034), the Fundamental Research Funds for the Central Universities, China (Nos.2020CDCGRJ072, 2020CDJQY-A021, and JUSRP121073), China Postdoctoral Science Foundation (No. 2020M673137), the Natural Science Foundation of Chongqing in China (No.cstc2020jcyj-bshX0114), the Science and Technology Development Fund of Macau (No.0047/2020/A1), Faculty Research Grant Projects of MUST (No.FRG-20-008-FI).

References

[1] Z. Xu, K. Zhao, M. Yan, P. Yuan, L. Xu, Y. Lei, X. Zhang, Imbalanced metric learning for crashing fault residence prediction, J. Syst. Softw. (JSS) (2020) 110763.

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

[3] Z. Xu, T. Zhang, J. Keung, M. Yan, X. Luo, X. Zhang, L. Xu, Y. Tang, Feature selection and embedding based cross project framework for identifying crashing fault residence, Inf. Softw. Technol. (IST) (2020) 106452.

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

[5] N. Chen, S. Kim, Star: Stack trace based automatic crash reproduction via symbolic execution, IEEE Trans. Softw. Eng. (TSE) 41 (2) (2014) 198–220.

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

[7] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, A. Larsson, JCHARMING: A bug reproduction approach using crash traces and directed model checking, in: Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE, 2015, pp. 101–110.

[8] M. Soltani, A. Panichella, A. Van Deursen, A guided genetic algorithm for automated crash reproduction, in: Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE), IEEE, 2017, pp. 209–220.

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

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

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

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

[13] K. Gao, T.M. Khoshgoftaar, H. Wang, An empirical investigation of filter attribute selection techniques for software quality classification, in: 2009 IEEE International Conference on Information Reuse & Integration, IEEE, 2009, pp. 272–277.

[14] K. Muthukumaran, A. Rallapalli, N.B. Murthy, Impact of feature selection techniques on bug prediction models, in: Proceedings of the 8th India Software Engineering Conference, 2015, pp. 120–129.

[15] K. Gao, T.M. Khoshgoftaar, H. Wang, N. Seliya, Choosing software metrics for defect prediction: an investigation on feature selection techniques, Softw. - Pract. Exp. 41 (5) (2011) 579–606.

[16] H. Wang, T.M. Khoshgoftaar, A. Napolitano, A comparative study of ensemble feature selection techniques for software defect prediction, in: 2010 Ninth International Conference on Machine Learning and Applications, IEEE, 2010, pp. 135–140.

[17] H. Wang, T.M. Khoshgoftaar, J. Van Hulse, K. Gao, Metric selection for software defect prediction, Int. J. Softw. Eng. Knowl. Eng. (IJSEKE) 21 (02) (2011) 237–257.

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

[19] J. Stuckman, J. Walden, R. Scandariato, The effect of dimensionality reduction on software vulnerability prediction models, IEEE Trans. Reliab. 66 (1) (2017) 17–37.

[20] B. Ghotra, S. McIntosh, A.E. Hassan, A large-scale study of the impact of feature selection techniques on defect classification models, in: Proceedings of the 14th IEEE/ACM International Conference on Mining Software Repositories (MSR), IEEE, 2017, pp. 146–157.

[21] M. Kondo, C.-P. Bezemer, Y. Kamei, A.E. Hassan, O. Mizuno, The impact of feature reduction techniques on defect prediction models, Empir. Softw. Eng. (EMSE) 24 (4) (2019) 1925–1963.

[22] J. Jiarpakdee, C. Tantithamthavorn, C. Treude, The impact of automated feature selection techniques on the interpretation of defect models, Empir. Softw. Eng. (EMSE) 25 (5) (2020) 3590–3638.

[23] A.O. Balogun, S. Basri, S.J. Abdulkadir, A.S. Hashim, Performance analysis of feature selection methods in software defect prediction: a search method approach, Appl. Sci. 9 (13) (2019) 2764.

[24] Q. Yu, J. Qian, S. Jiang, Z. Wu, G. Zhang, An empirical study on the effectiveness of feature selection for cross-project defect prediction, IEEE Access 7 (2019) 35710–35718.

[25] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, Benchmarking classification models for software defect prediction: A proposed framework and novel findings, IEEE Trans. Softw. Eng. (TSE) 34 (4) (2008) 485–496.

[26] C. Seiffert, T.M. Khoshgoftaar, J. Van Hulse, A. Folleco, An empirical study of the classification performance of learners on imbalanced and noisy software quality data, Inform. Sci. 259 (2014) 571–595.

[27] B. Ghotra, S. McIntosh, A.E. Hassan, Revisiting the impact of classification techniques on the performance of defect prediction models, in: Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE), 1, IEEE, 2015, pp. 789–800.

[28] C. Tantithamthavorn, S. McIntosh, A.E. Hassan, K. Matsumoto, The impact of automated parameter optimization on defect prediction models, IEEE Trans. Softw. Eng. (TSE) 45 (7) (2018) 683–711.

[29] X. Chen, Y. Zhao, Z. Cui, G. Meng, Y. Liu, Z. Wang, Large-scale empirical studies on effort-aware security vulnerability prediction methods, IEEE Trans. Reliab. 69 (1) (2019) 70–87.

[30] M. Aniche, E. Maziero, R. Durelli, V. Durelli, The effectiveness of supervised machine learning algorithms in predicting software refactoring, 2020, arXiv Preprint arXiv:2001.03338.

[31] X. Chen, D. Zhang, Y. Zhao, Z. Cui, C. Ni, Software defect number prediction: Unsupervised vs supervised methods, Inf. Softw. Technol. (IST) 106 (2019) 161–181.

