The Journal of Systems and Software 207 (2024) 111858

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Towards an understanding of intra-defect associations: Implications for defect prediction☆

Yangyang Zhao [a], Mingyue Jiang [a], Yibiao Yang [b], Yuming Zhou [b], Hanjie Ma [a], Zuohua Ding [a,*]

[a] Zhejiang Sci-Tech University, HangZhou, China
[b] Nanjing University, Nanjing, China

ARTICLE INFO

Keywords:
Defect prediction
Intra-defect association
Data processing
Software quality

ABSTRACT

In previous studies, when collecting defect data, if the fix of a defect spans multiple modules, each involved module is labeled as defective. In this context, the defect prediction models are built based on the features of each individual module, ignoring the potential associations between the modules involved in the same defect(referred to as "*intra-defect associations*"). Considering the possibility of numerous cross-module defects in practice, we hypothesize that these intra-defect associations could play a crucial role in enhancing defect prediction performance. Unfortunately, there is no empirical evidence to know that. To this end, we are motivated to conduct a comprehensive study to explore the implications of intra-defect associations for defect prediction. We first examine the proportion of cross-module defects and the relationships between the involved modules. The results reveal that, at function level, the majority of defects occur across functions, with most of the cross-module defects exhibiting implicit dependencies. Inspired by these findings, we propose a novel data processing approach for building defect prediction models. This approach leverages the intra-defect associations by merging the involved modules into new instances with mean or median variables to augment the training data. The experimental results indicate that considering intra-defect associations can significantly improve the defect prediction performance in both the ranking and classification scenarios. This study provides valuable insights into the implications of intra-defect associations for defect prediction.

1. Introduction

In the software development process, defects are prevalent and inevitable. They usually disrupt the normal execution of programs, leading to unexpected and potentially harmful outcomes, which pose threats to the security and reliability of software products (Tripathy and Naik, 2011). Such failures can result in severe consequences including property damage monetary loss, or even human casualty (Wong et al., 2010, 2017). Hence, the timely detection and repair of defects is crucial for developing high-quality software products. Unfortunately, in practice, identifying and fixing software defects can be time-consuming and resource-intensive, accounting for 50% to 75% of the total software development costs (LaToza et al., 2006). Moreover, research has shown that the majority of defects tend to occur in a small number of software modules (Boehm and Basili, 2001). This observation suggests that focusing on testing and debugging a few select modules can effectively uncover most defects, especially when resources are limited. As a solution, software defect prediction has been proposed to optimize the allocation of limited resources by recommending the software modules most prone to defects.

In software defect prediction, datasets are typically gathered from software repositories, such as version control systems and issue tracking systems. In the dataset, an *instance* usually represents a *software module*, which could be a package, file, class or function at various levels of granularity. Each instance consists of multiple independent variables and a dependent variable. The independent variables are known as software metrics, which quantitatively evaluate different attributes of the software, such as size and complexity. The dependent variable represents the defect data of the module, usually indicating whether defects are present or not. Traditionally, the dependent variable is set to 1 if at least one post-release defect is detected in the module, and 0 otherwise. This binary classification approach allows for straightforward judgments regarding the presence or absence of defects in each module. Using the dataset containing software metrics and historical defect data, a defect prediction model can be built to classify whether a new module is defective or not (Song et al., 2010).

As can be seen, defect data plays a crucial role in defect prediction. In previous studies, when constructing defect datasets, a fix that spans

☆ Editor: W. Eric Wong.
* Corresponding author.
E-mail address: zouhuading@hotmail.com (Z. Ding).

https://doi.org/10.1016/j.jss.2023.111858
Received 15 February 2023; Received in revised form 1 August 2023; Accepted 25 September 2023
Available online 30 September 2023

multiple modules is often treated as multiple separate defects, with each involved module considered defective (Walkinshaw and Minku, 2018). Subsequently, a defect prediction model is developed based on the features of each individual module. For example, if a patch that fixes defect d_1 involves modification in two modules m_1 and m_2, both m_1 and m_2 are labeled as defective with a dependent variable of 1, leading to the counting of two defects in this case. Here, d_1 is a ***cross-module defect*** (i.e. a defect occurring across modules), and m_1 and m_2 are ***intra-defect modules*** (i.e. modules involved in the same defect). In terms of the cross-module defect d_1,it is intuitively assumed that there are associations between the involved modules m_1 and m_2, as they are modified simultaneously to fix d_1. These potential associations between the intra-defect modules are referred to as ***intra-defect associations***, which have not been considered in the prior studies when building defect prediction models. This gives rise to the question that whether the features of a single module are sufficient to predict the fault proneness. Hypothetically, if there are a substantial number of cross-module defects in practice (*hypothesis-1*), and certain relationships exist between the intra-defect modules(*hypothesis-2*), then we infer that these specific relationships could be leveraged to enhance defect prediction performance(*hypothesis-3*). Unfortunately, little is currently known on these hypotheses.

We attempt to fill this gap by comprehensively investigating the value of intra-defect associations on defect prediction. To achieve this, we first verify *hypothesis-1* and *hypothesis-2* by studying the proportion of cross-module defects and the relationships between the intra-defect modules. Subsequently, based on these verifications, we further explore the effectiveness of considering intra-defect associations when building defect prediction models to confirm *hypothesis-3*. More specifically, Our study aims to answer the following research questions:

RQ1*: To what extent do the defects occur within a single module or across modules?*

RQ2*: What are the relationships between the intra-defect modules?*

RQ3*: What is the value of intra-defect associations on the performance of fault-proneness prediction?*

We collect defect data from open-source repositories and conducting an empirical investigation into the frequency of cross-module defects at the granularity of files and functions. Subsequently, we explore the relationships between intra-defect modules and classify them into two categories: *explicit dependency* (when there is a directed dependency between the involved modules, e.g. function f_1 calls f_2, where f_1 and f_2 are involved in the same defect) and *implicit dependency* (when there is no directed dependency between the involved modules, e.g. even though there is no direct invocation or data dependency between function f_1 and f_2, they are simultaneously modified to fix a specific defect). Additionally, we design a DD metric to further evaluate the degree of direct dependencies for defects with explicit dependencies. The experimental results show that: (1) At the file level, 81% of defects occur within a single file. While at the function level, a considerable number of defects span across functions, with an average percentage of 60.5%; (2) The majority of cross-module defects (about 90%) exhibit only implicit dependencies, indicating the absence of direct dependencies between the involved modules. Even for the minor defects with explicit dependencies, the degree of direct dependencies is relatively low, with a DD value ranging from 0.006 to 0.083.

Based on the findings from RQ1 and RQ2, we propose a novel data processing approach for building defect prediction models. This approach leverages the intra-defect associations by merging the involved modules into new instances with the mean or median variables to augment the training data. To conduct a comprehensive evaluation, we build three types of multivariate logistic regression models: "C" model (utilizing the intra-defect associations), "T" model (based on the features of each individual module, as done in prior studies), and "S" model (employing SMOTE technique for data processing). We conduct a comparison between the performance of the C model and the T model in both ranking and classification scenarios. Based on nine widely used systems, the results reveal that C model is significantly more effective than T model in at least 4 systems at the significant level of 0.05, and exhibits positive improvements over T model in at least five systems. In addition, to further validate our findings, we conduct additional comparisons between C model and S model, as well as S model and T model, to investigate whether intra-defect associations significantly contribute to the performance improvement of C mode. The results demonstrate that the C model significantly outperforms the S model in at least 4 datasets, with positive improvements in at least 6 datasets. Moreover, S model exhibits superior predictive performance compared to the T model in the majority of systems. Notably, despite the improvement caused by data augmentation, the enhancement achieved by incorporating intra-defect associations is considerable, highlighting its significance for defect prediction. Overall, our findings indicate that, considering the intra-defect associations can significantly improve the defect prediction performance in both the fault-proneness ranking and classification scenarios.

This comprehensive study sheds light on the occurrence patterns and relationships between intra-defect modules, and provides an in-depth understanding on the value of intra-defect associations for fault-proneness prediction.

Paper outline The rest of this paper is organized as follows. Section 2 explains the research motivation. Section 3 presents the experimental methodology, including the research questions, the studied systems, the data collection, the intra-defect dependency measurement, the modeling technique and the evaluation methods. In Section 4, we reports the experimental results. Section 5 describes the threats to the validity of our study, followed by Section 6 presenting related work. Finally, Section 7 concludes the paper.

2. Motivation

Software defect data, as labels in the labeled training data, is crucial for training defect prediction models. In previous works, two primary types of defect data have been utilized to construct prediction models: (1) the defect data collected by researchers themselves from real-word systems; (2) the publicly available defect datasets, such as PROMISE, AEEEM (D'Ambros et al., 2012), and ECLIPSE (Zimmermann et al., 2007).

For the first type, defect data is typically mined from open-source software repositories, such as version control systems (e.g. Github, CVS) and issue tracking systems (e.g. JIRA, BUGZILLA). These repositories are widely used for project management during software development. Specifically, version control systems store information about software developments, such as source codes and change logs. On the other hand, issue tracking systems contain detailed defect information, such as bug reports and their status. This information is valuable for software quality assurance, especially for defect prediction (Bachmann et al., 2010). To collect defect data from these repositories, existing studies mostly leveraged the SZZ algorithm (Śliwerski et al., 2005) and its variants (Kim et al., 2006; Da Costa et al., 2016) to infer bug-inducing commits based on bug-fixing commits. The main steps are as follows: (1) identify the bug-fixing commits by establishing links between bugs and their fixes; (2) leverage the diff command to locate the modified lines purely for fixing bugs; (3) trace back through the code change history to identify the commits that induce defective lines using the annotate and blame commands; and (4) finally, filter out the innocent changes, such as modifications to blank lines and comments (Kim et al., 2006; Da Costa et al., 2016). Secondly, in terms of the publicly available defect datasets, the PROMISE repository offers several datasets for empirical software engineering study. The AEEEM dataset, collected by D'Ambros et al. (2012), is used to benchmark different defect prediction models. The ECLIPSE bug dataset, collected by Zimmermann et al. (2007), contains defect data from multiple Eclipse releases (e.g. 2.0, 2.1 and 3.0). These publicly available defect datasets are widely utilized as

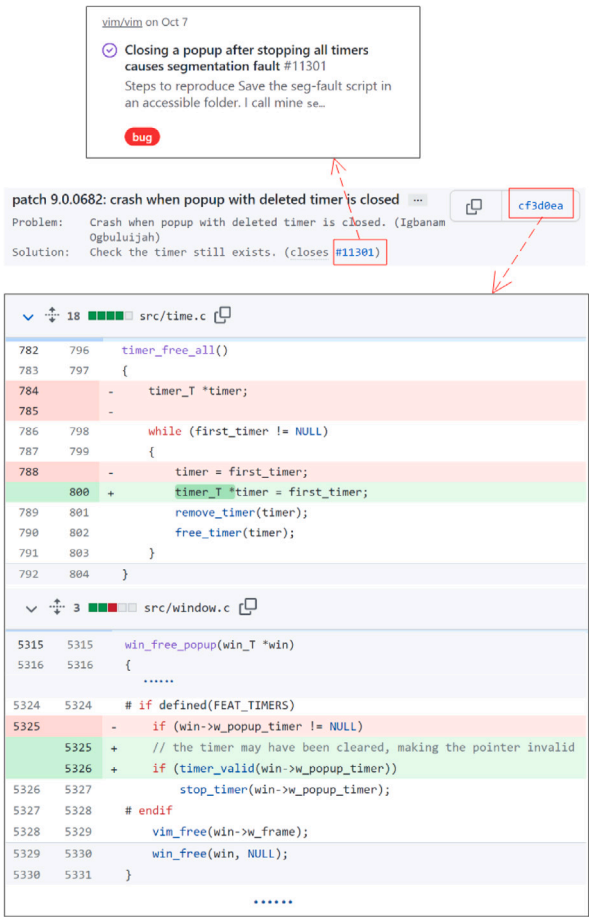

Fig. 1. A real bug-fixing example on Vim.

benchmark datasets and are also collected from version archives and bug tracking systems.

