Retrieving Affected Versions by Leveraging the Defects Life Cycle

BAILEY VANDEHEI, California Polytechnic State University
DANIEL ALENCAR DA COSTA, University of Otago, New Zealand
DAVIDE FALESSI, California Polytechnic State University

Two recent studies explicitly suggest labeling defective classes in releases by using the realistic approach, i.e., using affect version (AV) as provided by developers in the issue tracker such as JIRA. However, no study investigated whether the AV can actually be used (i.e., it is available and consistent), nor how to retrieve it when unavailable. The aim of this study is threefold: 1) to measure the proportion of defects on which the realistic method is usable, 2) to propose a method to retrieve the AV of a defect, and hence making the realistic approach usable when the AV is unavailable, 3) to compare the accuracy of the proposed method versus three SZZ implementations. The assumption of our method is that defects have a stable life cycle in terms of the proportion of the number of versions needed to discover and to fix a defect. Results related to 212 open-source projects from the Apache ecosystem, featuring a total of about 125,000 defects, show that the realistic method cannot be used in the majority (51%) of defects. Therefore, it is important to develop automated methods to retrieve AV. Results related to 76 open-source projects from the Apache ecosystem, featuring a total of about 6,250,000 classes affected by 60,000 defects and spread over 4,000 versions and 760,000 commits, show that the proportion of the number of versions between its discovery and its fix is pretty stable (STDV <2) across defects of the same project. Moreover, the proposed method resulted significantly more accurate than all three SZZ implementations, in retrieving AV, in labeling classes as defective, and in developing defects repositories to perform feature selection. In conclusion, when the realistic method is unusable, the proposed method is a valid automated alternative to SZZ for retrieving the origin of a defect.

CCS Concepts: • **Software and its engineering → Software defect analysis**.

Additional Key Words and Phrases: Affected version, SZZ, defect origin, developing defects repository

ACM Reference Format:
Bailey Vandehei, Daniel Alencar da Costa, and Davide Falessi. 2018. Retrieving Affected Versions by Leveraging the Defects Life Cycle. 1, 1 (May 2018), 32 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The manner in which defects are introduced into code, and the sheer volume of defects in software, are typically beyond the capability and resources of most development teams [18, 27, 32, 41, 57, 60]. Due to this problem, researchers have explored machine learning approaches to predict (1) whether a defect is likely to occur in a software module (i.e, a binary response); and (2) how many post-deployment defects are likely to occur (a count response) [15, 24, 28, 35, 41, 42, 54].

Authors' addresses: Bailey Vandehei, bvandehe@calpoly.edu, California Polytechnic State University, San Luis Obispo, California; Daniel Alencar da Costa, danielcalencar@otago.ac.nz, University of Otago, Dunedin, New Zealand; Davide Falessi, dfalessi@calpoly.edu, California Polytechnic State University, San Luis Obispo, California.

Defect prediction models are built using complexity features (i.e., lines of code, Halstead metrics, McCabe's complexity, and CK metrics) [2, 21, 37] and process features [45]. Predicting the occurrence of defects is useful because a development team could better focus the limited testing effort.

Before any defect prediction can be performed, it is important to create a repository containing the features and the associated defects. Our work focuses on the automated methods for the creation of defect prediction datasets. We are interested in methods for establishing the origin of a defect. Researchers provided means to create [17, 71], collect [11] and select [19, 40, 51] datasets for associating software defects to process and product features. However, existing research has shown that the general quality of software defect datasets are not without flaws [3, 22, 29, 46, 56]. For example, Bird et al. [3] demonstrated the existence of non-negligible bias in the features that are used to build defect prediction models. Tantithamthavorn et al. [56] have also shown that cleaning the datasets prior to performing defect predictions can increase the ability to better identify the defective modules. Indeed, the general accuracy of a defect prediction model depends on the quality of the underlying datasets [31, 52].

One main limitation of the defect prediction models is the granularity of the predictions (e.g., whether a module is defective or not), which is often too coarse to be useful [26]. To face this limitation, researchers have explored *Just-In-Time* (JIT) defect predictions [14], in which the goal of the prediction is to indicate whether a newly produced commit will be defective or clean. Nevertheless, JIT prediction models can only be feasible if the exact origins of a defect are known [14].

To identify the origins of a defect, researchers have proposed the SZZ approach [55]. However, the state-of-art of the SZZ approach is far from being ideal [7, 48, 50]. For example, da Costa et al. [7] highlighted that current SZZ implementations cannot determine the origins of defects that were fixed by solely adding code. Additionally, SZZ is also incapable of identifying the origins of defects of the regression type [61]. Finally, Rodríguez-Pérez et al. [49] revealed that only a significant minority of defects can have their origins traceable in the source code repository, thus limiting the applicability of SZZ.

Two recent studies [7, 68] suggest the use of *Affected-Versions* (AVs) of defect reports, as provided by the Issue Tracker Systems such as JIRA, to better label defective modules, instead of solely relying on SZZ. However, these studies also hint that the availability of AVs is scarce [7, 68], i.e., only a few defect reports provide AV information. Therefore, in this work, we propose a first-of-its-kind approach to retrieve AVs. To achieve our goal, we first investigate the extent to which AVs are usable, i.e., available and consistent, in open-source projects. Second, we propose, evaluate, and compare a novel and automated methods for retrieving AVs, including the earliest possible AV (i.e., the origin of a the defect). Our intuition is that *defects have a stable life cycle* in terms of the proportion of the number of versions needed to discover and to fix a defect. The idea is that defects that quickly manifest themselves as a fault (i.e., report creation) are easiest to find and fix than defects that are dormant [1, 5] over several releases. This is because, in order to fix a defect, the developers need to identify the change that leaded to a defect, and we assume that the more the change inducing the defect is remote the longer it takes to be identified and fixed. These assumptions of the stability of defects' life-cycle seem to have analogies with *diseases'* life-cycle [47].

