Received: 14 March 2022 | Revised: 7 July 2022 | Accepted: 30 August 2022

DOI: 10.1049/sfw2.12073

IET Software

The Institution of Engineering and Technology WILEY

ORIGINAL RESEARCH

An unsupervised cross project model for crashing fault residence identification

Xiao Liu[1,2] | Zhou Xu[1,2] | Dan Yang[1,2] | Meng Yan[1,2] | Weihan Zhang[2] | Haohan Zhao[2] | Lei Xue[3] | Ming Fan[1,4]

[1]Key Laboratory of Dependable Service Computing in Cyber Physical Society, Ministry of Education, Chongqing University, Chongqing, China

[2]School of Big Data and Software Engineering, Chongqing University, Chongqing, China

[3]School of Cyber Science and Technology, Sun Yat-Sen University, Shenzhen, China

[4]Ministry of Education Key Laboratory of Intelligent Networks and Network Security, Xi'an Jiaotong University, Xi'an, Shaanxi, China

Correspondence

Zhou Xu and Meng Yan, Key Laboratory of Dependable Service Computing in Cyber Physical Society, Ministry of Education, Chongqing University, Chongqing 400044, China.
Email: zhouxullx@cqu.edu.cn and mengy@cqu.edu.cn

Funding information

Open Foundation of Key Laboratory of Dependable Service Computing in Cyber Physical Society, Ministry of Education of China, Grant/Award Number: CPSDSC202004; National Key Research and Development Project, Grant/Award Number: 2021YFB1714200; National Nature Science Foundation of China, Grant/Award Number: 62002034; Fundamental Research Funds for the Central Universities, Grant/Award Numbers: 2022CDJKYJH001, xxj022019001, xzy012020009; Natural Science Foundation of Chongqing, Grant/Award Number: cstc2021jcyj-msxmX0538

Abstract

It is a critical quality assurance activity to effectively detect the root cause of faults causing the software crashes (i.e. crashing faults). Previous studies extracted features to characterise crash instances and built models to identify whether the residences of crashing faults locate inside the stack traces. These models all belong to supervised learning methods which require labelled crash data to be involved. In this study, the introduction of an unsupervised model, called **T**ransfer **S**pectral **C**lustering (**TSC**), for the task of crashing fault residence identification under the unlabelled data scenario is proposed. Unlike traditional unsupervised methods which are applied to individual project data, TSC transfers the knowledge of auxiliary unlabelled data from the source project to assist the clustering task on the unlabelled data from the target project. TSC is an unsupervised transfer learning method, and simultaneously considers the data manifold information of the individual project and feature manifold information across projects to facilitate the clustering effect. Extensive experiments are conducted on a benchmark dataset containing seven software projects. Five indicators were chosen for performance evaluation. The results show that TSC achieves better performance than four clustering based unsupervised methods, and competitive performance compared with eight supervised cross-project methods.

1 | INTRODUCTION

As the coming of information modernisation, software plays an essential role in our daily life with the rapid popularity of computer and mobile devices. However, in the process of software development and maintenance, programmers usually encounter software faults due to the surging complexity of modern software or some unanticipated factors [1]. The faults may cause the software to crash which is a kind of unexpected interruption of software [2]. In this case, it will take programmers a large amount of efforts to deal with the crashes through finding their root cause, which is a time and labour consuming process. Fortunately, some crash-related report information, such as stack traces recorded by the crash reporting system that most modern software is equipped with, is able to help programmers or testers to find out faults causing the crashes (short for crashing faults). Effectively identifying the residences of crashing faults is a critical software quality assurance activity which could facilitate the process of fixing the faulty source code and save time. In a previous study [3], this process is called **I**dentification of **C**rashing **F**ault **R**esidence (**ICFR**).

Recently, the information from stack traces is used to locate the residence of crashing faults. The reason is that the

stack traces contain a wealth of information, such as the type of crash exception as well as the function invocation sequence, which are associated with the crashing faults. In other words, the information provided by the stack traces can help find the root cause or residence of the crashing faults. As the most relevant and earliest study towards our work, Gu et al. [2] proposed an automatic crashing fault localisation method called CraTer which first extracted a set of self-defined features from the stack traces and source code, and then conducted a binary classification task based on these features and collected labels. Their goal is to predict whether new-submitted crashing faults locate in the stack traces. More specifically, the task is to determine whether the faulty code causing the crash matches one of code snippets of the code call sequence in the stack trace. If the crashing fault residence locates inside the stack trace, programmers only need to check the related code recorded in the stack trace. In this case, it will greatly improve the efficiency of debugging software for developers. If the crashing fault residence locates outside the stack trace, programmers need to check the function call sequence. In this case, it involves in examining a large amount of code to find the root cause of the crashing fault. This will waste a mass of limited test resources, which seriously hinders the debugging efficiency. Following Gu et al.'s work and using their shared dataset, Xu et al. [3] and Zhao et al. [4] proposed a series of methods to improve the residence identification performance for the crashing fault. By analysing these existing methods, they can roughly fall into two categories. One is within project identification, in which the training set and test set are from the crash data of the same project. The other one is cross-project identification, in which the training set comes from the crash data of external projects. However, the methods in these two scenarios all belong to supervised learning methods which need labelled crash data as the training set to participate the model construction process.

Nevertheless, the collection process of label information for the crash data is time-consuming, labour-intensive, and error-prone, but the unlabelled data are easily available. This inspires us to come up with a way to conduct ICFR on the unlabelled crash data. Previous studies used unsupervised learning methods, such as clustering algorithm or ranking models, to fulfil some tasks in the software engineering domain, such as defect prediction [5, 6] and test suite reduction [7]. However, these unsupervised methods are all applied to individual projects. This basic idea is simple, that is, using a clustering algorithm to group the unlabelled data from a project into two or multiple clusters. However, for software engineering tasks, considering the unlabelled data from other projects when applying the unsupervised methods to the unlabelled data from the at-hand project has not yet been explored.

In this work, we make a first attempt to explore whether the unlabelled data from the external project (aka the source project) can help promote the performance of unsupervised learning method when it is performed on the unlabelled data from the current project (aka the target project) for the ICFR task. In other words, this work aims at proposing a novel unsupervised methods for the ICFR task by combining the unlabelled data from not only the target project but also the source project, expecting better performance than this method alone on the unlabelled data from the target project. In this work, we introduce a novel unsupervised method based on the transfer learning strategy called **T**ransfer **S**pectral **C**lustering (**TSC**) [8] for this purpose. This method transfers the knowledge of auxiliary unlabelled crash data from the source project to augment the clustering effect on the unlabelled crash data from the target project. The advantage of the used TSC method is that it simultaneously considers the data manifold information of the ICFR task on the crash data of individual projects, and the feature manifold information of the shared ICFR task on the crash data across two projects. Therefore, the TSC method can not only alleviate the widespread scarcity problem of the data labels, but also can take advantage of the knowledge of unlabelled data from other projects with the same task to further improve clustering performance. In view of the merits of the TSC method, we select it as our unsupervised cross-project ICRF model.

To verify the effectiveness of the TSC method, we design two research questions as follows. *RQ1: Does our TSC method perform better than the unsupervised learning models over single project for ICFR task?* This question focuses on exploring whether our unsupervised learning method TSC can achieve better performance than other unsupervised learning methods for ICFR task. *RQ2: Is our TSC method superior to some classic supervised cross-project models for ICFR task?* This question is to investigate whether our unsupervised cross-project model can acquire similar or even better performance compared to the supervised cross-project models for this task.

To verify the effectiveness of the TSC method for ICFR task under both unlabelled data and the cross-project scenario, we conduct experiments on a publicly available benchmark dataset [2] consisting of crash data from seven open source software projects. This dataset simulates software crashes through seeding faults with program mutation since collecting and reproducing real-word crashes is not a trivial matter. We compare the TSC method with four clustering based unsupervised learning baseline methods which use only the unlabelled crash data from the target project and eight supervised cross-project baseline models which work on the assumption that the labels of crash data from the source project are known. Five performance indicators are carefully selected to comprehensively evaluate the effectiveness of our TSC based ICFR method and the 12 baseline methods. The detailed experimental results show that our TSC method achieves improvement of 23.8%, 27.8%, 17.6%, 12.3%, and 26.9% in terms of F-measure, G-measure, g-mean, Balance, and MCC compared with the best corresponding indicators of the four unsupervised baseline methods respectively. In addition, our TSC method achieves better performance than the eight supervised cross-project baseline models in terms of F-measure and g-mean, and competitive performance in terms of G-measure and Balance compared with one baseline method, but lower performance in terms of MCC than one baseline method. Overall, our TSC method utilising information of auxiliary

unlabelled crash data indeed performs better than the unsupervised learning methods which are applied only to unlabelled crash data of the individual project without auxiliary information. Besides, although our unsupervised cross-project TSC method is not fully superior to the supervised cross-project models, it is still competitive since it does not require any labelled crash data. In that sense, the performance results of the TSC method are not so unsatisfactory, that is, the results are acceptable.

In summary, we have the following main contributions for this work:

(1) In this work, we were the first to study the ICFR problem under the completely no label scenario using a unsupervised cross-project model to address the label scarcity issue, in which the labelled crash data are not always available for both the external project and the current project.
(2) By introducing an advanced TSC method which incorporates the transfer learning and clustering algorithm, we utilise the auxiliary unlabelled crash data from the source project to make the unlabelled crash data from the target project produce more effective clustering results. To the best of our knowledge, TSC is the first unsupervised cross-project model for the ICFR task.
(3) We evaluate the effectiveness of the used unsupervised cross-project ICFR model (i.e. TSC) on seven open-source software projects with five indicators. The experimental results show that TSC achieves promising performance over four unsupervised non-cross-project baseline methods on individual project and acceptable performance over eight supervised cross-project models.

The remainder of this paper is organised as follows. Section 2 briefly introduces the related work. Section 3 introduces the fundamentals of our work. Section 4 describes the details of our TSC based cross-project ICFR method. Section 5 presents our experimental setup. Section 6 reports and analyses the experimental results. Section 7 lists the potential threats to validity of our work. Finally, Section 8 concludes our work.