Traditionally, the above defect data, whether self-collected or publicly available, is obtained with the assumption that each defect is fixed by modifying a single module (e.g. file), therefore a fix that spans multiple modules is treated as multiple separate defects (Walkinshaw and Minku, 2018). However, in the actual development process, code changes for bug fixes may involve multiple modules. For instance, as shown in Fig. 1, the commit $cf3d0ea$ was submitted to fix the bug issue *#11301*, which involved modifications on multiple functions(i.e. *timer_free_all* and *win_free_popup*) in the files *time.c* and *window.c*, respectively. In this case, using the traditional approach would result in at least two defects being counted. When constructing defect datasets, for bug #11301, each involved module is considered defective. Taking the file level as an example, both *time.c* and *window.c* would be marked as defective. Subsequently, when building a predictive model, each of *time.c* and *window.c* is treated as a separate instance with its own features and a dependent variable of 1, ignoring the associations between them. In this context, it gives rise to the question that whether it is sufficient to predict defects based solely on features from individual modules .

Recently, Gesi et al. (2021) reported on the distribution of the number of edited files per commit, highlighting that the majority of commits involve less than 10 edited files. It aroused our curiosity on the distribution of the number of modules involved in a defect. However, little is currently known about that. Many prior works have assumed one fix per defect (Walkinshaw and Minku, 2018). If the fix spans multiple modules, the defect is a cross-module defect. For a given cross-module defect, we hypothesize that there should be associations among the involved modules, as they were modified simultaneously to fix the defect. Consequently, if there exist a number of cross-module defects in practice, this leads us to conjecture that considering the associations between intra-defect modules may affect the performance of fault-proneness prediction. However, there is currently insufficient empirical evidence to validate this hypothesis.

As discussed above, the existing defect prediction models typically rely on features from individual modules, overlooking the associations among different modules involved in a defect. This deficiency may limit the accuracy and applicability of current defect prediction models. Firstly, in complex software systems, defects can arise due to interactions and dependencies between different modules. For instance, a change in one module might introduce a defect that affects the behavior or performance of another module. Addressing such defects may require coordinated modifications to multiple modules. By not accounting for such cross-module defects, existing defect prediction models may fail to capture the full extent of potential issues, leading to potentially misleading predictions. Moreover, by neglecting intra-defect associations, the models overlook the potential spread of defects within a module or across different modules. This incomplete understanding of defect propagation may hamper the identification and mitigation of defects, as the models fail to consider the cascading effects and broader impact of defects within and across modules.

To this end, we are motivated to conduct a comprehensive study on cross-module defects and the implications of intra-defect associations on defect prediction.

3. Experimental methodology

3.1. Research question

In previous studies, it is the common practice to assume one fix per defect. If a patch purely for bug fixing spans multiple modules, each module is marked as buggy. Subsequently, the prediction models are built using the features of individual modules, ignoring the associations between the intra-defect modules. This raises the question that whether considering intra-defect associations can improve defect prediction performance. However, there is little empirical evidence to know that. In this study, we aim to comprehensively analyze the value of intra-defect associations for the fault-proneness prediction. To gain insights into cross-module defects and explore the significance of intra-defect associations, We will address the following three questions.

***RQ1**: To what extent do the defects occur within a single module or across modules?*

***RQ2**: What are the relationships between the intra-defect modules?*

***RQ3**: What is the value of intra-defect associations on the performance of fault-proneness prediction?*

To establish the necessary of studying intra-defect associations for defect prediction, we begin with *RQ1*, which investigates the occurrence of defects across modules in practice. If the results reveal a non-negligible proportion of defects spanning multiple modules, we hypothesize that the associations among these intra-defect modules could play a crucial role for defect prediction. Subsequently, further study is needed to determine whether incorporating such associations improves the prediction performance when constructing defect prediction models. RQ3 attempts to verify this. To guide the appropriate representation of intra-defect associations, *RQ2* is set to further explore the dependencies between modules that were fixed simultaneously. The insights gained from RQ2 will be influential in shaping the design of RQ3. For example, if a considerable number of defects exhibit explicit dependencies, and the degree of dependencies between the involved modules is substantial, then it becomes imperative to consider the information of direct dependencies to effectively represent intra-defect associations. On the other hand, if implicit dependencies are prevalent, we must explore how to leverage these implicit dependencies effectively. Based on the findings of RQ1 and RQ2, the primary objective

Table 1
Descriptive information of the studied systems.

Subjects	Version	Size(KSLOC)	#File	#Function
Vim	7	197	379	5323
Subversion	1.4.0	193	802	7386
Gstreamer	1.0.1	127	550	5726
Gst-plugins-base	1.10.0	227	651	7787
Libgcrypt	1.4.2	41	125	1247
Make	3.8	15	64	348
Gimp	1.3.0	617	1746	16998
Wine	1	1361	2951	49480
Libav	0.7	365	1395	9833

of RQ3 is to verify whether considering intra-defect associations is effective for improving the performance of fault-proneness prediction.

The exploration of the above three questions can help both researchers and practitioners to comprehensively understand the cross-module defects and intra-defect associations. Additionally, the findings will serve as crucial guidance for developing more effective fault-proneness prediction models in practical software development scenarios.

3.2. Subject systems

Our study is performed on nine well-known open-source systems written in C language, including Vim, Subversion, Gstreamer, Gst-plugins-base, Libgcrypt, Make, Gimp, Wine and Libav. Table 1 illustrates the descriptive information for them. The first column lists the names of the studied systems, followed by the selected versions in the second column. The last three columns present the total size (measured in thousands of source code lines), the number of files, and the number of functions for each system.

We select these systems for the following reasons. Firstly, these systems are widely recognized and used in the software development community, making our research findings more applicable and relevant to real-world scenarios. Secondly, these systems are well-documented and studied in the research community, making our study based on them more credible. Thirdly, the availability of their development process data and defect data enables us to perform thorough and in-depth analyses. Lastly, by selecting versions of these systems that have been released for at least 5 years, we ensure that they have undergone extensive testing and bug-fixing processes. This guarantees that we are working with mature versions where most of the bugs have been detected and addressed, making the collected defect data more comprehensive.

3.3. Data collection

In the publicly available defect datasets, such as PROMISE and AEEEM, modules are treated as individual instances and labeled based on whether they have defects or not. However, these datasets lack information about the relevant associations between modules involved in a defect, which is fundamental for our study. Therefore, they are not suitable for serving as experimental data in our study. We need to conduct a new data collection process to obtain the necessary defect data that aligns with the objectives of our research.

The process of data collection in our study is illustrated in Fig. 2. Following prior works (Hoang et al., 2019; McIntosh and Kamei, 2017), we identify the bug-inducing commit candidates using SZZ algorithm (Śliwerski et al., 2005), which is the most commonly used approach for mining defect data and has been demonstrated to be effective (Fan et al., 2019). The main steps are as follows. Firstly, we scan all historical data (from its appearance to 2022-08-01) stored in Github for each studied system, including all code commits and issues. From the collected data, we identify commits that addressed post-release defects (i.e. bug-fixing commits). Following the original

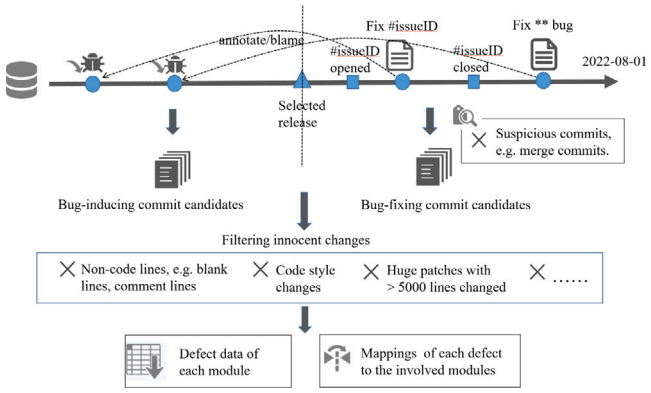

Fig. 2. The process of defect data collection.

SZZ algorithm, we mainly extract bug-fixing commit candidates in two ways: (1) by checking whether the commit message contains descriptions of bug fixes. If so, it is considered a fixing commit; (2) by establishing a link between a commit and an issue through the associated issue ID in the commit message. The commit is marked as a fixing commit if the linked issue: (a) is marked as a bug; (b) was opened before the commit and closed after the commit; and (c) contains a description such as "fix *#issueID*" in the commit message. Next, once we have identified the bug-fixing commits, we leverage the diff command to locate the modified lines related to bug fixing. Finally, we trace back through the code history using the annotate and blame commands to recognize the bug-inducing commits that introduced the buggy lines.

In addition, to enhance the data quality and mitigate the noise caused by the original SZZ algorithm, we do additional filtering processes during the data collection. As done in previous works (Da Costa et al., 2016; McIntosh and Kamei, 2017; Zeng et al., 2021), we remove suspicious commits that do not change the source code, including branch commits (which copy operations from one branch to another) and merge commits (which consist in applying the changes performed in one branch to another). Furthermore, in the remaining commits, as Kim et al. did (Kim et al., 2006), we filter out code style changes and ignore the bug-fixing changes on non-code lines, such as comment lines and blank lines. Moreover, we ignore too hug patches with more than 5000 lines changed, as they are less likely to be bug fixes and are more likely to be related to functionality changes or code refactorings.

After performing the above steps, many pairs of bug-inducing commits and bug-fixing commits are generated. To ensure that the selected defects are present in the chosen versions of the software, we filter these commit pairs based on the following condition: a defect is considered to exist in the current version only if the buggy code was introduced before the version's release and fixed after the version's release. In this way, we identify the valid bug-inducing commits that are relevant to the defects in the selected versions, while excluding those that are not directly associated with the current version's codebase.

$$T_{bugInducing} < T_{versionRelease} < T_{bugFixing} \quad (1)$$

Since it takes time for bugs to be found and fixed, if a bug in the current version has not been discovered or fully fixed, the involved modules might be treated as defect-free, which lead to false negatives (FN) in the defect dataset (Ahluwalia et al., 2019). To address the issue of false negatives in the defect dataset, we consider the time interval between bug appearance and bug fixing. Previous studies have reported that the time to fix a bug ranges from 100 to 200 days (Kim and Whitehead, 2006). Recently, (Zeng et al., 2021) also illustrated the median time interval for the defect from appearance to be fixed, which ranges from 243 to 526 days. To mitigate the impact of authentication latency, we carefully select versions for our study that have been

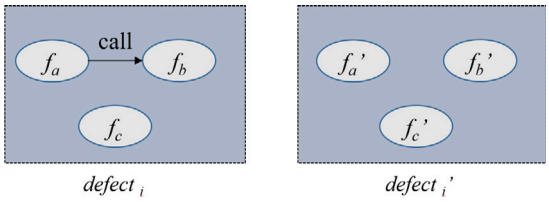

Fig. 3. An example of two defects with different types of relationships.

released for at least 5 years (as indicated in Table 1). This ensures that all (or at least most) bugs have been detected and fully addressed in these versions. Bugs that have not been discovered or fixed within this time interval are considered to have little impact on software quality.

Through the above processes, we finally collect the defect data, and establish mappings of each defect to the involved modules. With the collected data, we can study RQ1 by investigating the extent to which defects occur within single module or across multiple modules at the granularity of both files and functions.

