Supplementary material for:

Inconsistent defect labels: essence, causes, and influence

Shiran Liu, Zhaoqiang Guo, Yanhui Li, Chuanqi Wang,
Lin Chen, Zhongbin Sun, Yuming Zhou, and Baowen Xu

Appendix A. Causes of Inconsistent labels

A.1. Actual causes of inconsistent labels for motivational examples

In this section, we analyze the actual causes of inconsistent labels in the motivational examples presented in Section 3. For each of the motivational examples, we conducted a manual analysis to examine the actual cause. The reverse analysis of the causes of inconsistent labels in motivation examples can be helpful in identifying deficiencies in existing defect collection approaches (or even problems that have not been noted in literatures, such as rollback changes and branch-merge conflicts), and thereby identifying opportunities for improvement in defect collection approaches.

Example 1 (a semi-automatic SZZ-based approach [1]): the "XPathParser.java" instances in the Tika project. The labels of the "XPathParser.java" instances were collected from the issue reports in JIRA and the commits in GIT. As shown in Fig. 5, the corresponding instances were marked as "buggy" on versions 1.0~1.9 but marked as "clean" on versions 1.10~1.17. By a Git query command[1], we find that there are four commits involved in the "XPathParser.java" instances: C_1 (7adc256, 2009-04-28), C_2 (5220653, 2011-10-10), C_3 (398d0b1, 2015-06-29), and C_4 (ff6d3fc, 2015-06-29). Fig. 1 shows their relationships with the corresponding versions. As can be seen, C_3 is a BFC, while C_1 is identified as BIC. This is the reason why the "XPathParser.java" instance is marked as "buggy" on versions 1.0~1.9 but as "clean" on versions 1.10~1.17. However, the labels on versions 1.10~1.17 are indeed mislabels. By examining the commit log, we find that C_4 is a rollback change for recovering the code modified in C_3, as it appears that C_3 is an incomplete fix. As a result, the "XPathParser.java" instances on versions 1.10~1.17 have the same code as on versions 1.0~1.9. In other words, they have the same buggy code. As a result, for example 1, the actual cause is that a rollback change leads to the mislabels on version versions 1.10~1.17.

Example 2 (the time-window approach [2, 3]): the "SourceTree.java" instances in the Xalan project. The labels of the "SourceTree.java" instances were collected based on the BFCs identified by the BugInfo[2] tool. Since the download link of the BugInfo tool provided in [3] had been closed, we cannot obtain the BFC information to further analyze the actual cause. However, Jureczko et al. [2, 3] pointed out the limitations they faced when linking defective modules to versions: "*The defects are assigned to versions according to the bugfix date. It could be probably better to assign a defect to the version, where the defect has been found, but unfortunately, the source code version control system does not contain such information.*". In other words, they believe that linking defective modules to versions based on BIC and BFC is more accurate. In this sense, the Metrics-Repo-2010 data set may contain many mislabels. In addition, according to the official website of the Xalan project[3], the release date of the four versions Xalan-2.4, 2.5, 2.6, and 2.7 are 2002-09-03, 2003-04-14, 2004-02-29, and 2005-08-08, respectively. The time intervals between the four versions are 7, 10 and 17 months, respectively. Since the intervals between versions are not fixed, the use of a fixed time-window (e.g., 6 months) may exacerbate the inaccuracy of the time-window approach. Therefore, for example 2, we believe that the actual cause is that the inaccuracy of the time-window mechanism introduces mislabels and hence results in inconsistent labels.

Example 3 (the affected version approach [4]): the "KahaMessageStore.java" instances in the Activemq project. By inspection, we find an issue report AMQ-1529[4]: (1) reports a bug in "KahaMessageStore.java"; (2) the version numbers recorded in the *affected version* field are 5.0.0, 5.1.0, and 5.2.0; and (3) the developer stated: "*Also in KahaMessageStore.java should be added removing the blob files when the message is removed.*" Furthermore, "KahaMessageStore.java" instances have the same code on versions 5.0.0, 5.1.0, and 5.2.0. Therefore, the "KahaMessageStore.java" instance on version 5.1.0 should be "buggy" instead of "clean". As a

- *S. Liu, Z. Guo, Y. Li, C. Wang, L. Chen, Y. Zhou, and B. Xu are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu 210023, China. E-mail: {shiranliu, gzq, DZ1533030}@smail.nju.edu.cn; {yanhuili, lchen, zhouyuming, bwxu}@nju.edu.cn.*
- *Z. Sun is with the Mine Digitization Engineering Research Center of Ministry of Education, China University of Mining and Technology, Xuzhou, Jiangsu 221116, China. E-mail: zhongbin@cumt.edu.cn.*
(Corresponding authors: Yanhui Li, Lin Chen, and Yuming Zhou)

[1] Query command: git log -- tika-core/src/main/java/org/apache/tika/sax/xpath/XPathParser.java
[2] http://kenai.com/projects/buginfo
[3] http://archive.apache.org/dist/xml/xalan-j/
[4] https://issues.apache.org/jira/browse/AMQ-1529

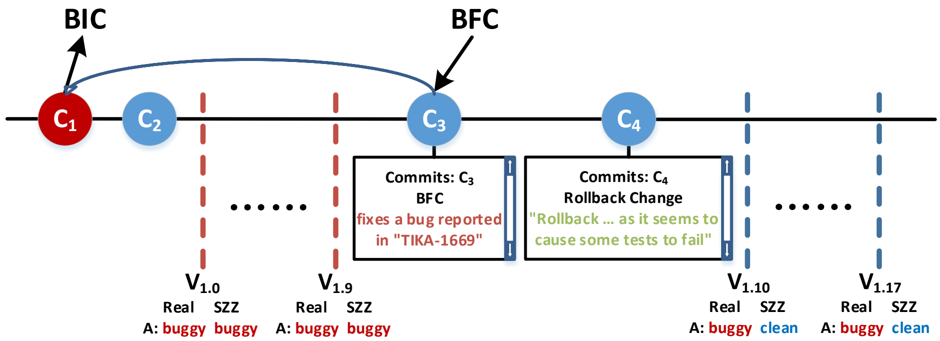

Fig. 1. SZZ-based approach: Schematic diagram of producing inconsistent labels in the “XPathParser.java” instances

result, for example 3, the actual cause is that the used affected version approach leverages only the earliest and the latest version numbers to link defective modules to versions, introducing mislabels and hence resulting in inconsistent labels (Note: this mislabeling problem may result from the issue report missing the version number 5.1.0 in the early maintenance phase).

Example 4 (our replicated MA-SZZ [6] approach): the “PostgreSqlInterpreterTest.java” instances in the Zeppelin project. The labels of the “PostgreSqlInterpreterTest.java” instances were collected from the issue reports in JIRA and the commits in GIT. As shown in Fig. 8, only version 0.6.0 was marked as “buggy”, and the other four versions (0.5.5, 0.5.6, 0.6.1, and 0.6.2) were marked as “clean”. By a Git query command, we find that there are eight commits involved in the “PostgreSqlInterpreterTest.java” instances: C_1 (bb96f42, 2015-08-09), C_2 (cd227fb, 2015-08-24), C_3 (54d4f48, 2015-10-16), C_4 (34e1385, 2016-01-17), C_5 (d85c7a1, 2016-01-18), C_6 (ed5b471, 2016-07-05), C_7 (3c93645, 2016-11-29), and C_8 (fa3f9f72, 2017-02-01). Fig. 2 shows their relationships with the corresponding versions. As can be seen, C_6 is a BFC. By MA-SZZ, C_5 is identified as BIC. This is the reason why MA-SZZ marks the “PostgreSqlInterpreterTest.java” instance as “buggy” on version 0.6.0 but as “clean” on the other four versions (0.5.5, 0.5.6, 0.6.1, and 0.6.2). However, the labels on 0.5.5, 0.5.6, 0.6.1, and 0.6.2 are indeed mislabels. Furthermore, on the one hand, by examining the commit log, we find that: C_4 is a commit deleting “PostgreSqlInterpreterTest.java” but C_5 is a rollback change for recovering this file. As a result, the “PostgreSqlInterpreterTest.java” instances on versions 0.5.5 and 0.5.6 have the same code as on 0.6.0. In other words, they have the same buggy code. On the other hand, the “PostgreSqlInterpreterTest.java” instances on version 0.6.1 and version 0.6.2 do not contain the code introduced by C_6. Therefore, it is reasonable to believe that there are branch-merge conflicts for versions 0.6.1 and 0.6.2. When these conflicts occur, developers only kept the content of “PostgreSqlInterpreterTest.java” from the derived branch rather than the master branch. As a result, for example 4, a rollback change leads to the mislabels on version versions 0.5.5 and 0.5.6, while the branch-merge conflicts lead to the mislabels on versions 0.6.1 and 0.6.2.

As can be seen from motivational example 2, not all the existing defect data sets can find enough defect data information and background information to analyze the cause of each inconsistent label. In fact, due to a variety of objective factors such as age factor / limited information published / maintenance or changes of defect data information, it is difficult for other researchers who are not original collectors of existing defect data sets to completely and accurately restore the actual causes of all inconsistent labels in the existing data sets. In addition, it is extremely time-consuming to manually analyze the causes of inconsistent labels (requiring a manual review of every line of code). Therefore, the quantitative analysis of the actual causes of inconsistent labels may need and deserve the participation of as many researchers as possible, and perhaps even need the collaborative efforts of the whole software engineering community toward this end.

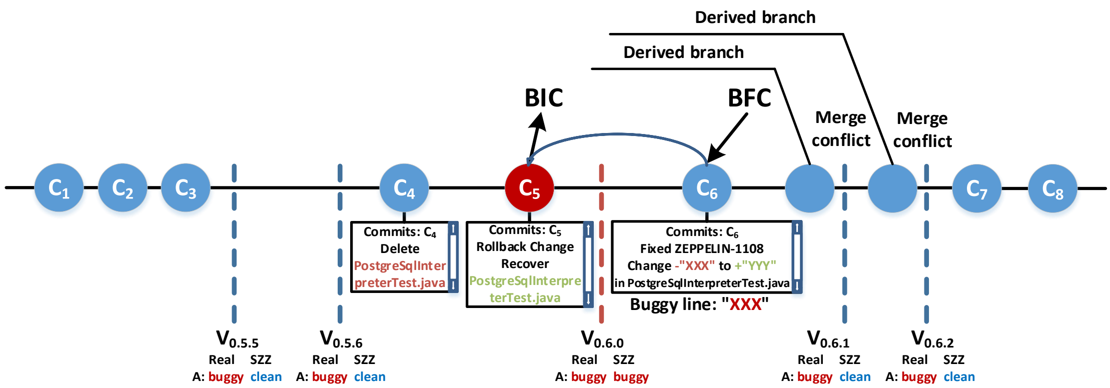

Fig. 2. Schematic diagram of producing inconsistent labels in the “PostgreSqlInterpreterTest.java” instances

TABLE 1
Summary of forms of a pair of inconsistent labels caused by different factors

Source	Occurrence Stage	Factor type	Cause		Mislabel	Real Labels	Inconsistent Labels
Extrinsic bug(s) (Note: #1)	When a non-code-change external factor had changed that affect the defect status of the module: Changes in requirements, run-time environment, dependencies on the run-time environment, and bugs in external APIs				FN	(buggy, buggy)	(clean, buggy) (buggy, clean)
					FP and FN	(clean, buggy)	(buggy, clean)
					Non-mislabeled	(clean, buggy)	(clean, buggy)
Source	**Occurrence Stage**	**Factor type**	**Cause**		**Mislabel**	**Real Labels**	**Inconsistent Labels**
Mislabeling (Note: #2)	First step: identify BFCs	Incomplete or incorrect data in VCS and ITS (i.e. errors in issue reports and/or commits)	Misidentified/unrecognized BFCs		FP	(clean, clean)	(clean, buggy) (buggy, clean)
					FN	(buggy, buggy)	(clean, buggy) (buggy, clean)
	Second step: analyze BFCs to determine which modules in a version are defective	Factors specific to a defect label collection approach (Note: #3)	SZZ-based	Imperfect code backtracking (non-source-code modification)	FP	V1.1, V1.2 (clean, clean)	V1.1, V1.2 (clean, buggy)
						V1.2, V1.3 (clean, clean)	V1.2, V1.3 (buggy, clean)
				Rollback change (type I)	FN	V1.1, V1.2 (buggy, buggy)	V1.1, V1.2 (clean, buggy)
				Rollback change (type II)		V1.2, V1.3 (buggy, buggy)	V1.2, V1.3 (buggy, clean)
			Time-window	Mixed-purpose BFC(s)	FP	V1.1, V1.2 (clean, clean)	V1.1, V1.2 (clean, buggy)
						V1.2, V1.3 (clean, clean)	V1.2, V1.3 (buggy, clean)
				Incorrect time window length	FN	V1.1, V1.2 (buggy, buggy)	V1.1, V1.2 (clean, buggy)
			Affected version	Mixed-purpose BFC(s)	FP	V1.1, V1.2 (clean, clean)	V1.1, V1.2 (clean, buggy)
						V1.2, V1.3 (clean, clean)	V1.2, V1.3 (buggy, clean)
				Ignorance of non-earliest affected version(s)	FN	V1.1, V1.2 (buggy, buggy)	V1.1, V1.2 (buggy, clean)
				Missing version(s)	FN	V1.1, V1.2 (buggy, buggy)	V1.1, V1.2 (clean, buggy)
				Error record(s)	FP	V1.1, V1.2 (clean, clean)	V1.1, V1.2 (clean, buggy)
					FN	V1.3, V1.4 (buggy, buggy)	V1.3, V1.4 (clean, buggy)
		Factors common to all defect label collection approaches (Note: #4)	Branch-merge conflict		FN	V1.1, V1.2 (buggy, buggy)	V1.1, V1.2 (buggy, clean)

#1 Inconsistent labels caused by extrinsic bugs can also be seen in Fig. 8 of our paper.
#2 Mislabeling here refers to mislabeling caused by factors other than extrinsic bug(s). Formally, the mislabeling corresponds to two situations: "buggy" is marked "clean", and "clean" is marked "buggy".
#3 Please refer to Fig. 9-14 (Section 4.1) for situations that generating inconsistent labels.
#4 Please refer to Fig. 15-18 (Section 4.2) for situations that generating inconsistent labels.
#5 The labels marked in red in the last column represent mislabels.

A.2. Summary of forms of a pair of inconsistent labels caused by different factors

Our study combines theoretical analysis and reverse analysis to distill the causes behind inconsistent labeling. On the one hand, by reading literature, we theoretically analyze which factors at each phase in a label collection process can lead to inconsistent labels. On the other hand, by manually tracing back the label collection process for real inconsistent label examples, we reversely analyze which factors can lead to inconsistent labeling.

From Section 3.2, we can see that extrinsic bugs and mislabeling are two sources of inconsistent defect labels. On the one hand, extrinsic bugs are caused by many factors in external factors outside the code of a project, including changes in requirements, dependencies on the run-time environment, changes to the environment, and bugs in external APIs [7, 8, 9]. If there is no mislabeling, extrinsic bugs will lead to inconsistent but correct defect labels. On the other hand, mislabeling can also lead to inconsistent defect labels, regardless of whether intrinsic or extrinsic bugs are involved. The factors causing mislabeling can be classified to two categories: (1) incomplete or incorrect data in VCS and ITS; and (2) an imperfect defect label collection mechanism or implementation. In practice, multiple factors with respect to extrinsic bugs and mislabeling may be tangled together, leading to inconsistent defect labels in a multi-version-project data set.

When collecting defect labels, all the SZZ-based, time-window, and affected version approaches are not aware of bug categories (intrinsic and extrinsic). As mentioned before, at a high level, they consist of two steps: (1) identify BFCs usually by linking commits for fixing bugs recorded in VCS to issue reports recorded in ITS; and (2) analyze BFCs to determine which modules in a version are defective. At the first step, incorrect or incomplete data in VCS and ITS may result in many missing links (i.e., many BFCs are not found) and many incorrect links (i.e., many identified BFCs are not correct). For the former, the main reason is that developers may forget to write specific keywords in the logs of commits in VCS or leave links for commit log in issue report description in ITS [10-18]. It is also possible that some BFCs are not recorded in VCS or some issues are not recorded in ITS [19, 20]. For the latter, the main reason is that many issues reported as bugs in ITS are actually requests for new features, bad documentation, or refactoring [21]. In particular, BFCs may contain non-fixing changes

[22, 23]. At the second step, even if all the BFCs are correctly identified at the first step, an inaccurate defect label collection mechanism can also introduce mislabels (see Sections 4.1 and 4.2 of our paper).

Table 1 summarizes the factors behind inconsistent labeling that we have found so far, including the source (the 1st column), the occurrence stage (the 2nd column), the type of mislabel (the 3rd column), the real label (the 4th column), and the form of inconsistent label (the last column).

In Table 1, the discovery and confirmation process for each factor is as follows:

- The discovery and confirmation of the **"extrinsic bug(s)"** factor came from the literature [7]. Inspired by [7], we make a theoretical analysis that the extrinsic bug(s) is one of the causes of inconsistent labels. Please see Section 2.2 and the example shown in Fig. 8 in Section 3.2 of our paper for details.
- The discovery and confirmation of the **"non-source code modification"** factor is inspired by literatures [6, 24-26] (see the example shown in Fig. 9 in Section 4.1 and Section 8.1 of our paper for details) and our manual verification. By changing our MA-SZZ code to original SZZ, we verified that non-source code modification factor do lead to mislabels and further lead to inconsistent labels.
- The discovery and confirmation of the **"rollback change"** factor is derived from our manual analysis of BIC and BFC commits of motivational examples 1 and 4 (see the above Appendix A.1 for details).
- The **inaccuracy of the time-window mechanism** is demonstrated by literatures [2-4]. Please see Section 2.2 and the example shown in Fig. 11 in Section 4.1 of our paper and the motivational example 2 in the above Appendix A.1 for details.
- The discovery and confirmation of the **"mixed-purpose BFC(s)" factor** is inspired by literatures [22, 23]. We find that the time-window approach and affected version approach do not exclude mixed-purpose BFC(s), which can lead to inconsistent labels. Please see the examples shown in Fig. 9 and Fig. 11-14 in Section 4.1 of our paper for details.
- The discovery of the **"ignorance of non-earliest affected version(s)"** stems from our analysis of the method description used in literature [5]. By manually analyzing the developer's records of issue reports, we find that the reported bugs affect not only the earliest version, but also other versions that were recorded. Please the example shown in Fig. 12 in Section 4.1 of our paper and the motivational example 3 in the above Appendix A.1 for details.
- The discovery and confirmation of the **"missing version(s)"** factor is derived from our manual analysis of issue reports of motivational examples 3. Please see the above Appendix A.1 for details.
- The discovery and confirmation of the **"error record(s)"** factor is inspired by literature [1] and our manual verification. The literature [1] showed that the *affected version* field can be filled with incorrect values. Through manual analysis of issue reports, we confirm the existence of incorrect version records in the *affected version* field. Please the example shown in Fig. 14 in Section 4.1 of our paper and the motivational example 3 in the above Appendix A.1 for details.
- The discovery and confirmation of the **"branch-merge conflict"** factor is derived from our manual analysis of BIC and BFC commits of motivational examples 4. During the forward and backward tracking of BIC and BFC, we find the phenomenon of branch missing. Further, by analyzing all branch histories in GIT, we confirm that the "branch-merge conflict" factor is one of the causes of inconsistent labels. Please see the motivational example 4 in the above Appendix A.1 for details.

Appendix B. TSILI: A three stage inconsistent label identification approach

This section introduces our proposed TSILI approach, including objective, algorithm flow, effectiveness analysis, time complexity, running time, influencing factors and implication.