2 | RELATED WORK

2.1 | Stack trace analysis

Crash, a serious software failure, is an event when the program itself stops or aborts unexpectedly. It is usually caused by faults in the software. Modern software generally equips the crash reporting system which can automatically generate the crash report when a crash occurs. The crash report mainly contains the stack trace information of the crash, that is, function invocation sequences when software executing. As the stack traces can be utilised to reproduce the crash, it can help to understand the root causes of crashes [9].

The stack trace is composed of a set of frame objects, that is, an initial frame object which describes the exception of crash and other frame objects which include the function call sequences. The top one is called the first frame and the bottom one is called the last frame in this work. Each frame except for the initial frame mainly consists of the class name, function name, and the code line number which describes the location of the execution point. Some frames also contain information about the function arguments.

Previous studies analysed stack traces for various tasks, for example, crash report clustering [10, 11], crash reproduction [9, 12–14], and crash residence identification [2, 15].

2.2 | Crash fault residence identification

Gu et al. [2] were the first to explore this issue by proposing an automatic method CraTer. They constructed a training data based on known crashes by extracting a set of features for instance representation. They used sampling strategy to relief the class imbalanced issue and employed six classifiers as the classification model. The built model is used to identify the residence of the newly-submitted crashes. By utilising this collected benchmark dataset, Xu et al. [3] extended Gu et al's work to cross-project scenario by utilising a state-of-the-art feature embedding based transfer learning model to alleviate the data distribution difference across projects and also considering their corresponding weights. Then, they further extended their own work by adding a feature selection stage to preprocess the crash data before performing the feature embedding [15]. Zhao et al. [4] employed a two-step framework for ICFR task by first using a consistency based feature selection method to obtain a reduced feature set, then introducing a simplified deep forest method to construct classification model. Xu et al. [16] applied an advanced imbalanced metric learning method to address the class imbalanced issue of the crash data. Zhao et al. [17] conducted a comprehensive investigation to explore the impact of 24 feature selection methods on 21 classification models for the ICFR task.

Though these studies focussed on the same task as our work, that is, identifying whether the residence of crashing fault falls in the stack traces or not, the difference between them and our study is that the proposed models in these work all belong to the supervised within project or supervised cross-project model while our method is an unsupervised cross-project model. To our best knowledge, our work was the first to introduce such a model to solve the ICFR task.

2.3 | Root cause analysis via machine learning techniques

Lal et al. [18] proposed a machine learning based approach to find the root cause of bugs in newly developed software. They conducted experiments on Eclipse and the results showed the effectiveness of the proposed method. Kahles et al. [19] employed the machine learning technique to automatically analyse the root cause in agile software test environments. They extracted 188 relevant features from totally 1271 failed

test cases for experiments and the results illustrated that the model obtained an accuracy of 88.9%. Ma et al. [20] developed a big data-driven system that employed the supervised machine learning technique for enhancing the performance of root cause analysis. They first employed the data mining technique to build a feature-based model to represent the distinct types of quality issues and then utilised the machine learning model to predict the root causes. The experimental results showed the proposed data-driven model can reduce the prediction time and cost. Hangal et al. [21] developed a new model called DIDUCE that could instrument programs and observe their runtime behaviours. It began with the strongest hypothesises and could gradually weaken the hypothesis to explore the new rare behaviours that might damage the program operation. Based on such information, they found that DIDUCE is effective in detecting software bugs. Abdelrahman et al. [22] first employed seven anomaly detection models with two performance indicators followed by the Boosting technique to detect the anomalies. Then, they used a statistical root cause analysis on such anomalies to visualise the possible causes. Their experimental results demonstrated the effectiveness of their proposed method. Soldani et al. [23] surveyed the existing techniques for anomaly detection and root cause analysis in multi-service applications, including log-based methods, visualisation-based methods, distributed tracing-based, and monitoring-based methods. They further summarised the setup, accuracy, explainability, and countermeasures of these methods.

Different from previous studies that focussed on analysing the root causes of quality deviations or bugs in software, our aim is to identify whether the software crashes located in the stack trace or not because the information derived from stack traces is more helpful for developers to locate the bug lines with fewer efforts.

2.4 | Software defect prediction via unsupervised learning techniques

Nam et al. [24] proposed the novel CLA and CLAMI approaches for software defect prediction task. The approaches automatically classified the unlabelled dataset by means of the magnitude of metric values. Their experimental results showed the effective performance compared with the supervised learning techniques. Liu et al. [25] proposed the CCUM method that employed the code churn features to construct an unsupervised learning model for software defect prediction. They conducted experiments with three settings including cross-validation, time-wise cross-validation, and cross-project prediction, and the results demonstrated the effectiveness of the CCUM model. Yan et al. [26] conducted an empirical study that explored the prediction performance of supervised and unsupervised learning techniques under effort-aware file-level defect prediction models. The experimental results showed the effectiveness of unsupervised learning techniques under cross-project scenarios. Fan et al. [27] analysed the prediction performance of the CLIFF&MORPH model for cross-company defect prediction task. Their experimental results illustrated that the unsupervised ManualDown technique obtained better performance. Huang et al. [28] conducted a replication exploration of Yang et al.'s work [29] that proposed an unsupervised method called LT. Their results found that LT needed partitioners to check many code changes involving a large number of code context. Chang et al. [30] proposed a novel approach called CISC for software defect prediction task. It first incorporated spectral clustering with Ng-Jordan-Weiss technique to obtain a low dimensional unlabelled dataset. Then, the chaotic and immune clone selection method was employed and the layer chaotic mutation was developed to improve the variety of antibody. Their experimental results demonstrated the effectiveness of the CISC method. Li et al. [31] empirically investigated the performance of 49 unsupervised learning techniques for software defect prediction task. They grouped these unsupervised models into 14 categories and adopted the Matthews Correlation Coefficient as the basic indicator. Their experimental results illustrated the inconsistent and insufficient of the published experimental results in original papers.

Although previous studies adopted unsupervised learning techniques for software defect prediction task, to the best of our knowledge, we are the first to introduce such model for ICFR task, that is, identifying whether the crashing faults located in the stack trace under the cross-project scenario.

2.5 | Summary

In this section, we discuss the related work of our study from four aspects. First, we introduce the knowledge of crash and stack trace analysis as the background of our work. Then, we present how existing studies deal with the same ICFR task and explain the differences between this work and other studies. In addition, we discuss the related topics such as root cause analysis and software defect prediction, and compare them with ours.

3 | FUNDAMENTALS

In this section, we introduce the used corpora for the experiments in our work, which is an public available benchmark dataset shared by Gu et al. [2]. This dataset consists of crashing fault instances from seven open-source and real-world software projects, that is, Apache Commons **Codec**, Apache Commons **Collections**, Apache Commons **IO**, **Jsoup**, **JSqlParse**, **Mango** and **Ormlite**-Core. Table 1 lists some basic information of this dataset, including the selected version for each project (number in parentheses), the total number of crashing fault instance (**# Crash**), the number of crashing fault instances whose residences locate outside the stack traces (**# OutTrace**), and the number and proportion of crashing fault instances whose residences locate inside the stack traces (**# InTrace** and **% InTrace**). The construction of this dataset mainly consists of three steps: that is, generating crashes,

TABLE 1 Basic statistics of the used seven projects

Projects	# Crashes	# OutTrace	# InTrace	% InTrace
Codec (1.1)	610	433	177	29.02 %
Collections (4.1)	1350	1077	273	20.22 %
IO (2.5)	686	537	149	21.72 %
Jsoup (1.11.1)	601	481	120	19.97 %
JSqlParser (0.9.7)	647	586	61	9.43 %
Mango (1.5.4)	733	680	53	7.23 %
Ormlite (5.1)	1303	977	326	25.02 %

extracting features, and labelling each crash instance. The operations for each step are briefly described below:

3.1 | Generating crashes

Generating program mutants by mutation operation. Gu et al. [2] used the mutation techniques to seed faults into the seven software projects to simulate the real crashes following previous work [32, 33]. More specifically, they used seven default mutation operation in a state-of-the-art mutation testing system to generate program mutants by making a single-point change on the corresponding program.

Removing useless mutants. The program mutants that do not lead to the software crashes need to be discarded. The deletion rules are that the ones which can pass all the test cases and the ones whose stack traces only include AssertionFailedError, ComparisonFailure, or test cases are filtered out. The remained mutants are treated as the crashing fault instances.

3.2 | Feature extraction

In order to characterise the crashing fault instances, Gu et al. [2] extracted a total of 89 features for representation. The feature extraction is done with the help of a static program analysis framework, called Spoon, which is an open-sourced and extensible Java compiler and designed for analysing, rewriting, transforming, and compiling Java source code [34]. The 89 features are derived from a total of five families:

- 11 features are related to the stack traces, which characterises the difficulty of dealing with corresponding crashes. For convenience, the names of these 11 feature2 are abbreviated $\{ST_1, ST_2, \ldots, ST_{11}\}$.
- 23 features are extracted from the function and class in the first frame, which represents the program state messages when the program crashes. The names of these 23 feature are abbreviated $\{FF_1, FF_2, \ldots, FF_{23}\}$.
- 23 features are mined from the function and class in the last frame, which shows the information of the initial function call. The names of these 23 feature are abbreviated $\{LF_1, LF_2, \ldots, LF_{23}\}$.
- 16 features are gained via normalising the 16 features related to the top function in the first frame with LOC (Lines Of Code). The names of these 16 feature are abbreviated $\{NFF_1, NFF_2, \ldots, NFF_{16}\}$.
- features are obtained by normalising the 16 feature related to the bottom function in the last frame with LOC. The names of these 16 feature are abbreviated $\{NLF_1, NLF_2, \ldots, NLF_{16}\}$.

The details for each feature can be found in Table 2.

3.3 | Rules of label assignment

In general, the contained main items of each frame in the stack traces can be treated as a triplet, that is, [the class name, the function name, line number of the code]. When the information of a crashing fault instance exactly matches a triple of one frame in the stack trace, the label of the corresponding instance is assigned as *InTrace*, that is, the residence of the crashing fault is identified as inside the stack trace. Otherwise, the label of the instance is identified as *OutTrace*, that is, the residence of the crashing fault is identified as outside the stack trace.