3.4. Internal dependencies measurement

In terms of RQ2, we aim to explore the relationships between the intra-defect modules. In this study, our hypothesis is that there are associations among the modules involved in a single defect, as they are simultaneously modified to fix a bug. We categorize these relationships into two types: *explicit dependency* and *implicit dependency*. Explicit dependency refers to a directed dependency between the intra-defect modules, such as when function f_a directly calls function f_b, and both functions are involved in the same defect. On the other hand, implicit dependency implies that there is no direct directed dependency between the intra-defect modules, but they are modified simultaneously to fix the defect. Fig. 3 illustrates an example of two defects with different types of relationships. In this figure, $defect_i$ is fixed by modifying functions f_a, f_b, and f_c, while $defect_i'$ is resolved by modifying functions f_a', f_b', and f_c'. For each cross-module defect, a similar dependency graph can be created, with modules as nodes and the relationships between modules as edges. We consider $defect_i$ as a defect with an explicit dependency, as function f_a calls function f_b. In contract, $defect_i'$ is treated as a defect with only implicit dependencies.

To answer RQ2, we first study the proportion of defects with explicit or implicit dependencies between the involved modules. Furthermore, we define a metric called "DD" (i.e. the degree of explicit intra-defect dependencies) to quantitatively evaluate the degree of directed dependencies between the intra-defect modules. "DD" is inspired by Vernazza et al.'s component cohesion metric (Vernazza et al., 2000). The corresponding definition of "DD" is in Eq. (2). Numerically, for a given cross-module defect, DD is equal to the ratio of the number of directed dependencies between its involved modules to the maximum possible number of directed dependencies (i.e. $n * (n-1)$), where n represents the number of modules that were modified simultaneously to fix the defect. To illustrate the calculation of DD, we take $defect_i$ in Fig. 3 as an example. With the dependency graph of $defect_i$, we can get the value of DD for $defect_i$, which is 1/(3*2). The higher the complexity of the graph, the greater the value of DD. By employing the DD metric, we can gain valuable insights into the extent of explicit dependencies between the intra-defect modules, shedding light on the nature of cross-module defects and how the involved modules are interconnected in the context of defect fixing.

$$\mathrm{DD} = \sum_{i=1}^{n} \sum_{j=1, j\neq i}^{n} depend(m_i, m_j)/(n * (n-1)) \tag{2}$$

where $depend(m_i, m_j) = \begin{cases} 1 \text{ if } m_i \text{ directly depends on } m_j \\ 0 \text{ otherwise} \end{cases}$

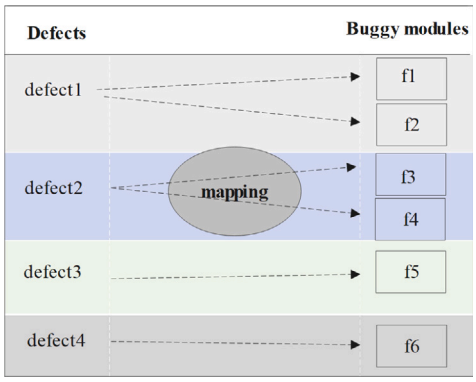

Fig. 4. An example of defect-to-module mappings.

3.5. Modeling technique

In order to study RQ3, we utilize multivariate logistic regression analysis to investigate the value of intra-defect associations on fault-proneness prediction. Logistic regression is a prominent algorithm known for its simplicity and low computational complexity compared to more intricate models. It has become a preferred choice for constructing fault-proneness prediction models in various research studies (Zhao et al., 2017; Yang et al., 2014; Zeng et al., 2021), as it is easy to use and understand. Notably, it has been observed that Logistic regression consistently ranks among the top-performing modeling techniques for defect prediction (Hall et al., 2011; Zeng et al., 2021). Moreover, in a comprehensive benchmarking study, logistic regression demonstrated comparable performance to 16 other classifiers, emphasizing its effectiveness and reliability (Lessmann et al., 2008).

In terms of multivariate logistic regression, we collect the data at the function level(inspired by the findings of RQ1). In the datasets, each instance consists of multiple independent variables and one dependent variable. The independent variables in this study consist of the most widely used traditional metrics, including one size metric, 11 structural complexity metrics, 11 Halstead complexity metrics, and 3 code churn metrics. Table 2 lists the description for each metric. The dependent variable Y in this study can only take the value of 1 or 0, indicating buggy or clean. We build multivariate logistic regression models to predict the probability of $Y = 1$, which indicates post-release fault-proneness.

To investigate the value of intra-defect associations, we build three kinds of multivariate logistic regression models: (1) "T" model (based on the features of each individual module as the existing works did); (2) "C" model (leveraging the intra-defect associations); and (3) "S" model (employing SMOTE technique for data processing). The approaches for building these models are described in detail below.

To illustrate the model building approaches, we present a simplified function-level example as shown in Fig. 4, which demonstrates several cases through the mappings of the defect to the involved defective modules. As depicted in Fig. 4, four defects are illustrated. Among them, $defect1$ occurs in functions $f1$ and $f2$, while $defect2$ involves functions $f3$ and $f4$. On the other hand, $defect3$ occurs within function $f5$, and $defect4$ occurs in function $f6$. It can be seen that, in this example, $defect1$ and $defect2$ are cross-module defects, while the others are single-module defects.

T model: In the traditional approach, if there is at least one post-release defect detected in a module, regardless of whether the defect occurs solely in this module or spans other modules, the dependent variable for this module is set to 1. As a result, all six functions illustrated in Fig. 4 are considered defective and marked as 1. The defect prediction model is built using the features of individual modules, without considering the associations between modules involved

Table 2
The most commonly used traditional code metrics.

Characteristic	Metrics	Description
Size	SLOC	Source lines of code in a function (excluding blank lines and comment lines)
Structural complexity	FANIN	Number of calling functions plus global variables read
	FANOUT	Number of called functions plus global variables set.
	NPATH	Number of unique paths though a body of code
	Cyclomatic	Cyclomatic complexity
	CyclomaticModified	Modified cyclomatic complexity
	CyclomaticStrict	Strict cyclomatic complexity
	Essential	The Cyclomatic complexity after iteratively replacing all well structured control structures with a single statement
	Knots	Measure of overlapping jumps
	MaxNesting	Maximum nesting level of control constructs
	MaxEssentialKnots	Maximum Knots after structured programming constructs have been removed.
	MinEssentialKnots	Minimum Knots after structured programming constructs have been removed.
Halstead complexity	n1	The number of unique operators
	n2	The number of unique operands:
	N1	The total number of operators
	N2	The total number of operands
	n	The sum of the number of unique operators and operands
	V	The information contents of the program, measured in mathematical bits
	D	The difficulty level or error proneness
	L	The inverse of the error proneness of the program
	E	The effort to implement
	T	The time to implement or understand a program
	B	The number of delivered bug
Code Churn	Added	Added source lines of code, normalized by function size
	Deleted	Deleted source lines of code, normalized by function size
	Changed	Modified source lines of code, normalized by function size

in a defect. Following the traditional approach used in prior studies, "T" model is built in this way, which is illustrated in Fig. 5. In the collected data, each instance represents a function, and consists of its own features (i.e. X_{fi}) and its label (i.e. 1 or 0). In terms of the example in Fig. 4, all the six defective functions are positive instances with a label of 1. The data is randomly divided into two parts for training and testing. As an example, f_1, f_2 and f_5 are used for training, and f_3, f_4 and f_6 are used for testing. Subsequently, the model is trained on the training data using multivariate logistic regression, and then evaluated on the testing data by comparing the predicted labels with the actual labels.

C model: Different from the traditional approach, we propose a novel data processing method to build the "C" model by leveraging the intra-defect associations of defects. The data processing algorithm is illustrated in Fig. 6, and the details are as follows. (1) *Constrained division*. After the data collection, we obtain the mappings of each defect to the involved modules. Based on the mapping information M, we first bind the intra-defect modules together. With these cross-module constraints, we randomly divide the dataset P into two parts, namely $P1$ (for training) and $P2$ (for testing). This division ensures

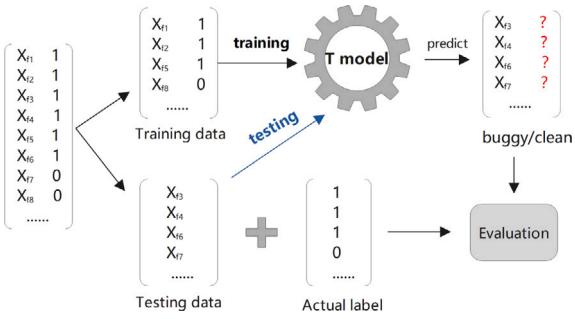

Fig. 5. The traditional approach for building "T" model.

C model building algorithm

Input:
P={(X_1,y$_1$),(X_2,y$_2$),...,(X_m,y$_m$)} // Dataset P with m instances, each contains a set of features X_i and a label y_i, i.e. (X_i,y_i)
D={d_1,d_2,...,d_n} // Defect set D with n defects
M={(d_1,M_1),(d_2,M_2),...,(d_n,M_n)} // Mappings of each defect d_i to the involved modules M_i
Output: Y'={y_1',y_2',...,y_k'} // Predicted results

$P1$, $P2$ = constrainedDivision (P) // $P1$ for training, $P2$ for testing
$P1'$=$P1$ and $P2'$={}
// Leverage the cross-module associations on the training data
foreach d_i in D do:
  if $P1$ contains M_i: // M_i is the set of modules involved in defect d_i
    S = []
    foreach m_j in M_i do:
      S.addRow(X_{mj})
    $P1'$.add((average(S),1)) // Add a new instance by averaging the X of the involved modules
    $P1'$.add((median(S),1)) // Add a new instance by calculating the median value of X of the involved modules
C=trainingModel($P1'$)
foreach i_i in $P2$ do: // Iterate over all instances in the dataset $P2$ to filter the instances used in training set
  if i_j is not used in $P1$:
    $P2'$.add(i_j)
Y'=predict(C, $P2'$) // Test C model in $P2'$
return Y' // The predicted labels for the instances in $P2'$, Y'={y_1',y_2',...,y_k'}, where k = | $P2'$|

Fig. 6. Algorithm of data processing approach for C model.

that the bound modules are placed in the same part for either testing or training as much as possible. The purpose of this process is to ensure that modules related to the same defect receive appropriate consideration, thereby reducing potential biases introduced by module separation. By binding the relevant modules together, we can capture the intra-defect associations between them. We aim to avoid the situation where all modules involved in cross-module defects are intentionally placed in the training set, resulting in only modules involved in single-module defects and non-defective modules in the test set. Such an unrealistic situation could introduce bias in the performance evaluation. Consequently, the partitioning process is random. (2) *Intra-defect associations utilization*. In order to explore an appropriate way to leverage intra-defect associations, we conduct a preliminary study of RQ1 and RQ2. The experimental results, which are detailed in the following Section 4, reveal that most defects involve modules with no direct dependencies. Therefore, as a preliminary attempt, we represent the intra-defect associations by considering the implicit dependencies. Specifically, we merge the intra-defect modules as new instances to take advantage of the potential associations. For each bound set of modules M_i, we create two cross-module instances. One is obtained by averaging the independent variables X of the involved modules, and the other is obtained by calculating the median values of the independent variables X of the involved modules. As reported by Boehm and Basili (2001), 80 percent of defects comes from 20 percent of modules, resulting in data imbalance issues where most modules are defect-free. To this end, we keep the original single-module instances and augment the new cross-module instances in the training data. After this process, the intra-defect associations are effectively utilized. With the processed

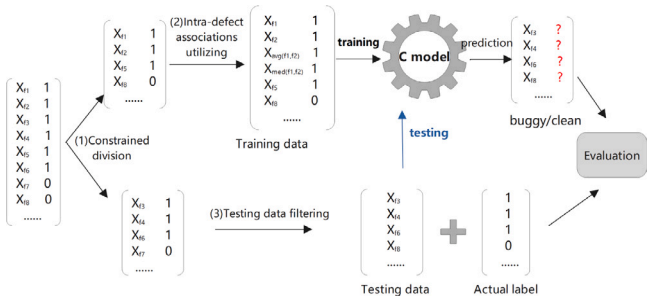

Fig. 7. The approach for building C model.

training data $P1'$, we can build "C" model to predict the fault proneness of functions. (3) *Testing data filtering*. In the process of constrained division, a function involved in multiple defects may be bound to different functions multiple times. As a result, it could appear in both the training set and the test set simultaneously when randomly divided. To ensure an unbiased evaluation, we further remove the instances that have already been used in the training set. (4) *Evaluation*. After the aforementioned processing, the "C" model, built using the training data $P1'$, is used to predict whether the modules in the testing data $P2'$ are defective or not. The performance of "C" model is then evaluated by comparing the real labels and the predicted labels of $P2'$.