Our results obtained in 212 Apache open-source projects reveal that AV information is lacking in the majority of defects (51%). Therefore, it is important to investigate automated methods for retrieving AVs. Our results obtained in 76 Apache open-source projects demonstrate that our proposed method is more accurate than previously proposed SZZ methods in terms of retrieving AVs. Additionally, our methods are more accurate in labeling classes as defective and in developing defects datasets for performing feature selections.

The remainder of this paper is structured as follows. We explain the background material and related work in Section 2. In Section 3, we describe our study design. We present our obtained results in Section 4. In Section 5, we discuss our results. We explain the threats to validity of this study in Section 6, while we provide our conclusions in Section 7.

2 RELATED WORK & BACKGROUND

We provide in this section the key concepts to understand our research context.

Śliwerski et al. [55] proposed the first implementation of the SZZ approach, which strived to find the origins of a defect (i.e., the bug-introducing changes). SZZ exploits the versioning system annotation mechanism (e.g. git blame) to determine–for the source code lines that have been changed in a defect fix–when they have last been changed before the fix. The SZZ algorithm consists of three main steps. We demonstrate these steps by using the HADOOP-7770[1] defect as an example (shown in Figure 1). HADOOP-7770 was caused because the developers used the wrong object to provide a file path, which incurred a `FileNotFoundException`. **Step 1** of SZZ (shown in Figure 1) consists of finding the commit that fixed the defect (i.e., the bug-fixing change). In the case of HADOOP-7770, the bug-fixing change was performed in commit `1190532`[2] by changing `getFileChecksum(f)` to `getFileChecksum(res.remainingPath)`. SZZ can use several mechanisms to find bug-fixing changes [55].[3] Afterwards, in **Step 2**, SZZ analyzes the diff patch of the bug-fixing change in order to locate the *faulty code*. In this step, SZZ assumes that the code removed in a patch is the code that expresses the defect. In the case of HADOOP-7770, the removed code in the diff patch was the `getFileChecksum(f);` code. Finally, once the *faulty code* has been identified, SZZ traces the code history to find when the *faulty code* was introduced (i.e., **Step 3**). **Step 3** of SZZ can be implemented by using, for example, the `blame` operation that is present in most Version Control Systems (VCSs, such as Git or Subversion). In Figure 1, SZZ uses the *git blame* command to find that commit `1100026`[4] is the commit that introduced the `getFileChecksum(f);` and, hence, the code that potentially introduced the bug (i.e., the bug-introducing commit).

Several other studies strove on estimating the origin of defects. Kim et al. [30] presented algorithms to automatically and accurately identify bug-introducing changes which improved over SZZ. da Costa et al. [7] proposed three criteria and evaluated five SZZ implementations. They concluded that current SZZ implementations still lack mechanisms to accurately identify bug-introducing changes. Yatish et al. [68] presented the realistic approach to estimate the origin of a defect. This approach relies on the use of the AV and is the main motivation of the present work. Neto et al. [43] found that 19.9% of lines that are removed during a fix are related to refactoring operations and, therefore, their respective bug-introducing changes are false positives. Falessi and Moede [11] presented the Pilot Defects Prediction Dataset Maker (PDPDM), a desktop application for measuring metrics for using in defect prediction. PDPDM avoids the use of outdated datasets, and it allows researchers and practitioners to create defect datasets without writing any code. Rodríguez-Pérez et al. [50] investigated the complex phenomenon of defect introduction and defect fix. They showed that less than 30% of defects can actually be traced to its origins by assuming that "a given defect was introduced by the lines of code that were modified to fix it". Our research complements the prior research in defect introduction by providing methods to estimate the AV information. AVs can then be used to evaluate or improve approaches such as SZZ [7].

Extensive research has been invested in building and evaluating datasets for defect prediction. Shepperd et al. [53] investigated five studies that have used the NASA dataset for building defect prediction models. The goal of their

[1] https://issues.apache.org/jira/browse/HADOOP-7770
[2] http://svn.apache.org/viewvc?view=revision&revision=1190541
[3] A popular approach to identify bug-fixing changes is by using simple heuristics, such as searching for the "fix" or "fixed" keywords in a commit log [10].
[4] http://svn.apache.org/viewvc?view=revision&revision=1100026