B.1. Objective and principle of inconsistent label detection

The objective of inconsistent label detection is to find cross-version instances with inconsistent labels (i.e., the same code but different defect labels) from multi-version defect data sets. For intuitive understanding, we use the example shown in Fig. 3 to describe. Assume that a multi version defect data set has five versions (V_1~V_5); X_1, X_2 and X_3 are three cross-version instances, in which X_2 only exists in V_1, V_3, V_4 and V_5. The defect label of an instance is represented by "0" (non-defective) or "1" (defective). If the code of an instance is the same on both versions, we use a black line to connect it. In this context, the defect labels of X_1 on (V_1, V_2, V_3) are inconsistent labels, because the code of X_1 on (V_1, V_2, V_3) is the same, but the defect labels are different. Although the code of X_1 on (V_4, V_5) is the same, the defect labels of X_1 on (V_4, V_5) are not inconsistent labels because the defect labels are the same. Similarly, the defect labels of X_2 on (V_1, V_3) are inconsistent labels. The defect labels of X_3 on (V_1, V_3) and (V_2, V_4) are inconsistent labels. For a single version, the instances with inconsistent labels on V_1 and V_3 are (X_1, X_2, X_3), the instances with inconsistent labels on V_2 are (X_1, X_3), the instances with inconsistent labels on V_4 are (X_3), and the instances with inconsistent labels are not included on V_5. By detecting inconsistent labels, we can investigate the degree of inconsistent labels on each version of a collected defect data set, so as to assist in evaluating the label quality of a collected defect data set.

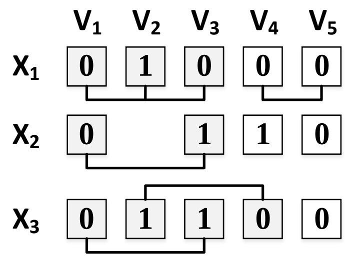

Fig. 3. Schematic diagram of inconsistent label detection

B.2. Algorithm

To automatically identify inconsistent labels in a multi-version-project defect data set, we propose an efficient and effective approach called **T**hree **S**tage **I**nconsistent **L**abel **I**dentification (TSILI). For the simplicity of presentation, we assume that a project consists of n versions. For each version V_i, let DV_i be the corresponding defect data set and SV_i be the corresponding source code database, $1 \leq i \leq n$. In this context, the task of TSILI is to identify inconsistent labels for the multi-version-project defect data set consisting of DV_1, DV_2, …, and DV_n (in the following, DV_i will be called a component defect data set). Specifically, for each component defect data set DV_i, TSILI aims to add a feature to indicate which instances have inconsistent labels, $i \leq n$. To this end, TSILI proceeds as follows. At the first stage (see Section B.2.1), given the inputs of a multi-version-project defect data set and the corresponding source code databases, an information table is generated to record those instances whose source codes can be found. At the second stage, the elements in the information table are analyzed to identify instances with inconsistent labels, i.e. those cross-version instances with the same name, the same source code, but different defect labels (see Section B.2.2). At the third stage, for each instance in the multi-version-project defect data set, a feature is added to indicate whether its label is inconsistent or not based on the inconsistent label information recorded in the information table (see Section B.2.3).

B.2.1. Stage 1: Generate an Information Table Recording Instances in all Versions

For the first stage, the inputs are a multi-version-project defect data set (i.e. DV_1, DV_2, …, and DV_n) and the corresponding source code databases (i.e. SV_1, SV_2, …, and SV_n).

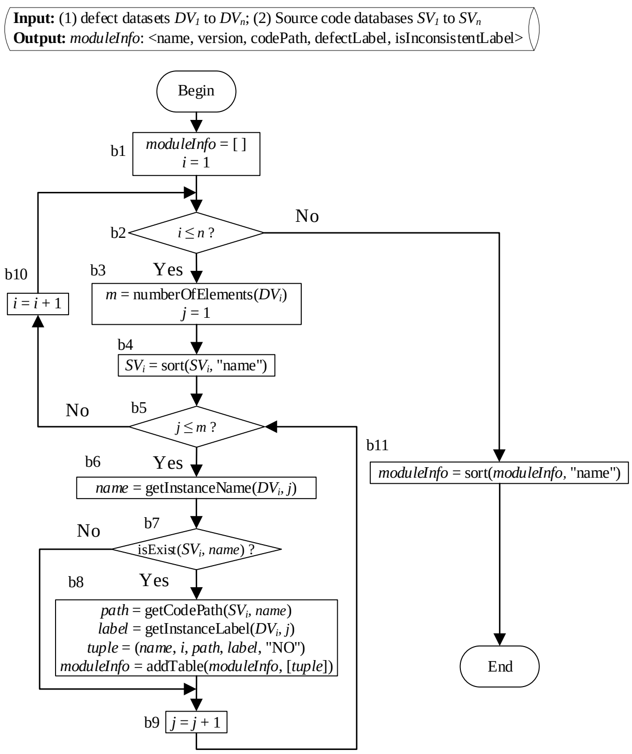

Fig. 4. The flowchart of generating an information table recording all instances in all versions

The output is an information table *moduleInfo* recording the following information for those instances whose source codes can be found in the source code databases: (1) "name": the name of an instance in multi-version-project defect data set (i.e. the name of the corresponding module in the source code databases); (2) "version": the version number of the project that the corresponding module of an instance belongs to; (3) "codePath": the path of the file in which the source code of the corresponding module of an instance is located; (4) "defectLabel": the defect label of an instance in the multi-version-project defect data set ("0" is clean and "1" is buggy); and (5) "isInconsistentLabel": whether the defect label is an inconsistent label ("NO" means non-inconsistent label, while "YES" means an inconsistent label).

As shown in Fig. 4, the first stage proceeds as follows. First, set *moduleInfo* as an empty table (b1). Second, analyze the multi-version-project defect data set and the corresponding source code databases to generate *moduleInfo* (b2~b10). Specifically, for each instance in DV_i, examine whether its name appears in SV_i (b7). If the answer is "Yes", obtain the path of the corresponding source code recorded in SV_i and the corresponding defect label recorded in DV_i. With such information, a five-tuple (i.e. <name, version, codePath, defectLabel, isInconsistentLabel>) is generated and added to *moduleInfo*, in which "isInconsistentLabel" is set as "NO" (b8). Third, sort the elements (each element is a five-tuple, corresponding to an instance) in *moduleInfo* in ascending order according to the instance name (b11). As a result, the instances with the same name on different versions (i.e. cross-version instances) are grouped together. This will facilitate the identification of cross-version instances with inconsistent labels in the next stage.

B.2.2. Stage 2: Identify Cross-version Instances with Inconsistent Labels

For the second stage, the input is the information table *moduleInfo* generated in the first stage, in which all the instances have the same "NO" value for the attribute "isInconsitentLabel". The output is an updated *moduleInfo* in which inconsistent labels have been recorded, i.e. instances with inconsistent labels have a value "Yes" for the attribute "isInconsitentLabel".

As shown in Fig. 5, the second stage proceeds as follows. First, divide the elements (i.e. instances) in *moduleInfo* into different groups by their "name" values (b1). Within each group, the instances have the same instance name but are from different versions, i.e. they are cross-version instances. Second, for each group, identify which instances have the same source code but different defect labels (b2~b19). Specifically, for each group, obtain the corresponding set of defect labels *labelSet*. If |*labelSet*| == 1, skip this group as there is no inconsistent label (b19). Otherwise, the instances in the group are partitioned into equivalence classes by comparing their source codes (b6~b18). Within each equivalence class, the instances not only have the same name but also have the same source code. In order to reduce the influence of code format, the following measures are taken to format the code (b9) before the partition (b11): filter out comments and replace consecutive whitespaces and newlines with one whitespace character. For each

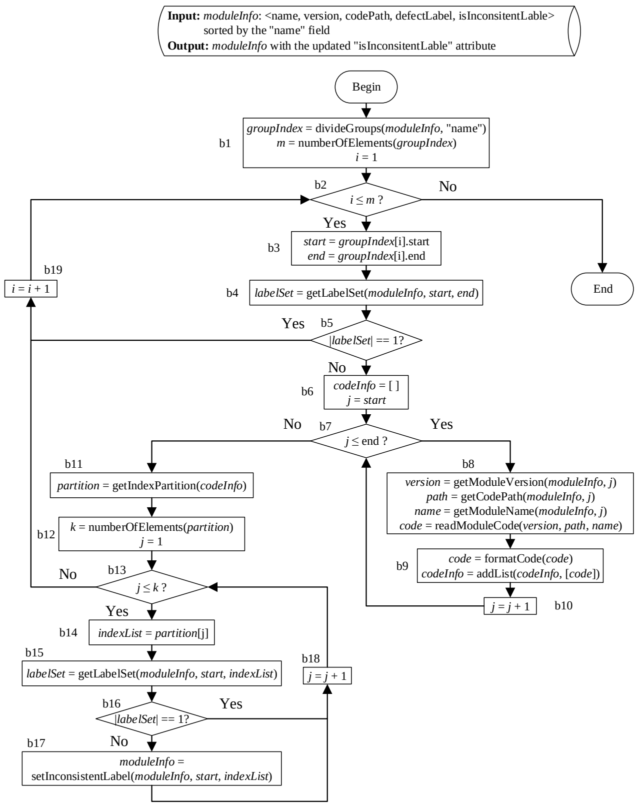

Fig. 5. The flowchart of identifying cross-version instances with inconsistent labels

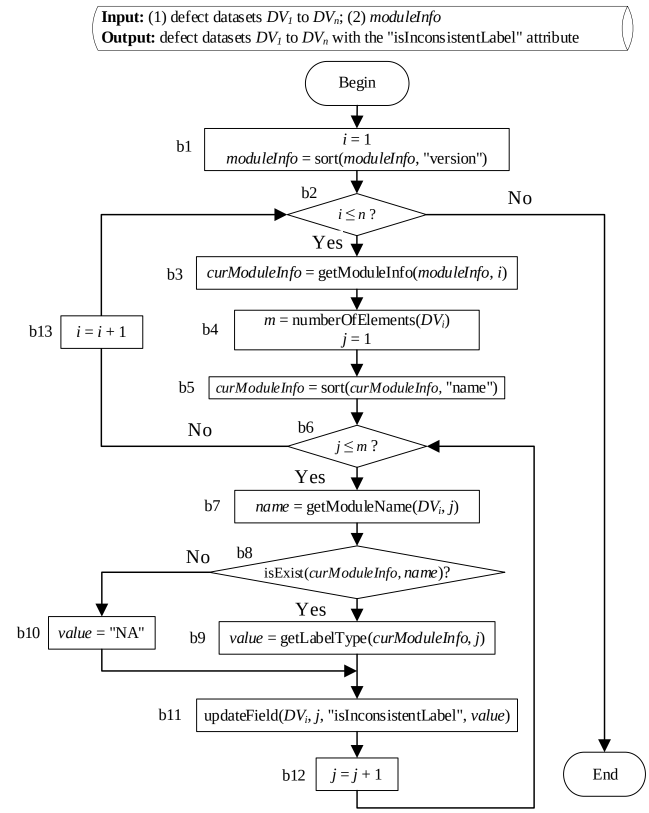

Fig. 6. The flowchart of augmenting the multi-version-project defect data set with inconsistent labels

equivalence class, examine whether all the instances have the same label (b16). If the answer is "No", this means that the instances in this equivalence class have inconsistent labels and update *moduleInfo* to record this information accordingly (b17). When the second stage terminates, it outputs *moduleInfo* with the updated "isInconsitentLabel" attribute.

B.2.3. Stage 3: Augmenting the Multi-version-project Defect Data Set with Inconsistent Labels

For the third stage, the input is the multi-version-project defect data set (i.e. DV_1, DV_2, …, and DV_n) and the information table *moduleInfo* generated at the second stage (in which all the instances have the "Yes" or "NO" value for the feature "isInconsitentLabel"). The output is the multi-version-project defect data set augmented with the feature "isInconsitentLabel" that indicates whether an instance has an inconsistent label: "NO" means non-inconsistent, "YES" means inconsistent, and "NA" means unknown (for an instance, if the corresponding source code cannot be found, its feature "isInconsitentLabel" will be assigned an "NA" value).

As shown in Fig. 6, the third stage proceeds as follows. First, sort the elements in *moduleInfo* according to their "version" values so that the instances whose corresponding modules belong to the same version are grouped together (b1). Second, for each component defect data set, add a feature "isInconsitentLabel" and set its value for each instance based on the inconsistent label information recorded in *moduleInfo* (b2~b13). Specifically, for DV_i, take all the elements belonging to the *i*th version from *moduleInfo* to *curModuleInfo* (b3). For each instance in DV_i, examine whether its name appears in *curModuleInfo* (b8). If the answer is "Yes", set its "isInconsitentLabel" as the corresponding value recorded in *curModuleInfo*; otherwise, set its "isInconsitentLabel" as a value of "NA" (b9~b11). When the third stage terminates, it outputs a multi-version-project defect data set augmented with inconsistent label information.

B.3. Effectiveness and time complexity

Effectiveness. For TSILI, the core is to examine whether two cross-version instances have the exactly same (non-comment, non-blank) source code after excluding the influence of code format. In order to check the correctness of code comparison, we compare the results obtained from TSILI and Bcompare[5] (a commercial code comparison tool). On six different multi-version-project defect data sets, TSILI found a total of 10194 instances with inconsistent labels, in which 3628 different module names are involved (see Section 6.2). We extracted a statistically significant sample that was created based on 3628 with a confidence level of 99% and a margin of error of 5%, resulting in a stratified sample of 563 module names[6]. For each of 563 module names, we first sampled a pair of the corresponding cross-version instances with inconsistent labels. Then, we used Bcompare to compare their code. We found, after ignoring the factors such as comments and line feeds, Bcompare always produced the same result as reported by

[5] https://www.scootersoftware.com

[6] https://www.calculator.net/sample-size-calculator.html?type=1&cl=99&ci=5&pp=50&ps=3628&x=96&y=19

TSILI, i.e., the two modules were believed to have the same code. This is in line with expectations, because the code comparison function in TSILI is based on the Python APIs provided by the Understand[7] tool (a code analysis tool), while Bcompare is based on its own underlying API implementation. Both Understand and Bcompare are mature commercial software, which ensures the correctness of code comparison. Through the above inspections, we ensure the effectiveness of our TSILI algorithm (at least it did not produce false positive under our investigation). However, there may be a small number of inconsistent labels that cannot be recognized by TSILI due to objective factors, which will be discussed in Section B.5.

Time complexity. As mentioned above, for TSILI, the inputs are a multi-version-project defect data set (i.e. DV_1, DV_2, …, and DV_n) and the corresponding source code databases (i.e. SV_1, SV_2, …, and SV_n). In order to facilitate the complexity analysis of TSILI, we assume that: (1) SV_i consists of s_i modules and DV_i consists of d_i instances, $1 \le i \le n$; (2) e_i is the number of instances in the intersection of SV_i and DV_i, $1 \le i \le n$; (3) $s = \max(\{s_i \mid 1 \le i \le n\})$ and $d = \max(\{d_i \mid 1 \le i \le n\})$; and (4) $S = \sum_{i=1}^{n} s_i$ and $D = \sum_{i=1}^{n} d_i$. Let E be the number of elements in the information table *moduleInfo*. Therefore, we have $E = \sum_{i=1}^{n} e_i$, $E \le S$, and $E \le D$. Furthermore, assume that l is the number of characters of the module that has the most characters in the source code databases. Note that, for a given software version V_i, if the defect data collection process is accurate, we should have $e_i = s_i = d_i$, $1 \le i \le n$. In practice, however, due to a variety of unknown reasons, s_i may be different from d_i, $1 \le i \le n$. In the following, for the simplicity of presentation, the former is called the "IDEAL" condition, while the latter is called the "REAL" condition.

The total time complexity of TSILI is equal to the sum of the time complexities of its three stages.

- Stage 1 has a time complexity of $O(S\times\log(S) + D\times\log(s))$. The first stage consists of three parts: initialize *moduleInfo* (b1), generate *moduleInfo* (b2~b10), and sort *moduleInfo* (b11). The first part has a time complexity of $O(1)$, while the third part has a time complexity of $O(E\times\log(E))$ (e.g. using quick sort). At the second part, each instance is examined one time, in which all statements but b4 and b7 have a complexity of $O(1)$. As for b4, each execution requires $s_i\times\log(s_i)$ comparisons. Since b4 is executed n times, the total number of comparisons is: $s_1\times\log(s_1) + \ldots + s_n\times\log(s_n) < S\times\log(S)$. As for b7, since SV_i has been sorted, each search is a binary search and requires at most $\log(s)$ comparisons. Since b7 is executed D times, the total number of comparisons is $D\times\log(s)$. Therefore, the time complexity of the second part is at most $O(S\times\log(S) + D\times\log(s))$. As a result, stage 1 at most has a time complexity: $O(1) + O(E\times\log(E)) + O(S\times\log(S) + D\times\log(s)) = O(S\times\log(S) + D\times\log(s))$.
- Stage 2 has a time complexity of $O(E\times\log(n)\times l)$. The second stage consists of two parts: group instances (b1) and identify inconsistent labels (b2~b19). Since *moduleInfo* has already been sorted, the first part has a time complexity of $O(E)$. The second part iterates over each group: if the instances in the current group do not have the same label (b3~b5), then get the code (a filtered string) for each instance (b6~b10), partition instances into equivalence classes by the same code (b11), iterate over each equivalence class to examine the label (b12~b18). For the second part, "formatCode" in b9 and "getIndexPartition" in b11 dominate the time complexity. In "formatCode", the code is parsed to filter out comments, newlines, and extra spaces. For each execution, its time complexity is proportional to the code length (at most l). Since "formatCode" is executed at most E times, the total time complexity of "formatCode" in b9 is at most $O(E\times l)$. In "getIndexPartition", the instances in the same group are compared by their codes to obtain equivalence classes. For a group with x instances, the equivalence class partition can be performed by $x \times \log(x) \times l$ comparisons, i.e. each element needs $\log(x) \times l$ comparisons. Since x is at most n and the total number of instances in *moduleInfo* is E, the total time complexity of "getIndexPartition" in b11 is $O(E\times\log(n)\times l)$. As a result, stage 2 at most has a time complexity: $O(E) + O(E\times l) + O(E\times\log(n)\times l) = O(E\times\log(n)\times l)$.
- Stage 3 has a time complexity of $O(E\times\log(E) + D\times\log(d))$. The third stage consists of two parts: sort *moduleInfo* (b1) and augment the multi-version-project data set (b2~b13). The first part has a time complexity of $O(E\times\log(E))$. At the second part, each instance in the multi-version-project data set is examined one time, in which "sort" in b5 and "isExist" in b8 dominates the time complexity. For "sort" in b5, each execution requires $e_i\times\log(e_i)$ comparisons. Since b5 is executed n times, the total number of comparisons is: $e_1\times\log(e_1) + \ldots + e_n\times\log(e_n) < E\times\log(E)$. For "isExist" in b8, since *curModuleInfo* has already been sorted, each search is a binary search and requires at most $\log(d)$ comparisons. Since b8 is executed at most D times, the total number of comparisons is at most $D\times\log(d)$. Therefore, the time complexity of the second part is at most $O(E\times\log(E) + D\times\log(d))$. As a result, stage 3 at most has a time complexity: $O(E\times\log(E)) + O(E\times\log(E) + D\times\log(d)) = O(E\times\log(E) + D\times\log(d))$.

Therefore, under the "REAL" condition, at the worst case, the total time complexity of TSILI is: $O(S\times\log(S) + D\times\log(s)) + O(E\times\log(n)\times l) + O(E\times\log(E) + D\times\log(d)) = O(S\times\log(S) + D\times\log(s) + E\times\log(n)\times l + D\times\log(d))$. Under the "IDEAL" condition, at the worst case, the total time complexity of TSILI will become $O(D\times\log(D) + D\times\log(n)\times l)$ (since $d < D$, $s = d$, and $E = S = D$).