4 | METHOD

4.1 | Notations

Our TSC based cross-project ICFR model involves two projects, that is, the source project and the target project. We define the data from the source project as the data matrix $S = [s_1, \ldots, s_{n_1}]$ containing n_1 crashing instances with n_f features, where $s_i \in R^{n_f \times n_1}$. Similarly, the data from the target project is defined as the data matrix $T = [t_1, \ldots, t_{n_2}]$ containing n_2 crashing instances with n_f features, where $t_j \in R^{n_f \times n_2}$.

4.2 | Transfer Spectral Clustering (TSC)

In the context of unsupervised cross-project ICFR task, TSC clusters one project data with the help of another project data. TSC is expected to achieve better performance on both project data than clustering them separately. For this purpose, it needs to find the low dimensional embeddings for the two project data to enable they are not only smooth on the data manifold, but also maximise the task relationship [35]. In the obtained low dimensional embeddings, both project data can be partitioned into the desired number of groups. Assuming that $G^{(s)}$ and $G^{(t)}$ are defined as the k nearest neighbour graphs generated on each project data, the corresponding affinity matrices $M^{(s)}$ and $M^{(s)}$ are denoted as follows:

$$M_{ij}^{(s)} = \begin{cases} 1, & \text{if } s_i \in nei(s_j) \text{ or } s_j \in nei(s_i) \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

TABLE 2 The detailed information of 89 features

Feature	Description
ST	Features related to the stack trace
ST_1	The type of the exception in the crash
ST_2	The frame number in the stack trace
ST_3	The class number in the stack trace
ST_4	The function number in the stack trace
ST_5	Whether there is an overloaded function in the stack trace
ST_6	The name length in the top class
ST_7	The name length in the top function
ST_8	The name length in the bottom class
ST_9	The name length in the bottom function
ST_10	The Java file number in the project
ST_11	The class number in the project
FF (and LF)	Features related to the **F**irst **F**rame (**FF**) and the **L**ast **F**rame (**LF**)
FF_1 (LF_1)	The local variable number in the top/bottom class
FF_2 (LF_2)	The field number in the top/bottom class
FF_3 (LF_3)	The function number (except constructor functions) in the top/bottom class
FF_4 (LF_4)	The imported package number in the top/bottom class
FF_5 (LF_5)	Whether the top/bottom class is inherited from others
FF_6 (LF_6)	The comment LOC in the top/bottom class
FF_7 (LF_7)	The top/bottom function LOC
FF_8 (LF_8)	The parameter number in the top/bottom function
FF_9 (LF_9)	The local variable number in the top/bottom function
FF_{10} (LF_{10})	The if-statement number in the top/bottom function
FF_{11} (LF_{11})	The loop number in the top/bottom function
FF_{12} (LF_{12})	The for statement number in the top/bottom function
FF_{13} (LF_{13})	The for-each statement number in the top/bottom function
FF_{14} (LF_{14})	The while statement number in the top/bottom function
FF_{15} (LF_{15})	The do-while statement number in the top/bottom function
FF_{16} (LF_{16})	The try block number in the top/bottom function
FF_{17} (LF_{17})	The catch block number in the top/bottom function
FF_{18} (LF_{18})	The finally block number in the top/bottom function
FF_{19} (LF_{19})	The assignment statement number in the top/bottom function
FF_{20} (LF_{20})	The function call number in the top/bottom function
FF_{21} (LF_{21})	The return statement number in the top/bottom function
FF_{22} (LF_{22})	The unary operator number in the top/bottom function
FF_{23} (LF_{23})	The binary operator number in the top/bottom function
NFF (and NLF)	Features **n**ormalised by LOC from **FF** (**NFF**) and **LF** (**NLF**)
NFF_1 (NLF_1)	FF_8/FF_7 (LF_8/LF_7)
NFF_2 (NLF_2)	FF_9/FF_7 (LF_9/LF_7)
NFF_3 (NLF_3)	FF_{10}/FF_7 (LF_{10}/LF_7)

TABLE 2 (Continued)

Feature	Description
…	…
NFF_{15} (NLF_{15})	FF_{22}/FF_7 (LF_{22}/LF_7)
NFF_{16} (NLF_{16})	FF_{23}/FF_7 (LF_{23}/LF_7)

$$M_{ij}^{(t)} = \begin{cases} 1, & \text{if } t_i \in nei(t_j) \text{ or } t_j \in nei(t_i) \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

where $nei(s_j)$ and $nei(t_j)$ denote the neighbour sets of s_j and t_j respectively. Then we need to obtain the low dimensional embeddings that are smooth on the graphs. We define $E^{(s)} \in R^{n_1 \times k}$ and $E^{(t)} \in R^{n_2 \times k}$ as embeddings of the two project data in which each row represents an embedding for the corresponding crash instance. In this case, the smoothness of the $E^{(s)}$ over $G^{(s)}$ is measured as:

$$\frac{1}{2}\sum_{i,j=1}^{n_1} M_{ij}^{(s)} \left\| \frac{1}{\sqrt{D_{ii}^{(s)}}} E_i^{(s)} - \frac{1}{\sqrt{D_{jj}^{(s)}}} E_j^{(s)} \right\|^2 \quad (3)$$

where $E_i^{(s)}$ is the i-th row of the $E^{(s)}$ and $D^{(s)} = \text{diag}(M^{(s)}\mathbf{1})$ and $\mathbf{1}$ is the vector 1. This formula can be simplified into the following form $\text{tr}\left(E^{(s)^{\mathrm{T}}}\left(I - M_N^{(s)}\right)E^{(s)}\right)$, where $M_N^{(s)} = D^{(s)^{-\frac{1}{2}}} M^{(s)} D^{(s)^{-\frac{1}{2}}}$. If we add the following constraint $E^{(s)^{\mathrm{T}}}E^{(s)} = I$, the smoothness of $E^{(s)}$ over $G^{(s)}$ can be further measured as follows:

$$\text{tr}\left(E^{(s)^{\mathrm{T}}} M_N^{(s)} E^{(s)}\right) \quad (4)$$

The greater value means smoother of the embedding on the data manifold, that is, on the corresponding graph. Similarly, the smoothness of $E^{(t)}$ on $G^{(t)}$ can also be measured as follows:

$$\text{tr}\left(E^{(t)^{\mathrm{T}}} M_N^{(t)} E^{(t)}\right) \quad (5)$$

where $M_N^{(t)} = D^{(t)^{-\frac{1}{2}}} M^{(t)} D^{(t)^{-\frac{1}{2}}}$ and $D^{(t)} = \text{diag}(M^{(t)}\mathbf{1})$.

As mentioned above, in addition to seeking for the smoothness on the graph, TSC also aims to maximise the task relationship, that is, the embedding obtained over one project data can promote the acquisition of the embedding over another project data. This goal can be treated as a graph co-clustering issue which aims to find minimum cut partitions in a bipartite graph, in which the graph nodes contain the information of crash instances and features [36]. We define a data matrix $A \in R^{d \times n}$, where d and n denote the number of features and crash instances respectively. The corresponding affinity matrix of graph is defined as follows:

$$W = \begin{bmatrix} 0 & A \\ A^{\mathrm{T}} & 0 \end{bmatrix} \quad (6)$$

The graph Laplacian derived from the above formula is defined as

$$L = \begin{bmatrix} D_1 & -A \\ -A^{\mathrm{T}} & D_2 \end{bmatrix} \quad (7)$$

where $D_1 = \text{diag}(A\mathbf{1})$ and $D_2 = \text{diag}\left(A^{\mathrm{T}}\mathbf{1}\right)$. Then, we get the objective function of graph co-clustering as follows:

$$\min_z \quad z^{\mathrm{T}} L z \qquad s.t. \quad \begin{cases} x^{\mathrm{T}} D_1 x = 1, \\ y^{\mathrm{T}} D_2 y = 1. \end{cases} \quad (8)$$

where x and y denote the embeddings for features and crash instances respectively, and $z = \begin{bmatrix} x \\ y \end{bmatrix}$. The objective function can be equivalent to the following formula:

$$\max_{x,y} \quad x^{\mathrm{T}} A y \qquad s.t. \quad \begin{cases} x^{\mathrm{T}} D_1 x = 1, \\ y^{\mathrm{T}} D_2 y = 1. \end{cases} \quad (9)$$

According to the graph co-clustering theory, the objective function can be rewritten as follows:

$$\Phi\left(E^{(s)}, E^{(t)}, E^{(f)}\right) = \text{tr}\left(E^{(f)^{\mathrm{T}}} S E^{(s)}\right) + \text{tr}\left(E^{(f)^{\mathrm{T}}} T E^{(t)}\right) \quad (10)$$

where $E^{(s)}$ and $E^{(t)}$ are the embeddings for the crash instances of two projects as mentioned above, and $E^{(f)}$ denotes the embeddings for features. A larger Φ value means better co-clustering effect on the two project data.

Therefore, in order to pursue better smoothness on the data manifold and maximise the task relationship, the final objective function combines the above three formulas simultaneously as follows:

$$g\left(E^{(s)}, E^{(t)}, E^{(f)}\right) = \max_{E^{(s)}, E^{(t)}, E^{(f)}} \text{tr}\left(E^{(s)^{\mathrm{T}}} M_N^{(s)} E^{(s)}\right) + \text{tr}\left(E^{(t)^{\mathrm{T}}} M_N^{(t)} E^{(t)}\right) + \lambda\left(\text{tr}\left(E^{(f)^{\mathrm{T}}} S_N E^{(s)}\right) + \text{tr}\left(E^{(f)^{\mathrm{T}}} T_N E^{(t)}\right)\right) s.t. \begin{cases} E^{(s)^{\mathrm{T}}} E^{(s)} = I, \\ E^{(t)^{\mathrm{T}}} E^{(t)} = I, \\ E^{(f)^{\mathrm{T}}} E^{(f)} = I, \end{cases}$$

where $S_N = \mathrm{diag}(S\mathbf{1})^{-\frac{1}{2}} S \mathrm{diag}(S^{\mathrm{T}}\mathbf{1})^{-\frac{1}{2}}$ is the normalised version of S, and $T_N = \mathrm{diag}(T\mathbf{1})^{-\frac{1}{2}} T \mathrm{diag}\left(T^{\mathrm{T}}\mathbf{1}\right)^{-\frac{1}{2}}$ is the normalised version of T. $M_N^{(s)}$ and $M_N^{(t)}$ are the affinity matrices corresponding to the normalised versions of S and T respectively. The λ ($\lambda > 0$) is a trade off parameter between the two goals (i.e. the smoothness on the data manifold and the co-clustering effect).