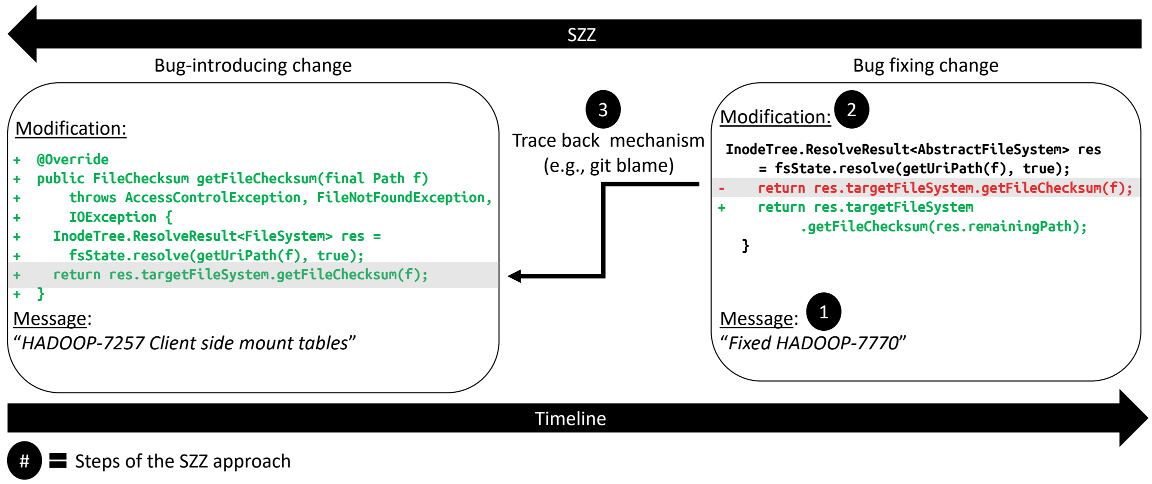

Fig. 1. An example of the SZZ functioning. Step 1 consists on identifying the bug-fixing changes. Step 2 localizes the faulty code, which is the code removed in the bug-fixing change. Finally, in Step 3, SZZ traces the code history to find the bug-introducing changes.

work was to verify whether the different versions of the NASA dataset yield consistent results. Shepperd et al. [53] observed that different versions of the same dataset (e.g., NASA) may produce different results for defect prediction and, therefore, researchers should be cautious before choosing a dataset. Nam and Kim [42] proposed the CLA and CLAMI approaches to automatically label unlabelled defect prediction datasets, relieving researchers from the manual effort. The approaches work based on the magnitude of metric values and obtain average prediction performances of around 0.64 (F-measure) and 0.72 (AUC).

Other studies focused on how to select repositories to mine. Nagappan et al. [40] combine ideas from representativeness and diversity, and introduce a measure called sample coverage, which is the percentage of projects in a population that are similar to the given sample. They conclude that papers should discuss the target population of the research (universe) and dimensions that potentially can influence the outcomes of research (space). Gousios and Spinellis [19] proposed the Alitheia Core analysis platform, which pre-processes repository data into an intermediate format that allows researchers to provide custom analysis tools. Rozenberg et al. [51] proposed RepoGrams to support researchers in qualitatively comparing and contrasting software projects over time using a set of software metrics. RepoGrams uses an extensible, metrics-based, visualization model that can be adapted to a variety of analyses. Falessi et al. [13] presented STRESS, a semi-automated and fully replicable approach that allows researchers to select projects by configuring the desired level of diversity, fit, and quality.

A significant effort has been spent in measuring the noise in defects repositories and its impact on the follow-up analyses. Bird et al. [3] found that bias is a critical problem that threatens both the effectiveness of processes that rely on biased datasets to build prediction models and the generalizability of hypotheses tested on biased data. Kim et al. [29] measured the impact of noise on defect prediction models and provide guidelines for acceptable noise levels. They also propose a noise detection and elimination algorithm to address this problem. However, the noise studied and removed is supposed to be random. Herzig et al. [22] reported that 39% of files marked as defective actually never had a defect. They discuss the impact of this misclassification on earlier studies and recommend manual data validation for future studies. Rahman et al. [46] showed that size always matters just as much as bias direction, and in fact, much more than bias direction when considering information-retrieval measures such as AUCROC and F-score. This indicates that, at least for prediction models, even when dealing with sampling bias, simply finding larger samples can sometimes be sufficient. Tantithamthavorn et al. [56] found that: (1) issue report mislabelling is not random; (2) precision is rarely impacted by mislabelled issue reports, suggesting that practitioners can rely on the accuracy of modules labelled as defective by models that are trained using noisy data; (3) however, models trained on noisy data typically achieve about 60% of the recall of models trained on clean data. Similarly to the aforementioned papers, we measured the extent of noise (i.e. classes mislabeled) and its impact on analyzing a repository in terms of features selection.

Another line of research in defect prediction has proposed the usage of a machine learning model to predict whether an upcoming commit is defective or clean [16, 25, 27, 28, 64, 67]. This area of research was eventually coined as *Just-in-time* defect prediction (JIT). Kim et al. [28] proposed the usage of JIT models in their seminal work. In order to label their datasets, the authors used the output from SZZ. Fukushima et al. [16] and Kamei et al. [25] advanced the area and explored the usage of cross-project JIT models to help software projects without enough historical data to build their own models. In our work, we aim at estimating AV information and verifying whether our methods can improve the accuracy on labeling defective classes compared to SZZ-based approaches. It is worth to note that we do not use our approaches for building JIT models as we envision to do so in future work.

Other researchers argue that, more important than predicting the number of potential defects, is to predict the ranking of modules with respect to their defect proneness [44, 65, 66, 69, 70]. Yang et al. [65] proposed the use of Learning-to-Rank (LTR) algorithms to rank the defect proneness of software modules in their seminal work. Later, Yang et al. [66] expanded the their seminal work to (i) test more LTR algorithms.