B.4. The running time of our TSILI algorithm

The method we adopted to generate the source code databases required by the TSILI algorithm is to download the

TABLE 2
Time consuming of the TSILI algorithm

Dataset	Project (versions)	n, totalIns, sumSLOC	Running Time	Experimental environment
Metrics-Repo-2010	Log4j (1.0, 1.1, 1.2)	n=3, totalIns=411, sumSLOC=74857	≈ 25 seconds	Inter(R) Core(TM) i7-7700 CPU @ 3.6GHz and 16G RAM
JIRA-RA-2019	Hive (0.9.0, 0.10.0, 0.12.0)	n=3, totalIns=5285, sumSLOC=974774	≈ 11 minutes	
ECLIPSE-2007	Eclipse (2.0, 2.1, 3.0)	n=3, totalIns=25203, sumSLOC=3089619	≈ 3 hours	

[7] http://scitools.com

codes corresponding to each version from the official website of each target project, and then use the Understand[8] tool to parse the code to generate source code databases (.udb file). We write a Python[9] script to implement the TSILI algorithm. In the second stage of TSILI, the source code of each module is parsed and filtered (b9 in Fig. 5) based on the Python API (application programming interface) provided by the Understand tool.

In order to observe the time required for the TSILI algorithm to detect inconsistent labels on projects of different orders of magnitude, we selected Log4j, Hive, and Eclipse projects from multi-version-project defect data sets Metrics-Repo-2010 [2], JIRA-RA-2019 [4], and ECLIPSE-2007 [5], respectively, and then ran TSILI and recorded the time spent. Table 2 lists the details of these three projects and the single thread running time of TSILI. The 2nd column lists the versions included in each project. The 3rd column lists the order of magnitude of the project size, where *n*, *totalIns*, and *sumSLOC* represent the number of versions, the total number of instances of all versions, and the total number of code lines of all versions, respectively. The 4th column reports the running time of TSILI. These three projects (Log4j, Hive, and Eclipse) were selected because they represented orders of magnitude of the size of the projects in their respective data sets (the ECLIPSE-2007 data set only has the Eclipse project). In addition, the size of the total number of instances of these three projects varies in turn by one order of magnitude, which is conducive to observe the running time of TSILI under different orders of magnitude data sets.

Table 2 shows that the running time of TSILI is positively correlated with the size of the data set. For the project with hundreds of instances (i.e., Log4j project), the running time is at the second level. For the projects with ten thousands of instances (i.e., Eclipse project), the running time is at the hour level. Because TSILI is an offline algorithm, the hour-level (even minute-level or second-level) running time is acceptable in practice.

B.5. What factors would influence the number of inconsistent labels identified by the TSILI algorithm?

Section B.3 shows that our TSILI algorithm is effective in identifying inconsistent labels. The rationale of TSILI is simple: it only needs to compare the source code and defect labels of cross-version instances, without any other complex procedures and additional information. Therefore, the following question naturally arises: Can TSILI identify all identify inconsistent labels in a defect data set? The answer is No. Because, in practice, there are three factors that can influence the number of inconsistent labels that the TSILI algorithm can identify. First, TSILI cannot be applied to cross-version instances that have no source code. For example, for the Camel project in the JIRA-HA-2019 [4] and JIRA-RA-2019 [4] data sets, the cross-version instance "package-info.java" contains only comment statements. In addition, due to unknown reasons, about 1%~9% instance cannot be found in the source codes downloaded from the official websites. Specifically, for ECLIPSE-2007 [5], JIRA-HA-2019, and JIRA-RA-2019, about <1% instances in 77% (60/78) versions cannot be found; for IND-JLMIV+R-2020 [1], about 1%~9% instances in 13% (51/395) versions cannot be found; for METRICS-REPO-2010 [2], about 1%~7% instances in 78% (25/32) versions cannot be found[10]. In this case, for these instances, it is not possible to apply TSILI to identify inconsistent labels. Second, if the path name or file name of a cross-version instance has changed between versions, the TSILI algorithm is unable to detect inconsistent labels, even if its source code remain no change. In our implementation, TSILI identifies a cross-version instance (or module) among versions based on its full name (path name + file name). If the full name is changed, a cross-version instance will be treated as two different instances. In this case, TSILI may miss inconsistent labels. In our study, we did observe such a phenomenon in the investigated multi-version-project defect data sets, although it did not happen often. Third, the number of versions used will influence the number of inconsistent labels that TSILI can identify. As reported in [19, 20], for a project, it was common that a considerable proportion of bugs in a low version would not be discovered until in high versions. In our study, for ECLIPSE-2007, METRICS-REPO-2010, JIRA-HA-2019, and JIRA-RA-2019, each project contains only three to five versions. Nonetheless, TSILI still found a large number of inconsistent labels. Therefore, it is reasonable to believe that more inconsistent labels can be identified if more versions are analyzed, i.e., the multi-version-project defect data sets investigated in our study should contain more inconsistent labels than reported in Section 5.1 in our paper. This means that the influence of inconsistent labels may be underestimated in our study.

B.6. Can software metrics be used as a proxy of source code to identify inconsistent labels?

In TSILI, inconsistent labels are regarded as found if a cross-version instance has the same source code but different labels in different versions. During this process, there is a need to compare source code to identify cross-version instances. However, in practice, it is common to see that a data set only provides for each instance a number of software metrics (i.e. features) and a label indicating whether it is defective. In other words, source code is external information for a defect data set, which needs to be acquired additionally. In this context, an interesting question naturally arises: Can we use software metrics as a proxy of source code to identify cross-version instances? Indeed, in previous studies [27-29], it is not uncommon to see that software metric information is used to identify "inconsistent instances" in a defect data set. In [27, 29], if two instances in a version had identical values for all features but different labels, they were called "inconsistent instances". In [28], inconsistent cross-version instances were also examined. In their view, inconsistent instances are problematic in the context of machine learning and hence should

[8] https://scitools.com
[9] https://www.python.org
[10] The exceptions are that in IND-JLMIV+R-2020, only 73% and 78% instances of "santuario-java-1.2.0" and "commons-math-1.1" can be found; In METRICS-REPO-2010, only 85% and 86% instances of "log4j-1.0" and "xerces-1.4" can be found, respectively.

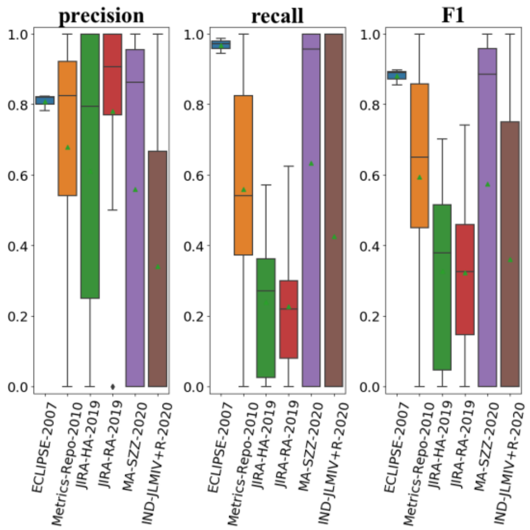

Fig. 7. Distribution of prediction performance scores of SMI for detecting inconsistent labels

be excluded when building and evaluating a defect prediction model. As can be seen, the concept of their inconsistent cross-version instances is very similar to the concept cross-version instances with "inconsistent labels" in our study. The difference is that the former uses software metrics rather than source code to identify cross-version instances. At a glance, it seems that we could use software metrics to replace source code to identify cross-version instances in our study. In fact, such a replacement is problematic due to the following two-fold reasons. On the one hand, the fact that two instances have identical source code does not necessarily mean that they have identical values for all features (i.e. metrics). The reason is that many metrics depend on the use context of an instance rather than only on its source code. For example, two functions with identical code may have different values for the "called-by number" feature. On the other hand, the fact that two instances have identical values for all features does not necessarily mean that they have identical source code. For example, it is possible that two instances with different source code have identical values for all features. As a result, if we use software metrics as a proxy to identify cross-version instances, it is possible to miss real "inconsistent labels" or report false "inconsistent labels", thus leading to a low accuracy of inconsistent label identification. Given this situation, we should not use software metrics as a proxy to identify inconsistent labels.

We next use the inconsistent labels identified by TSILI as the ground truth to empirically understand how good software metrics can be used as a proxy for inconsistent label identification. For the simplicity of presentation, we use SMI to denote the method that uses software metrics to identify inconsistent labels. Fig. 7 reports for SMI the *precision*, *recall*, and F_1 on six multi-version-project data sets. Here, *precision* denotes the percentage of inconsistent labels identified by SMI that are also identified by TSILI, *recall* denotes the percentage of inconsistent labels identified by TSILI that are also identified by SMI, and F_1 is the harmonic mean of *precision* and *recall*. As can be seen, for five out of the six data sets, all the three indicators vary in a large range. For ECLIPSE-2007, all the three indicators have a small variance, as the inconsistent label ratios are low on all the three versions. Furthermore, on the one hand, for most data sets, the median/mean recall is low, indicating a high false negative (i.e., real inconsistent labels are incorrectly identified as non-inconsistent labels by SMI). On the other hand, for most data sets, the median/mean precision is around or less than 0.8, indicating there is non-negligible false positive (i.e., non-inconsistent labels are incorrectly identified as inconsistent labels by SMI). Overall, the above results reveal that SMI has a low application value, even though it has a lower computation cost than TSILI.

B.7. Implications

Our work contributes a validated automatic approach TSILI to detect inconsistent labels in defect data sets. In practice, inconsistent labels can be used as a risk indicator to evaluate the quality of defect data sets. For practitioners and researchers, this work (TSILI) has the following important implications:

(1) **Our work provides practitioners a simple but effective way to examine the quality of to examine the quality of labels in multi-version-project defect data sets before building defect prediction models.** This is helpful for them to exclude the potential label noise and hence obtain a quality defect prediction model. In practice, there are two approaches to examining the quality of a data set. The first approach is to apply a noise detection approach such as CLNI [30] to identify noisy instances, while the second approach uses manually-curated data in VCS/ITS to identify noisy instances [31]. However, for the former, it is inevitable to report false noisy instances, i.e. actually non-noisy identified as noisy. For the latter, it is time-consuming to manually curate the data. Compared with the first approach, Our TSILI approach does not report false noisy instances (i.e. false inconsistent labels). Compared with the second approach, Our TSILI approach is a light-weight approach and can be automated, as no manual curation is needed. As a result, we suggest using TSILI to examine the quality of multi-version-project defect data sets in practice.

(2) **Our work provides a means of generating (partial) noise ground truth, which can assist researchers in evaluating the effectiveness of existing noise identification approaches.** In the literature, multiple approaches have been proposed to detect the noise in a defect data set [27-30]. However, it is difficult to evaluate their effectiveness due to the following two reasons. On the one hand, it is time consuming to obtain the ground truth manually [21, 32]; on the other hand, it is difficult to obtain the ground truth for the data sets lacking background information (such as not knowing the tools, approaches, issue reports, and commits used). In this context, we recommend using our proposed TSILI approach to automatically identify inconsistent labels. As mentioned earlier, inconsistent labels mean noise data. Therefore, the identified inconsistent labels can be used as the (partial) noise ground truth to assist in the evaluation of noise identification approaches.

Appendix C. Data sets information and software metrics used in the MA-SZZ-2020

C.1. Multi-version-project defect data sets

In our study, we used the six multi-version-project defect data sets to conduct the experiment: ECLIPSE-2007 [5], Metrics-Repo-2010 [2, 3], JIRA-HA-2019 [4], JIRA-RA-2019 [4], MA-SZZ-2020, and IND-JLMIV+R-2020 data set [1].

- ECLIPSE-2007. This data set corresponds to one project with three versions. According to [5], an affected version approach was employed to collect the defect label data. After identifying BFCs by matching regular expressions with comments of commits, the first release listed in the "*version*" field of the corresponding issue reports in BUGZILLA was used to link defective modules to versions. During this process, the analyzed issue reports were limited to those reported in the first six months after release.
- Metrics-Repo-2010. This data set corresponds to 12 projects with 43 versions[11]. According to [2, 3], a time-window approach was used to link defective modules to versions. For each target version of interest, a tool called BugInfo was employed to identify its BFCs by regular expression matching. During this process, the time window was set to the period between the release time of the target version and the release time of the next version.
- JIRA-HA-2019. This data set corresponds to 9 projects with 32 versions. According to [4], a time-window approach (Yatish et al. regarded it as a heuristic approach, abbreviated as HA) was used to link defective modules to versions. For each target version of interest, a collection of regular expressions was applied on commit logs to identify BFCs. During this process, the time window was set as a 6-month period after the version of interest was released.
- JIRA-RA-2019. This data set was collected from the same projects as used in JIRA-HA-2019. However, an affected version approach (rather than a time-window approach) was used to link defective modules to versions. In [4], Yatish et al. regarded it as a realistic approach (abbreviated as RA). For each target version of interest, they first retrieved these issue reports in JIRA whose "*affected version*" fields listed the target version as the earliest affected version. Then, they leveraged the traceable links (provided by JIRA) between issue reports and code commits to identify BFCs. During this process, the complete history of the target version after release was analyzed in order to reduce false negative modules.
- MA-SZZ-2020. This data set corresponds to 5 projects with 50 versions. We first used MA-SZZ [6], the state-of-the-art SZZ variant, to collect BICs. When identifying BICs, the following changes were excluded: (1) non-semantic code changes (e.g. changes of annotations, spaces, and blank lines); (2) format changes (e.g. moving the bracket), and (3) meta-changes, including branch change (copy the project state from one branch to another), merge-change (apply change activity from one branch to another), property change (only impact file properties stored in the VCS). After that, we leveraged BFCs and their corresponding BICs to link defective modules to versions. Similar to [1, 4], we select Apache Java projects with Git VCS and JIRA ITS as the subject projects, because: (1) "Apache projects must have reached a certain level of maturity in order to be considered as a top-level project" [1]; and (2) they have a high (traceable) link rate between issue reports to commits[12]. In particular, they should have the property that a version with a smaller release number has an earlier release time. This property is important for the projects under consideration, as it helps accurately link defective modules to versions. To further ensure the maturity and popularity of projects, a project under consideration was required to have: (1) at least 10 versions; (2) at least 1000 stars on GitHub; and (3) at least 100 issue reports[13], each having a "Bug" *Type*, a "resolved" or "closed" *status*, and a "fixed" *Resolution*. Consequently, we obtained the following five projects: Zeppelin, Shiro, Maven, Flume, and Mahout. For each considered version of the five projects, we used a tool called "Understand" to collect 44 code metrics.
- IND-JLMIV+R-2020. This data set consists of 395 versions of 38 projects[14]. According to [1], a semi-automatic defect label collection approach was employed to collect the defect label data, which is called IND-JLMIV+R. In nature, IND-JLMIV+R is an improved SZZ-based approach combined with manual validation. It uses manual validation to identify BFCs which can be used as the ground truth and uses a variety of improved heuristic methods to improve the accuracy of SZZ approach in identifying BICs. Specifically, this approach reduces the introduction of mislabels in five ways. First, this approach uses manual classification to validate types of issue reports and correct type errors. Second, use "JIRA link" to establish the link between a BFC and an issue report. "JIRA link" refers to the fixed format (<project-ID>) used for the identifier of issue report in JIRA, and the corresponding BFC also uses the same format to indicate the fixed bug. Using "JIRA link" helps to avoid misidentification of BFCs. Third, the heuristic rules of "JIRA link" and original SZZ approach are used to search for BFCs, and then manual validation is carried out to select correct BFCs that can be used as the ground truth. Fourth, use the RA-SZZ approach [25], one of the latest SZZ variants, to identify BICs. The accuracy of RA-SZZ approach is further improved by using the RefactoringMiner [33] tool instead of RefDiff [34]. Fifth, according to the creation time of pairs of BICs and BFCs, link defective modules to ver-

[11] Note that the original Metrics-Repo-2010 data sets also include Ivy project with 1.1, 1.4, and 2.0 versions. However, we were unable to find their corresponding code from the official website and hence excluded them from our experiment.

[12] According to [18], "Apache developers are meticulous in their efforts to insert bug references in the change logs of the commits".

[13] All the issue reports had a "Created Date" less than "2019-12-03 12:00", as this was the time point we collected the data.

[14] Note: the original dataset contains 398 versions. We did not use three (santuario-java-1.5.9, parquet-mr-1.8.0, and parquet-mr-1.9.0) of them because of the lack of corresponding versions on the official website or GitHub, or because the version code cannot be parsed with the Understand tool. Considering the large number of versions of this data set, we refer the readers to Herbold [1] for the details.

TABLE 3
List of metrics in the MA-SZZ-2020 data set

Type	Name	Definition	Tool for measuring metrics
Size Metrics	SLOC (loc in data set)	the non-commentary source lines of code in a class	We used the Perl script developed in previous studies [38, 39] to collect metrics based on the udb database, where the udb database is generated by the commercial software Understand.
	NMIMP	the number of methods implemented in a class	
	NumPara	sum of the number of parameters of the methods implemented in a class	
	NM	the number of methods in a class, both inherited and non-inherited	
	NAIMP	the number of attributes in a class excluding inherited ones	
	NA	the number of attributes in a class including both inherited and non-inherited	
	Stms	the number of declaration and executable statements in the methods of a class	
	Nmpub	number of public methods implemented in a class	
	NMNpub	number of non-public methods implemented in a class	
	NIM	Number of Instance Methods	
	NCM	Number of Class Methods	
	NLM	Number of Local Methods	
	AvgSLOC	Average Source Lines of Code	
Complexity Metrics	CDE	Class Definition Entropy	
	CIE	Class Implementation Entropy	
	WMC	Weighted Method Per Class	
	SDMC	Standard Deviation Method Complexity	
	AvgWMC	Average Weight Method Complexity	
	CCMax	Maximum cyclomatic complexity of a single method of a class	
	NTM	Number of Trivial Methods	
Coupling Metrics	CBO	Coupling Between Object	
	DAC	Data Abstraction Coupling: Type is the number of attributes of other classes.	
	DACquote	Data Abstraction Coupling: Type is the number of other classes.	
	ICP	Information-flow-based Coupling	
	IHICP	Information-flow-based inheritance Coupling	
	NIHICP	Information-flow-based non-inheritance Coupling	
Inheritance Metrics	NOC	Number Of Child Classes	
	NOP	Number Of Parent Classes	
	DIT	Depth of Inheritance Tree	
	AID	Average Inheritance Depth of a class	
	CLD	Class-to-Leaf Depth	
	NOD	Number Of Descendants	
	NOA	Number Of Ancestors	
	NMO	Number of Methods Overridden	
	NMI	Number of Methods Inherited	
	NMA	Number Of Methods Added	
	SIX	Specialization IndeX = NMO * DIT / (NMO + NMA + NMI)	
	PII	Pure Inheritance Index.	
	SPA	static polymorphism in ancestors	
	SPD	static polymorphism in descendants	
	DPA	dynamic polymorphism in ancestors	
	DPD	dynamic polymorphism in descendants	
	SP	static polymorphism in inheritance relations	
	DP	dynamic polymorphism in inheritance relations	

sions (SZZ-based approach described in Section 2.1). After the above steps, the IND-JLMIV+R approach can be expected to have a high accuracy in defect data collection. In this sense, the IND-JLMIV+R-2020 data set collected by IND-JLMIV+R is expected to contain little or even no inconsistent labels.

C.2. Software metrics used in the MA-SZZ-2020

Table 3 describes the size, complexity, coupling, and inheritance metrics in the MA-SZZ-2020 we collected. Note that for IND-JLMIV+R-2020, we use the same 44 code metrics as MA-SZZ-2020 to conduct our experiments, as the number of features in the original IND-JLMIV+R-2020 data set is far more than the number of instances. Prior studies found that when the EPV (Events Per Variable) index [35] is less than a certain value (usually 5 or 10), the performance of a model will be very unstable [36]. Due to the limitation of space, the metrics of other data sets can be found in the data sets in the online appendix[15] or refer to the original literature.

In Table 3, column "Type" represents the type to which each metric belongs to, column "Name" gives the acronym of each metric, column "Definition" provides an informal description of the corresponding metric, and column "Tool for measuring metrics" gives the source of the tool that we measure metrics from. Note that inheritance metrics are indeed a form of coupling metrics. In practice, however, many researchers distinguish inheritance metrics from coupling metrics. Our study follows a metric classification framework similar to that in Briand et al. [37].