In terms of the simplified example, the main processes for building C model are illustrated in Fig. 7. We first bind $f1$ and $f2$, $f3$ and $f4$ respectively, as $defect1$ spans $f1$ and $f2$, and $defect2$ spans $f3$ and $f4$. In this way, they can be allocated to the same data set for either training or testing. Under such cross-module constraints, the dataset is randomly divided into two parts. As an illustrative example, $f1$ and $f2$ are grouped together and randomly assigned to the training set, while $f3$ and $f4$ are grouped together and randomly assigned to the training set. In the training set, we leverage the intra-defect associations of $defect1$ by merging the involved modules $f1$ and $f2$ into two new instances. One consists of the mean values of each independent variable of $f1$ and $f2$ (i.e. $X_{avg(f1,f2)}$, equal to $(X_{f1} + X_{f2})/2$), and a label marked as 1. The other consists of the median values of each independent variable of $f1$ and $f2$ (i.e. $X_{med(f1,f2)}$), as well as a positive label. After utilizing the intra-defect associations, C model is built on the processed training data. For the second part for testing, we filter the instances that have already been used in the training data. Since this is not the case in this simplified example, no filtering is required. Subsequently, we use the independent variables of each function in the testing data as input to C model to predict whether the function is buggy or clean. By comparing the predicted labels with the actual labels, we can evaluate the C model's effectiveness in predicting the fault proneness.

S model: As show in Fig. 7, new positive instances with intra-defect associations are augmented in the training data to build the C model. Even if the results show that the C model outperforms the T model as expected, it remains confusing whether it is indeed caused by the data augmentation or the intra-defect associations. Therefore, to avoid biased conclusions, we build a S model, in which the training dataset is augmented using the most commonly used over-sampling technique SMOTE (Chawla et al., 2002). SMOTE, as a pioneer in dealing with data imbalances (Fernández et al., 2018), is considered to be one of the most influential algorithms for data preprocessing (García et al., 2016). When building S model, the data is first randomly divided into two parts for training and testing. Specifically, we process the training data by employing SMOTE to add new positive instances. Then the S model is trained on the processed training data and tested on the testing data.

As can be seen, the C model is essentially an extension of the T model, incorporating the proposed processing approach of introducing intra-defects associations. Through the comparison between the C model and the T model, we could verify whether this processing approach indeed enhanced the prediction performance. On the other hand, S model is essentially an extension of the T model with data augmentation. Comparing the C model with the S model allows us to determine whether the performance improvement in the C model results from the consideration of intra-defect associations o or data augmentation.

When building the above models, we use the p-value as the criterion to perform a backward stepwise variable selection. In the backward approach, the variable selection begins with a model that contains all variables under consideration. The least significant variables will be removed one after the other. Repeat this process until the model fails to improve. Furthermore, we the Cook's distance to diagnose the influence observations. In our study, if the Cook's distance of an observation is greater than 1, the observation is regarded to be influential and subsequently is excluded from the model building.

Due to the small proportion of post-release defective functions in the datasets, we adopt 30 times 3-fold cross-validation (instead of 10-fold cross-validation) for each project to evaluate the performance of the prediction models. At each 3-fold cross-validation, we randomly divide the data set into three parts. Particularly, since the data division for C model is constrained by mappings of defects to the involved modules, we perform special processing to ensure a balanced distribution of modules across the folds. Specifically, if the number of modules in the randomly selected bound set is less than the number needed for the current fold, all of them will be allocated to this fold. On the contrary, if the number is exceeded, the required number of modules will be randomly selected from the set, and the others will be allocated to other folds. With these 3 folds, 2 folds are used as training data for training and the remaining fold is used as testing data for testing. Each part is used as the testing data to compute the effectiveness of the prediction models built on the rest of the dataset. We repeat the 3-fold cross-validation process 30 times to mitigate possible sampling bias in random splits. Based on effectiveness values obtained from 30 times 3-fold cross-validation, we use the Wilcoxon's signed-rank test to examine whether "C" model significantly outperforms other models.

3.6. Evaluation methods

Since effort-aware performance indicators are more informative than non-effort-aware indicators, recently many studies use effort-aware indicators to evaluate the effectiveness of defect prediction models (Rahman and Devanbu, 2013; Zhou et al., 2018; Xu et al., 2021; Qu et al., 2021). Unlike non-effort-aware evaluation, effort-aware performance evaluation assumes that the costs of applying quality assurance activities vary by modules. In line with prior studies (Rahman and Devanbu, 2013; Zhou et al., 2018; Qu et al., 2021), we also employ the SLOC in a module as the proxy of the inspecting or testing effort of this module. We study RQ3 in the two typical application scenarios, including ranking scenario and classification scenario. More specifically, we compare the effort-aware ranking performance of "C" models against other models to investigate the value of intra-defect associations on ranking performance. In addition, we compare the effort-aware classification performance of "C" models against other models to study whether considering intra-defect associations can improve the classification effectiveness of defect prediction.

(1) Ranking scenario

In the ranking scenario, we evaluate the effort-aware ranking performance of defect prediction models using the most extensively used cost effectiveness measure(CE) (Arisholm et al., 2010). CE is calculated based a "SLOC-based" Alberg diagram, as shown in Fig. 8. The x-axis in the diagram shows the cumulative percentage of SLOC of the selected module from the ranking list, and the y-axis denotes the cumulative percentage of defects detected in these modules. The three curves represent the optimal model, the prediction model and the random

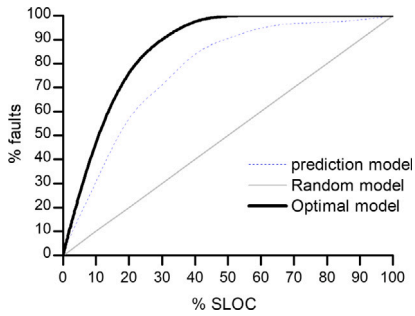

Fig. 8. SLOC-based Alberg diagram.

model respectively. Particularly, the modules in the optimal model are ranked in descending order according to their defect densities.

Based on this diagram, for a given model m and a given top $\pi \times 100\%$ percentage of SLOC, the corresponding CE is defined as:

$$CE_{\pi}(m) = \frac{Area_{\pi}(m) - Area_{\pi}(random\ model)}{Area_{\pi}(optimal\ model) - Area_{\pi}(random\ model)}$$

where $Area_{\pi}(m)$ denotes the area under the curve corresponding to model m at the cutoff of π. In this study, we use the CE at the cut-off $\pi = 0.1$, 0.2 and 0.3 to evaluate the effort-aware ranking performance of a model.

(2) Classification scenario

In the classification scenario, we evaluate the effort-aware classification performance of defect prediction models using the SLOC inspection reduction measure (ER) (Shin et al., 2010). For a given model m, the corresponding ER is defined as:

$$ER(m) = \frac{Effort(random) - Effort(m)}{Effort(random)}$$

where $Effort(m)$ presents the ratio of the total SLOC in the predicted defective modules to the total SLOC in the system. $Effort(random)$ is the ratio of the total SLOC to inspect or test to the total SLOC in the system using a random selection model and achieving the same recall as model m. Similar to F1-measure, we also employ BPP (Menzies et al., 2006), BCE (Schein et al., 2003) and MFM (Rahman et al., 2012) methods to determine the classification threshold. In addition, we also employ "ER-AVG" to evaluate the average effort reduction of a model over all possible thresholds (Zhou et al., 2014).

4. Experimental results

4.1. RQ1: The frequency of cross-module defects

To answer RQ1, we collect the defect data for each studied project using the approach described in Section 3.3. We extracted the modules involved in each defect,establishing mappings of defects to their corresponding modules. Specifically, a module is considered buggy if at least one post-release defect occurs in it, otherwise bug-free. A defect is considered as cross-module defect if it occurs across multiple modules.

Table 3 summarizes the statistics of the collected defect data. The second column lists the total number of bugs present in the studied projects. The third and fourth columns report the number of buggy files and buggy functions respectively. The fifth column presents the number of defects occurring within a single file, followed by the corresponding percentage in the last column. As can be seen from Table 3, the majority of defects in the studied projects occur within a single file, with percentages ranging from 63% to 92%. On average, around 81% of defects are localized within a single file, indicating that defects are more likely to be confined to a specific file in the projects.

Furthermore, to gain insight into the cross-module defects at different granularities, we further study the cross-file defects at the file granularity and the single-file defects at the function granularity. Fig. 9 illustrates the average frequency of defects involving different number of modules. On average, approximately 81% of defects occur within a single file. More specifically, among these single-file defects, about half of them span across multiple functions, with 24.86% involving two functions, 8.88% involving three functions, and the remaining spanning even more functions. On the other hand, cross-file defects are relatively less common across all the studied systems, with an average percentage of 19% . Among these cross-file defects, 52.65% of them propagate to 2 files, while the rest propagate 3 or more files.

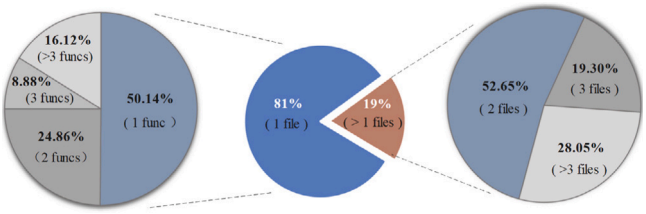

Fig. 9. The distribution of the number of modules involved in a defect.

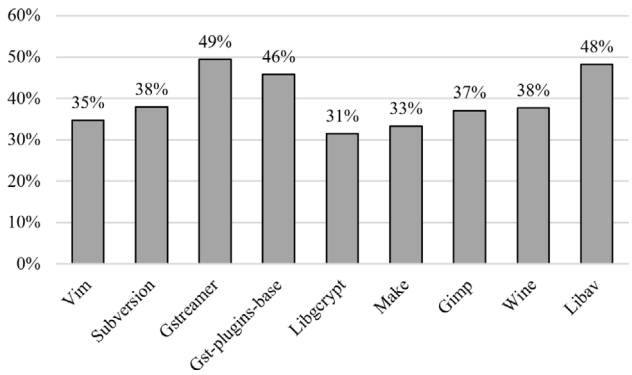

Fig. 10. Proportion of defects within one function.

In addition, the proportion of defects occurring within a single function is explored in detail in Table 4 and Fig. 10. On average, approximately 39.5% of defects are confined to a single function. The percentage varies across all the projects, ranging from 31% to 49%, with a standard deviation of 6.63. As can be seen from the results, in the studied projects, the majority of defects span multiple functions, with a minimum percentage of 51% and an average percentage of 60.5%. To statistically verify this observation, we conduct a significance test using Student's T-test. The null hypothesis H_0 assumes that the probability of a defect occurring in a single module is not less than 50%. The result from Table 4 indicates that the t-test yields a p-value of 0.001, which is lower than the significance level of 0.05. Therefore, we can reject the null hypothesis H_0, and accept the alternative hypothesis that the probability of defects occurring within a single module is less than 50%, which reinforces the significance of the observed result.