3 EXPERIMENTAL DESIGN

We present in this section the concepts that are shared across our research questions. Figure 2 illustrates the key terms by using the defect QPID-4462[5] as an example. The defect is first injected in the code at the *Introducing Version* (**IV**), i.e., the V0.18 version in Figure 2. Afterwards, a failure is experienced and a defect report is created to describe the defect. We refer to the version related to the creation of the defect report as the *Opening Version* (**OV**), i.e., the V0.20 version in Figure 2. Next, in a given future version, the defect is fixed by changes performed in one or more classes. We refer to the version related to the fix of the defect as the *Fixing Version* (**FV**), i.e., the V0.22 version in Figure 2. An AV is any version in which the defect could have been experienced, i.e., any version *affected* by the defect. Thus, the AVs in our example are those in the range [IV, FV), i.e., the V0.18 and V0.20 versions in Figure 2. The V0.22 version is not an AV since it contains the fix and is not affected by the defect.

The OV is available in all defect reports as it is generated by the issue tracker at the creation the report. The FV is available in defect reports where developers have mentioned the defect report ID in the log of the commit that fixes the defect. For example, commit *732ab160852f943cd847646861dd48370dd23ff3* is the last commit including *[QPID-4462]* in its log. Since this commit was performed at *2013-03-31T21:51:49+00:00*, we can infer that it has been performed between versions V0.20 and V0.22.

3.1 Research Questions

In this section, we describe our RQs and their respective motivation for our research.

[5]https://issues.apache.org/jira/browse/QPID-4462

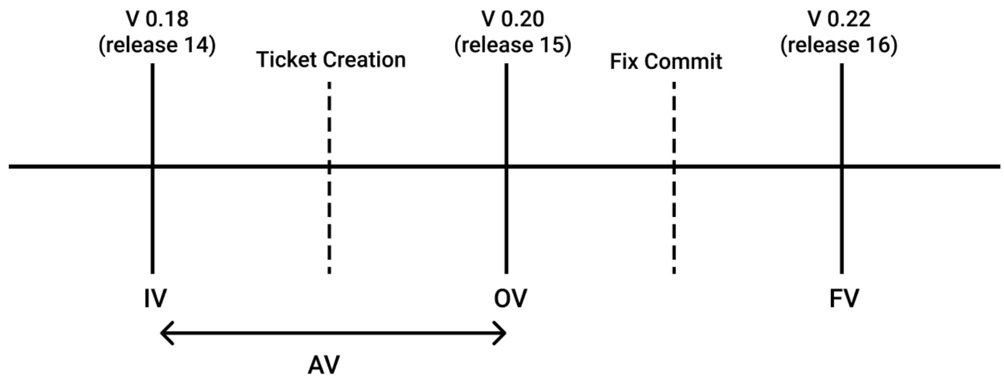

Fig. 2. Example of the life-cycle of a defect: Introduction Version (IV), Opening Version (OV), Fixed Version (FV), and Affected Versions (AV). Note, versions V0.19 and V0.21 were only "baselines" and not "user-intended" versions and, hence, were excluded

3.2 RQ1: Are AVs available and consistent?

Two recent SZZ studies [7, 68] suggest using AVs to identify the origin of a defect and hence create defect datasets. However, how often do developers actually report the AV field? In this research question, we investigate the extent to which AVs are usable, i.e., whether they are available and consistent. Thus, our dependent variable is the percentage of *available & consistent* AVs. An AV information is *available* if it is provided in the ticket related to a defect. An AV information is *consistent* when the earliest AV occurs before the OV. The rationale is that the defect must have affected a version that occurred at least at the moment when the defect report had been created. That is, a defect cannot have been injected *after* the defect had been experienced—experiencing a defect is what motivates the defect report creation. To measure the availability and consistency of AVs, we follow the following eight steps:

(1) We retrieve the JIRA and Git URL of all existing Apache projects.[6] We focused on Apache projects rather than GitHub projects because Apache projects have a higher quality of defect annotation and are unlikely toy projects [39]. Finally, Apache projects use JIRA as their issue tracker, which allows us to study the consistency and availability of AV information.
(2) We filter out projects which are not tracked in JIRA or not versioned in Git. This leads to 212 projects.
(3) As recently done by Borg et al. [36], for each project, we count the number of issue reports by performing the following query to the JIRA repository: *Type == “defect” AND (status == “Closed” OR status ==“Resolved”) AND Resolution ==“Fixed”*. This gave us a total of about 235,000 defects.
(4) We exclude issue reports not having a related Git commit fixing it.
(5) We exclude defects that are not post-release. Post-release defects are also known in the industry as production defects, i.e., defects that were experienced by users after a release of the project has been shipped. Thus, a defect that is injected and fixed in the same version is not a post-release defect. For brevity, in the remainder of this paper, we refer to post-release defects simply as defects. After steps 4 and 5, we are left with a total of 125,000 defects.
(6) For each issue report, we check its AV **availability**, i.e., the presence of the AV field, by performing the following query to the JIRA repository: Affect Version $\neq$ “Null”. Thus, each issue report is tagged as available or unavailable.
(7) For each issue report, we check its AV **consistency**, i.e., if $IV \leq OV$.

[6]https://people.apache.org/phonebook.html

(8) For each project, we compute the percentage of *available*, and *available & consistent* AV across its defects' issue reports, and we report the distribution of such percentages across projects as boxplots.