[32] X. Chen, Z. Yuan, Z. Cui, D. Zhang, X. Ju, Empirical studies on the impact of filter-based ranking feature selection on security vulnerability prediction, IET Softw. 15 (1) (2021) 75–89.

[33] H. Liu, R. Setiono, Chi2: Feature selection and discretization of numeric attributes, in: Proceedings of 7th IEEE International Conference on Tools with Artificial Intelligence, IEEE, 1995, pp. 388–391.

[34] I. Guyon, A. Elisseeff, An introduction to variable and feature selection, J. Mach. Learn. Res. 3 (Mar) (2003) 1157–1182.

[35] S. Fong, J. Liang, R. Wong, M. Ghanavati, A novel feature selection by clustering coefficients of variations, in: Proceedings of the 9th International Conference on Digital Information Management, IEEE, 2014, pp. 205–213.

[36] A. Ahmad, L. Dey, A feature selection technique for classificatory analysis, Pattern Recognit. Lett. 26 (1) (2005) 43–56.

[37] T.M. Cover, Elements of Information Theory, John Wiley & Sons, 1999.

[38] J.R. Quinlan, C4. 5: Programs for Machine Learning, Elsevier, 2014.

[39] S.S. Kannan, N. Ramaraj, A novel hybrid feature selection via symmetrical uncertainty ranking based local memetic search algorithm, Knowl.-Based Syst. 23 (6) (2010) 580–585.

[40] I. Kononenko, Estimating attributes: analysis and extensions of RELIEF, in: European Conference on Machine Learning, Springer, 1994, pp. 171–182.

[41] R.C. Holte, Very simple classification rules perform well on most commonly used datasets, Mach. Learn. 11 (1) (1993) 63–90.

[42] I. Guyon, J. Weston, S. Barnhill, V. Vapnik, Gene selection for cancer classification using support vector machines, Mach. Learn. 46 (1–3) (2002) 389–422.

[43] M.A. Hall, Correlation-Based Feature Selection of Discrete and Numeric Class Machine Learning, University of Waikato, Department of Computer Science, 2000.

[44] M. Dash, H. Liu, H. Motoda, Consistency based feature selection, in: Pacific-Asia Conference on Knowledge Discovery and Data Mining, Springer, 2000, pp. 98–109.

[45] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, L. Seinturier, Spoon: A library for implementing analyses and transformations of java source code, Softw. - Pract. Exp. 46 (9) (2016) 1155–1179.

[46] Y. Fan, X. Xia, D. Lo, A.E. Hassan, Chaff from the wheat: Characterizing and determining valid bug reports, IEEE Trans. Softw. Eng. (TSE) (2018).

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

[48] C. Fang, Z. Liu, Y. Shi, J. Huang, Q. Shi, 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), 2020, pp. 516–527.

[49] J. Han, J. Pei, M. Kamber, Data Mining: Concepts and Techniques, Elsevier, 2011.

[50] X. Wu, V. Kumar, The Top Ten Algorithms in Data Mining, CRC press, 2009.

[51] S.B. Kotsiantis, I. Zaharakis, P. Pintelas, Supervised machine learning: A review of classification techniques, Emerg. Artif. Intell. Appl. Comput. Eng. 160 (1) (2007) 3–24.

[52] Y. Singh, A.S. Chauhan, Neural networks in data mining, J. Theor. Appl. Inf. Technol. 5 (1) (2009).

[53] A.J. Myles, R.N. Feudale, Y. Liu, N.A. Woody, S.D. Brown, An introduction to decision tree modeling, J. Chemom. J. Chemom. Soc. 18 (6) (2004) 275–285.

[54] M. Shepperd, Q. Song, Z. Sun, C. Mair, Data quality: Some comments on the nasa software defect datasets, IEEE Trans. Softw. Eng. (TSE) 39 (9) (2013) 1208–1215.

[55] J. Fürnkranz, Separate-and-conquer rule learning, Artif. Intell. Rev. 13 (1) (1999) 3–54.

[56] T. Cover, P. Hart, Nearest neighbor pattern classification, IEEE Trans. Inform. Theory 13 (1) (1967) 21–27.

[57] N. Cristianini, J. Shawe-Taylor, et al., An Introduction To Support Vector Machines and Other Kernel-Based Learning Methods, Cambridge university press, 2000.

[58] Z.-H. Zhou, Ensemble learning, Encycl. Biom. 1 (2009) 270–273.

[59] S. Shivaji, E.J. Whitehead Jr, R. Akella, S. Kim, Reducing features to improve bug prediction, in: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2009, pp. 600–604.

[60] S. Shivaji, E.J. Whitehead, R. Akella, S. Kim, Reducing features to improve code change-based bug prediction, IEEE Trans. Softw. Eng. (TSE) 39 (4) (2012) 552–569.

[61] C. Tantithamthavorn, S. McIntosh, A.E. Hassan, K. Matsumoto, An empirical comparison of model validation techniques for defect prediction models, IEEE Trans. Softw. Eng. (TSE) 43 (1) (2016) 1–18.

[62] F. Zhang, Q. Zheng, Y. Zou, A.E. Hassan, Cross-project defect prediction using a connectivity-based unsupervised classifier, in: Proceedings of the 38th IEEE/ACM International Conference on Software Engineering (ICSE), IEEE, 2016, pp. 309–320.

[63] E.G. Jelihovschi, J.C. Faria, I.B. Allaman, ScottKnott: a package for performing the Scott-Knott clustering algorithm in R, TEMA (SãO Carlos) 15 (1) (2014) 3–17.