Appendix D. Supplementary results

In Sections 5.2 (influence on prediction performance) and 5.3 (influence on model evaluation), "CC vs. NC" and "NC vs. NN", are respectively used to investigate the influence of inconsistent labels in the training set on the model performance and the influence of inconsistent labels in the test set on the model evaluation. In this section, we report the results of more indicators for reference.

D.1. Performance indicator

Performance Indicators. In the defect prediction scenario,

[15] http://github.com/sticeran/InconsistentLabels

TABLE 4
The comparison of prediction performance: CC vs. NC

Learner	Metric	Median CC	Median NC	+/0/–	P-value	Effect size
LR	*precision*	0.23	0.208	78/2/17	2.27E-14	0.697 (L)
	F_1	0.354	0.333	74/1/22	1.77E-10	0.600 (L)
	IFA	0	0	20/59/18	0.64	0.037
NB	*precision*	0.223	0.194	74/1/22	3.19E-11	0.621 (L)
	F_1	0.352	0.306	68/1/28	7.77E-07	0.472 (M)
	IFA	0	0	20/56/21	0.549	0.013
RF	*precision*	0.689	0.654	58/13/26	1.43E-04	0.363 (M)
	F_1	0.729	0.699	55/12/30	0.002	0.286 (S)
	IFA	0	0	12/70/15	0.29	-0.057
MLP	*precision*	0.37	0.357	63/1/33	4.84E-04	0.331 (M)
	F_1	0.498	0.497	60/0/37	0.003	0.275 (S)
	IFA	0	0	29/40/28	0.492	-0.002
ADT	*precision*	0.3	0.256	62/2/33	5.12E-04	0.330 (M)
	F_1	0.4	0.358	64/1/32	1.12E-04	0.369 (M)
	IFA	0	0	14/56/27	0.01	-0.235 (S)
SVM	*precision*	0.217	0.192	73/1/23	9.46E-12	0.635 (L)
	F_1	0.334	0.31	73/0/24	8.27E-09	0.547 (L)
	IFA	2	2	16/43/38	0.002	-0.289 (S)

(1) +/0/–: CC has a larger/the same/a smaller performance value compared with NC
(2) L: large, M: moderate, S: small

TABLE 6
The comparison of prediction performance: NC vs. NN

Learner	Metric	Median NC	Median NN	+/0/–	P-value	Effect size
LR	*precision*	0.246	0.273	19/9/78	5.22E-10	-0.572 (L)
	F_1	0.369	0.371	47/0/59	0.085	-0.167 (S)
	IFA	0	0	2/104/0	0.5	0.137 (S)
NB	*precision*	0.24	0.247	14/5/87	4.96E-11	-0.600 (L)
	F_1	0.34	0.36	40/0/66	0.029	-0.211 (S)
	IFA	0	0	2/102/2	1	0.001
RF	*precision*	0.641	0.64	45/16/45	0.567	0.056
	F_1	0.676	0.632	80/1/25	9.47E-10	0.564 (L)
	IFA	0	0	4/100/2	0.688	0.078
MLP	*precision*	0.384	0.389	25/10/71	5.45E-07	-0.471 (M)
	F_1	0.461	0.472	55/0/51	0.704	0.037
	IFA	0	0	5/98/3	0.367	0.072
ADT	*precision*	0.278	0.294	19/11/76	1.10E-08	-0.531 (L)
	F_1	0.352	0.375	51/0/55	0.88	-0.015
	IFA	0	0	3/100/3	0.812	0.002
SVM	*precision*	0.24	0.261	19/6/81	1.14E-09	-0.562 (L)
	F_1	0.364	0.369	44/0/62	0.028	-0.213 (S)
	IFA	2	2	8/94/4	0.169	0.117 (S)

(1) +/0/–: NC has a larger/the same/a smaller performance value compared with NN
(2) L: large, M: moderate, S: small

TABLE 5
The sum of IFA: CC vs. NC

Learner	Metric	Sum CC	Sum NC
LR	*IFA*	81	90
NB		115	81
RF		64	71
MLP		122	122
ADT		130	254
SVM		632	718

TABLE 7
The sum of IFA: NC vs. NN

Learner	Metric	Sum NC	Sum NN
LR	*IFA*	98	87
NB		120	119
RF		121	121
MLP		132	119
ADT		325	315
SVM		607	579

for each instance in a test set, a defect prediction model outputs whether the instance is defective. Consequently, there are four outcomes: *TP* (the instance is actually defective and is predicted to be defective), *TN* (the instance is actually non-defective and is predicted to be non-defective), *FP* (the instance is actually non-defective and is predicted to be defective), and *FN* (the instance is actually defective and is predicted to be non-defective). In addition to the four performance indicators (*recall*, *pf*, *inspect*, and *MCC*) reported in our paper, we additionally use the following three commonly used performance indicators to measure the performance of a model.

- *Precision*: the proportion of correctly identified defective instances among all predicted defective instances. It is defined as: $precision = TP / (TP + FP)$.
- F_1 [40]: the harmonic mean of precision (i.e. $p = |TP| / (|TP|+|FP|)$) and recall (i.e. $r = |TP| / (|TP|+|FN|)$), i.e. $2 \times p \times r / (p + r)$.
- Initial False Alarms (*IFA*) [41]. *IFA* measures the number of instances until the first true defective instance is found when instances are ranked by their defect-proneness. A low *IFA* value indicates that few non-defective instances are ranked at the top, while a high *IFA* value indicates that developers will spend unnecessary effort on non-defective instances. The intuition behinds this measure is that developers may stop inspecting if they could not get promising results (i.e., find defective instances) within the first few inspected instances [42]. *IFA* is computed as: *k*.

Notably, the literature [43] stated that for class imbalanced data, the precision is an unstable indicator. Therefore, the above three performance indicators are for reference only.

D.2. Supplementary results for Sections 5.2 (influence on prediction performance)

In this section, we report the results of more indicators (*precision*, F_1 and *IFA*) for reference. Table 4 summarizes the results from the comparison of CC and NC. The third column reports for CC and NC the median performance values. The fourth column reports how many times CC has a larger (+), the same (0), and a smaller (−) performance value compared with NC. The fifth column reports the p-value from the Wilcoxon signed-rank test [44] with a Benjamini-Hochberg [45] corrected pvalue: a p-value < 0.05 indicates that CC is significantly better than NC in defect prediction. In this table, a statistically insignificant p-value is shown in gray background. Note that the higher the value of *precision* or F_1 is, the better the model performance is. The lower the value of *IFA* is, the better the model performance is. The last column reports the effect size to indicate whether their difference is practical important. According to [46], the effect size r for the Wilcoxon signed-rank test is computed by $Z/\sqrt{N}$, where Z is from the Wilcoxon signed-rank test and N is the total size of training-test pairs. By convention, the effect size is considered trivial for $|r| < 0.1$, small for $0.1 \le |r| < 0.3$, moderate for $0.3 \le |r| < 0.5$, and large for $|r| \ge 0.5$.

Consequently, as can be seen from Table 4, on all three indicators, CC is significantly better than NC as a whole. Note that for *IFA*, there is no significant difference between CC and NC on four classifiers (LR, NB, RF, and MLP). The reason is that on these classifiers, the value of *IFA* on most training set-test set pairs is 0, so there is no room for significant performance improvement. Table 5 lists the sum of IFAs of CC and NC on 97 training set-test set pairs. As can be seen from Table 5, on all classifiers except SVM, the sum of IFAs of CC or NC is slightly more than 97 (less than 2 on average). Except NB, the sum of IFAs of CC is lower than NC. Therefore, in general, CC is still better than NC.

D.3. Supplementary results for Section 5.3 (influence on model evaluation)

In this section, we report the results of more indicators (*precision*, F_1 and *IFA*) for reference. Table 6 summarizes the results from the comparison of NC and NN. In this table, a statistically insignificant p-value is shown in gray background (a p-value < 0.05 indicates that there is a statically significant difference in the prediction performance between NC and NN). Consequently, as can be seen from Table 6, in most cases, inconsistent labels in a test data set can lead to a considerable evaluation bias on the real performance: either overestimate or underestimate. Note that for *IFA*, there is no significant difference between NC and NN on all six classifiers. The reason is that on these classifiers, the value of *IFA* on most training set-test set pairs is 0, whether for NC or NN. Table 7 lists the sum of IFAs of NC and NN on 106 training set-test set pairs. As can be seen from Table 7, on all classifiers except SVM and ADT, the sum of IFAs of NC or NN is slightly more than 106 (less than 2 on average). Thus, there is no room for significant performance overestimation or underestimation for *IFA*.

References (A-D)

[1] S. Herbold, A. Trautsch, F. Trautsch. Issues with SZZ: an empirical assessment of the state of practice of defect prediction data collection. arXiv preprint arXiv:1911.08938v2, 2020.

[2] M. Jureczko, D. Spinellis. Using object-oriented design metrics to predict software defects. RELCOMEX 2010: 69-81.

[3] M. Jureczko, L. Madeyski. Towards identifying software project clusters with regard to defect prediction. PROMISE 2010, article no. 9: 1–10.

[4] S. Yatish, J. Jiarpakdee, P. Thongtanunam, C. Tantithamthavorn. Mining software defects: should we consider affected releases? ICSE 2019: 654-665.

[5] T. Zimmermann, R. Premraj, A. Zeller. Predicting defects for Eclipse. PROMISE 2007, article no 9: 1-7.

[6] D.A. Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, A.E. Hassan. A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. IEEE Transactions on Software Engineering, 43(7), 2017: 641-657.

[7] G. Rodríguez-Pérez, G. Robles, A. Serebrenik, A. Zaidman, D.M. Germán, J.M. González-Barahona. How bugs are born: A model to identify how bugs are introduced in software components. Empirical Software Engineering, 25(2), 2020: 1294-1340.

[8] D.M. Germán, G. Robles, A.E. Hassan. Change impact graphs: determining the impact of prior code changes. SCAM 2008: 184-193.

[9] G. Rodríguez-Pérez, A. Zaidman, A. Serebrenik, G. Robles, J.M. González-Barahona. What if a bug has a different origin?: Making sense of bugs without an explicit bug introducing change. ESEM 2018: 52:1-52:4.

[10] J. Aranda, G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. ICSE 2009: 298-308.

[11] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, P.T. Devanbu. Fair and balanced?: Bias in bug-fix datasets. ESEC/SIGSOFT FSE 2009: 121-130.

[12] N. Bettenburg, S. Just, A. Schröter, C. Weiß, R. Premraj, T. Zimmermann. Quality of bug reports in Eclipse. ETX 2007: 21-25.

[13] T.H.D. Nguyen, B. Adams, A.E. Hassan. A case study of bias in bug-fix datasets. WCRE 2010: 259-268.

[14] A. Bachmann, C. Bird, F. Rahman, P.T. Devanbu, A. Bernstein. The missing links: bugs and bug-fix commits. SIGSOFT FSE 2010: 97-106.

[15] R. Wu, H. Zhang, S. Kim, S. Cheung. ReLink: Recovering links between bugs and changes. SIGSOFT FSE 2011: 15-25.

[16] A.T. Nguyen, T.T. Nguyen, H.A. Nguyen, T.N. Nguyen. Multi-layered approach for recovering links between bug reports and fixes. SIGSOFT FSE 2012: 63.

[17] G. Mausa, T.G. Grbac, B.D. Basic. Data collection for software defect prediction - an exploratory case study of open source software projects. MIPRO 2015: 463-469.

[18] T.F. Bissyandé, F. Thung, S. Wang, D. Lo, L. Jiang, L. Réveillère. Empirical evaluation of bug linking. CSMR 2013: 89-98.

[19] T. Chen, M. Nagappan, E. Shihab, A.E. Hassan. An empirical study of dormant bugs. MSR 2014: 82-91.

[20] A. Ahluwalia, D. Falessi, M.D. Penta. Snoring: A noise in defect prediction datasets. MSR 2019: 63-67.

[21] K. Herzig, S. Just, A. Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. ICSE 2013: 392-401.

[22] H.A. Nguyen, A.T. Nguyen, T.N. Nguyen. Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization. ISSRE 2013: 138-147.

[23] C.Mills, J.Pantiuchina, E.Parra, G.Bavota, S.Haiduc. Are bug reports enough for text retrieval-based bug localization? ICSME 2018: 381-392.

[24] S. Kim, T. Zimmermann, K. Pan, E.J. Whitehead. Automatic identification of bug-introducing changes. ASE 2006: 81-90.

[25] E.C. Neto, D.A. Costa, U. Kulesza. The impact of refactoring changes on the SZZ algorithm: An empirical study. SANER 2018: 380-390.

[26] M. Wen, R. Wu, Y. Liu, Y. Tian, X. Xie, S. Cheung, Z. Su. Exploring and exploiting the correlations between bug-inducing and bug-fixing commits. ESEC/SIGSOFT FSE 2019: 326-337.

[27] D. Gray, D. Bowes, N. Davey, Y. Sun, B. Christianson. The misuse of the NASA metrics data program data sets for automated software defect prediction. EASE 2011: 96-103.

[28] Z. Sun, J. Li, H. Sun. An empirical study of public data quality problems in cross project defect prediction. arXiv preprint arXiv:1805.10787, 2018.

[29] M. Shepperd, Q. Song, Z. Sun, C. Mair. Data quality: some comments on the NASA software defect datasets. IEEE Transactions on Software Engineering, 39(9), 2013: 1208-1215.

[30] S. Kim, H. Zhang, R. Wu, L. Gong. Dealing with noise in defect prediction. ICSE 2011: 481-490.

[31] C. Tantithamthavorn, S. McIntosh, A.E. Hassan, A. Ihara, K. Matsumoto. The impact of mislabelling on the performance and interpretation of defect prediction models. ICSE 2015: 812-823.

[32] H. Tu, Z. Yu, T. Menzies. Better data labelling with EMBLEM (and how that impacts defect prediction). IEEE Transactions on Software Engineering, 2020, accepted.

[33] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, D. Dig. Accurate and efficient refactoring detection in commit history. ICSE 2018: 483-494.

[34] D. Silva, M.T. Valente. RefDiff: detecting refactorings in version histories. MSR 2017: 269-279.

[35] P. Peduzzi, J. Concato, E. Kemper, T.R. Holford，A.R. Feinstein. A simulation study of the number of events per variable in logistic regression analysis. Journal of clinical epidemiology, 49(12), 1996: 1373-1379.

[36] C. Tantithamthavorn, S. McIntosh, A.E. Hassan, K. Matsumoto. An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. IEEE Transactions on Software Engineering, 43(1), 2017: 1-18.

[37] L.C. Briand, J. Wust, J.W. Daly, D.V. Porter. Exploring the relationships between design measures and software quality in object-oriented systems. Journal of Systems and Software, 51(3), 2000: 245-273.

[38] Y. Zhou, B. Xu, H. Leung, L. Chen. An in-depth study of the potentially confounding effect of class size in fault prediction. ACM Transactions on Software Engineering and Methodology, 23(1), 2014: 1-51.

[39] Y. Zhou, H.K.N Leung, B. Xu. Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness. IEEE Transactions on Software Engineering, 35(5), 2009: 607-623.

[40] J. Han, M. Kamber, J. Pei. Data mining: concepts and techniques, 3rd edition. Morgan Kaufmann 2011, ISBN 978-0123814791.

[41] Q. Huang, X. Xia, D. Lo. Supervised vs Unsupervised Models: A Holistic Look at Effort-Aware Just-in-Time Defect Prediction. ICSME 2017: 159-170.

[42] C. Parnin, A. Orso. Are automated debugging techniques actually helping programmers? ISSTA 2011: 199-209.

[43] T. Menzies, A. Dekhtyar, J.S.D. Stefano, J. Greenwald. Problems with Precision: A Response to “Comments on ‘Data Mining Static Code Attributes to Learn Defect Predictors’”. IEEE Transactions on Software Engineering, 33(9), 2007: 637-640.

[44] M. Hollander, D.A. Wolfe. Nonparametric statistical methods. New York: John Wiley & Sons, (1973).

[45] Y. Benjamini, Y. Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. Journal of the Royal Statistical Society Series B, 57, (1995), pp. 289-300.

[46] C.O. Fritz, P.E. Morris, J.J. Richler. Effect size estimates: current use, calculations, and interpretation. Journal of experimental psychology: General, 141(1), 2012: 2.

Appendix E. To what extent might previous studies be potentially influenced by inconsistent labels?

Our experimental results show that existing multi-version-project defect data sets (such as ECLIPSE-2007 [A1], Metrics-Repo-2010 [A2], JIRA-HA-2019 [A5], JIRA-RA-2019 [A5], and IND-JLMIV+R-2020 [A6]) contain inconsistent labels. In particular, for a defect prediction model, the existence of such inconsistent labels may considerably change its prediction ability, evaluation, and model interpretation. This raises concerns on the reliability of the experimental results or conclusions reported in previous studies that used these data sets. Therefore, the following question naturally arises: how many previous studies might be potentially influenced by these multi-version-project defect data sets? In this section, we answer this question by investigating the number of previous studies that used them as the subject data sets to conduct their experiments.

In order to avoid ambiguity, we separate the two concepts of “number of citations” and “number of literatures that had experimented with the collected data sets”. The “number of citations” refers to the total number of citations by other literatures for the original papers that published the target multi-version-project defect data set. The “number of literatures that had experimented with the collected data sets” refers to the total number of other literatures, which not only cited the original papers of the target multi-version-project defect data set, but also used the target multi-version-project defect data set in their experiments. Generally, “number of citations” cannot be used as a proxy for the frequency of a data set usage, because other literatures may only introduce the methods or ideas of the cited paper. Therefore, we additionally count the “number of literatures that had experimented with the collected data sets” as a proxy for the frequency of a data set usage.

Table 8 summarizes the “number of citations” and “number of literatures that had experimented with the collected data sets” for the existing multi-version-project defect data sets investigated in our study. The first column lists the data sets. The second column lists the original paper(s) publishing each multi-version-project defect data set. The third column reports how many other literatures cite the original literature, i.e., “number of citations” (reported by Google scholar, March 26, 2021). The fourth column reports the total number of other literatures (written in English) that use the corresponding data sets to conduct their experiments, i.e., “number of literatures that had experimented with the collected data sets” (inspected by the first author and confirmed by the seventh author). Note that the Metrics-Repo-2010 data set was first published in [A2]. However, most literature cite [A3] and [A4] as its source. The reason was that [A3] was published in a well-known international conference on Predictive Models in Software Engineering (PROMISE), aiming to share publicly accessible data sets. In particular, Metrics-Repo-2010 was put on the corresponding promise repository website [A4]. Given this situation, we use them (i.e. [A2], [A3], and [A4]) as three sources to count the number of (different) citations. As can be seen, JIRA-RA-2019 (JIRA-HA-2019) and IND-JLMIV+R-2020 data sets were used by few studies (the “number of literatures that had experimented with the collected data sets” are 6 and 5 respectively), as they are two recently published multi-version-project defect data sets. However, ECLIPSE-2007 and Metrics-Repo-2010 data sets were widely used in previous studies (the “number of literatures that had experimented with the collected data sets”

TABLE 8
Literatures potentially influenced by target multi-version-project defect data sets

Defect data sets	Source	Number of citations	Number of literatures that had experimented with the target data sets	List number range of literatures that had experimented with the target data sets
ECLIPSE-2007	[A1]	817	144	[1~144]
Metrics-Repo-2010	[A2, A3, A4]	453	264	[145~408]
JIRA-HA-2019 / JIRA-RA-2019	[A5]	17	6	[409~414]
IND-JLMIV+R-2020	[A6]	9	5	[415~419]

is 144 and 264 respectively). This indicates that inconsistent labels have a potentially wide influence on previous studies. At the end of this section, we lists all the literatures we investigated.

It is important to note that the influence of inconsistent labels on the existing literature may be overestimated, although the "number of literatures that had experimented with the collected data sets" is a more secure proxy for estimating the potential influence of inconsistent labels than the "number of citations". This is because our study does not conduct replication experiments to investigate the specific influence of inconsistent labels on each of the existing literatures (it is a large amount of work that could not be done in our study alone). It is still an open problem to investigate the actual influence of each multi-version-project defect data set with inconsistent labels on the existing literatures, pending an in-depth or extensive empirical study in the future.