3.3 RQ2: Do methods have different accuracy for labeling affected versions?

If the AVs are unusable, then they must be retrieved. Since AVs are in the range are those in the range [IV, FV), and since we always know FV, retrieving the AVs of a defect actually translates into estimating its IV. One approach to estimate the IV of a defect is to employ the SZZ algorithm. The oldest defect-introducing commit produced by SZZ can be considered as the IV, whereas all other defect-introducing commits can be used to label the consecutive versions before the defect-fixing commit as other AVs (the defect-fixing commit itself is not considered in the labeling process, of course). However, existing research have highlighted substantial limitations of the SZZ approach [7, 48, 50].

In this research question investigates which methods are most accurate for labeling versions as AV or not at the defect level. Thus, we investigate the following null hypothesis:

•H_{10}: *different methods obtain the same accuracy for labeling AVs.*

3.3.1 Independent variables. Our independent variable is the method used to retrieve the AV, i.e., to label a version as affected or not by a specific defect. In this work, we consider a total of 10 methods:

(1) **SZZ**: As previously discussed in Section 2, SZZ is an algorithm that, given a fix commit, determines the possible defect-introducing commits. In our methods, we assume the oldest defect-introducing commit to be the IV. Specifically, among the possible ways to use SZZ, we considered the following methods:
   (a) **SZZ_Basic**: We use the SZZ algorithm [55] to determine when the defect has been introduced, and we assume as AVs all versions between the IV and the FV (not including FV). In the example in Figure 2, SZZ_B identified three defect-introducing commits with the following dates: 2012-05-19T08:54:25, 2012-10-06T05:38:51, 2012-11-05T10:03:36. Among these, 2012-05-19T08:54:25 is the oldest date, which falls into version 0.18 labeled as the IV. Therefore, the AVs are 0.18 and 0.20. Versions 0.18 and 0.20 were correctly identified as defective (true positives), and therefore, this method receives 100% accuracy for this defect.
   (b) **SZZ_U**: We rely on an open implementation of SZZ by Borg et al. [4] and we set the depth to one. This SZZ implementation does not discard cosmetic changes (since it supports all programming languages). However SZZ_U uses Jaccard distances to map moving lines. In the example in Figure 2, SZZ_U identified one defect-introducing commit dated 2012-05-18T20:54:25, which falls into version 0.16 labeled as the IV. Therefore, the AVs are 0.16, 0.18, and 0.20. Versions 0.18 and 0.20 were correctly identified as defective (true positives) and version 0.16 was incorrectly identified as defective (false positives).
   (c) **SZZ_RA**: We use a refactoring-aware SZZ algorithm implemented by Da Costa [7]. This algorithm tracks defect-introducing commits and filters out refactoring operations. However, this implementation only analyzes java files, so the defect-introducing commits for non-java files are determined by SZZ_U. In the example in Fig. 2, SZZ_RA identified one defect-introducing commit dated 2012-05-18T16:54:25 which falls into version 0.16 labeled as the IV. Therefore, the AVs are 0.16, 0.18, and 0.20. Versions 0.18 and 0.20 were correctly identified as defective (true positives) and version 0.16 was incorrectly identified as defective (false positives).

(2) **Simple**: It simply assumes that the IV corresponds to OV. The rationale is that, by definition, all versions from OV to FV (not including FV) are AV. However, versions before OV can also be AV. Therefore, we expect this heuristic to achieve a 100% Precision but a lower Recall. Specifically, this heuristic would identify 0.20 as IV in Figure 2. Therefore, it would miss 0.18 (false negative) and would correctly identify 0.20 (true positives) as AVs.

(3) **SZZ_X+:** All the SZZ based methods can be merged with the *Simple* method. Therefore, for each SZZ method, we created a SZZ_X+, where a version is defective if SZZ_X labeled it as defective or Simple labeled it as defective. Hence, we are merging the defects' life cycle information with the SZZ based method. The rationale is that if Simple labels a version as defective, then the version is actually defective by definition.
To illustrate how this works, we will use a new example, WICKET-4071[7]. The AVs indicated on this defect report are: 1.4.6, 1.4.7, 1.4.8, 1.4.19, 1.4.10, and 1.5-M1. The OV is 1.4.8 and FV is 1.4.11. Simple would classify versions 1.4.8, 1.4.19, 1.4.10, and 1.5-M1 as defective (true positives) and would miss versions 1.4.6 and 1.4.7 (false negatives). SZZ_B would classify 1.4.10 and 1.5-M1 as defective (true positives) and miss versions 1.4.6, 1.4.7, 1.4.8, and 1.4.19 (false negatives). However, SZZ_B+ would classify versions 1.4.8, 1.4.19, 1.4.10, and 1.5-M1 as defective (true positives) and would miss versions 1.4.6 and 1.4.7 (false negatives).

(4) **Proportion**. It assumes a stable proportion (P), among defects of the same project, between the number of affected versions between IV and FV, and the number of versions between OV and FV. The rationale is that the life-cycle might be consistent among defects of the same projects. Thus, in some projects, defects require a number of versions to be found and a number to be fixed. Our intuition is that the proportion among these numbers is somehow stable across defects of the same project. Of course, defects of the same projects may vary and, hence, we do not expect this method to be perfectly accurate. Since FV and OV are known for every defect, the idea is to compute P on previous defects, and then use it for defect reports where AVs are not available nor consistent. Thus, we define P as $(FV - IV)/(FV - OV)$. Therefore, we can calculate the IV as $FV - (FV - OV) * P$.
Among the possible ways to use Proportion we considered the following methods:

(a) **Proportion_Incremental:** In this method, we ordered the defects by fix date. For each version R within a project, we used the average P among defects fixed in versions 1 to R-1. Using the example in Figure 2, the $P\_Increment$, computed as the average P among defects in versions 1 to 15, is 1.7775. Therefore, $IV = 16 - (16 - 15) * 1.7775$, which is 14.2225. Hence, this method would correctly identify 0.20 as defective (true positive), but incorrectly classify 0.18 as not defective (false negative).