Overall, by considering multiple perspectives, we observe that at the file level, the majority of defects occur within a single file. However, at the function level, most defects span across functions, with an average percentage of 60.5%. As a result, due to the high frequency of defects across functions, we are motivated to conduct the first systematic study at the function granularity to explore the implications of intra-defect associations on defect prediction performance.

4.2. RQ2: The relationships between intra-defect modules

To answer RQ2, we further study the relationships between intra-defect modules. In this study, we think there are associations between modules, as they are simultaneously modified to fix a bug. Motivated

Table 3
Statistics of the collected defect data.

Subjects	#bugs	#buggyFiles	#buggyFunctions	#bugs within one file	%bugs within one file
vim_7.0	300	63	866	253	84%
subversion_1.4.0	664	169	1405	553	83%
gstreamer_1.0.1	156	53	267	144	92%
gst-plugins-base_1.10.0	83	63	182	68	82%
libgcrypt_1.4.2	35	35	160	22	63%
make_3.80	108	26	157	68	63%
gimp_1.3.0	666	437	2491	495	74%
wine_1.0	1100	608	3408	910	83%
libav_0.7	640	443	1557	543	85%

Table 4
Descriptive statistics of specific defects.

	Mean	Media	Min.	Max.	Stdev.	Student's t-test	
						t	p-value
Percentage of defects within one module	39.5	37.64	31.43	49.36	6.63	−4.747	0.001
Percentage of defects with expilict dependencies	10.12	11.11	2.22	15.41	4.7	−25.48	6.02E−09

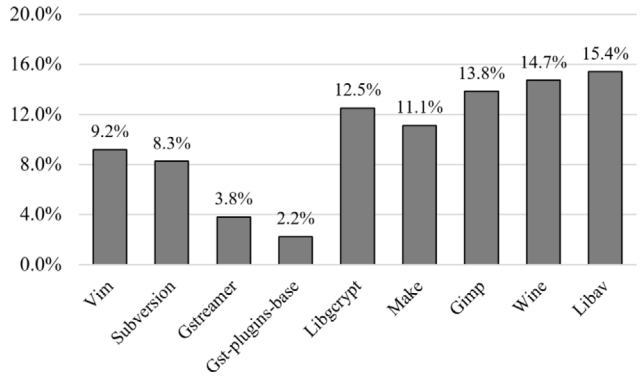

Fig. 11. Percentage of defects with explicit dependencies.

by the results of RQ1, we study the intra-defect relationships at function level.

Among the cross-function defects, we first investigate the proportion of defects with explicit dependencies. The results are reported in Table 4 and Fig. 11. As can be seen, across all the studied systems, the percentages of defects with explicit dependencies are consistently less than 16%, ranging from 2.22% to 15.42%, with an average percentage of 10.1%. This indicates that, the majority of cross-module defects, accounting for approximately 90%, have only implicit dependencies, lacking direct connections between the intra-defect modules. Similar to RQ1, we utilize Student's T-test to verify the observation. The null hypothesis H_0 states that the probability of a defect with explicit dependencies is not less than 50%. However, as shown in Table 4, the p-value is extremely small (i.e. 6.02E−09), well below the significance level of 0.05. As a result, we can reject the null hypothesis H_0, and accept the alternative hypothesis that the probability of a defect with explicit dependencies is less than 50%.

For the minor defects with explicit dependencies, we delve deeper by designing a DD metric, as described in Section 3.4, to examine the degree of direct dependencies. Fig. 12 illustrates the results using a boxplot, which displays the median, the 25th, and the 75th percentiles of the DD values. It can be seen that, the range of DD values mainly falls between 0.006 and 0.083, with the median value of 0.018, which suggests that the degree of explicit intra-defect dependencies is relatively low.

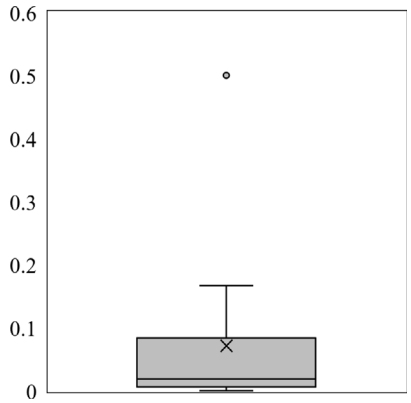

Fig. 12. Distribution of DD values for defects with explicit dependencies.

The results of RQ2 reveal that, at the function level, the majority of cross-module defects exhibit only implicit associations, without direct dependencies between the involved modules. Additionally, for the minor defects with explicit associations, the degree of directed dependencies is also relatively low. The findings offer valuable insights for representing intra-defect associations in RQ3. Inspired with these findings, we choose to represent the intra-defect associations by considering the implicit dependencies.

4.3. RQ3: The value of intra-defect associations on prediction performance

Since the findings of RQ2 indicate that most cross-module defects have only implicit dependencies, we decide to leverage the intra-defect associations by taking advantage of these implicit dependencies. This is achieved by merging the intra-defect modules into new instances through a mean and median process, thereby augmenting the training data. To answer RQ3, we construct three types of defect prediction models, including C model (leveraging the intra-defect associations), T model (using features of each individual module as existing works did), and S model (employing the SMOTE technique for data processing). We first compare the performance of C model against T model to investigate whether the proposed data-processing approach is more effective for defect prediction than the traditional approach based on the features of each single module. Furthermore, we conduct additional comparisons between C model and S model, as well as S model and T model, to investigate whether intra-defect associations significantly contribute to the performance improvement. To evaluate the effectiveness of these models, we use the CE and ER measures respectively to measure the effort-aware ranking and classification performance from cross-validations. The results are presented in Tables 5 and 6.

Table 5 illustrates the effort-aware ranking performance of the three models in terms of CE at the cut-offs π = 0.1, 0.2, and 0.3. Each sub-table with ten columns displays prediction effectiveness, including the average CE value and corresponding standard deviation for each

Table 5
Effort-aware ranking performance.

System	T model	S model	C model	S model *vs* T model		C model *vs* S model		C model *vs* T model	
				%improved	Sig.	%improved	Sig.	%improved	Sig.
(a) CE at π = 0.1									
Vim 7.0	0.068±0.025	0.068±0.037	0.069 ± 0.025	0		1.471		0.442	
Subversion 1.4.0	0.240±0.024	0.265±0.021	0.267 ± 0.025	10.417	***	0.755		11.261	√
Gstreamer 1.0.1	0.126±0.039	0.131±0.041	0.298 ± 0.056	3.968	*	127.481	***	137.284	√
Gst-plugins-base 1.10.0	0.102±0.031	0.169±0.037	0.177 ± 0.047	65.686	***	4.734	**	72.896	√
Libgcrypt 1.4.2	0.026±0.040	0.026±0.035	0.025 ± 0.062	0		-3.846		-3.346	
Make 3.80	0.241±0.091	0.240±0.078	0.247 ± 0.065	-0.415		2.917	*	2.608	
Gimp 1.3.0	0.135±0.014	0.135±0.026	0.133 ± 0.037	0		-1.481		-1.701	
Wine 1.0	0.168±0.012	0.188±0.016	0.191 ± 0.018	11.905	***	1.596	**	13.808	√
Libav 0.7	0.378±0.019	0.372±0.017	0.370 ± 0.020	-1.587	**	-0.538	**	-2.235	×
Average	0.165	0.177	0.197	7.412	~	11.481	~	19.69%	~
Win/tie/los	~	~	~	**4/3/2**	**4/4/1**	**6/0/3**	**4/4/1**	**6/0/3**	**4/4/1**
(b) CE at π = 0.2									
Vim 7.0	0.102±0.034	0.117±0.025	0.118 ± 0.035	14.706	***	0.855	*	15.624	√
Subversion 1.4.0	0.358±0.034	0.364±0.054	0.366 ± 0.026	1.676	***	0.549		2.248	√
Gstreamer 1.0.1	0.210±0.048	0.213±0.037	0.340 ± 0.059	1.429		59.624	***	61.966	√
Gst-plugins-base 1.10.0	0.185±0.041	0.223±0.046	0.240 ± 0.054	20.541	***	7.623	**	29.789	√
Libgcrypt 1.4.2	0.053±0.044	0.080±0.044	0.082 ± 0.058	50.943	***	2.5		55.086	√
Make 3.80	0.247±0.083	0.248±0.056	0.266 ± 0.076	0.405		7.258		7.91	
Gimp 1.3.0	0.156±0.014	0.161±0.021	0.145 ± 0.034	3.205	**	-9.938	***	-7.184	×
Wine 1.0	0.220±0.013	0.300±0.033	0.302 ± 0.021	36.364	**	0.667	*	37.055	√
Libav 0.7	0.587±0.021	0.577±0.021	0.576 ± 0.024	-1.704	***	-0.173		-1.897	×
Average	0.235	0.254	0.271	7.79	~	6.658	~	14.97%	~
Win/tie/los	~	~	~	**8/0/1**	**6/2/1**	**7/0/2**	**4/4/1**	**7/0/2**	**6/1/2**
(c) CE at π = 0.3									
Vim 7.0	0.127±0.037	0.129±0.035	0.160 ± 0.031	1.575	*	24.031	***	25.392	√
Subversion 1.4.0	0.476±0.045	0.476±0.051	0.477 ± 0.030	0		0.21		0.282	
Gstreamer 1.0.1	0.311±0.057	0.341±0.034	0.412 ± 0.065	9.646	**	20.821	***	32.521	√
Gst-plugins-base 1.10.0	0.259±0.048	0.285±0.052	0.287 ± 0.058	10.039	***	0.702		11.082	√
Libgcrypt 1.4.2	0.100±0.049	0.136±0.037	0.149 ± 0.058	36	***	9.559	**	49.119	√
Make 3.80	0.258±0.086	0.351±0.066	0.382 ± 0.067	36.047	***	8.832	**	48.277	√
Gimp 1.3.0	0.194±0.016	0.201±0.014	0.166 ± 0.033	3.608	**	-17.413	***	-14.387	×
Wine 1.0	0.269±0.015	0.338±0.022	0.344 ± 0.023	25.651	***	1.775		27.943	√
Libav 0.7	0.716±0.015	0.705±0.017	0.701 ± 0.026	-1.536	***	-0.567		-2.089	×
Average	0.301	0.329	0.342	9.299	~	3.916	~	13.63%	~
Win/tie/loss	~	~	~	**7/1/1**	**7/1/1**	**7/0/2**	**4/4/1**	**7/0/2**	**6/1/2**

model in the second to fourth columns. The performance comparison is summarized in the fifth to tenth columns, showing the improvement and significance for each group. Specifically, the "%improved" column indicates the improvement (in percentage terms) of the former model in the group over the latter one in terms of the average CE values. For the comparison between C model and T model, the "Sig". column denotes whether C model has significantly better (denoted by √) or worse (denoted by ×) performance than T model in defect prediction at the significance level of 0.05, determined by the Wilcoxon's signed-rank test. For the other two groups, the "Sig". columns illustrates whether the former model exhibits significantly better (with positive improvement) or worse (with negative improvement) performance than the latter model in defect prediction at the significance level of 0.05 (denoted by ***), 0.10 (denoted by **), or 0.20 (denoted by *) using the Wilcoxon's signed-rank test. In particular, the "Average" row lists the average CE of each model, and the average improvement of each group across the studied systems. The last row "Win/tie/loss" in each sub-table describes the number of systems for which the improvement of the former model over the latter model is positive/zero/negative (under the "%improved" column), and the number of systems for which the former model is significantly better/ equal/ worse than the latter model (under the "Sig". column). Similarly, Table 6 illustrate the effort-aware classification performance of these models in terms of four ER indicators.