4.3 | Solution process for TSC

Here, we describe the process of solving the TSC method. It is not feasible to give a global optimal solution for the above objective function as it is a non-convex optimisation. Thus, we can only obtain a local solution. In order to solve this constraint function, $E^{(s)}$ and $E^{(t)}$ are first initialised with the top k eigenvectors of $M_N^{(s)}$ and $M_N^{(T)}$, respectively, while $E^{(f)}$ is initialised with the top k left singular vectors of the matrix $[S_N\ T_N]$. The solution of this objective function can be obtained by updating $E^{(s)}$, $E^{(t)}$, and $E^{(f)}$ which can lead the increase in the value of the objective function. More specifically, the update formulas are based on the partial derivatives of g (i.e. $g(E^{(s)}, E^{(t)}, E^{(f)})$) with respect to $E^{(s)}$, $E^{(t)}$, and $E^{(f)}$ as follows:

$$E_{i+1}^{(s)} = \Psi\left(E_i^{(s)} + t\frac{\partial g}{\partial E^{(s)}} \Big/ \left\|\frac{\partial g}{\partial E^{(s)}}\right\|\right) \tag{11}$$

$$E_{i+1}^{(t)} = \Psi\left(E_i^{(t)} + t\frac{\partial g}{\partial E^{(t)}} \Big/ \left\|\frac{\partial g}{\partial E^{(t)}}\right\|\right) \tag{12}$$

$$E_{i+1}^{(f)} = \Psi\left(E_i^{(f)} + t\frac{\partial g}{\partial E^{(f)}} \Big/ \left\|\frac{\partial g}{\partial E^{(f)}}\right\|\right) \tag{13}$$

where t is the iteration step length, $\Psi(Z) = UI_{n,m}V^{\mathrm{T}}$ is the projection of a matrix $Z \in R^{n\times m}$ on Stiefel manifold, in which the singular value decomposition of Z is $Z = U\sum V^{\mathrm{T}}$ [37]. In the singular value decomposition formula, U is a $m \times m$ matrix, $\sum$ is the semi-positive definite diagonal matrix with the size of $m \times n$ and the corresponding elements on the diagonal are singular values of Z, and V is a $n \times n$ matrix. In addition, U and V are unitary matrices, that is, $U^{\mathrm{T}}U = I$ and $V^{\mathrm{T}}V = I$ [38].

The updating process terminates until achieving the convergence. At last, the crash data of the two projects are grouped into pre-designated clusters. In our ICFR task, the cluster number is set to 2 since our crashing fault instance only have two types of labels, that is, the crashing fault locates inside the stack trace (with label *InTrace*) or outside the stack trace (with label *OutTrace*). We have released the source code of our TSC-based cross-project ICFR method at https://github.com/sepine/TSC so that other researchers could easily reproduce our experiments in the future.

5 | EXPERIMENTAL SETUP

5.1 | The overall framework

Figure 1 provides an overview of the framework of this study. First, the unlabelled crash instances from the source project and the target project are input into the used TSC method, and the output are two clusters of the target project data. The empirical labelling strategy is used to label the instances in the two clusters. According to the clustering results, a set of indicators are calculated to evaluate the performance of our used TSC method. Finally, a statistic test method is employed to significantly analyse the performance results of both our TSC methods and the baseline methods.

5.2 | Performance indicators

As the goal of our ICFR task is to identify whether the residences of crash instances is inside or outside the stack trace under the unsupervised scenario, it can be treated as a binary clustering problem, that is, performing clustering on binary data. As stated in a previous study [39], a work also for a typical software engineering task, the performance indicators commonly-used for the binary classification task can also be applied to measure the effectiveness of binary clustering task. In this work, we carefully choose the same five indicators as that in the work of Xu et al. [3] which proposed a supervised cross-project model for ICFR task. The five indicators including F-measure, G-measure, g-mean, Balance, and

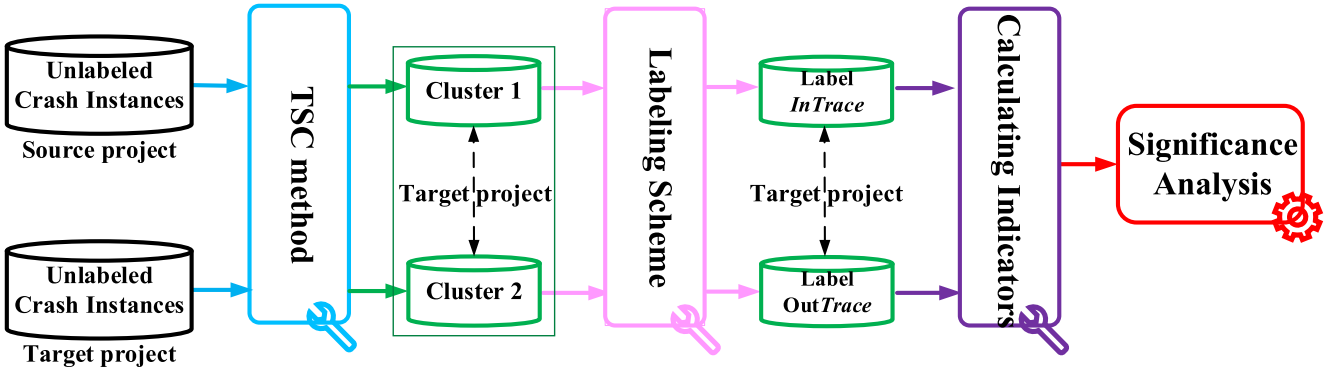

FIGURE 1 An overall of our experimental framework

Matthews correlation coefficient (**MCC**) allow us to do a comprehensive performance evaluation for our TSC based unsupervised ICFR model. We will introduce how to calculate these indicators below.

First, we give four types of output results of the unsupervised models:

- a crash instance whose true label is *InTrace* is identified as *InTrace* by the model.
- a crash instance whose true label is *InTrace* is identified as *OutTrace* by the model.
- a crash instance whose true label is *OutTrace* is identified as *OutTrace* by the model.
- a crash instance whose true label is *OutTrace* is identified as *InTrace* by the model.

The number of corresponding crash instances under above four output results are defined as the following terms: **T**rue **P**ositive (**TP**), **F**alse **N**egative (**FN**), **T**rue **N**egative (**TN**), and **F**alse **P**ositive (**FP**). Note that, in this work, we treat the crash instances with label *InTrace* as the positive instances since we always want to detect as many of these types of instances as possible to speed up the efficiency of fault localisation and avoid wasting resources reviewing irrelevant code. The reason for this refers to the studies of defect prediction which treat the faulty code instances as positive ones since the developers and testers want to detect as many of such instances as possible to allocate test resources appropriately [40].

Based on the above terms, three basic transitional indicators are calculated as follows. $Precision = \frac{TP}{TP+FP}$, that is, the ratio of crash instances with true labelled *InTrace* that are identified as *InTrace* to the total number of crash instances that are identified as *InTrace*. $Recall = \frac{TP}{TP+FN}$, that is, the ratio of crash instances with true label *InTrace* that are correctly identified to the total number of crash instances whose true labels are *InTrace*. Recall is also called **PD** (**P**robability of **D**etection). **P**robability of **F**alse alarm $\left(PF = \frac{FP}{FP+TN}\right)$, that is, the ratio of crash instances with true label *InTrace* that are incorrectly identified to the total number of crash instances whose true labels are *InTrace*.

Based on the above items, the used five performance indictors in this work are calculated as follows:

F-measure is a harmonic mean of Precision and Recall:

$$F\text{-}measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{14}$$

G-measure is a harmonic mean of PD and (1-PF):

$$G\text{-}measure = \frac{2 \times PD \times (1-PF)}{PD + (1-PF)} \tag{15}$$

g-mean is a geometric mean of PD and (1-PF):

$$g\text{-}mean = \sqrt{PD \times (1-PF)} \tag{16}$$

Balance is a trade-off between PF and Recall:

$$Balance = 1 - \sqrt{\frac{(0-PF)^2 + (1-Recall)^2}{2}} \tag{17}$$

MCC is essentially a correlation coefficient describing actual and output results:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \tag{18}$$

The first four indicators range from 0 to 1, while the last indicator ranges from −1 to 1. Larger values for all five indicators imply better performance. When $MCC = 1$, it manifests the perfect identification. When $MCC = 0$, it denotes that the identification results is worse than that of random guess. When $MCC = -1$, it means that the identification results is completely inconsistent with the actual ones.

5.3 | Labelling scheme for clusters

The output of the TSC method is two clusters, and we can know which crash instances belong to the same cluster. But this method does not tell us which group of crash instances is labelled as *InTrace*. Inspired by the heuristic from a defect prediction study [5] which assigned label to the cluster based on the feature values of each cluster, in this work, we first calculate the average feature values of the obtained two clusters, and assign the label *InTrace* (*OutTrace*) to the crash instances whose cluster has the lower (larger) average feature values. The rationale for this setting is explained in the Section 7.