(b) **Proportion_ColdStart:** In the case that a project is new or has very few fixed defects, computing an average P within the project's fixed defects is ineffective. Therefore, we utilize the average P values of other projects. For each studied project, we compute the average P across all defects within the project. We label each of these projects as $P\_PROJECT$ where $PROJECT$ is the project's ID. Next, for each project, we take the median of the $P\_PROJECT$ values among all other projects to use as the $P\_ColdStart$. Using the example in Figure 2, the indexes of the 0.18, 0.20, and 0.22 versions are 14, 15, and 16, respectively. The $P\_ColdStart$ computes as the median of the other $P\_PROJECT$ proportions, as 1.8089. Therefore, $IV = 16 - (16 - 15) * 1.8089$ which is 14.1911. Hence, this method would correctly identify 0.20 as defective (true positive), but incorrectly classify 0.18 as not defective (false negative).

(c) **Proportion_MovingWindow:** In this method, we ordered the defects by their fix date. For each defect within a project, we used the average P among the last 1% of fixed defects. Using the example in Figure 2, the $P\_MovingWindow$ is computed as the average P among the last 1% of defects. There are 1,192 defects in the project of Figure 2. Therefore, there are around 12 defects at the 1% of defects. The average P among the last 12 fixed defects is 2.167. Therefore, $IV = 16 - (16 - 15) * 2.167$ which is 13.833. Hence, this method would correctly identify 0.18 and 0.20 as defective (true positive), giving 100% accuracy for this defect.

[7]https://issues.apache.org/jira/browse/WICKET-4071

3.3.2 Dependent variables. Our dependent variable is the accuracy for labeling versions of a project as affected, or not, by a defect. We use the following set of metrics:

- True Positive(TP): The version is actually defective and is labeled as defective.
- False Negative(FN): The version is actually defective and is labeled as non-defective.
- True Negative(TN): The version is actually non-defective and is labeled as non-defective.
- False Positive(FP): The version is actually non-defective and is labeled as defective.
- **Precision** : $\frac{TP}{TP+FP}$
- **Recall** : $\frac{TP}{TP+FN}$
- **F1** : $\frac{2*Precision*Recall}{Precision+Recall}$
- Cohen's **Kappa** : A statistic that assesses the classifier's performance against random guessing [6]. $Kappa = \frac{Observed-Expected}{1-Expected}$ where
  - Observed: The proportionate agreement. $\frac{TP+TN}{TP+TN+FP+FN}$
  - Expected: The probability of random agreement. $P_{Yes} + P_{No}$ where
    - P_{Yes}: Probability of positive agreement. $\frac{TP+FP}{TP+TN+FP+FN} * \frac{TP+FN}{TP+TN+FP+FN}$
    - P_{No}: Probability of negative agreement. $\frac{TN+FP}{TP+TN+FP+FN} * \frac{TN+FN}{TP+TN+FP+FN}$
- **Matthews** Correlation Coefficient : $\frac{TP*TN-FP*FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$

Since we have binary classifications that are thresholds independent, we do not use Area Under the Receiver Operating Characteristic metric.

3.3.3 Measurement procedure. We began by selecting the projects with the highest proportion of usable (i.e., available and consistent) AVs. We selected projects with at least 100 defects that were linked with git and contained available and consistent AVs. Then, we filtered out projects with less than 6 versions. Lastly, we filtered out projects where the percent of available and consistent AVs are less than 50%. This left us with 76 projects. For each project, we followed the steps below. See Figure 3 for an overview of this process.

(1) We retrieved the versions of the project and their release dates from JIRA. We numbered these versions beginning with the oldest version as version 1.
(2) We used the defects, of which their reports provided available and consistent AVs in RQ1. For each defect, we determined the IV (i.e., the version of the first AV labeled by JIRA), OV (i.e., the version of the ticket creation), FV (i.e., the fix version), and the fix commit hash by Git. We ordered the defects by fix date.
(3) For each defect, we labeled versions 1 to FV as defective or not by each of the following methods:
   (a) **Simple:**
      (i) We set IV equal to OV.
      (ii) For each defect, we label each version before the IV as not defective. We label each version from the IV to the FV as defective. The FV is labeled not defective.
   (b) **SZZ:**
      (i) We ran each SZZ implementation on the project by supplying the Git directory and a list of defects and their fix commit.

(ii) For each defect, SZZ outputs all possible defect-introducing commits. We compute the corresponding version for each defect-introducing commit. We chose the oldest version to be the IV.

(iii) For each defect, we label each version before the IV as not defective. We label each version from the IV to the FV as defective. The FV is labeled not defective.

(c) **Proportion_ColdStart:**

(i) We computed the average P across the project's defects, i.e., $P = (FV - AV)/(FV - OV)$. If FV equals OV, then $FV - OV$ is set to one to avoid divide by zero cases.

(ii) We computed the $P\_ColdStart$, i.e., the median P of all other projects.

(iii) For each defect, we computed the IV as $IV = (FV - OV) * P\_ColdStart$. If FV equals OV, the IV equals FV. However, recall we excluded defects that were not post-release. Therefore, we set $FV - OV$ equal to 1 to assure IV is not equal to FV.