(1) The performance of C model compared to T model

Ranking performance: In terms of CE at π = 0.1, the results shown in Table 5(a) reveal that the C model performs significantly better than "T" model in four out of the nine data sets, performs similarly in four data sets, and performs worse in one dataset. Regarding "%improved", C model demonstrates positive improvements on six datasets, with an average improvement of 19.69%. Notably, the improvement is over 137% in Gstreamer 1.0.1. In terms of CE at π = 0.2, C model is significantly more effective than T model in six systems, and exhibits substantial improvements over T model in seven systems, with an average improvement of over 14% . Similar findings are evident at the cutoff of π = 0.3, where the C model consistently outperforms the T model in most studied systems, and the average improvement remains positive.

Overall, these results presented in Table 5 clearly show that, in the ranking scenario, C model has significantly better performance than T model on the inspection or testing of defects at the top cutoffs of the fault-proneness ranking list. These results indicate that augmenting new instances with intra-defect associations can result in substantial improvements of the fault-proneness prediction effectiveness in the ranking scenario under cross-validation evaluation.

Classification performance: Table 6 summarizes the effort-aware classification performance of C model and T model. As shown in Table 6(a), the performance of C model is significantly better than T model in five out of the nine systems, with an average improvement of 3.98% in terms of ER-BPP. Similarly, in Table 6(b), C model performs better in five datasets, similarly in two datasets, and worse in two datasets. The improvements are positive in six datasets, with an average improvement of 7.58%. Notably, C model achieves a considerable improvement of 82.796% in terms of ER-BCE in Libgcrypt 1.4.2. In addition, for ER-MFM, C model exhibits significantly better performance in seven datasets, and similar performance in one datasets. The improvements range from 15% to 85% for most systems. On average, C model obtains 49.1% effort reduction in code inspection or testing,compared to 37.1% effort reduction achieved by T model, resulting in a considerable average improvement of 32.21%. Furthermore, for ER-AVG, it is obvious

Table 6
Effort-aware classification performance.

System	T model	S model	C model	S model *vs* T model		C model *vs* S model		C model *vs* T model	
				%improved	Sig.	%improved	Sig.	%improved	Sig.
(a) ER-BPP									
Vim 7.0	0.324 ± 0.08	0.298 ± 0.140	0.274 ± 0.162	-8.025	**	-8.054	**	-15.617	×
Subversion 1.4.0	0.715 ± 0.029	0.726 ± 0.047	0.730 ± 0.041	1.538	***	0.551	*	2.026	√
Gstreamer 1.0.1	0.608 ± 0.055	0.728 ± 0.075	0.731 ± 0.052	19.737	***	0.412		20.175	√
Gst-plugins-base 1.10.0	0.586 ± 0.064	0.591 ± 0.034	0.653 ± 0.076	0.853		10.491	***	11.563	√
Libgcrypt 1.4.2	0.393 ± 0.071	0.425 ± 0.073	0.531 ± 0.164	8.142	**	24.941	***	35.199	√
Make 3.80	0.44 ± 0.146	0.410 ± 0.098	0.403 ± 0.089	-6.818	***	-1.707	*	-8.521	×
Gimp 1.3.0	0.439 ± 0.025	0.431 ± 0.124	0.432 ± 0.041	-1.822		0.232		-1.589	
Wine 1.0	0.6 ± 0.024	0.623 ± 0.052	0.625 ± 0.026	3.833	***	0.321	*	4.215	√
Libav 0.7	0.834 ± 0.009	0.761 ± 0.009	0.758 ± 0.015	-8.753	***	-0.394		-9.191	×
Average	0.549	0.555	0.571	1.093	~	2.884	~	3.98%	~
Win/tie/loss	~	~	~	**5/0/4**	**4/2/3**	**6/0/3**	**4/3/2**	**5/0/4**	**5/1/3**
(b) ER-BCE									
Vim 7.0	0.323 ± 0.085	0.319 ± 0.063	0.280 ± 0.242	-1.238		-12.226		-13.107	
Subversion 1.4.0	0.720 ± 0.028	0.741 ± 0.026	0.750 ± 0.039	2.917	***	1.215	*	4.068	√
Gstreamer 1.0.1	0.608 ± 0.052	0.718 ± 0.057	0.726 ± 0.049	18.092	***	1.114	*	19.249	√
Gst-plugins-base 1.10.0	0.585 ± 0.061	0.629 ± 0.064	0.630 ± 0.059	7.521	***	0.159		7.689	√
Libgcrypt 1.4.2	0.286 ± 0.151	0.321 ± 0.091	0.523 ± 0.163	12.238	**	62.928	***	82.796	√
Make 3.80	0.416 ± 0.11	0.423 ± 0.117	0.441 ± 0.092	1.683		4.255		6.111	
Gimp 1.3.0	0.448 ± 0.029	0.418 ± 0.025	0.411 ± 0.036	-6.696	***	-1.675		-8.128	×
Wine 1.0	0.576 ± 0.019	0.621 ± 0.083	0.64 ± 0.026	7.813	***	3.06	*	11.129	√
Libav 0.7	0.852 ± 0.008	0.856 ± 0.062	0.778 ± 0.014	0.469		-9.112	***	-8.718	×
Average	0.535	0.561	0.575	4.819	~	2.636	~	7.58%	~
Win/tie/loss	~	~	~	**7/0/2**	**5/3/1**	**6/0/3**	**4/4/1**	**6/0/3**	**5/2/2**
(c) ER-MFM									
Vim 7.0	0.019 ± 0.018	0.020 ± 0.058	0.022 ± 0.172	5.263		10		17.104	
Subversion 1.4.0	0.621 ± 0.038	0.684 ± 0.072	0.716 ± 0.038	10.145	***	4.678	*	15.363	√
Gstreamer 1.0.1	0.562 ± 0.065	0.592 ± 0.071	0.809 ± 0.063	5.338	**	36.655	***	43.907	√
Gst-plugins-base 1.10.0	0.541 ± 0.197	0.631 ± 0.158	0.812 ± 0.232	16.636	***	28.685	***	50.225	√
Libgcrypt 1.4.2	0.191 ± 0.073	0.221 ± 0.023	0.352 ± 0.561	15.707	**	59.276	***	84.509	√
Make 3.80	0.125 ± 0.107	0.179 ± 0.067	0.172 ± 0.089	43.2	***	-3.911		37.323	√
Gimp 1.3.0	0.200 ± 0.025	0.301 ± 0.026	0.309 ± 0.071	50.5	***	2.658	*	54.768	√
Wine 1.0	0.298 ± 0.022	0.469 ± 0.108	0.488 ± 0.027	57.383	***	4.051		63.738	√
Libav 0.7	0.785 ± 0.009	0.732 ± 0.041	0.738 ± 0.016	-6.752	***	0.82		-6.096	×
Average	0.371	0.425	0.491	14.572	~	15.383	~	32.21%	~
Win/tie/loss	~	~	~	**8/0/1**	**7/1/1**	**8/0/1**	**5/4/0**	**8/0/1**	**7/1/1**
(d) ER-AVG									
Vim 7.0	0.327 ± 0.074	0.331 ± 0.062	0.266 ± 0.058	1.223	*	-19.637	***	-18.541	×
Subversion 1.4.0	0.668 ± 0.038	0.718 ± 0.076	0.723 ± 0.038	7.485	***	0.696		8.214	√
Gstreamer 1.0.1	0.602 ± 0.057	0.713 ± 0.091	0.720 ± 0.043	18.439	***	0.982	**	19.649	√
Gst-plugins-base 1.10.0	0.538 ± 0.061	0.542 ± 0.045	0.663 ± 0.045	0.743		22.325	***	23.335	√
Libgcrypt 1.4.2	0.202 ± 0.186	0.295 ± 0.136	0.438 ± 0.244	46.04	***	48.475	***	117.418	√
Make 3.80	0.372 ± 0.114	0.402 ± 0.074	0.401 ± 0.050	8.065	***	-0.249		7.914	√
Gimp 1.3.0	0.489 ± 0.023	0.457 ± 0.122	0.431 ± 0.044	-6.544	**	-5.689	**	-11.872	×
Wine 1.0	0.589 ± 0.015	0.661 ± 0.032	0.683 ± 0.024	12.224	***	3.328	*	15.968	√
Libav 0.7	0.796 ± 0.01	0.730 ± 0.050	0.732 ± 0.014	-8.291	***	0.274		-8.072	×
Average	0.509	0.539	0.562	5.804	~	4.29	~	10.39%	~
Win/tie/loss	~	~	~	**7/0/2**	**6/1/2**	**6/0/3**	**4/3/2**	**6/0/3**	**6/0/3**

that C model is more effective in most datasets, resulting in 56.2% effort reduction in code inspection or testing, while *T* model achieves 50.9% effort reduction. The average improvement of C model over *T* model is more than 10%.

As evident from the results, C model exhibits better performance than *T* model when classifying fault-prone functions. This finding suggests that augmenting new instances with intra-defect associations is more effective than the traditional approach of using the instances with the features of each single module in the fault-proneness classification scenario.

From the above results, it can be observed that the C model demonstrates superior predictive performance compared to the *T* model. This performance difference may be attributed to the strategic incorporation of intra-defect associations in the C model, which leads to variations in the selected features. By leveraging intra-defect associations during data processing, C model could capture additional patterns and implicit dependencies among intra-defect modules, which in turn influence the selection of relevant features for defect prediction. Although both the C model and the *T* model utilize the same set of features for modeling, the selected features differ between them. This discrepancy in feature selection may have contributed to the difference in their predictive performance. As a result, the differences in feature selection between the C model and the *T* model may play a significant role in their distinct predictive performances.

Particularly, in terms of "C model vs *T* model" in Tables 5 and 6, it can be seen that the performance of C model in Libav 0.7 and Gimp 1.3.0 is not significantly better than *T* model. One possible reason for this may be the higher proportion of defects with explicit dependencies in Libav 0.7, as shown in Fig. 11. Similarly, Gimp 1.3.0 exhibits a relatively high proportion of such defects, along with higher DD values compared to other studied systems. These findings indicate that the relationships between intra-defect modules in these projects are more complex than others. As a result, the data processing approach utilized in C model, which focuses on the implicit dependencies, may not fully capture the direct dependencies between modules, leading to an impact on the prediction performance. Additionally, *T* model has achieved relatively good performance in Libav 0.7, with high evaluation indicators, making further significant improvements challenging.

(2) The performance of C model compared to S model

Furthermore, we build S model which leverages the most commonly used over-sampling technique SMOTE to augment the positive

instances. By comparing the performance of C model against S model, we investigate whether intra-defect associations significantly contribute to the performance improvement of C model. The seventh and eighth columns in Tables 5 and 6 presents the comparison results between the C model and S model.

In terms of the ranking performance, we can observe from Table 5 that, C model outperforms S model in all CE indicators. Specifically,for each indicator, the C model performs significantly better than the S model in 4 datasets, similarly in 4 datasets, and worse in the remaining 1 dataset at the significance level of at least 0.2. The performance improvements are consistently positive in at least 6 of data sets.

About the classification performance, Table 6 indicates that C model is significantly better than S model in at least four systems under all studied ER indicators, with positive improvements in at least 6 datasets. The results for ER-AVG clearly show that the C model outperforms the S model in most datasets, leading to a significant reduction of 56.2% in effort required for code inspection or testing, compared to 53.9% achieved by the S model. On average, the C model exhibits a 4.29% improvement over the S model.