5.4 | Significance analysis

In order to analyse the significance differences between our TSC based ICFR model and the baseline methods, we choose to use Friedman test with an improved Nemenyi test for statistic significant analysis as suggested by Herbold et al. [41]. Friedman is a commonly-used test to compare the overall performance of a set of methods over multiple project data. However, this test only gives a conclusion on whether there exists a difference between the performance among these methods based on a p value. If so (i.e. the p value is lower than 0.05), a post-hoc test (such as the used Nemenyi test) is needed to find out which methods have statistically significant

differences in performance. The Nemenyi test is applicable to the scenario when all methods are compared to each other. The Nemenyi test compares the difference of average ranking of each method with a **C**ritical **D**ifference (**CD**) which is calculated as $CD = q_\alpha \sqrt{\frac{r(r+1)}{6b}}$, where r and b are the number of methods and project data respectively, and q_α is a critical value related to significant level α (usually $\alpha = 0.05$) and r [42]. If the ranking difference between two methods is greater that CD, it indicates that the method with the high average ranking is statistically superior to the method with the low average ranking. Otherwise, there is no statistical difference between the two methods. The reason of using the improved version of the Nemenyi test in this work is to avoid generating overlapped groups after dividing these methods. The output of the improved Nemenyi test is a CD diagram as depicted in Figure 2, in which the methods (such as method 1, method 2, and method 3 or method 1, method 3, and method 4) with significant differences are drawn in different colours while the methods (such as method 2 and method 4) without significant differences are drawn in the same colour.

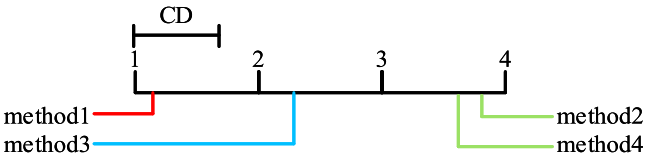

FIGURE 2 An example of the CD diagram

6 | EXPERIMENTAL RESULTS

6.1 | RQ1: does our TSC method perform better than the unsupervised learning models over single project for ICFR task?

Motivation: Since our TSC based ICFR method does not involve any labelled crash data, thus, in essence, it belongs to a kind of unsupervised learning model. We design this question to investigate whether our unsupervised TSC method could achieve better performance than other unsupervised learning methods for the ICFR task. In addition, our TSC method is applied to two unlabelled project data. In this question, we also set experiment applying methods to single project data to observe their performance for the ICFR task. If our TSC method wins out, it implies that the information provided by auxiliary unlabelled crash data indeed have the potential to help improve the performance of unsupervised methods.

Methods: As far as we know, no researchers have proposed to use unsupervised methods for the ICFR task yet, either under within or cross-project scenarios. Therefore, we cannot find baseline methods from previous studies for performance comparison. In this case, we choose some typical clustering algorithm based unsupervised learning methods which are applied to single project for comparison. Finally, we employ 4 clustering methods, including **M**ini **B**atch k-**M**eans (**MBM**) [43], **B**alanced **i**terative **r**educing and **c**lustering using **h**ierarchies (**Birch**) [44], **S**pectral **B**i**C**lustering (**SBC**) [45], and **G**aussian **M**ixture **M**odel (**GMM**) using Expectation Maximisation algorithm [46]. According to a family classification summarised in a previous study [39], the used four baseline unsupervised methods come from different family. More specifically, MBM, Birch, SBC, and GMM belong to the partition based, hierarchy based, graph-theory-based, and model based clustering families respectively. The brief description of the four baseline methods are below.

- **MBM:** an improved k-means. It reduces the computational cost via exploiting a fixed size subset of whole dataset.
- **Birch:** a clustering method generally for large datasets. It uses a single scan to gain high quality clustering results and improves performance with following scans.
- **SBC:** It assumes that the input data matrix has a hidden checkerboard structure, and divides the rows and columns so that the corresponding blockwise-constant checkerboard matrix can well approximate the original matrix.
- **GMM:** It is a parametric probability density function which assumes that the input data obey the mixture Gaussian distribution in clustering task.

Note that the baseline method SBC is a spectral clustering method being only applied to single project data (i.e. the target project data) which does not involve data from the source project. From this point of view, it can be treated as a non-cross-project version of TSC, that is, TSC without auxiliary data.

Results: In the experiments, for our TSC method, each project is treated as the target project once while other projects are treated as the source projects in turn. For the four comparative unsupervised baseline methods, they are only applied to the target project. In Figure 3, we report the radar charts of the average performance values of the five indicators for our TSC methods and four unsupervised baseline methods on the cross-project pairs of each project being treated as the target project from Figure 3a–g as well as across all projects in Figure 3h. Table 3 reports the average performance of the five indicators for these five methods across all projects. Figure 4 visualises the analysis results of Friedman test with Nemenyi test on the performance of our TSC method and four unsupervised baseline methods. From Figure 3, Figure 4, and Table 3, we have the following observations:

First, from Figure 3a to Figure 3g, we can observe that, in terms of F-measure, our TSC method achieves better performance than all four unsupervised baseline methods on projects Codec, Collection, IO, and JSqlParser when they are individually treated as the target project; in terms of G-measure, our TSC method performs better than all four unsupervised baseline methods on projects Collection and IO when they are individually treated as the target project; in terms of g-mean and Balance, our TSC method performs better than four other baseline methods on projects Collection, IO and JSqlParser as well; in terms of MCC, our TSC method performs better than four other baseline methods on projects Codec, Collection, IO and JSqlParser.

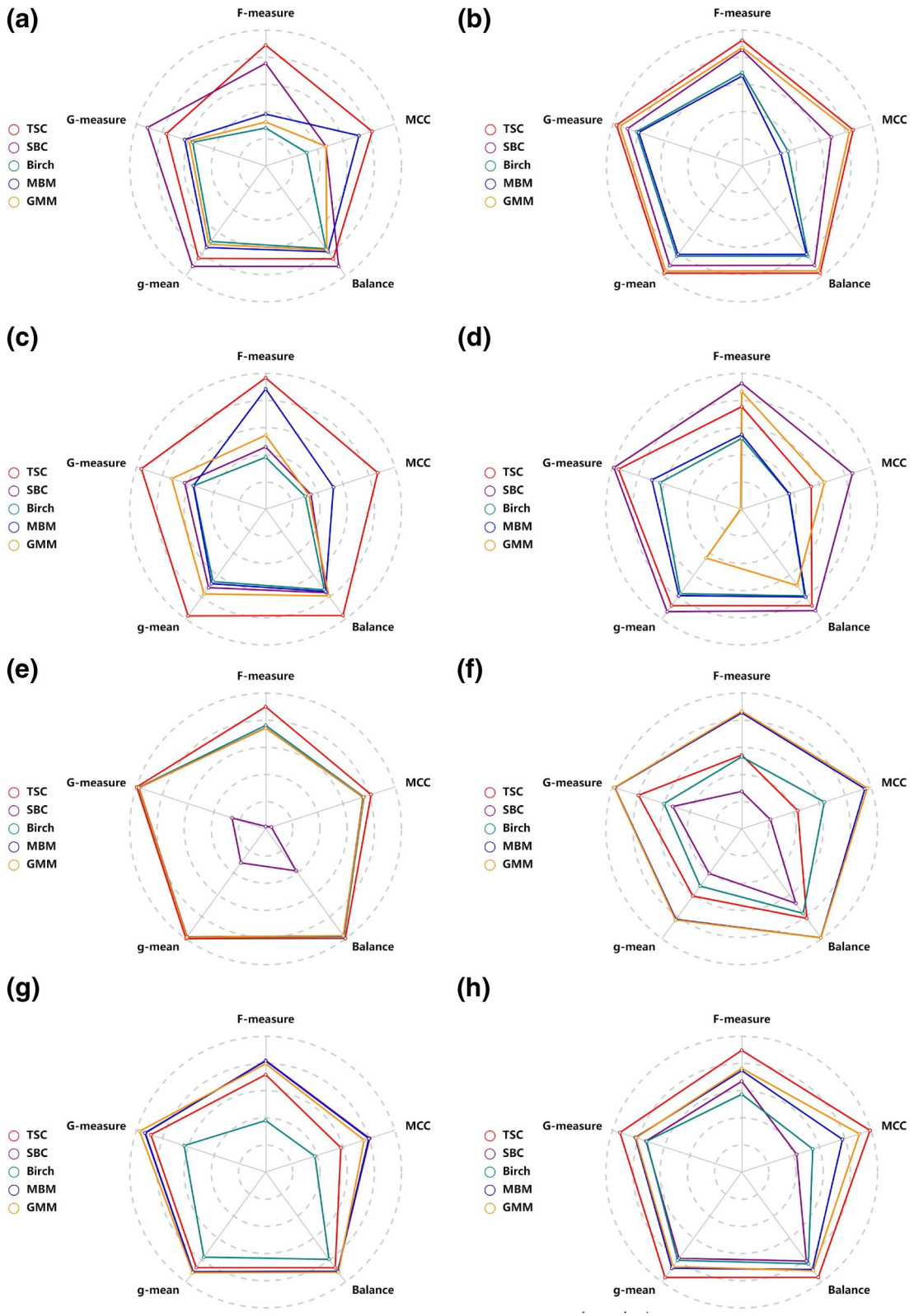

FIGURE 3 Radar charts of average values of five indicators on the cross-project pairs of each project and across all projects in terms of TSC and four unsupervised baseline methods. (a) Radar charts of five indicators when Codec is treated as the target project. (b) Radar charts of five indicators when Collections is treated as the target project. (c) Radar charts of five indicators when IO is treated as the target project. (d) Radar charts of five indicators when Jsoup is treated as the target project. (e) Radar charts of five indicators when JSqlParser is treated as the target project. (f) Radar charts of five indicators when Mango is treated as the target project. (g) Radar charts of five indicators when Ormlite-Core is treated as the target project. (h) Radar charts of five indicators across all project when each one is treated as the target project

Second, from Figure 3h, we can find that in terms of all five indicators, our TSC methods achieves the best average performance compared with all four unsupervised baseline methods across all projects when they are individually treated as the target project in turn. Combining the specific performance values given in Table 3, by comparing with the best indicator values among the four baseline methods, our TSC achieves improvements by 23.8% in terms of F-measure, by 27.8% in terms of G-measure, by 17.6% in terms of g-mean, by 12.3% in terms of Balance, and by 26.9% in terms of MCC.