(iv) For each defect, we label each version before the IV as not defective. We label each version from the IV to the FV as defective. The FV is labeled not defective.

(d) **Proportion_Increment:**

(i) For each version R, we computed $P\_Increment$ as the average P among defects fixed in versions 1 to R-1.

(ii) We used the P_ColdStart for P_Increment values containing less than 5 defects in the average.

(iii) For each defect in each version, we computed the IV as $IV = (FV - OV) * P\_Increment$. If FV equals OV, the IV equals FV. However, recall we excluded defects that were not post-release. Therefore, we set $FV - OV$ equal to 1 to assure IV is not equal to FV.

(iv) For each defect, we label each version before the IV as not defective. We label each version from the IV to the FV as defective. The FV is labeled not defective.

(e) **Proportion_MovingWindow:**

(i) For each defect, we computed P_MovingWindow as the average P among the last 1% of defects. The defects are ordered by their fix date.

(ii) We used the P_ColdStart for P_MovingWindow values containing less than 1% of defects in the average.

(iii) For each defect, we computed the IV as $IV = (FV - OV) * P\_MovingWindow$. If FV equals OV, the IV equals FV. However, we excluded defects that were not post-release. Therefore, we set $FV - OV$ equal to 1 to assure IV is not equal to FV.

(iv) For each defect, we label each version before the IV as not defective. We label each version from the IV to the FV as defective. The FV is labeled not defective.

(f) **+:**

(i) For each SZZ method, we combined it with Simple. For each defect, we labeled each version as defective if SZZ_X or Simple labeled the version as defective.

(4) We then computed the actual defectiveness of versions 1 to FV for each defect. We label each version before the IV, labeled by JIRA developers, as not defective. We label each version from the IV to the FV as defective. The FV is labeled not defective.

(5) For each method, we compared the classification to the actual classification and computed the TP, TN, FP, FN, Precision, Recall, F1, Matthews, and Kappa across the project's version-defect pairs.

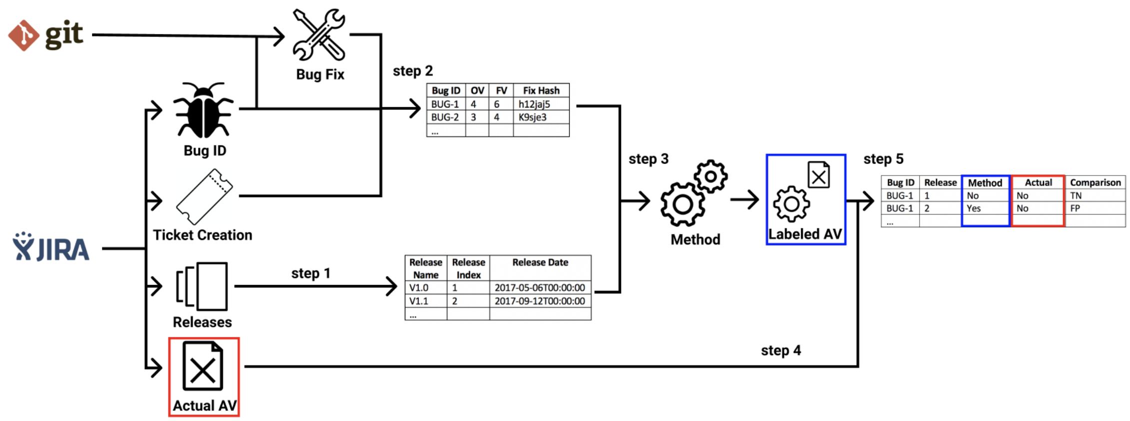

Fig. 3. The process to measure the accuracy of methods in labeling defective classes.

3.3.4 Hypothesis testing. To test hypothesis H_{10}, in RQ2, we used the Kruskal–Wallis test [33], which is a non-parametric test (i.e., a normally distributed data is not required) to check whether three or more distributions are significantly different. This test is necessary because we compare 10 distributions of values (i.e., one for each studied method). For example, we compare whether our 10 studied methods for estimating AVs significantly differ in terms of precision or recall.

Although Kruskal–Wallis provides an indication of whether a significant difference between distributions exists, it does not indicate which specific pairs of distributions are significantly different between each other. For this purpose, we perform a follow-up Dunn test [9], which is a *post-hoc* test to indicate which specific pairs of distributions significantly differ between each other. To account for the chance of errors due to multiple comparisons, we perform a Holm-Bonferroni correction of our $p - values$ [23]. Finally, to verify the effect-size of our observed differences, we compute Scott–Knott Effect Size Differences (ESD) as proposed by Tantithamthavorn et al. [58]. The Scott–Knott ESD procedure clusters different distributions into ranks by considering the differences in effect-size of the distributions [34]. We use the Scott–Knott ESD procedure because it delivers further insights on the strength of the differences between our studied distributions. The output of the Scott–Knott ESD procedure is, for example, which methods fall in the top ranks of precision or recall.

3.4 RQ3: Do methods have different accuracy for labeling defective classes?

In this research question, we investigate which methods have the highest accuracy for labeling defective classes. In other words, in this research question we observe the accuracy of the realistic approach [68], in labeling classes, when the AVs are retrieved by different methods. This investigation is important as the accuracy of defect prediction activities is highly correlated with the soundness of the datasets that are used [31, 52]. We propose the following hypothesis for this RQ:

• H_{20}: *different methods obtain the same accuracy for labeling classes.*

3.4.1 Independent variables. The independent variable is represented by the same methods shown in RQ2 (see Section 3.3.1). However, in this research question, the retrieved AVs (as performed in RQ2) is used to label classes as defective or not.