The comparison between the C and S models demonstrates that considering intra-association during data augmentation enhances the defect prediction performance. This result confirms the significance of incorporating intra-association for defect prediction, highlighting its positive influence on predictive accuracy.

(3) The performance of S model compared to T mode The fifth and sixth columns in Tables 5 and 6 presents the comparison results between the S model and *T* model. Based on the experimental results, it is obvious that the S model exhibits superior predictive performance compared to the *T* model in both ranking and classification scenarios across the majority of systems, demonstrating a positive improvement. This consistent finding aligns with previous research conclusions, reinforcing the notion that data augmentation plays a crucial role in enhancing the predictive performance (Chen et al., 2020; Yu et al., 2023). The incorporation of the most commonly used oversampling technique, SMOTE, in the S model proves to be advantageous in effectively augmenting positive instances and contributing to the performance improvement. When combined with the results of "C model vs S model", it becomes apparent that the improvement achieved by incorporating intra-defect associations compared to most commonly data augmentation technique is significant.

In summary, the above results for RQ3 indicate that considering intra-defect associations is valuable for improving the performance of defect prediction models in both ranking and classification scenarios.

5. Validity

In this section, we discuss the most important threats to the construct, internal, and external validity of our study.

Construct validity Since the quality of the datasets influences the performance of defect prediction models, ensuring the accuracy of both the dependent and independent variables is paramount for maintaining the construct validity of our study. In terms of the dependent variable, we primarily use SZZ algorithm to obtain the defect data . The SZZ algorithm is the most commonly used approach for mining defect data from open-source software, and have been extensively utilized in previous studies (Zeng et al., 2021). While some limitations of SZZ have been identified, such as the potential introduction of mislabeling data (Herbold et al., 2019; Rodríguez-Pérez et al., 2018), these imperfections do not significantly impact the performance of defect prediction, thus validating the effectiveness of SZZ (Fan et al., 2019). Moreover, to mitigate these potential limitations and further enhance the accuracy of the dependent variable, we implemented additional processing steps as introduced in Section 3.3. Specifically, we excluded changes related to coding style, comments, and blank lines, as well as filtered out large patches. Furthermore, to verify the reliability of the dependent variables, we conducted a manual examination of a small sample of modules, and found that the dependent variables are reliable. On the other hand, regarding the independent variables, we utilized the "Understand" tool, a well-established commercial source code analysis tool, to collect metrics. We use its perl API to create data collection scripts. To ensure the reliability of the data collection process, the perl scripts have been checked doubly by the authors to ensure the reliability. By taking these measures to address potential inaccuracies in both the dependent and independent variables, we attempt to improve data accuracy and maintain the construct validity of our study.

Internal validity The most important threat to the internal validity of our study mainly lies in how to leverage the intra-defect associations when building prediction models, and how to design experiments to obtain valuable conclusions. With the hypothesis verification, we find that, for most cross-module defects, there are not direct dependencies between the involved modules. Inspired by this finding, in order to leverage the intra-defect associations, we attempt to take advantage of the implicate dependencies by merging the intra-defect modules into new instances. Since the independent variables in this study consist of the most commonly used traditional metrics, most of which are counts, the use of mean or median can prevent the merged value from changing too much, causing huge variance and generating outliers, so as to ensure the accuracy of the model. Therefore, in this study, we use the mean and median methods to merge the intra-defect modules. The independent variables of each generated instance are derived from the mean/median values of its involved modules. In order to evaluate the effectiveness of the above processing approach, in addition to the traditional *T* model, we also build a S model based on SMOTE method to verify the value of intra-defect associations in improving defect prediction performance, and thus mitigate the bias caused by data augmentation. Although a simple method is adopted to leverage the intra-defect associations, valuable findings are discovered. This study is a preliminary attempt to investigate the value of intra-defect associations on the performance of fault-proneness prediction. We do think that there should be other potential and possibly more effective ways to consider intra-defect associations, such as summing or other complex data processing methods. We will do more attempts in the future.

Regarding the choice of modeling technique, we carefully selected the logistic regression as our modeling technique, as it is a well-established and widely used approach known for its good performance in fault-proneness prediction (Zeng et al., 2021; Hall et al., 2011; Lessmann et al., 2008). To ensure the robustness of our findings, we also conducted experiments using an alternative technique, namely random forests. Interestingly, even with this different modeling approach, the consideration of intra-defect associations consistently led to improved performance, without affecting our conclusions. This finding further reinforces the validity of our study.

External validity Our experiments are conducted based on nine widely used open-source systems written in C language. One possible threat to the external validity is that our findings may not be generalized to other kinds of systems, such as object-oriented systems or systems implemented with dynamic programming languages. Nevertheless, it is important to acknowledge that this issue is inherent to all empirical studies, rather than specific to our research. The mitigation of such threats can only be achieved through the replication of our study across a diverse range of systems in future research. The second threat is that our findings of RQ3 are limited to the function granularity. Since the frequency of cross-file defects is not high in the studied systems, we investigate the value of intra-defect associations on defect prediction at the level of function. It is possible that intra-defect associations at different granularities may have different effects on defect prediction performance. To mitigate these threats, we will replicate our study using a wide variety of systems and from other granularities in the future work.

6. Related works

In this section, we briefly summarize the related work from the following aspects:

Data manipulation The quality of data sets is crucial for defect prediction performance. In most defect prediction studies, the dependent variable in the data sets can take on the value of 1 or 0, indicating defective or not. The dependent variable is usually set to 1 if at least one post-release defect is detected in the module, and otherwise 0. For example, (Herzig, 2014) investigated the usefulness of test execution metrics as software quality indicators for building defect prediction models. In their study, a module is considered defective if at least one post-release code fix edited it. Ahluwalia et al. (2019) collected defect data in the similar way, they labeled a class as defective if it has at least one post-release defect. Zhao et al. (2015, 2017) studied the value of modularization metrics and context-based cohesion metrics for package-level defect prediction, and considered a package defective if at least one fault is discovered in it. Pornprasit and Tantithamthavorn (2022) considered the lines defective if they are modified for fixing a post-release defect, otherwise non-defective. For the above studies, their models were built based on the features of each single module consistently.

Defect data is important for building defect prediction models. Recently, the defect data is collected mainly based on the bug fixing statements or bug issue links in the commit logs. As a result, there may be noises in the defect data collected in this way, due to non-standard descriptions, mislabeled issues, or non-code changes, etc. Rahman et al. (2013), Wu et al. (2011), Le et al. (2015), Xie et al. (2019). These noisy defect data will affect the reliability of defect prediction models. Consequently, researchers have raised concerns about the quality of the data. Tantithamthavorn et al. (2015) performed a case study of 3931 manually-curated issue reports, and studied the impact of issue report mislabeling on the performance and interpretation of defect prediction models, and found that mislabeling rarely impacts the prediction reliability, suggesting that the effectiveness of the models trained on noisy data can be trusted. Herzig et al. (2016) analyzed the impact of tangled changes on defect prediction, and found that many bug fixes consist of multiple tangled changes, and suggested that untangling tangled code changes can lead to more accurate defect prediction models. Ahluwalia et al. (2019) studied the nature of sleeping defects and snoring classes, and analyzed the severity of the sleeping defects and of the snoring classes in defect datasets for defect prediction. Fan et al. (2019) investigated the performance of JIT defect prediction models trained on different mislabeled data sets, which were collected using four SZZ variants. They demonstrated the effectiveness and accuracy of SZZ algorithm.

In addition, software defect data has the class imbalance issue, which makes defect prediction more challenging. To handle the class imbalance issues, (Ryu et al., 2016) proposed a method called the value-cognitive boosting with support vector(VCB-SVM), and compared the defect prediction effectiveness of this method against the existing cross-project defect prediction(CPDP) techniques not prepared for the class imbalance and the class imbalance techniques not prepared for the cross-project learning. Tan et al. (2015) employed the re-sampling techniques (e.g. simple duplicate, SMOTE,etc.) to deal with the class imbalance problem by adding positive instances, and found they were effective for improving change classification performance. Feng et al. (2021) proposed a complexity-based oversampling technique (COSTE) to address the class imbalance problem, which adopts the similar complexity as a basis to select instances to generate synthetic instances. Besides, SMOTE is considered to be one of the most influential algorithms for data preprocessing, which has been widely used in many areas, including defect prediction (García et al., 2016).

Defect prediction In terms of the granularity levels of defect prediction models, prior studies proposed defect prediction approaches at various granularity. For example, (Zhao et al., 2017) studied the effectiveness of considering client usage context in package cohesion metrics for defect prediction at the package level. Jiarpakdee et al. (2020) investigated the effects of 12 widely used feature selection techniques on the performance as well as interpretation of defect models at file level. At class level, Okutan and Yıldız (2014) leveraged Bayesian networks to study the most effective metrics for defect prediction and the probabilistic influential relationships among metrics and defectiveness. Yang et al. (2014) performed a validation of the value of slice-based cohesion metrics on the performance of defect-proneness prediction at the function level. In terms of commit level, CC2Vec, a neural network model, was recently proposed for predicting defect-introducing commits. Wong et al. (2000) reported a study to predict fault-proneness at the design level. For fine-grained predictions, Pornprasit and Tantithamthavorn (2021) subsequently proposed a line-level Just-In-Time defect prediction approach (JITLine) to predict defect-inducing commits and defective lines. They found JITLine is faster than CC2Vec and more accurate than the baseline NLP approach. In addition, they further proposed a deep learning approach (DeepLineDP) for line-level defect prediction, which can automatically capture the surrounding tokens and lean the hierarchical structure of source code to identify defective files and lines (Pornprasit and Tantithamthavorn, 2022).

7. Conclusion

In this paper, we perform a comprehensive investigation on the implications of intra-defect associations for fault-proneness prediction. Based on nine open-source C systems, we organize our experiments with 3 research questions. We first investigate the proportion of defects that occur across modules at the file and function granularity respectively. The results indicate that, at the file granularity, the majority of defects (about 81%) occur within a single file. However, at the function granularity, a considerable number of defects (averaging 60.5%) span across functions. Then we further examine the relationships between the involved modules of cross-module defect. The results show that, for most cross-module defects(approximately 90%), there are not direct dependencies between the involved modules. Even for the minor defects with explicit dependencies, the degree of directed dependencies is also relatively low, with a median DD value of 0.018. Based the above key findings, we propose a data processing method that leverages the intra-defect associations by merging the intra-defect modules into new instances with mean or median variables. To gain deep insights into intra-defect associations, We build multivariate logistic regression models to analyze the value of intra-defect associations for fault-proneness prediction. From the results, we find that considering the intra-defect associations is substantially more effective than the traditional data processing approaches when predicting fault proneness in both the ranking scenario and the classification scenario. We believe our results are significant for improving the understanding of the value of intra-defect associations, and suggest a way for developing better fault-proneness prediction models.

In the future work, we will try other potential ways to represent the intra-defect associations, and study the impact of different representations on defect prediction performance. Besides, we will replicate our experiments on a variety of systems, especially the object-oriented software systems.

CRediT authorship contribution statement

Yangyang Zhao: Conceptualization, Methodology, Investigation, Writing – original draft. **Mingyue Jiang:** Validation, Writing – review & editing. **Yibiao Yang:** Methodology, Validation. **Yuming Zhou:** Methodology, Review & editing, Supervision. **Hanjie Ma:** Writing – review & editing. **Zuohua Ding:** Writing – review & editing, Supervision.

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.

Data availability

Data will be made available on request.

Acknowledgments

This work was supported by the National Nature Science Foundation of China (Grant No. 62132014, 62172205 and 62072194), Zhejiang Provincial Key Research and Development Program of China (No. 2022C01045), and Zhejiang Provincial Natural Science Foundation of China (No. LQ23F020020).