The literatures of five target multi-version-project defect data sets

[A1] T. Zimmermann, R. Premraj, A. Zeller. Predicting defects for Eclipse. In Proceedings of the Third International Workshop on Predictor Models in Software Engineering, ser. PROMISE '07. IEEE Computer Society, 2007: 9–.

[A2] M. Jureczko, D. Spinellis. Using object-oriented design metrics to predict software defects. In Models and Methods of System Dependability. Oficyna Wydawnicza Politechniki Wrocławskiej, 2010: 69-81.

[A3] M. Jureczko, L. Madeyski. Towards identifying software project clusters with regard to defect prediction. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering, 2010: 1–10.

[A4] T. Menzies, B. Caglayan, E. Kocaguneli, J. Krall, F. Peters, B. Turhan. The promise repository of empirical software engineering data, June 2012.

[A5] S. Yatish, J. Jiarpakdee, P. Thongtanunam, C. Tantithamthavorn. Mining software defects: should we consider affected releases? ICSE 2019: 654-665.

[A6] S. Herbold, A. Trautsch, F. Trautsch. Issues with SZZ: An empirical assessment of the state of practice of defect prediction data collection. arXiv preprint arXiv:1911.08938v2, 2020.

The list of citations using the ECLIPSE-2007 data set

[1] Moser R, Pedrycz W, Succi G. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction[C]//Proceedings of the 30th international conference on Software engineering. ACM, 2008: 181-190.

[2] Zimmermann T, Nagappan N, Gall H, et al. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process[C]//Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. ACM, 2009: 91-100.

[3] Bird C, Bachmann A, Aune E, et al. Fair and balanced?: bias in bug-fix datasets[C]//Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. ACM, 2009: 121-130.

[4] Nagappan N, Zeller A, Zimmermann T, et al. Change bursts as defect predictors[C]//2010 IEEE 21st International Symposium on Software Reliability Engineering. IEEE, 2010: 309-318.

[5] Mende T, Koschke R. Effort-aware defect prediction models[C]//2010 14th European Conference on Software Maintenance and Reengineering. IEEE, 2010: 107-116.

[6] Tantithamthavorn C, McIntosh S, Hassan A E, et al. Automated parameter optimization of classification techniques for defect prediction models[C]//2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). IEEE, 2016: 321-332.

[7] Li M, Zhang H, Wu R, et al. Sample-based software defect prediction with active and semi-supervised learning[J]. Automated Software Engineering, 2012, 19(2): 201-230.

[8] Zhou Y, Xu B, Leung H. On the ability of complexity metrics to predict fault-prone classes in object-oriented systems[J]. Journal of Systems and Software, 2010, 83(4): 660-674.

[9] Tantithamthavorn C, McIntosh S, Hassan A E, et al. An empirical comparison of model validation techniques for defect prediction models[J]. IEEE Transactions on Software Engineering, 2017, 43(1): 1-18.

[10] Zhang H. An investigation of the relationships between lines of code and defects[C]//2009 IEEE International Conference on Software Maintenance. IEEE, 2009: 274-283.

[11] Khoshgoftaar T M, Gao K, Seliya N. Attribute selection and imbalanced data: Problems in software defect prediction[C]//2010 22nd IEEE International Conference on Tools with Artificial Intelligence. IEEE, 2010, 1: 137-144.

[12] Shihab E, Jiang Z M, Ibrahim W M, et al. Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project[C]//Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. ACM, 2010: 4.

[13] Zhang H. On the distribution of software faults[J]. IEEE Transactions on Software Engineering, 2008, 34(2): 301-302.

[14] Bettenburg N, Hassan A E. Studying the impact of social structures on software quality[C]//2010 IEEE 18th International Conference on Program Comprehension. IEEE, 2010: 124-133.

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

[16] Yang X, Tang K, Yao X. A learning-to-rank approach to software

defect prediction[J]. IEEE Transactions on Reliability, 2015, 64(1): 234-246.
[17] Nguyen T H D, Adams B, Hassan A E. Studying the impact of dependency network measures on software quality[C]//2010 IEEE International Conference on Software Maintenance. IEEE, 2010: 1-10.
[18] Premraj R, Herzig K. Network versus code metrics to predict defects: A replication study[C]//2011 International Symposium on Empirical Software Engineering and Measurement. IEEE, 2011: 215-224.
[19] Wang H, Khoshgoftaar T M, Seliya N. How many software metrics should be selected for defect prediction?[C]//Twenty-Fourth International FLAIRS Conference. 2011.
[20] Kpodjedo S, Ricca F, Galinier P, et al. Design evolution metrics for defect prediction in object oriented systems[J]. Empirical Software Engineering, 2011, 16(1): 141-175.
[21] Rodriguez D, Herraiz I, Harrison R. On software engineering repositories and their open problems[C]//2012 First International Workshop on Realizing AI Synergies in Software Engineering (RAISE). IEEE, 2012: 52-56.
[22] Liu W, Liu S, Gu Q, et al. Empirical studies of a two-stage data preprocessing approach for software fault prediction[J]. IEEE Transactions on Reliability, 2016, 65(1): 38-53.
[23] Khoshgoftaar T M, Gao K, Napolitano A, et al. A comparative study of iterative and non-iterative feature selection techniques for software defect prediction[J]. Information Systems Frontiers, 2014, 16(5): 801-822.
[24] Wang H, Khoshgoftaar T M, Van Hulse J. A comparative study of threshold-based feature selection techniques[C]//2010 IEEE International Conference on Granular Computing. IEEE, 2010: 499-504.
[25] Khoshgoftaar T M, Gao K, Napolitano A. An empirical study of feature ranking techniques for software quality prediction[J]. International journal of software engineering and knowledge engineering, 2012, 22(02): 161-183.
[26] Elish M O, Al-Yafei A H, Al-Mulhem M. Empirical comparison of three metrics suites for fault prediction in packages of object-oriented systems: A case study of Eclipse[J]. Advances in Engineering Software, 2011, 42(10): 852-859.
[27] Zhou Y, Xu B, Leung H, et al. An in-depth study of the potentially confounding effect of class size in fault prediction[J]. ACM Transactions on Software Engineering and Methodology (TOSEM), 2014, 23(1): 10.
[28] Caglayan B, Bener A, Koch S. Merits of using repository metrics in defect prediction for open source projects[C]//2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development. IEEE, 2009: 31-36.
[29] Wang H, Khoshgoftaar T M, Napolitano A. Software measurement data reduction using ensemble techniques[J]. Neurocomputing, 2012, 92: 124-132.
[30] Li W, Huang Z, Li Q. Three-way decisions based software defect prediction[J]. Knowledge-Based Systems, 2016, 91: 263-274.
[31] Shihab E. An exploration of challenges limiting pragmatic software defect prediction[D]. , 2012.
[32] Zeller A, Zimmermann T, Bird C. Failure is a four-letter word: a parody in empirical research[C]//Proceedings of the 7th International Conference on Predictive Models in Software Engineering. ACM, 2011: 5.
[33] Zimmerman T, Nagappan N, Herzig K, et al. An empirical study on the relation between dependency neighborhoods and failures[C]//2011 Fourth IEEE International Conference on Software Testing, Verification and Validation. IEEE, 2011: 347-356.
[34] Krishnan S, Strasburg C, Lutz R R, et al. Are change metrics good predictors for an evolving software product line?[C]//Proceedings of the 7th International Conference on Predictive Models in Software Engineering. ACM, 2011: 7.
[35] Ibrahim W M, Bettenburg N, Adams B, et al. On the relationship between comment update practices and software bugs[J]. Journal of Systems and Software, 2012, 85(10): 2293-2304.
[36] Moser R, Pedrycz W, Succi G. Analysis of the reliability of a subset of change metrics for defect prediction[C]//Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement. ACM, 2008: 309-311.
[37] Lu H, Kocaguneli E, Cukic B. Defect prediction between software versions with active learning and dimensionality reduction[C]//2014 IEEE 25th International Symposium on Software Reliability Engineering. IEEE, 2014: 312-322.
[38] Oyetoyan T D, Cruzes D S, Conradi R. A study of cyclic dependencies on defect profile of software components[J]. Journal of Systems and Software, 2013, 86(12): 3162-3182.
[39] Tantithamthavorn C, McIntosh S, Hassan A E, et al. The impact of automated parameter optimization on defect prediction models[J]. IEEE Transactions on Software Engineering, 2018:1-1.
[40] Rathore S S, Kumar S. Towards an ensemble based system for predicting the number of software faults[J]. Expert Systems with Applications, 2017, 82: 357-382.
[41] Wahyudin D, Ramler R, Biffl S. A framework for defect prediction in specific software project contexts[C]//IFIP Central and East European Conference on Software Engineering Techniques. Springer, Berlin, Heidelberg, 2008: 261-274.
[42] Gao K, Khoshgoftaar T M. Software Defect Prediction for High-Dimensional and Class-Imbalanced Data[C]//SEKE. 2011: 89-94.
[43] Wang H, Khoshgoftaar T M, Van Hulse J, et al. Metric selection for software defect prediction[J]. International Journal of Software Engineering and Knowledge Engineering, 2011, 21(02): 237-257.
[44] Pipitone J, Easterbrook S. Assessing climate model software quality: a defect density analysis of three models[J]. Geoscientific Model Development, 2012, 5(4): 1009-1022.
[45] Liu Y, Cheah W P, Kim B K, et al. Predict software failure-prone by learning Bayesian network[J]. International Journal of Advanced Science and Technology, 2008, 1(1): 35-42.
[46] Krishnan S, Strasburg C, Lutz R R, et al. Predicting failure-proneness in an evolving software product line[J]. Information and Software Technology, 2013, 55(8): 1479-1495.
[47] Tan X, Peng X, Pan S, et al. Assessing software quality by program clustering and defect prediction[C]//2011 18th working conference on Reverse Engineering. IEEE, 2011: 244-248.
[48] Chen J, Liu S, Liu W, et al. A two-stage data preprocessing approach for software fault prediction[C]//2014 Eighth International Conference on Software Security and Reliability (SERE). IEEE, 2014: 20-29.
[49] Marinescu C. Are the classes that use exceptions defect prone?[C]//Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution. ACM, 2011: 56-60.
[50] Gao K, Khoshgoftaar T M, Napolitano A. Impact of data sampling on stability of feature selection for software measurement data[C]//2011 IEEE 23rd International Conference on Tools with Artificial Intelligence. IEEE, 2011: 1004-1011.
[51] Rathore S S, Kumar S. Linear and non-linear heterogeneous ensemble methods to predict the number of faults in software systems[J]. Knowledge-Based Systems, 2017, 119: 232-256.
[52] Marinescu R, Marinescu C. Are the clients of flawed classes (also) defect prone?[C]//2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation. IEEE, 2011: 65-74.

[53] Tantithamthavorn C, Hassan A E. An experience report on defect modelling in practice: Pitfalls and challenges[C]//Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice. ACM, 2018: 286-295.
[54] Wang H, Khoshgoftaar T M, Napolitano A. An Empirical Study of Software Metrics Selection Using Support Vector Machine[C]//SEKE. 2011: 83-88.
[55] Gao K, Khoshgoftaar T M, Napolitano A. Combining Feature Subset Selection and Data Sampling for Coping with Highly Imbalanced Software Data[C]//SEKE. 2015: 439-444.
[56] Wang H, Khoshgoftaar T M, Wald R. Measuring robustness of feature selection techniques on software engineering datasets[C]//2011 IEEE International Conference on Information Reuse & Integration. IEEE, 2011: 309-314.
[57] Plosch R, Gruber H, Hentschel A, et al. On the relation between external software quality and static code analysis[C]//2008 32nd Annual IEEE Software Engineering Workshop. IEEE, 2008: 169-174.
[58] Marinescu C. Should we beware the exceptions? an empirical study on the eclipse project[C]//2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. IEEE, 2013: 250-257.
[59] Tantithamthavorn C, Hassan A E, Matsumoto K. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models[J]. IEEE Transactions on Software Engineering, 2018.
[60] Liu W, Liu S, Gu Q, et al. Fecs: A cluster based feature selection method for software fault prediction with noises[C]//2015 IEEE 39th Annual Computer Software and Applications Conference. IEEE, 2015, 2: 276-281.
[61] Kuo C S, Huang C Y. A study of applying the bounded generalized pareto distribution to the analysis of software fault distribution[C]//2010 IEEE International Conference on Industrial Engineering and Engineering Management. IEEE, 2010: 611-615.
[62] Devine T, Goseva-Popstojanova K, Krishnan S, et al. Assessment and cross-product prediction of software product line quality: accounting for reuse across products, over multiple releases[J]. Automated Software Engineering, 2016, 23(2): 253-302.
[63] Zhang H, Cheung S C. A cost-effectiveness criterion for applying software defect prediction models[C]//Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, 2013: 643-646.
[64] Zhang H, Wu R. Sampling program quality[C]//2010 IEEE International Conference on Software Maintenance. IEEE, 2010: 1-10.
[65] Wang H, Khoshgoftaar T M, Liang Q. A study of software metric selection techniques: stability analysis and defect prediction model performance[J]. International journal on artificial intelligence tools, 2013, 22(05): 1360010.
[66] Pipitone J. Software quality in climate modelling[J]. Master's thesis, Department of Computer Science, University of Toronto, 2010.
[67] Steff M, Russo B. Measuring architectural change for defect estimation and localization[C]//2011 International Symposium on Empirical Software Engineering and Measurement. IEEE, 2011: 225-234.
[68] Choudhary G R, Kumar S, Kumar K, et al. Empirical analysis of change metrics for software fault prediction[J]. Computers & Electrical Engineering, 2018, 67: 15-24.
[69] Wang H, Khoshgoftaar T M, Napolitano A. An empirical investigation on wrapper-based feature selection for predicting software quality[J]. International Journal of Software Engineering and Knowledge Engineering, 2015, 25(01): 93-114.
[70] Wang H, Khoshgoftaar T M, Seliya N. On the stability of feature selection methods in software quality prediction: an empirical investigation[J]. International Journal of Software Engineering and Knowledge Engineering, 2015, 25(09n10): 1467-1490.
[71] Wang H, Khoshgoftaar T M, Napolitano A. Stability of filter-and wrapper-based software metric selection techniques[C]//Proceedings of the 2014 IEEE 15th International Conference on Information Reuse and Integration (IEEE IRI 2014). IEEE, 2014: 309-314.
[72] Wang H, Khoshgoftaar T M, Napolitano A. An empirical study on the stability of feature selection for imbalanced software engineering data[C]//2012 11th International Conference on Machine Learning and Applications. IEEE, 2012, 1: 317-323.
[73] Khoshgoftaar T M, Gao K, Van Hulse J. Feature selection for highly imbalanced software measurement data[M]//Recent trends in information reuse and integration. Springer, Vienna, 2012: 167-189.
[74] GAO K, KHOSHGOFTAAR T M, WALD R. The use of under-and oversampling within ensemble feature selection and classification for software quality prediction[J]. International Journal of Reliability, Quality and Safety Engineering, 2014, 21(01): 1450004.
[75] Zhang H, Tan H B K, Marchesi M. The distribution of program sizes and its implications: An eclipse case study[C]//1st International Symposium on Emerging Trends in Software Metrics. 2009: 1-10.
[76] Khoshgoftaar T M, Gao K, Napolitano A. A Comparative Study of Different Strategies for Predicting Software Quality[C]//SEKE. 2011, 2011: 65-70.
[77] Mizuno O, Hata H. An empirical comparison of fault-prone module detection approaches: Complexity metrics and text feature metrics[C]//2010 IEEE 34th Annual Computer Software and Applications Conference. IEEE, 2010: 248-249.
[78] Wang H, Khoshgoftaar T M, Wald R, et al. A comparative study on the stability of software metric selection techniques[C]//2012 11th International Conference on Machine Learning and Applications. IEEE, 2012, 2: 301-307.
[79] Guo Y, Würsch M, Giger E, et al. An empirical validation of the benefits of adhering to the law of demeter[C]//2011 18th Working Conference on Reverse Engineering. IEEE, 2011: 239-243.
[80] Mizuno O, Hata H. An integrated approach to detect fault-prone modules using complexity and text feature metrics[M]//Advances in Computer Science and Information Technology. Springer, Berlin, Heidelberg, 2010: 457-468.
[81] Oyetoyan T D, Cruzes D S, Conradi R. Can refactoring cyclic dependent components reduce defect-proneness?[C]//2013 IEEE International Conference on Software Maintenance. IEEE, 2013: 420-423.
[82] Khoshgoftaar T M, Gao K, Napolitano A. An empirical study of predictive modeling techniques of software quality[C]//International Conference on Bio-Inspired Models of Network, Information, and Computing Systems. Springer, Berlin, Heidelberg, 2010: 288-302.
[83] Jiarpakdee J, Tantithamthavorn C, Treude C. AutoSpearman: Automatically Mitigating Correlated Software Metrics for Interpreting Defect Models[C]//2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2018: 92-103.
[84] Wang H, Khoshgoftaar T M. Measuring stability of threshold-based feature selection techniques[C]//2011 IEEE 23rd International Conference on Tools with Artificial Intelligence. IEEE, 2011: 986-993.
[85] Gao K, Khoshgoftaar T M, Napolitano A. The use of ensemble-based data preprocessing techniques for software defect prediction[J]. International Journal of Software Engineering and Knowledge Engineering, 2014, 24(09): 1229-1253.

[86] Khoshgoftaar T M, Gao K, Van Hulse J. A novel feature selection technique for highly imbalanced data[C]//2010 IEEE International Conference on Information Reuse & Integration. IEEE, 2010: 80-85.

[87] Khoshgoftaar T M, Gao K, Napolitano A. Improving software quality estimation by combining feature selection strategies with sampled ensemble learning[C]//Proceedings of the 2014 IEEE 15th International Conference on Information Reuse and Integration (IEEE IRI 2014). IEEE, 2014: 428-433.

[88] Khoshgoftaar T M, Gao K, Napolitano A. Exploring an iterative feature selection technique for highly imbalanced data sets[C]//2012 IEEE 13th International Conference on Information Reuse & Integration (IRI). IEEE, 2012: 101-108.

[89] Mizuno O, Hata H. A metric to detect fault-prone software modules using text filtering[J]. International Journal of Reliability and Safety, 2013, 7(1): 17-31.

[90] Altidor W, Khoshgoftaar T M, Gao K. Wrapper-based feature ranking techniques for determining relevance of software engineering metrics[J]. International Journal of Reliability, Quality and Safety Engineering, 2010, 17(05): 425-464.

[91] Kumari D, Rajnish K. Comparing Efficiency of Software Fault Prediction Models Developed Through Binary and Multinomial Logistic Regression Techniques[M]//Information Systems Design and Intelligent Applications. Springer, New Delhi, 2015: 187-197.

[92] Wang H, Khoshgoftaar T M, Wald R. Measuring stability of feature selection techniques on real-world software datasets[M]//Information Reuse and Integration in Academia and Industry. Springer, Vienna, 2013: 113-132.

[93] Oyetoyan T D, Cruzes D S, Conradi R. Transition and defect patterns of components in dependency cycles during software evolution[C]//2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE). IEEE, 2014: 283-292.

[94] Jia H, Shu F, Yang Y, et al. Data transformation and attribute subset selection: Do they help make differences in software failure prediction?[C]//2009 IEEE International Conference on Software Maintenance. IEEE, 2009: 519-522.

[95] Wang H, Khoshgoftaaar T M, Liang Q. Stability and classification performance of feature selection techniques[C]//2011 10th International Conference on Machine Learning and Applications and Workshops. IEEE, 2011, 1: 151-156.

[96] Mahmood Z, Bowes D, Hall T, et al. Reproducibility and replicability of software defect prediction studies[J]. Information and Software Technology, 2018, 99: 148-163.

[97] Mizuno O. On effects of tokens in source code to accuracy of fault-prone module prediction[C]//2013 International Computer Science and Engineering Conference (ICSEC). IEEE, 2013: 103-108.