Third, from Figure 4 which visualises the corresponding statistic test results for the five methods in terms of the five indicators, we can observe that the *p*-values of Friedman test (all less than 0.05) indicate that there exist statistically significant performance differences among the five methods in terms of all indicators. The CD diagrams show that our TSC method belongs to the top ranking group in terms of all indicators and achieves the best average ranking on four indicators except for F-measure. However, our TSC method has no significant differences compared with two baseline methods (i.e. GMM and MBM) in terms of F-measure and MCC, and one baseline method (i.e. GMM) in terms of other three indicators. Although the performance differences between our TSC and GMM are not statistically significant over all five indicators, in terms of the specific average values of the our TSC method and GMM across all project (i.e. F-measure 0.349-vs-0.282, G-measure 0.542-vs-0.418, g-mean 0.560-vs-0.460, Balance 0.558-vs-0.497, and MCC 0.118-vs-0.093), the performance value of our TSC method is still much better than GMM. Therefore, when the need of better performance is the primary consideration for ICFR in practical applications and unlabelled data of other projects are available, our TSC method is still recommended over GMM. In addition, the performance of GMM is affected by whether input data obeys mixture Gaussian distribution. While the performance of our TSC method is not affected by this constraint. Thus, when the distribution of crash data is not satisfied, our TSC method is preferred over GMM.

TABLE 3 Average performance for the five methods across all projects

Indicators	SBC	Birch	MBM	GMM	TSC
F-measure	0.234	0.186	0.274	0.282	**0.349**
G-measure	0.340	0.336	0.424	0.418	**0.542**
g-mean	0.385	0.403	0.476	0.460	**0.560**
Balance	0.409	0.435	0.486	0.497	**0.558**
MCC	−0.053	−0.015	0.054	0.093	**0.118**

Note: The bold values means the best prediction performance in terms of each indicators.

Answer to RQ1

Our TSC-based unsupervised method with the aid of unlabelled crash data from external projects indeed shows the performance superiority to the typical unsupervised methods without auxiliary data. This implies that it is possible to utilise auxiliary unlabelled crash data to improve performance for the ICFR task.

6.2 | RQ2: is our TSC method superior to some classic supervised cross-project models for ICFR task?

Motivation: Since our TSC based ICFR method involves two projects, that is, the source project and target project, to put it another way, it belongs to a cross-project model. Nevertheless, unlike traditional cross-project models that have been proposed for other software engineering tasks, our TSC method does not require the source project data to have labels. We set this question to explore whether our unsupervised cross-project model can achieve similar or even better results compared with the supervised cross-project models as the traditional ones. If our TSC method wins out, it implies that our TSC method has great competitiveness towards the supervised cross-project models.

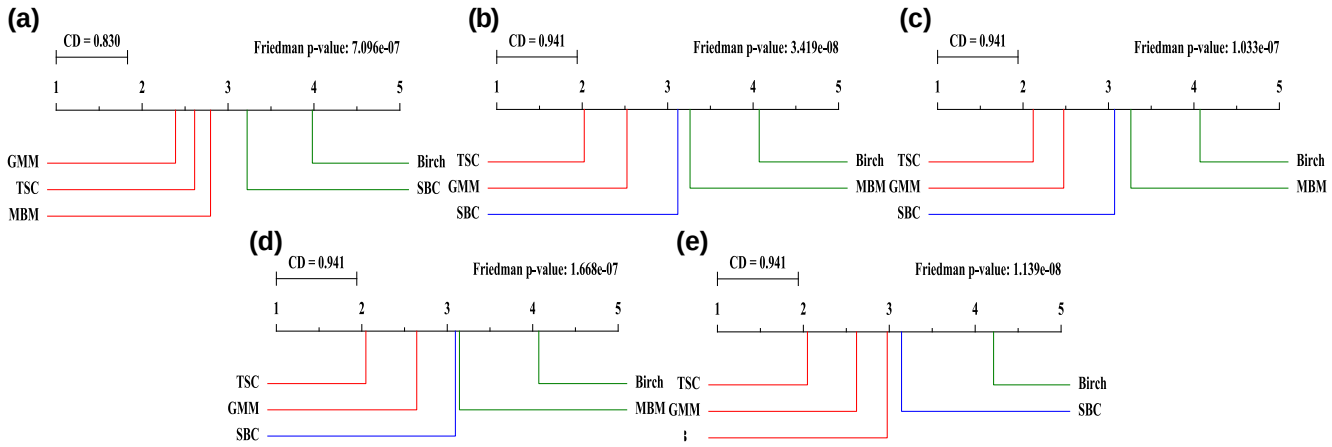

FIGURE 4 Visualised results of statistical test for our TSC method and four unsupervised methods in terms of five indicators. (a) Statistical test results for F-measure. (b) Statistical test results for G-measure. (c) Statistical test results for g-mean. (d) Statistical test results for Balance. (e) Statistical test results for MCC

Methods: Researchers have proposed different supervised cross-project models for software engineering tasks. For this question, we choose eight existing methods as our baseline methods. These methods could be divided into three types: four instance filtering based models (i.e. NN-Filter [47], Peter-Filter [48], Yu-Filter [49], and HISNN [50]), two transfer learning-based models (i.e. TCA+ [51] and TNB [52]) and two classifier combination-based models (i.e. CODEP [53] and ASCI [54]). The brief descriptions of these methods are below:

- **NN-Filter:** for each crash instance in the target project, NN-filter finds its nearest neighbour from the source project.
- **Peter-Filter:** it first clusters the mixed source and target project data, and then conducts the NN-filter operation in each clusters.
- **Yu-Filter:** it first clusters the mixed source and target project data, and then reserves the crash data of the source project in such clusters which contain as least one crash instance from the target project.
- **HISNN:** a hybrid instance selection method by learning both local and global knowledge.
- **TCA+:** it uses a component analysis method to relief the distribution difference between source and target projects.
- **TNB:** an improved naive Bayes method by considering the weighting information of the crash data in the source project.
- **CODEP:** it combines outputs of different classifiers, which is superior to the performance of all stand-alone classifiers.
- **ASCI:** it uses a strategy to adaptively choose the optimal classifier which can better predict the class.

Results: For the eight comparative supervised baseline methods, the only difference with our TSC method is that they assume that the labels of the source project data are known. In Figure 5, we report the box-plots of the performance values of five indicators for our TSC methods and eight supervised baseline methods across all 42 cross-project pairs from Figure 5a to Figure 5e. Table 4 reports the average performance of the five indicators for these nine methods across all projects. Figure 6 visualises the corresponding analysis results of Friedman test with Nemenyi test on the performance. From Figure 5, Table 3, and Figure 6, we have the following findings:

First, from Figure 5a to Figure 5e, we can observe that, in terms of the median F-measure, our TSC method performs the best compared with all eight supervised baseline methods; in terms of the median G-measure, g-mean, and Balance, our TSC methods achieves nearly the same performance as two supervised baseline methods, that is, Yu-Filter and TCA+, while achieves better performance than the other six supervised baseline methods; in terms of the median MCC, our TSC method achieves a little lower performance than three supervised baseline methods, that is, TCA+, CODEP, and ASCI, but higher performance than the other five baseline methods.

Second, according to the specific performance values given in Table 4, compared with the best indicator values among the eight baseline methods, our TSC achieves improvements by 7.4% in terms of F-measure, by 7.1% in terms of g-mean, and by 0.5% in terms of Balance. Overall, the improvement is slight. In addition, our TSC method is 0.2% lower than the best performance achieved by TCA+ in terms of G-measure, and 19.7% lower than the best performance achieved by ASCI in terms of MCC.

Third, from Figure 6, we can find that the p-values of Friedman test (all less than 0.05) demonstrate that there exist statistically significant performance differences among the nine methods in terms of all indicators. The CD diagrams show that our TSC method belongs to the top ranking group in terms of all indicators and achieves the second best or third best average rankings on all indicators. Besides, our TSC method has significant differences compared with four supervised baseline methods in terms of F-measure, with three supervised baseline methods in terms of G-measure and MCC, and with two supervised baseline methods in terms of g-mean and Balance.

Answer to RQ2

Although our unsupervised cross-project method TSC does not perform the best in all of performance indicators, it can slightly outperform all baseline supervised cross-project models on some indicators. Since our TSC method does not need to involve in any labelled data, it is highly competitive for the ICFR task in the scenario where labelled data are extremely scarce.

7 | THREATS TO VALIDITY

Threats to External Validity: The generalisation of the experimental results is the main threat to the external validity of our work. In order to alleviate such threat, we use an open-source benchmark dataset which is commonly used in previous studies [2–4, 15–17]. Nevertheless, the projects in the used dataset were developed with Java language. Thus, additional experiments need to be done to verify whether our TSC method can also work well on other projects developed with programming languages like C, C++ or Python for the ICFR task. Another threat lies in that the crashes in the used dataset were simulated using mutation technology, nevertheless, previous researches have proved that the crashes by mutation can be considered as alternative to the real ones in software testing [55]. In this work, we assume that the source and target projects have the same features, this is, we only focus on the cross-project ICFR task under the homogeneous scenario. This will threat the general adaptation of our methods. Considering the heterogeneous ICFR task will be our follow-up work.