References

Ahluwalia, A., Falessi, D., Di Penta, M., 2019. Snoring: A noise in defect prediction datasets. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories. MSR, IEEE, pp. 63–67.

Arisholm, E., Briand, L.C., Johannessen, E.B., 2010. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. J. Syst. Softw. 83 (1), 2–17.

Bachmann, A., Bird, C., Rahman, F., Devanbu, P., Bernstein, A., 2010. The missing links: bugs and bug-fix commits. In: FSE. pp. 97–106.

Boehm, B., Basili, V.R., 2001. Defect reduction top 10 list. Computer 34 (1), 135–137.

Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P., 2002. SMOTE: synthetic minority over-sampling technique. J. Artificial Intelligence Res. 16, 321–357.

Chen, J., Hu, K., Yu, Y., Chen, Z., Xuan, Q., Liu, Y., Filkov, V., 2020. Software visualization and deep transfer learning for effective software defect prediction. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 578–589.

Da Costa, D.A., McIntosh, S., Shang, W., Kulesza, U., Coelho, R., Hassan, A.E., 2016. A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. IEEE Trans. Softw. Eng. 43 (7), 641–657.

D'Ambros, M., Lanza, M., Robbes, R., 2012. Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empir. Softw. Eng. 17 (4), 531–577.

Fan, Y., Xia, X., Da Costa, D.A., Lo, D., Hassan, A.E., Li, S., 2019. The impact of mislabeled changes by SZZ on just-in-time defect prediction. IEEE Trans. Softw. Eng. 47 (8), 1559–1586.

Feng, S., Keung, J., Yu, X., Xiao, Y., Bennin, K.E., Kabir, M.A., Zhang, M., 2021. COSTE: Complexity-based OverSampling TEchnique to alleviate the class imbalance problem in software defect prediction. Inf. Softw. Technol. 129, 106432.

Fernández, A., Garcia, S., Herrera, F., Chawla, N.V., 2018. SMOTE for learning from imbalanced data: progress and challenges, marking the 15-year anniversary. J. Artificial Intelligence Res. 61, 863–905.

García, S., Luengo, J., Herrera, F., 2016. Tutorial on practical tips of the most influential data preprocessing algorithms in data mining. Knowl.-Based Syst. 98, 1–29.

Gesi, J., Li, J., Ahmed, I., 2021. An empirical examination of the impact of bias on just-in-time defect prediction. In: Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM, pp. 1–12.

Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S., 2011. A systematic literature review on fault prediction performance in software engineering. IEEE Trans. Softw. Eng. 38 (6), 1276–1304.

Herbold, S., Trautsch, A., Trautsch, F., Ledel, B., 2019. Issues with SZZ: An empirical assessment of the state of practice of defect prediction data collection. arXiv preprint arXiv:1911.08938.

Herzig, K., 2014. Using pre-release test failures to build early post-release defect prediction models. In: 2014 IEEE 25th International Symposium on Software Reliability Engineering. IEEE, pp. 300–311.

Herzig, K., Just, S., Zeller, A., 2016. The impact of tangled code changes on defect prediction models. Empir. Softw. Eng. 21 (2), 303–336.

Hoang, T., Dam, H.K., Kamei, Y., Lo, D., Ubayashi, N., 2019. Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories. MSR, IEEE, pp. 34–45.

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

Kim, S., Whitehead, Jr., E.J., 2006. How long did it take to fix bugs? In: Proceedings of the 2006 International Workshop on Mining Software Repositories. pp. 173–174.

Kim, S., Zimmermann, T., Pan, K., James, Jr., E., et al., 2006. Automatic identification of bug-introducing changes. In: 21st IEEE/ACM International Conference on Automated Software Engineering. ASE'06, IEEE, pp. 81–90.

LaToza, T.D., Venolia, G., DeLine, R., 2006. Maintaining mental models: a study of developer work habits. In: Proceedings of the 28th International Conference on Software Engineering. pp. 492–501.

Le, T.D.B., Linares-Vásquez, M., Lo, D., Poshyvanyk, D., 2015. Rclinker: Automated linking of issue reports and commits leveraging rich contextual information. In: 2015 IEEE 23rd International Conference on Program Comprehension. IEEE, pp. 36–47.

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

McIntosh, S., Kamei, Y., 2017. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. IEEE Trans. Softw. Eng. 44 (5), 412–428.

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

Okutan, A., Yıldız, O.T., 2014. Software defect prediction using Bayesian networks. Empir. Softw. Eng. 19 (1), 154–181.

Pornprasit, C., Tantithamthavorn, C.K., 2021. Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories. MSR, IEEE, pp. 369–379.

Pornprasit, C., Tantithamthavorn, C., 2022. DeepLineDP: Towards a deep learning approach for line-level defect prediction. IEEE Trans. Softw. Eng..

Qu, Y., Chi, J., Yin, H., 2021. Leveraging developer information for efficient effort-aware bug prediction. Inf. Softw. Technol. 137, 106605.

Rahman, F., Devanbu, P., 2013. How, and why, process metrics are better. In: 2013 35th International Conference on Software Engineering. ICSE, IEEE, pp. 432–441.

Rahman, F., Posnett, D., Devanbu, P., 2012. Recalling the" imprecision" of cross-project defect prediction. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. pp. 1–11.

Rahman, F., Posnett, D., Herraiz, I., Devanbu, P., 2013. Sample size vs. bias in defect prediction. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. pp. 147–157.

Rodríguez-Pérez, G., Robles, G., González-Barahona, J.M., 2018. Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the SZZ algorithm. Inf. Softw. Technol. 99, 164–176.

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

Schein, A.I., Saul, L.K., Ungar, L.H., 2003. A generalized linear model for principal component analysis of binary data. In: International Workshop on Artificial Intelligence and Statistics. PMLR, pp. 240–247.

Shin, Y., Meneely, A., Williams, L., Osborne, J.A., 2010. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Trans. Softw. Eng. 37 (6), 772–787.

Śliwerski, J., Zimmermann, T., Zeller, A., 2005. When do changes induce fixes? ACM SIGSOFT Softw. Eng. Notes 30 (4), 1–5.

Song, Q., Jia, Z., Shepperd, M., Ying, S., Liu, J., 2010. A general software defect-proneness prediction framework. IEEE Trans. Softw. Eng. 37 (3), 356–370.

Tan, M., Tan, L., Dara, S., Mayeux, C., 2015. Online defect prediction for imbalanced data. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 2. IEEE, pp. 99–108.

Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Ihara, A., Matsumoto, K., 2015. The impact of mislabelling on the performance and interpretation of defect prediction models. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 1. IEEE, pp. 812–823.

Tripathy, P., Naik, K., 2011. Software Testing and Quality Assurance: Theory and Practice. John Wiley & Sons.

Vernazza, T., Granatella, G., Succi, G., Benedicenti, L., Mintchev, M., 2000. Defining metrics for software components. In: Proceedings of the World Multiconference on Systemics, Cybernetics and Informatics, Vol. 1. pp. 16–23.

Walkinshaw, N., Minku, L., 2018. Are 20% of files responsible for 80% of defects? In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 1–10.

Wong, W.E., Debroy, V., Surampudi, A., Kim, H., Siok, M.F., 2010. Recent catastrophic accidents: Investigating how software was responsible. In: 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement. IEEE, pp. 14–22.

Wong, W.E., Horgan, J.R., Syring, M., Zage, W., Zage, D., 2000. Applying design metrics to predict fault-proneness: a case study on a large-scale software system. Softw. -Pract. Exp. 30 (14), 1587–1608.

Wong, W.E., Li, X., Laplante, P.A., 2017. Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures. J. Syst. Softw. 133, 68–94.

Wu, R., Zhang, H., Kim, S., Cheung, S.-C., 2011. ReLink: Recovering links between bugs and changes. In: ESEC/FSE '11. pp. 15–25.

Xie, R., Chen, L., Ye, W., Li, Z., Hu, T., Du, D., Zhang, S., 2019. DeepLink: A code knowledge graph based deep learning approach for issue-commit link recovery. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 434–444.

Xu, Z., Zhao, K., Zhang, T., Fu, C., Yan, M., Xie, Z., Zhang, X., Catolino, G., 2021. Effort-aware just-in-time bug prediction for mobile apps via cross-triplet deep feature embedding. IEEE Trans. Reliab. 71 (1), 204–220.

Yang, Y., Zhou, Y., Lu, H., Chen, L., Chen, Z., Xu, B., Leung, H., Zhang, Z., 2014. Are slice-based cohesion metrics actually useful in effort-aware post-release fault-proneness prediction? An empirical study. IEEE Trans. Softw. Eng. 41 (4), 331–357.

Yu, H., Sun, H., Tao, J., Qin, C., Xiao, D., Jin, Y., Liu, C., 2023. A multi-stage data augmentation and AD-ResNet-based method for EPB utilization factor prediction. Autom. Constr. 147, 104734.

Zeng, Z., Zhang, Y., Zhang, H., Zhang, L., 2021. Deep just-in-time defect prediction: how far are we? In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 427–438.

Zhao, Y., Yang, Y., Lu, H., Liu, J., Leung, H., Wu, Y., Zhou, Y., Xu, B., 2017. Understanding the value of considering client usage context in package cohesion for fault-proneness prediction. Autom. Softw. Eng. 24 (2), 393–453.

Zhao, Y., Yang, Y., Lu, H., Zhou, Y., Song, Q., Xu, B., 2015. An empirical analysis of package-modularization metrics: Implications for software fault-proneness. Inf. Softw. Technol. 57, 186–203.

Zhou, Y., Xu, B., Leung, H., Chen, L., 2014. An in-depth study of the potentially confounding effect of class size in fault prediction. ACM Trans. Softw. Eng. Methodol. (TOSEM) 23 (1), 1–51.

Zhou, Y., Yang, Y., Lu, H., Chen, L., Li, Y., Zhao, Y., Qian, J., Xu, B., 2018. How far we have progressed in the journey? an examination of cross-project defect prediction. ACM Trans. Softw. Eng. Methodol. (TOSEM) 27 (1), 1–51.

Zimmermann, T., Premraj, R., Zeller, A., 2007. Predicting defects for eclipse. In: Third International Workshop on Predictor Models in Software Engineering. PROMISE'07: ICSE Workshops 2007, IEEE, p. 9.

Yangyang Zhao received the PhD. degree from Nanjing University, China, in 2018. She is currently a lecturer at Zhejiang Sci-Tech University. Her current research interests include defect prediction, intelligent software engineering, software refactoring and empirical software engineering.

Mingyue Jiang received the B.Sc. degree in computer science and technology from Yunnan Normal University, and the PhD degree from Swinburne University of Technology, Australia. She is an associate professor of software engineering at the Zhejiang Sci-Tech University, China. Her current research interests include software testing and program repair.

Yibiao Yang received the Ph.D. degree from Nanjing University in Sept. 2016 and the B.Sc. degree from Southwest Jiaotong University in June 2010. He is currently an Associate Research Professor in Department of Computer Science and Technology at Nanjing University. His main research interest is automated technologies for software and systems: testing and analysis.

Yuming Zhou is a professor with the Department of Computer Science and Technology at Nanjing University. His main research interests are empirical software engineering and defect prediction.

Hanjie Ma received the PhD. degree from Zhejiang University. He is currently an associate professor at Zhejiang Sci-Tech University. His main research interests are Data Analysis and Mining, Embedded Software Systems.

Zuohua Ding received the Ph.D. degree from University of South Florida. He is currently a professor at Zhejiang Sci-Tech University. His main research interests include Requirements Modeling and Analysis, Software Testing and Reliability Evaluation, and Intelligent Software System.