[98] Gao K, Khoshgoftaar T M, Napolitano A. An empirical investigation of combining filter-based feature subset selection and data sampling for software defect prediction[J]. International Journal of Reliability, Quality and Safety Engineering, 2015, 22(06): 1550027.

[99] Zimmermann T, Nagappan N, Williams L, et al. An empirical study of the factors relating field failures and dependencies[C]//IEEE Fourth International Conference on Software Testing, Verification and Validation. 2011: 347-356.

[100] Gao K, Khoshgoftaar T M, Napolitano A. Aggregating data sampling with feature subset selection to address skewed software defect data[J]. International Journal of Software Engineering and Knowledge Engineering, 2015, 25(09n10): 1531-1550.

[101] Mizuno O, Hirata Y. Fault-prone module prediction using contents of comment lines[C]//Proc. of International Workshop on Empirical Software Engineering in Practice 2010 (IWESEP2010). 2010, 12: 39-44.

[102] Krishnan S. Evidence-based defect assessment and prediction for software product lines[J]. 2013.

[103] Gao K, Khoshgoftaar T M. Assessments of Feature Selection Techniques with Respect to Data Sampling for Highly Imbalanced Software Measurement Data[J]. International Journal of Reliability, Quality and Safety Engineering, 2015, 22(02): 1550010.

[104] Muthukumaran K, Dasgupta A, Abhidnya S, et al. On the effectiveness of cost sensitive neural networks for software defect prediction[C]//International Conference on Soft Computing and Pattern Recognition. Springer, Cham, 2016: 557-570.

[105] Han W, Lung C H, Ajila S A. Empirical investigation of code and process metrics for defect prediction[C]//2016 IEEE Second International Conference on Multimedia Big Data (BigMM). IEEE, 2016: 436-439.

[106] Gao K, Khoshgoftaar T M, Napolitano A. Investigating two approaches for adding feature ranking to sampled ensemble learning for software quality estimation[J]. International Journal of Software Engineering and Knowledge Engineering, 2015, 25(01): 115-146.

[107] Kumari D, Rajnish K. Investigating the Effect of Object-oriented Metrics on Fault Proneness Using Empirical Analysis[J]. International Journal of Software Engineering and Its Applications, 2015, 9(2): 171-188.

[108] Maheshwari S, Agarwal S. Three-way decision based Defect Prediction for Object Oriented Software[C]//Proceedings of the International Conference on Advances in Information Communication Technology & Computing. ACM, 2016: 4.

[109] Kumari D, Rajnish K. Evaluating The Impact Of Binary And Multinomial LR In Developing Software Fault Prediction Model Using OO-Metrics[J]. Global Journal of Pure and Applied Mathematics, 2015, 11(1): 437-462.

[110] Bettenburg N. Studying the Impact of Developer Communication on the Quality and Evolution of a Software System[D]. , 2014.

[111] Shriram C K, Muthukumaran K, Bhanu Murthy N L. Empirical Study on the Distribution of Bugs in Software Systems[J]. International Journal of Software Engineering and Knowledge Engineering, 2018, 28(01): 97-122.

[112] Muthukumaran K, Srinivas S, Malapati A, et al. Software Defect Prediction Using Augmented Bayesian Networks[C]//International Conference on Soft Computing and Pattern Recognition. Springer, Cham, 2016: 279-293.

[113] Li H Y, Li M, Zhou Z H. Towards one reusable model for various software defect mining tasks[J].

[114] Jiarpakdee J, Tantithamthavorn C, Treude C. Artefact: An R Implementation of the AutoSpearman Function[C]//2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2018: 711-711.

[115] Wang H, Khoshgoftaar T M, Napolitano A. Choosing the Best Classification Performance Metric for Wrapper-based Software Metric Selection for Defect Prediction[C]//SEKE. 2014: 540-545.

[116] Buchari M A, Mardiyanto S, Hendradjaya B. Implementation of Chaotic Gaussian Particle Swarm Optimization for Optimize Learning-to-Rank Software Defect Prediction Model Construction[C]//Journal of Physics: Conference Series. IOP Publishing, 2018, 978(1): 012079.

[117] Nagappan M, Murphy B, Vouk M. Which code construct metrics are symptoms of post release failures?[C]//Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics. ACM, 2011: 65-68.

[118] Herraiz Tabernero I, Shihab E, Nguyen T H D, et al. Impact of Installation Counts on Perceived Quality: A Case Study on Debian[J]. 2011.

[119]Verma R, Gupta A. An approach of attribute selection for reducing false alarms[C]//2012 CSI Sixth International Conference on Software Engineering (CONSEG). IEEE, 2012: 1-7.
[120]Rathore S S, Kumar S. An Approach for the Prediction of Number of Software Faults Based on the Dynamic Selection of Learning Techniques[J]. IEEE Transactions on Reliability, 2018 (99): 1-21.
[121]Zeller A, Zimmermann T, Bird C. Failure is a four-letter word[J]. 2011.
[122]Hashem A. A COMPREHENSIVE EMPIRICAL VALIDATION OF PACKAGE-LEVEL METRICS FOR OO SYSTEMS[D]. King Fahd University of Petroleum and Minerals, 2010.
[123]Khoshgoftaar T M, Gao K, Chen Y, et al. Comparing Feature Selection Techniques for Software Quality Estimation Using Data-Sampling-Based Boosting Algorithms[J]. International Journal of Reliability, Quality and Safety Engineering, 2015, 22(03): 1550013.
[124]Wang H, Khoshgoftaar T M, Wald R, et al. A novel dataset-similarity-aware approach for evaluating stability of software metric selection techniques[C]//2012 IEEE 13th International Conference on Information Reuse & Integration (IRI). IEEE, 2012: 1-8.
[125]Jiarpakdee J, Tantithamthavorn C, Treude C. The impact of automated feature selection techniques on the interpretation of defect models[J]. Empirical Software Engineering, 2020: 1-49.
[126]Goseva-Popstojanova K, Ahmad M, Alshehri Y. Software fault proneness prediction with group lasso regression: On factors that affect classification performance[C]//2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC). IEEE, 2019, 2: 336-343.
[127]Xu Z, Ye S, Zhang T, et al. MVSE: Effort-Aware Heterogeneous Defect Prediction via Multiple-View Spectral Embedding[C]//2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS). IEEE, 2019: 10-17.
[128]Huo X, Li M. On cost-effective software defect prediction: Classification or ranking?[J]. Neurocomputing, 2019, 363: 339-350.
[129]Pham V, Lokan C, Kasmarik K. A Better Set of Object-Oriented Design Metrics for Within-Project Defect Prediction[M]//Proceedings of the Evaluation and Assessment in Software Engineering. 2020: 230-239.
[130]Rahman A. Software Defect Prediction Using Rich Contextualized Language Use Vectors[D]. , 2019.
[131]Zheng W, Mo S, Jin X, et al. Software Defect Prediction Model Based on Improved Deep Forest and AutoEncoder by Forest[C]//SEKE. 2019: 419-540.
[132]Yang X. Evaluating Software Metrics for Sorting Software Modules in Order of Defect Count[C]//ICSOFT. 2019: 94-105.
[133]Li H Y, Li M, Zhou Z H. Towards one reusable model for various software defect mining tasks[C]//Pacific-Asia Conference on Knowledge Discovery and Data Mining. Springer, Cham, 2019: 212-224.
[134]Nair P R. Optimizing bug prediction in software testing using Super Learner[D]. Dublin, National College of Ireland, 2019.
[135]Bassuday K, Ahmed M. Fault Prediction in Android Systems through AI[J]. 2019.
[136]Tosun Misirli A, Murphy B, Zimmermann T, et al. An explanatory analysis on eclipse beta-release bugs through in-process metrics[C]//Proceedings of the 8th international workshop on Software quality. ACM, 2011: 26-33.
[137]Arshad A, Riaz S, Jiao L, et al. The empirical study of semi-supervised deep fuzzy C-mean clustering for software fault prediction[J]. IEEE Access, 2018, 6: 47047-47061.
[138]Kidwell B R. MiSFIT: Mining Software Fault Information and Types[J]. 2015.
[139]Shaikh M, Lee K S, Lee C G. Assessing the Bug-Prediction with Re-Usability Based Package Organization for Object Oriented Software Systems[J]. IEICE TRANSACTIONS on Information and Systems, 2017, 100(1): 107-117.
[140]Ferenc R, Tóth Z, Ladányi G, et al. A public unified bug dataset for java and its assessment regarding metrics and bug prediction[J]. Software Quality Journal, 2020: 1-60.
[141]Ferenc R, Siket I, Hegedűs P, et al. Employing Partial Least Squares Regression with Discriminant Analysis for Bug Prediction[J]. arXiv preprint arXiv:2011.01214, 2020.
[142]Wu Y, Yao J, Chang S, et al. LIMCR: Less-Informative Majorities Cleaning Rule Based on Naïve Bayes for Imbalance Learning in Software Defect Prediction[J]. Applied Sciences, 2020, 10(23): 8324.
[143]Rajbahadur G. UNDERSTANDING THE IMPACT OF EXPERIMENTAL DESIGN CHOICES ON MACHINE LEARNING CLASSIFIERS IN SOFTWARE ANALYTICS[D].
[144]Pham V, Lokan C, Kasmarik K. A Better Set of Object-Oriented Design Metrics for Within-Project Defect Prediction[M]//Proceedings of the Evaluation and Assessment in Software Engineering. 2020: 230-239.

The list of citations using the Metrics-Repo-2010 data set

[145]He Z, Shu F, Yang Y, et al. An investigation on the feasibility of cross-project defect prediction[J]. Automated Software Engineering, 2012, 19(2): 167-199.
[146]Peters F, Menzies T, Marcus A. Better cross company defect prediction[C]//Proceedings of the 10th Working Conference on Mining Software Repositories. IEEE Press, 2013: 409-418.
[147]He P, Li B, Liu X, et al. An empirical study on software defect prediction with a simplified metric set[J]. Information and Software Technology, 2015, 59: 170-190.
[148]Madeyski L, Jureczko M. Which process metrics can significantly improve defect prediction models? An empirical study[J]. Software Quality Journal, 2015, 23(3): 393-422.
[149]Turhan B, Mısırlı A T, Bener A. Empirical evaluation of the effects of mixed project data on learning defect predictors[J]. Information and Software Technology, 2013, 55(6): 1101-1118.
[150]Chen L, Fang B, Shang Z, et al. Negative samples reduction in cross-company software defects prediction[J]. Information and Software Technology, 2015, 62: 67-77.
[151]Jureczko M. Significance of different software metrics in defect prediction[J]. Software Engineering: An International Journal, 2011, 1(1): 86-95.
[152]Bennin K E, Keung J, Phannachitta P, et al. Mahakil: Diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction[J]. IEEE Transactions on Software Engineering, 2018, 44(6): 534-550.
[153]Ryu D, Jang J I, Baik J. A transfer cost-sensitive boosting approach for cross-project defect prediction[J]. Software Quality Journal, 2017, 25(1): 235-272.
[154]Shatnawi R. Deriving metrics thresholds using log transformation[J]. Journal of Software: Evolution and Process, 2015, 27(2): 95-113.
[155]Alenezi M, Magel K. Empirical evaluation of a new coupling metric: Combining structural and semantic coupling[J]. International Journal of Computers and Applications, 2014, 36(1): 34-44.
[156]Hosseini S, Turhan B, Mäntylä M. A benchmark study on the effectiveness of search-based data selection and feature selection for cross project defect prediction[J]. Information and Software Technology, 2018, 95: 296-312.
[157]Yu L. Using negative binomial regression analysis to predict software faults: A study of apache ant[J]. 2012.
[158]Mizuno O, Hirata Y. A cross-project evaluation of text-based

fault-prone module prediction[C]//2014 6th International Workshop on Empirical Software Engineering in Practice. IEEE, 2014: 43-48.
[159] Kaur A, Kaur K. Performance analysis of ensemble learning for predicting defects in open source software[C]//2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI). IEEE, 2014: 219-225.
[160] Hosseini S, Turhan B, Mäntylä M. Search based training data selection for cross project defect prediction[C]//Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering. ACM, 2016: 3.
[161] Lumpe M, Vasa R, Menzies T, et al. Learning better inspection optimization policies[J]. International Journal of Software Engineering and Knowledge Engineering, 2012, 22(05): 621-644.
[162] Shatnawi R. The application of ROC analysis in threshold identification, data imbalance and metrics selection for software fault prediction[J]. Innovations in Systems and Software Engineering, 2017, 13(2-3): 201-217.
[163] Alenezi M, Zarour M. Modularity measurement and evolution in object-oriented open-source projects[C]//Proceedings of the The International Conference on Engineering & MIS 2015. ACM, 2015: 16.
[164] Poon W N, Bennin K E, Huang J, et al. Cross-project defect prediction using a credibility theory based naive bayes classifier[C]//2017 IEEE International Conference on Software Quality, Reliability and Security (QRS). IEEE, 2017: 434-441.
[165] Xu Z, Liu J, Luo X, et al. Cross-version defect prediction via hybrid active learning with kernel principal component analysis[C]//2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2018: 209-220.
[166] Xu Z, Liu J, Luo X, et al. Software defect prediction based on kernel PCA and weighted extreme learning machine[J]. Information and Software Technology, 2019, 106: 182-200.
[167] Bennin K E, Keung J, Monden A. Impact of the distribution parameter of data sampling approaches on software defect prediction models[C]//2017 24th Asia-Pacific Software Engineering Conference (APSEC). IEEE, 2017: 630-635.
[168] Bennin K E, Keung J W, Monden A. On the relative value of data resampling approaches for software defect prediction[J]. Empirical Software Engineering, 2019, 24(2): 602-636.
[169] Mizuno O. On effects of tokens in source code to accuracy of fault-prone module prediction[C]//2013 International Computer Science and Engineering Conference (ICSEC). IEEE, 2013: 103-108.
[170] Bluemke I, Stepień A. Experiment on defect prediction[C]//International Conference on Dependability and Complex Systems. Springer, Cham, 2015: 25-34.
[171] Gupta S, Gupta D L. Fault Prediction using Metric Threshold Value of Object Oriented Systems[J]. International Journal of Engineering Science, 2017, 13629.
[172] Shatnawi R. Synergies and conflicts among software quality attributes and bug fixes[J]. International Journal of Information Systems and Change Management, 2017, 9(1): 3-21.
[173] Yang Y. Software Defect Prediction Model Research for Network and Cloud Software Development[C]//2017 5th International Conference on Mechatronics, Materials, Chemistry and Computer Engineering (ICMMCCE 2017). Atlantis Press, 2017.
[174] Shatnawi R. Improving Software Fault Prediction With Threshold Values[C]//2018 26th International Conference on Software, Telecommunications and Computer Networks (SoftCOM). IEEE, 2018: 1-6.
[175] Yuand Y T, Zhange T. Software Defect Prediction Based on Kernel PCA and Weighted Extreme Learning Machine[J].
[176] Bispo A, Prudêncio R, Véras D. Instance Selection and Class Balancing Techniques for Cross Project Defect Prediction[C]//2018 7th Brazilian Conference on Intelligent Systems (BRACIS). IEEE, 2018: 552-557.
[177] Ryu D, Baik J. A Comparative Study on Similarity Measure Techniques for Cross-Project Defect Prediction[J]. KIPS Transactions on Software and Data Engineering, 2018, 7(6): 205-220.
[178] Watanabe T, Monden A, Yücel Z, et al. Cross-validation-based association rule prioritization metric for software defect characterization[J]. IEICE TRANSACTIONS on Information and Systems, 2018, 101(9): 2269-2278.
[179] Malhotra R, Bansal A J. Fault prediction considering threshold effects of object-oriented metrics[J]. Expert Systems, 2015, 32(2): 203-219.
[180] Bennin K E, Keung J, Monden A, et al. The significant effects of data sampling approaches on software defect prioritization and classification[C]//Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. IEEE Press, 2017: 364-373.
[181] Alenezi M, Banitaan S, Obeidat Q. Fault-proneness of open source systems: An empirical analysis[J]. Synapse, 2014, 1: 256.
[182] Alenezi M, Zarour M. Does software structures quality improve over software evolution? Evidences from open-source projects[J]. International Journal of Computer Science and Information Security, 2016, 14: 61.
[183] He P, He Y, Yu L, et al. An Improved Method for Cross-Project Defect Prediction by Simplifying Training Data[J]. Mathematical Problems in Engineering, 2018, 2018.
[184] Jureczko M, Magott J. QualitySpy: a framework for monitoring software development processes[J]. Journal of Theoretical and Applied Computer Science, 2012, 6(1): 35-45.
[185] Jureczko M, Madeyski L. Cross–project defect prediction with respect to code ownership model: An empirical study[J]. e-Informatica Software Engineering Journal, 2015, 9(1).
[186] Shatnawi R. Empirical study of fault prediction for open-source systems using the Chidamber and Kemerer metrics[J]. IET software, 2013, 8(3): 113-119.
[187] Haouari A T, Souici-Meslati L, Atil F, et al. Empirical comparison and evaluation of Artificial Immune Systems in inter-release software fault prediction[J]. Applied Soft Computing, 2020: 106686.
[188] Tsoukalas D, Kehagias D, Siavvas M, et al. Technical debt forecasting: An empirical study on open-source repositories[J]. Journal of Systems and Software, 2020: 110777.
[189] Shao Y, Liu B, Wang S, et al. Software defect prediction based on correlation weighted class association rule mining[J]. Knowledge-Based Systems, 2020: 105742.
[190] Bal P R, Kumar S. WR-ELM: Weighted Regularization Extreme Learning Machine for Imbalance Learning in Software Fault Prediction[J]. IEEE Transactions on Reliability, 2020.
[191] Tong H, Liu B, Wang S, et al. Transfer-learning oriented class imbalance learning for cross-project defect prediction[J]. arXiv preprint arXiv:1901.08429, 2019.
[192] Alqmase M, Alshayeb M, Ghouti L. Threshold Extraction Framework for Software Metrics[J]. Journal of Computer Science and Technology, 2019, 34(5): 1063-1078.
[193] Xu Z, Li S, Xu J, et al. LDFR: Learning deep feature representation for software defect prediction[J]. Journal of Systems and Software, 2019, 158: 110402.
[194] Hosseini S. ASCIENTIAE RERUM[J].
[195] Sun Z, Li J, Sun H. An empirical study of public data quality problems in cross project defect prediction[J]. arXiv preprint arXiv:1805.10787, 2018.
[196] He P, Li B, Ma Y. Towards cross-project defect prediction with imbalanced feature sets[J]. arXiv preprint arXiv:1411.4228, 2014.

[197] Tantithamthavorn C, McIntosh S, Hassan A E, et al. Automated parameter optimization of classification techniques for defect prediction models[C]//Proceedings of the 38th International Conference on Software Engineering. 2016: 321-332.

[198] Tantithamthavorn C, McIntosh S, Hassan A E, et al. An empirical comparison of model validation techniques for defect prediction models[J]. IEEE Transactions on Software Engineering, 2016, 43(1): 1-18.

[199] Xia X, Lo D, Pan S J, et al. Hydra: Massively compositional model for cross-project defect prediction[J]. IEEE Transactions on software Engineering, 2016, 42(10): 977-998.

[200] Zhang F, Zheng Q, Zou Y, et al. Cross-project defect prediction using a connectivity-based unsupervised classifier[C]//2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). IEEE, 2016: 309-320.

[201] Herbold S. Training data selection for cross-project defect prediction[C]//Proceedings of the 9th international conference on predictive models in software engineering. 2013: 1-10.

[202] Li J, He P, Zhu J, et al. Software defect prediction via convolutional neural network[C]//2017 IEEE International Conference on Software Quality, Reliability and Security (QRS). IEEE, 2017: 318-328.

[203] He Z, Peters F, Menzies T, et al. Learning from open-source projects: An empirical study on defect prediction[C]//2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. IEEE, 2013: 45-54.