Threats to Internal Validity: This kind of threats come from the possible faults during implementing the code for our method and the baseline methods. To minimise this threat, we

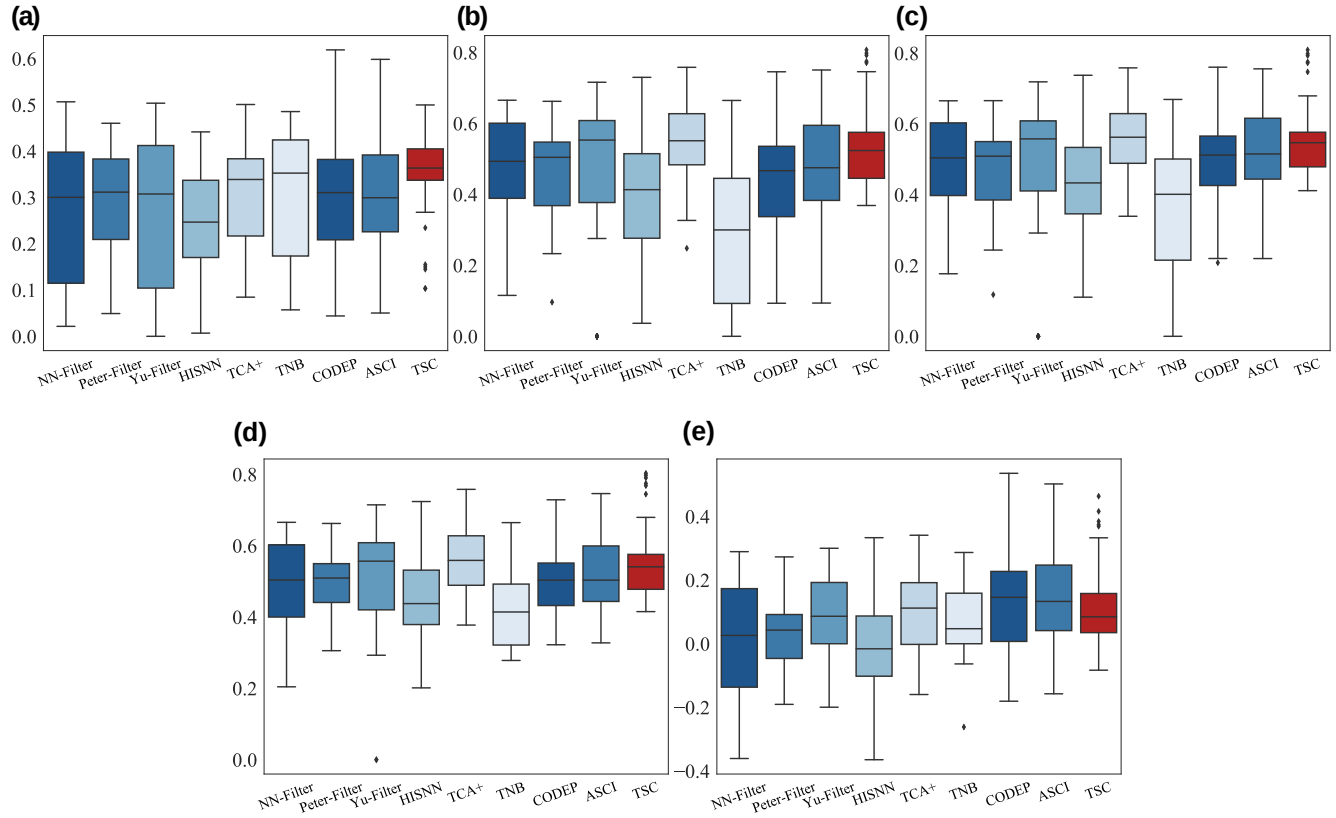

FIGURE 5 Box-plots of TSC and eight supervised baseline methods in terms of five indicators. (a) Box-plots of TSC and eight supervised baseline methods in terms of F-measure. (b) Box-plots of TSC and eight supervised baseline methods in terms of G-measure. (c) Box-plots of TSC and eight supervised baseline methods in terms of g-mean. (d) Box-plots of TSC and eight supervised baseline methods in terms of Balance. (e) Box-plots of TSC and eight supervised baseline methods in terms of MCC

TABLE 4 Average performance of all indicators for TSC and the eight supervised cross project methods across all projects

Indicators	NN-Filter	Peter-Filter	Yu-Filter	HISNN	TCA+	TNB	CODEP	ASCI	TSC
F-measure	0.272	0.284	0.267	0.244	0.315	0.325	0.298	0.314	**0.349**
G-measure	0.459	0.465	0.441	0.392	0.543	0.289	0.445	0.479	0.542
g-mean	0.471	0.471	0.445	0.424	0.428	0.351	0.497	0.523	**0.560**
Balance	0.478	0.491	0.493	0.446	0.555	0.420	0.499	0.524	**0.558**
MCC	0.006	0.026	0.091	−0.013	0.105	0.069	0.135	0.147	0.118

Note: The bold values means the best prediction performance in terms of each indicators.

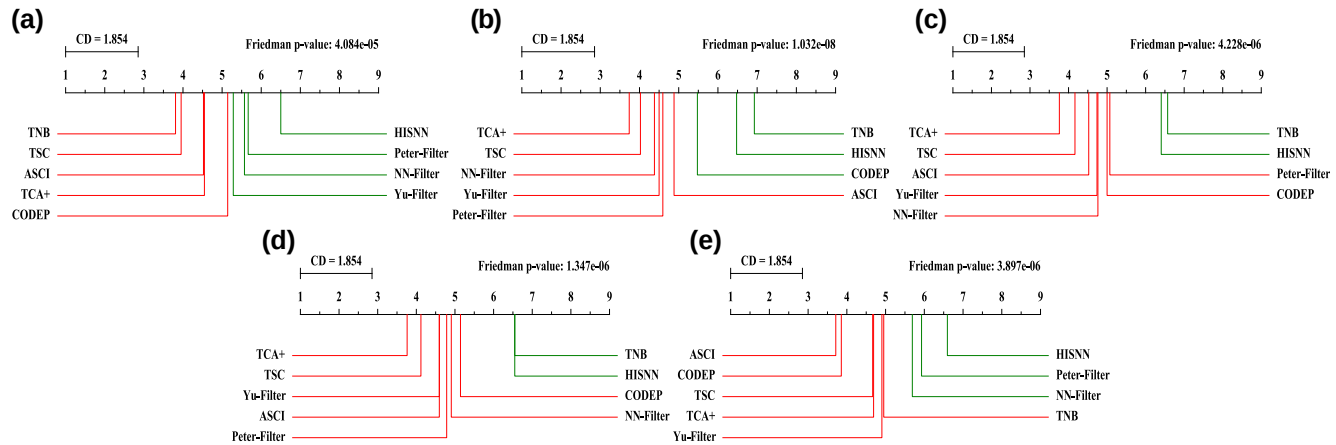

FIGURE 6 Visualised results of statistical test for our TSC method and eight supervised cross-project methods in terms of five indicators. (a) Statistical test results for F-measure. (b) Statistical test results for G-measure. (c) Statistical test results for g-mean. (d) Statistical test results for Balance. (e) Statistical test results for MCC

TABLE 5 Average feature values for crash instances with different labels

Label	Codec	Collections	IO	Jsoup	JSqlParser	Mango	Ormlite
OutTrace	10.04	20.11	21.22	10.76	17.89	14.60	11.97
InTrace	9.64	19.91	18.80	10.30	8.42	13.36	10.63

have double-checked the code manually, and try to use the off-the-shelf methods in the third-party libraries or the ones with existing source code for comparison.

Threats to Construct Validity: Such threats focus on the suitability of the evaluation indicators, statistic test methods and the used labelling strategy. We carefully choose five comprehensive indicators to give a thorough appraisal of the performance and make the result analysis more persuasive. Besides, we use an improved Nemenyi post-hoc test to generate more reasonable method division without overlapping. As mentioned in Subsection 5.3, we set the label of the cluster with smaller average feature values as *InTrace*. We analyse the benchmark dataset and count the average feature values for the crash instances with label *InTrace* and *OutTrace* respectively for each project as showed in Table 5. We can observe that the average feature values for crash instances with label *InTrace* is lower than that with label *OutTrace* over all projects, and the difference on several projects is clear. From this point of view, our labelling strategy for the clusters is rational.

Threats to Conclusion Validity: This kind of threats focus on the experimental results and the conclusion of our work. As the proposed TSC model is an unsupervised learning method, we thus compare it with four unsupervised learning techniques to show its effectiveness. In addition, as our TSC model has the potential to use the unlabelled data from source project to promote the clustering effect in target project, we also compare it with eight classic supervised cross-project techniques. The experimental results illustrate the superiority of our TSC method for ICFR task. More detailed analysis of other supervised and unsupervised learning techniques for ICFR task are also important and we leave it as the future work.

8 | CONCLUSION

If crash defects can be identified in the stack traces, only a small amount of code needs to be reviewed to find the root cause of the crashing faults, which greatly improves testing efficiency and reduces the fault localisation efforts. As the process of label collection is cumbersome and error-prone, using unsupervised models would be an alternative way to assist in software quality assurance activities. In this work, we were the first to introduce a both unsupervised and cross-project model for identifying the residence of crashing fault. This model, called TSC, transfer knowledge from auxiliary unlabelled crash data of other projects to facilitate more effective cluster results on unlabelled crash data of the project we care about. TSC aims to obtain a feature embedding that can link different clustering tasks from multiple project data and hence improve the clustering performance of each other. The experimental results over a benchmark dataset consisting of seven open-source projects show that our TSC method outperforms four typical clustering-based unsupervised methods in terms of five indicators and also performs well in terms of several indicators compared with eight supervised cross-project models.

In future, we plan to extend our TSC method to other software engineering tasks, such as defect prediction and software change prediction. In addition, we try to incorporate the solution of class imbalanced issue into the TSC method to make it more effective.

AUTHOR CONTRIBUTIONS

Xiao Liu: Writing – original draft, Data curation, Software. Zhou Xu: Methodology, Supervision. Dan Yang: Project administration. Mang Yan: Methodology, Conceptualisation, Writing – review & editing. Weihan Zhang: Visualisation. Haohan Zhao: Formal analysis. Lei Xue: Writing – review & editing. Ming Fan: Writing – review & editing.

ACKNOWLEDGEMENTS

This work was supported in part by the Open Foundation of Key Laboratory of Dependable Service Computing in Cyber Physical Society, Ministry of Education of China (No. Grant CPSDSC202004), the National Key Research and Development Project (No. 2021YFB1714200), the National Nature Science Foundation of China (No.62002034), the Fundamental Research Funds for the Central Universities (No. 2022CDJKYJH001), the Natural Science Foundation of Chongqing (No. cstc2021jcyj-msxmX0538), and the Fundamental Research Funds for the Central Universities (xxj022019001, xzy012020009).

CONFLICT OF INTEREST

The authors have no Conflict of Interest.

DATA AVAILABILITY STATEMENT

Data available on request from the authors.

PERMISSION TO REPRODUCE MATERIALS FROM OTHER SOURCES

None.