[204] Herbold S, Trautsch A, Grabowski J. A comparative study to benchmark cross-project defect prediction approaches[J]. IEEE Transactions on Software Engineering, 2017, 44(9): 811-833.

[205] Tantithamthavorn C, McIntosh S, Hassan A E, et al. The impact of automated parameter optimization on defect prediction models[J]. IEEE Transactions on Software Engineering, 2018, 45(7): 683-711.

[206] Hosseini S, Turhan B, Gunarathna D. A systematic literature review and meta-analysis on cross project defect prediction[J]. IEEE Transactions on Software Engineering, 2017, 45(2): 111-147.

[207] Peters F, Menzies T, Layman L. LACE2: Better privacy-preserving data sharing for cross project defect prediction[C]//2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. IEEE, 2015, 1: 801-811.

[208] Bowes D, Hall T, Petrić J. Software defect prediction: do different classifiers find the same defects?[J]. Software Quality Journal, 2018, 26(2): 525-552.

[209] Krishna R, Menzies T, Fu W. Too much automation? The bellwether effect and its implications for transfer learning[C]//Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. 2016: 122-131.

[210] Ryu D, Baik J. Effective multi-objective naïve Bayes learning for cross-project defect prediction[J]. Applied Soft Computing, 2016, 49: 1062-1077.

[211] Palomba F, Zanoni M, Fontana F A, et al. Toward a smell-aware bug prediction model[J]. IEEE Transactions on Software Engineering, 2017, 45(2): 194-218.

[212] Arar Ö F, Ayan K. Deriving thresholds of software metrics to predict faults on open source software: Replicated case studies[J]. Expert Systems with Applications, 2016, 61: 106-121.

[213] Herbold S, Trautsch A, Grabowski J. Global vs. local models for cross-project defect prediction[J]. Empirical software engineering, 2017, 22(4): 1866-1902.

[214] Zhou Y, Yang Y, Lu H, et al. How far we have progressed in the journey? an examination of cross-project defect prediction[J]. ACM Transactions on Software Engineering and Methodology (TOSEM), 2018, 27(1): 1-51.

[215] Krishna R, Menzies T. Bellwethers: A baseline method for transfer learning[J]. IEEE Transactions on Software Engineering, 2018, 45(11): 1081-1105.

[216] Kawata K, Amasaki S, Yokogawa T. Improving relevancy filter methods for cross-project defect prediction[C]//2015 3rd International Conference on Applied Computing and Information Technology/2nd International Conference on Computational Science and Intelligence. IEEE, 2015: 2-7.

[217] Ma W, Chen L, Yang Y, et al. Empirical analysis of network measures for effort-aware fault-proneness prediction[J]. Information and Software Technology, 2016, 69: 50-70.

[218] Zhang F, Keivanloo I, Zou Y. Data transformation in cross-project defect prediction[J]. Empirical Software Engineering, 2017, 22(6): 3186-3218.

[219] Amasaki S, Kawata K, Yokogawa T. Improving cross-project defect prediction methods with data simplification[C]//2015 41st Euromicro Conference on Software Engineering and Advanced Applications. IEEE, 2015: 96-103.

[220] Yan M, Fang Y, Lo D, et al. File-level defect prediction: Unsupervised vs. supervised models[C]//2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, 2017: 344-353.

[221] Turabieh H, Mafarja M, Li X. Iterated feature selection algorithms with layered recurrent neural network for software fault prediction[J]. Expert systems with applications, 2019, 122: 27-42.

[222] Boucher A, Badri M. Software metrics thresholds calculation techniques to predict fault-proneness: An empirical comparison[J]. Information and Software Technology, 2018, 96: 38-67.

[223] Huang J, Keung J W, Sarro F, et al. Cross-validation based K nearest neighbor imputation for software quality datasets: An empirical study[J]. Journal of Systems and Software, 2017, 132: 226-252.

[224] Li Z, Jing X Y, Zhu X, et al. Heterogeneous defect prediction through multiple kernel learning and ensemble learning[C]//2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2017: 91-102.

[225] Liu C, Yang D, Xia X, et al. A two-phase transfer learning model for cross-project defect prediction[J]. Information and Software Technology, 2019, 107: 125-136.

[226] Chen X, Zhang D, Zhao Y, et al. Software defect number prediction: Unsupervised vs supervised methods[J]. Information and Software Technology, 2019, 106: 161-181.

[227] Krishna R, Menzies T, Layman L. Less is more: Minimizing code reorganization using XTREE[J]. Information and Software Technology, 2017, 88: 53-66.

[228] Chen X, Shen Y, Cui Z, et al. Applying feature selection to software defect prediction using multi-objective optimization[C]//2017 IEEE 41st annual computer software and applications conference (COMPSAC). IEEE, 2017, 2: 54-59.

[229] Herbold S. CrossPare: A tool for benchmarking cross-project defect predictions[C]//2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW). IEEE, 2015: 90-96.

[230] Boucher A, Badri M. Using software metrics thresholds to predict fault-prone classes in object-oriented software[C]//2016 4th Intl Conf on Applied Computing and Information Technology/3rd Intl Conf on Computational Science/Intelligence and Applied Informatics/1st Intl Conf on Big Data, Cloud Computing, Data Science & Engineering (ACIT-CSII-BCD). IEEE, 2016: 169-176.

[231] Xiaoxing Yang, Wushao Wen. Ridge and Lasso Regression Models for Cross-Version Defect Prediction. IEEE Trans. Reliab. 67(3): 885-896 (2018).

[232] Zhang Y, Lo D, Xia X, et al. Combined classifier for cross-project

defect prediction: an extended empirical study[J]. Frontiers of Computer Science, 2018, 12(2): 280-296.

[233] Herbold S. A systematic mapping study on cross-project defect prediction[J]. arXiv preprint arXiv:1705.06429, 2017.

[234] He P, Li B, Zhang D, et al. Simplification of training data for cross-project defect prediction[J]. arXiv preprint arXiv:1405.0773, 2014.

[235] Li Y, Huang Z, Wang Y, et al. Evaluating data filter on cross-project defect prediction: Comparison and improvements[J]. IEEE Access, 2017, 5: 25646-25656.

[236] Zhou Xu, Shuai Li, Yutian Tang, Xiapu Luo, Tao Zhang, Jin Liu, Jun Xu. Cross version defect prediction with representative data via sparse subset selection. ICPC 2018: 132-143.

[237] Ni C, Chen X, Wu F, et al. An empirical study on pareto based multi-objective feature selection for software defect prediction[J]. Journal of Systems and Software, 2019, 152: 215-238.

[238] Kondo M, Bezemer C P, Kamei Y, et al. The impact of feature reduction techniques on defect prediction models[J]. Empirical Software Engineering, 2019, 24(4): 1925-1963.

[239] Hussain S, Keung J, Khan A A, et al. Performance evaluation of ensemble methods for software fault prediction: An experiment[C]//Proceedings of the ASWEC 2015 24th Australasian Software Engineering Conference. 2015: 91-95.

[240] Boucher A, Badri M. Predicting fault-prone classes in object-oriented software: an adaptation of an unsupervised hybrid SOM algorithm[C]//2017 IEEE International Conference on Software Quality, Reliability and Security (QRS). IEEE, 2017: 306-317.

[241] Hussain S, Keung J, Khan A A, et al. Detection of fault-prone classes using logistic regression based object-oriented metrics thresholds[C]//2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C). IEEE, 2016: 93-100.

[242] Porto F, Minku L, Mendes E, et al. A systematic study of cross-project defect prediction with meta-learning[J]. arXiv preprint arXiv:1802.06025, 2018.

[243] Zhang X, Ben K, Zeng J. Cross-entropy: A new metric for software defect prediction[C]//2018 IEEE International Conference on Software Quality, Reliability and Security (QRS). IEEE, 2018: 111-122.

[244] Herbold S. Comments on ScottKnottESD in response to" An empirical comparison of model validation techniques for defect prediction models"[J]. IEEE Transactions on Software Engineering, 2017, 43(11): 1091-1094.

[245] Chen D, Fu W, Krishna R, et al. Applications of psychological science for actionable analytics[C]//Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2018: 456-467.

[246] Tang H, Lan T, Hao D, et al. Enhancing defect prediction with static defect analysis[C]//Proceedings of the 7th Asia-Pacific Symposium on Internetware. 2015: 43-51.

[247] Li Z, Jing X Y, Zhu X. Heterogeneous fault prediction with cost-sensitive domain adaptation[J]. Software Testing, Verification and Reliability, 2018, 28(2): e1658.

[248] Liu Y, Li Y, Guo J, et al. Connecting software metrics across versions to predict defects[C]//2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2018: 232-243.

[249] Qiu S, Lu L, Jiang S. Multiple-components weights model for cross-project software defect prediction[J]. IET Software, 2018, 12(4): 345-355.

[250] Yang J, Qian H. Defect prediction on unlabeled datasets by using unsupervised clustering[C]//2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS). IEEE, 2016: 465-472.

[251] Fan G, Diao X, Yu H, et al. Software Defect Prediction via Attention-Based Recurrent Neural Network[J]. Scientific Programming, 2019, 2019.

[252] Zhou T, Sun X, Xia X, et al. Improving defect prediction with deep forest[J]. Information and Software Technology, 2019, 114: 204-216.

[253] Prateek S, Pasala A, Aracena L M. Evaluating performance of network metrics for bug prediction in software[C]//2013 20th Asia-Pacific Software Engineering Conference (APSEC). IEEE, 2013, 1: 124-131.

[254] Sousuke Amasaki. On Applicability of Cross-project Defect Prediction Method for Multi-Versions Projects. PROMISE 2017: 93-96.

[255] Morasca S, Lavazza L. Risk-averse slope-based thresholds: Definition and empirical evaluation[J]. Information and Software Technology, 2017, 89: 37-63.

[256] Mutlu B, Sezer E A, Akcayol M A. End-to-End hierarchical fuzzy inference solution[C]//2018 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE). IEEE, 2018: 1-9.

[257] Sousuke Amasaki. Cross-Version Defect Prediction using Cross-Project Defect Prediction Approaches: Does it work? PROMISE 2018: 32-41.

[258] Goyal R, Chandra P, Singh Y. Fuzzy inferencing to identify degree of interaction in the development of fault prediction models[J]. Journal of King Saud University-Computer and Information Sciences, 2017, 29(1): 93-102.

[259] Chen J, Hu K, Yang Y, et al. Collective transfer learning for defect prediction[J]. Neurocomputing, 2019.

[260] Li Z, Jing X Y, Zhu X, et al. Heterogeneous defect prediction with two-stage ensemble learning[J]. Automated Software Engineering, 2019, 26(3): 599-651.

[261] Porto F R, Simao A. Feature Subset Selection and Instance Filtering for Cross-project Defect Prediction-Classification and Ranking[J]. CLEI Electron. J., 2016, 19(3): 4.

[262] Herbold S. Benchmarking cross-project defect prediction approaches with costs metrics[J]. arXiv preprint arXiv:1801.04107, 2018.

[263] Pan C, Lu M, Xu B, et al. An Improved CNN Model for Within-Project Software Defect Prediction[J]. Applied Sciences, 2019, 9(10): 2138.

[264] Wen W, Zhang B, Gu X, et al. An empirical study on combining source selection and transfer learning for cross-project defect prediction[C]//2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF). IEEE, 2019: 29-38.

[265] Catolino G, Palomba F, Fontana F A, et al. Improving change prediction models with code smell-related information[J]. Empirical Software Engineering, 2020, 25(1): 49-95.

[266] Cheikhi L, Abran A. An Analysis of the PROMISE and ISBSG Software Engineering Data Repositories[J]. Int. Journal of Computers and Technology, 2014, 13(5).

[267] Shriram C K, Muthukumaran K, Bhanu Murthy N L. Empirical study on the distribution of bugs in software systems[J]. International Journal of Software Engineering and Knowledge Engineering, 2018, 28(01): 97-122.

[268] Zhang X, Ben K, Zeng J. Using Cross-Entropy Value of Code for Better Defect Prediction[J]. International Journal of Performability Engineering, 2018, 14(9).

[269] Alenezi M, Abunadi I. Quality of open source systems from product metrics perspective[J]. arXiv preprint arXiv:1511.03194, 2015.

[270] Sohan M F, Kabir M A, Jabiullah M I, et al. Revisiting the class

imbalance issue in software defect prediction[C]//2019 International Conference on Electrical, Computer and Communication Engineering (ECCE). IEEE, 2019: 1-6.

[271] Li K, Xiang Z, Chen T, et al. Understanding the Automated Parameter Optimization on Transfer Learning for CPDP: An Empirical Study[J]. arXiv preprint arXiv:2002.03148, 2020.

[272] Yucalar F, Ozcift A, Borandag E, et al. Multiple-classifiers in software quality engineering: Combining predictors to improve software fault prediction ability[J]. Engineering Science and Technology, an International Journal, 2020, 23(4): 938-950.

[273] Chen D, Chen X, Li H, et al. Deepcpdp: Deep learning based cross-project defect prediction[J]. IEEE Access, 2019, 7: 184832-184848.

[274] Krishna R, Menzies T. Actionable= Cluster+ Contrast?[C]//2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW). IEEE, 2015: 14-17.

[275] Hosseini S, Turhan B. An exploratory study of search based training data selection for cross project defect prediction[C]//2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE, 2018: 244-251.

[276] Muthukumaran K, Dasgupta A, Abhidnya S, et al. On the effectiveness of cost sensitive neural networks for software defect prediction[C]//International Conference on Soft Computing and Pattern Recognition. Springer, Cham, 2016: 557-570.

[277] Zhang D, Chen X, Cui Z, et al. Software defect prediction model sharing under differential privacy[C]//2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI). IEEE, 2018: 1547-1554.

[278] Chen X, Mu Y, Qu Y, et al. Do different cross-project defect prediction methods identify the same defective modules?[J]. Journal of Software: Evolution and Process, 2020, 32(5): e2234.

[279] Kaur I, Bajpai N. An Empirical Study on Fault Prediction using Token-Based Approach[C]//Proceedings of the International Conference on Advances in Information Communication Technology & Computing. 2016: 1-7.

[280] Huo X, Yang Y, Li M, et al. Learning Semantic Features for Software Defect Prediction by Code Comments Embedding[C]//2018 IEEE International Conference on Data Mining (ICDM). IEEE, 2018: 1049-1054.

[281] Ghosh S, Rana A, Kansal V. A Hybrid Nonlinear Manifold Detection Approach for Software Defect Prediction[C]//2018 7th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO). IEEE, 2018: 453-459.

[282] Rosli M M, Tempero E, Luxton-Reilly A. What is in our datasets? Describing a structure of datasets[C]//Proceedings of the Australasian Computer Science Week Multiconference. 2016: 1-10.

[283] Wu Y, Huang S, Ji H. An Information Flow-based Feature Selection Method for Cross-Project Defect Prediction[J]. International Journal of Performability Engineering, 2018, 14(6).

[284] Di Nucci D, De Lucia A. The role of meta-learners in the adaptive selection of classifiers[C]//2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE). IEEE, 2018: 7-12.

[285] Tumar I, Hassouneh Y, Turabieh H, et al. Enhanced binary moth flame optimization as a feature selection algorithm to predict software fault prediction[J]. IEEE Access, 2020, 8: 8041-8055.

[286] Malhotra R. An extensive analysis of search-based techniques for predicting defective classes[J]. Computers & Electrical Engineering, 2018, 71: 611-626.

[287] Chen X, Zhang D, Cui Z Q, et al. Dp-share: Privacy-preserving software defect prediction model sharing through differential privacy[J]. Journal of Computer Science and Technology, 2019, 34(5): 1020-1038.

[288] Sohan M F, Jabiullah M I, Rahman S S M M, et al. Assessing the Effect of Imbalanced Learning on Cross-project Software Defect Prediction[C]//2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT). IEEE, 2019: 1-6.

[289] Sundström A. Investigation into predicting unit test failure using syntactic source code features[J]. 2018.

[290] Aziz S R, Khan T, Nadeem A. Experimental Validation of Inheritance Metrics' Impact on Software Fault Prediction[J]. IEEE Access, 2019, 7: 85262-85275.

[291] Ghosh S, Rana A, Kansal V. Evaluating the Impact of Sampling-Based Nonlinear Manifold Detection Model on Software Defect Prediction Problem[M]//Smart Intelligent Computing and Applications. Springer, Singapore, 2020: 141-152.

[292] Krishna R, Menzies T. Learning actionable analytics from multiple software projects[J]. Empirical Software Engineering, 2020: 1-33.

[293] Yang Y, Yang J, Qian H. Defect prediction by using cluster ensembles[C]//2018 Tenth International Conference on Advanced Computational Intelligence (ICACI). IEEE, 2018: 631-636.

[294] Jiarpakdee J, Tantithamthavorn C, Treude C. The impact of automated feature selection techniques on the interpretation of defect models[J]. Empirical Software Engineering, 2020: 1-49.

[295] Raukas H. Some approaches for software defect prediction[D]. Bachelor's Thesis (9ECTS), Institute of Computer Science Computer Science Curriculum, University of TARTU, 2017.

[296] Kaur K. Statistical Comparison of Machine Learning Techniques for Predicting Software Maintainability and Defects[J]. Guru Gobind Singh Indraprastha University, 2016.

[297] Okutan A. Use of Source Code Similarity Metrics in Software Defect Prediction[J]. arXiv preprint arXiv:1808.10033, 2018.

[298] Rizwan M, Nadeem A, Sindhu M A. Empirical Evaluation of Coupling Metrics in Software Fault Prediction[C]//2020 17th International Bhurban Conference on Applied Sciences and Technology (IBCAST). IEEE, 2020: 434-440.

[299] Li K, Xiang Z, Chen T, et al. BiLO-CPDP: Bi-Level Programming for Automated Model Discovery in Cross-Project Defect Prediction[J]. arXiv preprint arXiv:2008.13489, 2020.

[300] Chen X, Mu Y, Ni C, et al. Revisiting Heterogeneous Defect Prediction: How Far Are We?[J]. arXiv preprint arXiv:1908.06560, 2019.

[301] Peng K, Menzies T. How to Improve AI Tools (by Adding in SE Knowledge): Experiments with the TimeLIME Defect Reduction Tool[J]. arXiv preprint arXiv:2003.06887, 2020.

[302] Yuan Z, Chen X, Cui Z, et al. ALTRA: Cross-Project Software Defect Prediction via Active Learning and Tradaboost[J]. IEEE Access, 2020, 8: 30037-30049.

[303] Li K, Xiang Z, Chen T, et al. Understanding the Automated Parameter Optimization on Transfer Learning for Cross-Project Defect Prediction: An Empirical Study[J].

[304] Sohan M F, Kabir M A, Rahman M, et al. Prevalence of Machine Learning Techniques in Software Defect Prediction[C]//International Conference on Cyber Security and Computer Science. Springer, Cham, 2020: 257-269.

[305] Krishnaa R, Menziesa T, Laymanb L. Recommendations for Intelligent Code Reorganization[J]. CoRR, 2016.

[306] Webster A. A Comparison of Transfer Learning Algorithms for Defect and Vulnerability Detection[R]. 2017.

[307] Amasaki S. Cross-version defect prediction: use historical data, cross-project data, or both?[J]. Empirical Software Engineering, 2020: 1-23.

[308]Deng J, Lu L, Qiu S, et al. A Suitable AST Node Granularity and Multi-Kernel Transfer Convolutional Neural Network for Cross-Project Defect Prediction[J]. IEEE Access, 2020, 8: 66647-66661.
[309]Ouellet A, Badri M. Empirical Analysis of Object-Oriented Metrics and Centrality Measures for Predicting Fault-Prone Classes in Object-Oriented Software[C]//International Conference on the Quality of Information and Communications Technology. Springer, Cham, 2019: 129-143.
[310]Qu Y, Liu T, Chi J, et al. node2defect: using network embedding to improve software defect prediction[C]//2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2018: 844-849.
[311]Ha D A, Chen T H, Yuan S M. Unsupervised methods for Software Defect Prediction[C]//Proceedings of the Tenth International Symposium on Information and Communication Technology. 2019: 49-55.
[312]Moudache S, Badri M. Software Fault Prediction Based on Fault Probability and Impact[C]//2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA). IEEE, 2019: 1178-1185.
[313]Ren J H, Liu F. Predicting Software Defects Using Self-Organizing Data Mining[J]. IEEE Access, 2019, 7: 122796-122810.
[314]Ahmed M R, Ali M A, Ahmed N, et al. The Impact of Software Fault Prediction in Real-World Application: An Automated Approach for Software Engineering[C]//Proceedings of 2020 the 6th International Conference on Computing and Data Engineering. 2020: 247-251.
[315]Bangash A A, Sahar H, Hindle A, et al. On the Time-Based Conclusion Stability of Software Defect Prediction Models[J]. arXiv preprint arXiv: 1911.06348v3, 2019.
[316]Sohan M F, Kabir M A, Rahman M, et al. Training Data Selection Using Ensemble Dataset Approach for Software Defect Prediction[C]//International Conference on Cyber Security and Computer Science. Springer, Cham, 2020: 243-256.
[317]Krishna Prasad R. Learning Actionable Analytics in Software Engineering[J]. 2019.
[318]Peng K, Menzies T. Defect Reduction Planning (using TimeLIME)[J]. arXiv preprint arXiv:2006.07416, 2020.
[319]Kumar S, Rathore S S. Evaluation of Techniques for Binary Class Classification[M]//Software Fault Prediction. Springer, Singapore, 2018: 39-57.
[320]Krishna R, Menzies T. From Prediction to Planning: Improving Software Quality with BELLTREE[J].
[321]Li H, Li X, Chen X, et al. Cross-project Defect Prediction via ASTToken2Vec and BLSTM-based Neural Network[C]//2019 International Joint Conference on Neural Networks (IJCNN). IEEE, 2019: 1-8.
[322]Zhang J, Wu J, Chen C, et al. CDS: A Cross–Version Software Defect Prediction Model With Data Selection[J]. IEEE Access, 2020, 8: 110059-110072.
[323]Kumar K, Jayadev Gyani D, Narsimha G. Software Defect Prediction using Ant Colony Optimization[J]. International Journal of Applied Engineering Research, 2018, 13(19): 14291-14297.
[324]Morasca S, Lavazza L. On the assessment of software defect prediction models via ROC curves[J]. Empirical Software Engineering, 2020: 1-43.
[325]Zhang Q, Wu B. Software Defect Prediction via Transformer[C]//2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC). IEEE, 2020, 1: 874-879.
[326]Hussain S, Khan A A, Bennin K E. Empirical investigation of fault predictors in context of class membership probability estimation[C]//Proceedings of the 31st Annual ACM Symposium on Applied Computing. 2016: 1550-1553.
[327]Gao H, Lu M, Pan C, et al. Empirical Study: Are Complex Network Features Suitable for Cross-Version Software Defect Prediction?[C]//2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS). IEEE, 2019: 1-5.
[328]Mutlu B, Sezer E A, Akcayol M A. Automatic Rule Generation of Fuzzy Systems: A Comparative Assessment on Software Defect Prediction[C]//2018 3rd International Conference on Computer Science and Engineering (UBMK). IEEE, 2018: 209-214.
[329]De Lucia A, Salza P. Parallel Genetic Algorithms in the Cloud[J]. 2017.
[330]Catal C, Erdogan M, Isik C. Software Defect Prediction in the Cloud[J].
[331]Srivastava K. Cohesion Based Testability Model: A New Perspective[J].
[332]Fujiwara T, Mizuno O, Leelaprute P. Fault-Prone Byte-Code Detection Using Text Classifier[C]//International Conference on Product-Focused Software Process Improvement. Springer, Cham, 2015: 415-430.
[333]Fan G, Diao X, Yu H, et al. Deep Semantic Feature Learning with Embedded Static Metrics for Software Defect Prediction[C]//2019 26th Asia-Pacific Software Engineering Conference (APSEC). IEEE, 2019: 244-251.
[334]AGRAWAL A. CROSS PROJECT DEFECT PREDICTION[D]. , 2018.
[335]Sara E, Laila C, Ali I. The Impact of SMOTE and Grid Search on Maintainability Prediction Models[C]//2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA). IEEE, 2019: 1-8.
[336]Chen H, Jing X Y, Li Z, et al. An Empirical Study on Heterogeneous Defect Prediction Approaches[J]. IEEE Transactions on Software Engineering, 2020.
[337]Ryu D, Baik J. Effective Harmony Search-Based Optimization of Cost-Sensitive Boosting for Improving the Performance of Cross-Project Defect Prediction[J]. KIPS Transactions on Software and Data Engineering, 2018, 7(3): 77-90.
[338]Rai P, Kumar S, Verma D K. Prediction of Software Effort Using Design Metrics: An Empirical Investigation[M]//Social Networking and Computational Intelligence. Springer, Singapore, 2020: 627-637.
[339]Bangash A A, Sahar H, Hindle A, et al. On the time-based conclusion stability of cross-project defect prediction models[J]. Empirical Software Engineering, 2020: 1-38.
[340]Yao Z, Song J, Liu Y, et al. Research on Cross-version Software Defect Prediction Based on Evolutionary Information[C]//IOP Conference Series: Materials Science and Engineering. IOP Publishing, 2019, 563(5): 052092.
[341]Kabir M A, Keung J W, Bennin K E, et al. A Drift Propensity Detection Technique to Improve the Performance for Cross-Version Software Defect Prediction[C]//2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC). IEEE, 2020: 882-891.
[342]Muthukumaran K, Murthy N L B, Janani P S. Empirical Study on the Distribution of Object-Oriented Metrics in Software Systems[C]//International Conference on Information and Software Technologies. Springer, Cham, 2019: 299-317.
[343]Peters F. LACE: Supporting Privacy-Preserving Data Sharing in Transfer Defect Learning[J]. 2014.
[344]Li J, Jing X Y, Wu F, et al. A Cost-Sensitive Shared Hidden Layer Autoencoder for Cross-Project Defect Prediction[C]//Chinese Conference on Pattern Recognition and Computer Vision (PRCV). Springer, Cham, 2019: 491-502.
[345]Di Nucci D. Methods and tools for focusing and prioritizing the testing effort[C]//2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2018: 722-726.

[346] Sun Y, Jing X Y, Wu F, et al. Adversarial Learning for Cross-Project Semi-Supervised Defect Prediction[J]. IEEE Access, 2020, 8: 32674-32687.
[347] Rathore S S, Kumar S. Homogeneous Ensemble Methods for the Prediction of Number of Faults[M]//Fault Prediction Modeling for the Prediction of Number of Software Faults. Springer, Singapore, 2019: 31-45.
[348] Tahir A, Bennin K E, MacDonell S G, et al. Revisiting the size effect in software fault prediction models[C]//Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. 2018: 1-10.
[349] Gong L, Jiang S, Jiang L. Conditional Domain Adversarial Adaptation for Heterogeneous Defect Prediction[J]. IEEE Access, 2020, 8: 150738-150749.
[350] Jiang K, Zhang Y, Wu H, et al. Heterogeneous defect prediction based on transfer learning to handle extreme imbalance[J]. Applied Sciences, 2020, 10(1): 396.
[351] Kaen E, Algarni A. Feature Selection Approach for Improving the Accuracy of Software Bug Prediction[J].
[352] Huo X, Li M. On cost-effective software defect prediction: Classification or ranking?[J]. Neurocomputing, 2019, 363: 339-350.
[353] Ren J, Liu F. A Novel Approach for Software Defect prediction Based on the Power Law Function[J]. Applied Sciences, 2020, 10(5): 1892.
[354] Maruf O M. The impact of parameter optimization of ensemble learning on defect prediction[J]. Computer Science Journal of Moldova, 2019, 79(1): 85-128.
[355] Xiaoxing Yang, Xin Li, Wushao Wen, Jianmin Su. An Investigation of Ensemble Approaches to Cross-Version Defect Prediction. SEKE 2019: 437-556.
[356] Peters F, Menzies T, Gong L, et al. Balancing privacy and utility in cross-company defect prediction[J]. IEEE Transactions on Software Engineering, 2013, 39(8): 1054-1068.
[357] Sarro F, Di Martino S, Ferrucci F, et al. A further analysis on the use of genetic algorithm to configure support vector machines for inter-release fault prediction[C]//Proceedings of the 27th annual ACM symposium on applied computing. 2012: 1215-1220.
[358] Rathore S S, Kumar S. Linear and non-linear heterogeneous ensemble methods to predict the number of faults in software systems[J]. Knowledge-Based Systems, 2017, 119: 232-256.
[359] Stuckman J, Walden J, Scandariato R. The effect of dimensionality reduction on software vulnerability prediction models[J]. IEEE Transactions on Reliability, 2016, 66(1): 17-37.
[360] Erturk E, Sezer E A. Iterative software fault prediction with a hybrid approach[J]. Applied Soft Computing, 2016, 49: 1020-1033.
[361] Foucault M, Falleri J R, Blanc X. Code ownership in open-source software[C]//Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering. 2014: 1-9.
[362] Gupta D L, Saxena K. Software bug prediction using object-oriented metrics[J]. Sādhanā, 2017, 42(5): 655-669.
[363] Shukla S, Radhakrishnan T, Muthukumaran K, et al. Multi-objective cross-version defect prediction[J]. Soft Computing, 2018, 22(6): 1959-1980.
[364] Xuan X, Lo D, Xia X, et al. Evaluating defect prediction approaches using a massive set of metrics: An empirical study[C]//Proceedings of the 30th Annual ACM Symposium on Applied Computing. 2015: 1644-1647.
[365] Qing H, Biwen L, Beijun S, et al. Cross-project software defect prediction using feature-based transfer learning[C]//Proceedings of the 7th Asia-Pacific Symposium on Internetware. 2015: 74-82.
[366] Ferenc R, Tóth Z, Ladányi G, et al. A public unified bug dataset for Java[C]//Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering. 2018: 12-21.
[367] Kaur A, Kaur K. Value and applicability of academic projects defect datasets in cross-project software defect prediction[C]//2016 2nd International Conference on Computational Intelligence and Networks (CINE). IEEE, 2016: 154-159.
[368] Gong L, Jiang S, Bo L, et al. A novel class-imbalance learning approach for both within-project and cross-project defect prediction[J]. IEEE Transactions on Reliability, 2019, 69(1): 40-54.
[369] Wang F, Huang J, Ma Y. A Top-k Learning to Rank Approach to Cross-Project Software Defect Prediction[C]//2018 25th Asia-Pacific Software Engineering Conference (APSEC). IEEE, 2018: 335-344.
[370] Du Y, Zhang L, Shi J, et al. Feature-grouping-based two steps feature selection algorithm in software defect prediction[C]//Proceedings of the 2nd International Conference on Advances in Image Processing. 2018: 173-178.
[371] Qiu S, Lu L, Cai Z, et al. Cross-Project Defect Prediction via Transferable Deep Learning-Generated and Handcrafted Features[C]//SEKE. 2019: 431-552.
[372] Goel L, Gupta S. Cross Projects Defect Prediction Modeling[M]//Data Visualization and Knowledge Engineering. Springer, Cham, 2020: 1-21.
[373] Cai Z, Lu L, Qiu S. An abstract syntax tree encoding method for cross-project defect prediction[J]. IEEE Access, 2019, 7: 170844-170853.
[374] Yu Q, Jiang S, Qian J, et al. Process metrics for software defect prediction in object-oriented programs[J]. IET Software, 2020.
[375] Geremia S, Tamburri D A. Varying defect prediction approaches during project evolution: A preliminary investigation[C]//2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE). IEEE, 2018: 1-6.
[376] Yang Z, Qian H. Automated Parameter Tuning of Artificial Neural Networks for Software Defect Prediction[C]//Proceedings of the 2nd International Conference on Advances in Image Processing. 2018: 203-209.
[377] Lu H. Semi-supervised and Active Learning Models for Software Fault Prediction[J]. 2015.
[378] Elahi E, Kanwal S, Asif A N. A new Ensemble approach for Software Fault Prediction[C]//2020 17th International Bhurban Conference on Applied Sciences and Technology (IBCAST). IEEE, 2020: 407-412.
[379] Cui C, Liu B, Wang S. Isolation Forest Filter to Simplify Training Data for Cross-Project Defect Prediction[C]//2019 Prognostics and System Health Management Conference (PHM-Qingdao). IEEE, 2019: 1-6.
[380] Rosli M. A Framework for Understanding and Evaluating the Quality of Data Sets in Empirical Software Engineering[D]. ResearchSpace@ Auckland, 2018.
[381] Kumar L, Sureka A. Analyzing fault prediction usefulness from cost perspective using source code metrics[C]//2017 Tenth International Conference on Contemporary Computing (IC3). IEEE, 2017: 1-7.
[382] Choeikiwong T, Vateekul P. Two Stage Model to Detect and Rank Software Defects on Imbalanced and Scarcity Data Sets[J]. IAENG International Journal of Computer Science, 2016, 43(3).
[383] Xu Z, Li L, Yan M, et al. A comprehensive comparative study of clustering-based unsupervised defect prediction models[J]. Journal of Systems and Software, 2021, 172: 110862.
[384] Sun Z, Li J, Sun H, et al. CFPS: Collaborative filtering based source projects selection for cross-project defect prediction[J]. Applied Soft Computing, 2021, 99: 106940.
[385] Esteves G, Figueiredo E, Veloso A, et al. Understanding machine

learning software defect predictions[J]. Automated Software Engineering, 2020, 27(3): 369-392.
[386] Shatnawi R. Comparison of threshold identification techniques for object-oriented software metrics[J]. IET Software, 2020, 14(6): 727-738.
[387] Mohamed F A, Salama C R, Yousef A H, et al. A Universal Model for Defective Classes Prediction Using Different Object-Oriented Metrics Suites[C]//2020 2nd Novel Intelligent and Leading Emerging Sciences Conference (NILES). IEEE, 2020: 65-70.
[388] Jin C. Cross-project software defect prediction based on domain adaptation learning and optimization[J]. Expert Systems with Applications, 2021, 171: 114637.
[389] Ferenc R, Tóth Z, Ladányi G, et al. A public unified bug dataset for java and its assessment regarding metrics and bug prediction[J]. Software Quality Journal, 2020: 1-60.
[390] Chen X, Mu Y, Liu K, et al. Revisiting heterogeneous defect prediction methods: How far are we?[J]. Information and Software Technology, 2021, 130: 106441.
[391] Pecorelli F, Di Nucci D. Adaptive selection of classifiers for bug prediction: A large-scale empirical analysis of its performances and a benchmark study[J]. Science of Computer Programming, 2021, 205: 102611.
[392] Eken B, Palma F, Ayşe B, et al. An empirical study on the effect of community smells on bug prediction[J]. Software Quality Journal, 2021, 29(1): 159-194.
[393] Alkhaeir T, Walter B. The effect of code smells on the relationship between design patterns and defects[J]. IEEE Access, 2020.
[394] Hassouneh Y, Turabieh H, Thaher T, et al. Boosted Whale Optimization Algorithm With Natural Selection Operators for Software Fault Prediction[J]. IEEE Access, 2021, 9: 14239-14258.
[395] Eivazpour Z, Keyvanpour M R. CSSG: A cost-sensitive stacked generalization approach for software defect prediction[J]. Software Testing, Verification and Reliability, 2021: e1761.
[396] Niu L, Wan J, Wang H, et al. Cost-sensitive Dictionary Learning for Software Defect Prediction[J]. Neural Processing Letters, 2020, 52(3): 2415-2449.
[397] Zhou X, Lu L. Defect Prediction via LSTM Based on Sequence and Tree Structure[C]//2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS). IEEE, 2020: 366-373.
[398] Zou Q, Lu L, Qiu S, et al. Correlation feature and instance weights transfer learning for cross project software defect prediction[J]. IET Software, 2021, 15(1): 55-74.
[399] Wang A, Zhang Y, Yan Y. Heterogeneous Defect Prediction Based on Federated Transfer Learning via Knowledge Distillation[J]. IEEE Access, 2021, 9: 29530-29540.
[400] Malhotra R, Jain J. Predicting Software Defects for Object-Oriented Software Using Search-based Techniques[J]. International Journal of Software Engineering and Knowledge Engineering, 2021, 31(02): 193-215.
[401] Li X, Yang X, Su J, et al. A Multi-Objective Learning Method for Building Sparse Defect Prediction Models[C]//2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS). IEEE, 2020: 204-211.
[402] Lavazza L, Morasca S. An Empirical Study of Thresholds for Code Measures[C]//2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2020: 346-357.
[403] Jahanshahi H, Cevik M, Başar A. Moving from cross-project defect prediction to heterogeneous defect prediction: a partial replication study[J]. arXiv preprint arXiv:2103.03490, 2021.
[404] Malhotra R, Jain J. Predicting defects in object-oriented software using cost-sensitive classification[C]//IOP Conference Series: Materials Science and Engineering. IOP Publishing, 2021, 1022(1): 012112.
[405] Wu Y, Yao J, Chang S, et al. LIMCR: Less-Informative Majorities Cleaning Rule Based on Naïve Bayes for Imbalance Learning in Software Defect Prediction[J]. Applied Sciences, 2020, 10(23): 8324.
[406] Wang H, Zhuang W, Zhang X. Software Defect Prediction Based on Gated Hierarchical LSTMs[J]. IEEE Transactions on Reliability, 2021.
[407] Yadav H S. Increasing Accuracy of Software Defect Prediction using 1-dimensional CNN with SVM[C]//2020 IEEE International Conference for Innovation in Technology (INOCON). IEEE, 2020: 1-6.
[408] Yan X, Zhang Y, Khan A A. An algorithm acceleration framework for correlation-based feature selection[C]//MATEC Web of Conferences. EDP Sciences, 2021, 336: 07011.

The list of citations list using the JIRA-HA-2019 / JIRA-RA-2019 data set

[409] S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, K. Matsumoto. Predicting Defective Lines Using a Model-Agnostic Technique. arXiv preprint arXiv:2009.03612v1, 2020.
[410] Kushani Perera, Jeffrey Chan, Shanika Karunasekera: A Framework for Feature Selection to Exploit Feature Group Structures. PAKDD (1) 2020: 792-804.
[411] Rajapaksha D, Tantithamthavorn C, Jiarpakdee J, et al. SQAPlanner: Generating Data-InformedSoftware Quality Improvement Plans[J]. arXiv preprint arXiv:2102.09687, 2021.
[412] Jiarpakdee J, Tantithamthavorn C, Grundy J. Practitioners' Perceptions of the Goals and Visual Explanations of Defect Prediction Models[J]. arXiv preprint arXiv:2102.12007, 2021.
[413] Jahanshahi H, Cevik M, Başar A. Moving from cross-project defect prediction to heterogeneous defect prediction: a partial replication study[J]. arXiv preprint arXiv:2103.03490, 2021.
[414] Khan S S, Niloy N T, Azmain M A, et al. Impact of Label Noise and Efficacy of Noise Filters in Software Defect Prediction[J].

The list of citations list using the IND-JLMIV+R-2020 data set

[415] Alexander Trautsch, Fabian Trautsch, Steffen Herbold, Benjamin Ledel, Jens Grabowski. The SmartSHARK Ecosystem for Software Repository Mining. arXiv preprint arXiv:2001.01606v1, 2020.
[416] Alexander Trautsch, Steffen Herbold, Jens Grabowski. A Longitudinal Study of Static Analysis Warning Evolution and the Effects of PMD on Software Quality in Apache Open Source Projects. arXiv preprint arXiv:1912.02179v3, 2020.
[417] Steffen Herbold, Alexander Trautsch, Benjamin Ledel. Large-Scale Manual Validation of Bugfixing Changes. MSR 2020.
[418] Herbold, S., (2020). Autorank: A Python package for automated ranking of classifiers. Journal of Open Source Software, 5(48), 2173. https://doi.org/10.21105/joss.02173.
[419] Herbold S, Trautsch A, Trautsch F. On the feasibility of automated prediction of bug and non-bug issues[J]. Empirical Software Engineering, 2020: 1-37.