ORCID

Zhou Xu https://orcid.org/0000-0003-3307-2994
Meng Yan https://orcid.org/0000-0002-9538-9121

REFERENCES

1. Shin, Y., Williams, L.: An empirical model to predict security vulnerabilities using code complexity metrics. In: Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 315–317 (2008)

2. Gu, Y., et al.: Does the fault reside in a stack trace? assisting crash localization by predicting crashing fault residence. J. Syst. Software. 148, 88–104 (2019). https://doi.org/10.1016/j.jss.2018.11.004
3. Xu, Z., et al.: Identifying crashing fault residence based on cross project model. In: 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), pp. 183–194. IEEE (2019)
4. Zhao, K., et al.: Predicting Crash Fault Residence via Simplified Deep Forest Based on a Reduced Feature Set. *arXiv preprint arXiv: 210401768*, (2021)
5. Zhang, F., et al.: Cross-project defect prediction using a connectivity-based unsupervised classifier. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp. 309–320. IEEE (2016)
6. Fu, W., Menzies, T.: Revisiting unsupervised learning for defect prediction. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (FSE), pp. 72–83 (2017)
7. Coviello, C., Romano, S., Scanniello, G.: Poster: cuter: Clustering-based test suite reduction. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), pp. 306–307. IEEE (2018)
8. Jiang, W., Chung, F.l.: Transfer spectral clustering. In: Joint European Conference on Machine Learning and Knowledge Discovery in Databases, pp. 789–803. Springer (2012)
9. Xuan, J., Xie, X., Monperrus, M.: Crash reproduction via test case mutation: let existing test cases help. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (FSE), pp. 910–913 (2015)
10. Dang, Y., et al.: Rebucket: a method for clustering duplicate crash reports based on call stack similarity. In: 2012 34th International Conference on Software Engineering (ICSE), pp. 1084–1093. IEEE (2012)
11. Dhaliwal, T., Khomh, F., Zou, Y.: Classifying field crash reports for fixing bugs: a case study of Mozilla firefox. In: 2011 27th IEEE International Conference on Software Maintenance (ICSM), pp. 333–342. IEEE (2011)
12. Chen, N., Kim, S.: Star: stack trace based automatic crash reproduction via symbolic execution. IEEE Trans. Software Eng. 41(2), 198–220 (2014). https://doi.org/10.1109/tse.2014.2363469
13. Nayrolles, M., et al.: A bug reproduction approach based on directed model checking and crash traces. J. Softw.: Evolution and Process (JSEP), 29(3), e1789 (2017). https://doi.org/10.1002/smr.1789
14. Nayrolles, M., et al.: Jcharming: a bug reproduction approach using crash traces and directed model checking. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 101–110. IEEE (2015)
15. Xu, Z., et al.: Feature selection and embedding based cross project framework for identifying crashing fault residence. Inf. Software Technol. 131, 106452 (2021). https://doi.org/10.1016/j.infsof.2020.106452
16. Xu, Z., et al.: Imbalanced metric learning for crashing fault residence prediction. J. Syst. Software. 170, 110763 (2020). https://doi.org/10.1016/j.jss.2020.110763
17. Zhao, K., et al.: A Comprehensive Investigation of the Impact of Feature Selection Techniques on Crashing Fault Residence Prediction Models. Information and Software Technology (IST), 106652, (2021)
18. Lal, H., Pahwa, G.: Root cause analysis of software bugs using machine learning techniques. In: 2017 7th International Conference on Cloud Computing, Data Science & Engineering-Confluence, pp. 105–111. IEEE (2017)
19. Kahles, J., et al.: Automating root cause analysis via machine learning in agile software testing environments. In: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), pp. 379–390. IEEE (2019)
20. Ma, Q., Li, H., Thorstenson, A.: A big data-driven root cause analysis system: application of machine learning in quality problem solving. Comput. Ind. Eng. 160, 107580 (2021). https://doi.org/10.1016/j.cie.2021.107580
21. Hangal, S., Lam, M.S.: Tracking down software bugs using automatic anomaly detection. In: Proceedings of the 24th International Conference on Software Engineering (ICSE), pp. 291–301. IEEE (2002)
22. Abdelrahman, O., Keikhosrokiani, P.: Assembly line anomaly detection and root cause analysis using machine learning. IEEE Access. 8, 189661–189672 (2020). https://doi.org/10.1109/access.2020.3029826
23. Soldani, J., Brogi, A.: Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: a survey. ACM Comput. Surv. 55(3), 1–39 (2022). https://doi.org/10.1145/3501297
24. Nam, J., Kim, S.: Clami: defect prediction on unlabeled datasets (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 452–463. IEEE (2015)
25. Liu, J., et al.: Code churn: a neglected metric in effort-aware just-in-time defect prediction. In: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 11–19. IEEE (2017)
26. Yan, M., et al.: File-level defect prediction: unsupervised vs. supervised models. In: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 344–353. IEEE (2017)
27. Fan, Y., et al.: The utility challenge of privacy-preserving data-sharing in cross-company defect prediction: an empirical study of the cliff&morph algorithm. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 80–90. IEEE (2017)
28. Huang, Q., Xia, X., Lo, D.: Supervised vs unsupervised models: a holistic look at effort-aware just-in-time defect prediction. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 159–170. IEEE (2017)
29. Zimmermann, T., et al.: Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (FSE), pp. 91–100. ACM (2009)
30. Chang, R., et al.: A novel method for software defect prediction in the context of big data. In: 2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA), pp. 100–104. IEEE (2017)
31. Li, N., Shepperd, M., Guo, Y.: A systematic review of unsupervised learning techniques for software defect prediction. Inf. Software Technol. 122, 106287 (2020). https://doi.org/10.1016/j.infsof.2020.106287
32. Zhang, L., Zhang, L., Khurshid, S.: Injecting mechanical faults to localize developer faults for evolving software. ACM SIGPLAN Not. 48(10), 765–784 (2013). https://doi.org/10.1145/2544173.2509551
33. Moon, S., et al.: Ask the mutants: mutating faulty programs for fault localization. In: Proceedings of the 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, pp. 153–162. IEEE (2014)
34. Pawlak, R., Noguera, C., Petitprez, N.: Spoon: Program Analysis and Transformation in Java. Inria (2006)
35. Ng, A.Y., Jordan, M.I., Weiss, Y.: On spectral clustering: analysis and an algorithm. In: Advances in Neural Information Processing Systems, pp. 849–856 (2002)
36. Dhillon, I.S.: Co-clustering documents and words using bipartite spectral graph partitioning. In: Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 269–274 (2001)
37. Manton, J.H.: Optimization algorithms exploiting unitary constraints. IEEE Trans. Signal Process. 50(3), 635–650 (2002). https://doi.org/10.1109/78.984753
38. Baker, K.: Singular Value Decomposition Tutorial, pp. 24. The Ohio State University (2005)
39. Xu, Z., et al.: A comprehensive comparative study of clustering-based unsupervised defect prediction models. J. Syst. Software 172, 110862 (2021). https://doi.org/10.1016/j.jss.2020.110862
40. Xu, Z., et al.: Ldfr: learning deep feature representation for software defect prediction. J. Syst. Software. 158, 110402 (2019). https://doi.org/10.1016/j.jss.2019.110402
41. Herbold, S., Trautsch, A., Grabowski, J.: A comparative study to benchmark cross-project defect prediction approaches. IEEE Trans. Software Eng. 44(9), 811–833 (2017). https://doi.org/10.1109/tse.2017.2724538

42. Pohlert, T.: The pairwise multiple comparison of mean ranks package (pmcmr). 27, 9 (2019)
43. Béjar Alonso, J.: K-Means vs Mini Batch k-Means: A Comparison (2013)
44. Zhang, T., Ramakrishnan, R., Livny, M.: Birch: an efficient data clustering method for very large databases. ACM Sigmod Record. 25(2), 103–114 (1996). https://doi.org/10.1145/235968.233324
45. Bolla, M.: Spectral Clustering and Biclustering: Learning Large Graphs and Contingency Tables. John Wiley & Sons (2013)
46. Reynolds, D.A.: Gaussian mixture models. Encyclopedia of biometrics 741, 659–663 (2009). https://doi.org/10.1007/978-0-387-73003-5_196
47. Turhan, B., et al.: On the relative value of cross-company and within-company data for defect prediction. Empir. Software Eng. 14(5), 540–578 (2009). https://doi.org/10.1007/s10664-008-9103-7
48. Peters, F., Menzies, T., Marcus, A.: Better cross company defect prediction. In: 2013 10th Working Conference on Mining Software Repositories (MSR), pp. 409–418 (2013)
49. Yu, X., et al.: A data filtering method based on agglomerative clustering. In: Proceedings of the 29th International Conference on Software Engineering and Knowledge Engineering (SEKE), pp. 392–397 (2017)
50. Ryu, D., Jang, J.I., Baik, J.: A hybrid instance selection using nearest-neighbor for cross-project defect prediction. J. Comput. Sci. Technol. 30(5), 969–980 (2015). https://doi.org/10.1007/s11390-015-1575-5
51. Nam, J., Pan, S.J., Kim, S.: Transfer defect learning. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 382–391. IEEE (2013)
52. Ma, Y., et al.: Transfer learning for cross-company software defect prediction. Inf. Software Technol. 54(3), 248–256 (2012). https://doi.org/10.1016/j.infsof.2011.09.007
53. Panichella, A., Oliveto, R., De.Lucia, A.: Cross-project defect prediction models: L'union fait la force. In: Proceedings of the 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), pp. 164–173 (2014)
54. Di Nucci, D., Palomba, F., De Lucia, A.: Evaluating the adaptive selection of classifiers for cross-project bug prediction. In: Proceedings of the IEEE/ACM 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), pp. 48–54. IEEE (2018)
55. Just, R., et al.: Are mutants a valid substitute for real faults in software testing?. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), pp. 654–665 (2014)

How to cite this article: Liu, X., et al.: An unsupervised cross-project model for crashing fault residence identification. IET Soft. 1–17 (2022). https://doi.org/10.1049/sfw2.12073