3.4.2 Dependent variables. The dependent variables are the same accuracy metrics presented in RQ2 (see Section 3.3.2), with the only difference that the unit upon which the accuracy is computed is the defectiveness of a class in a version. If at least one defect impacts the version-class pair, then the version-class pair is labeled as defective. This is demonstrated in Figure 4 where F1.java is deemed defective because it was touched by the fix for defect-3 in version 1 (i.e., at least one defect-fix touched F1.java in version 1).

In order to better explain the difference between RQ3 and RQ2, let's consider the case of methods A, B, and C, and a class that was affected by three defects in a certain version. Suppose that A is able to identify that the class was affected by one defect, B, by three defects, and C, by 4 defects. In this example, all three methods correctly identify the class in the version as defective and, therefore, all three methods result with perfect accuracy. However, for the purpose of RQ2, method B has a higher accuracy than methods A and C. The following metrics have been redefined for this RQ:

- True Positive(TP): The class in a version is actually defective and is labeled as defective.
- False Negative(FN): The class in a version is actually defective and is labeled as non-defective.
- True Negative(TN): The class in a version is actually non-defective and is labeled as non-defective.
- False Positive(FP): The class in a version is actually non-defective and is labeled as defective.

3.4.3 Measurement procedure. Figure 4 describes the process we use to label a class in a version as defective or not. The process is identical to what Yatish et al. [68] coined as the realistic approach. The only difference that the AV is assumed to be unavailable and, hence, it is retrieved by using a certain proposed method (see RQ2). The process consists of three steps:

(1) For each defect in RQ2, we computed a list of classes touched by the fix commit.
(2) For each method in RQ2, we labeled each version-class pair as defective if the version of the pair was determined to be an AV of at least one defect in RQ2 and the defect-fix commit of that defect involved the analyzed class. Otherwise, the version-class pair was labeled as not defective.
(3) We determined the observed/actual defectiveness of each version-class pair. To this end, we labeled each version-class pair as defective if the version of the pair was indicated as an AV of at least one defect by JIRA developers themselves, and the defect-fix commit of that defect touched the class. Otherwise, the version-class pair was labeled as not defective.
(4) For each proposed method, we compared its classifications to the observed/actual classification. Next, we computed the TP, FN, TN, FP, Precision, Recall, F1, Matthews, and Kappa metrics across the projects.

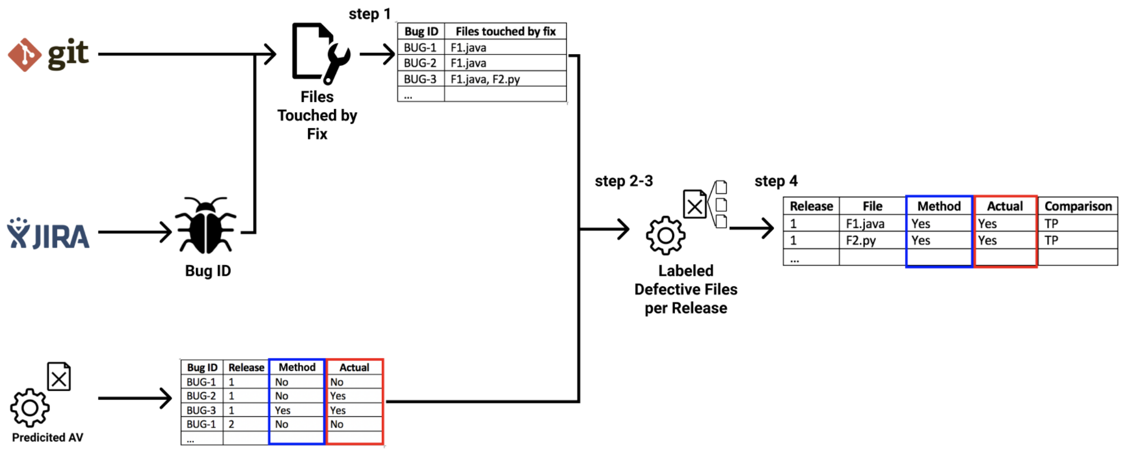

Fig. 4. The process to measure the accuracy of methods in labeling defective classes.

3.4.4 Hypothesis testing. To test hypothesis H_{20}, we use the same statistical machinery used in RQ2. We perform Kruskal-Wallis tests to check whether the distributions are significantly different. We perform Dunn tests to identify the specific pairs of distributions that are significantly different between themselves, and, finally, we perform the Scott-Knott ESD procedure to find the different ranks of significance based on effect-size measurements [58]

RQ4: Do methods lead to selecting different features?

To avoid the introduction of defects, it is important to understand what factors are associated with class defectiveness. Feature selection is the activity of identifying what features contribute the most in a prediction model for predicting whether a class is defective [20, 32]. If a feature is selected as important, then it is strongly associated with the defectiveness of classes. As such, these important features can provide practitioners and researchers with knowledge on how to avoid future defects [20, 71]. However, the noise within defect datasets can lead prediction models to an inaccurate selection of important features. Therefore, in this research question, we investigate the accuracy of methods for leading to accurate feature selection. We propose the following hypothesis:

•H_{30}*: different methods lead to the same level of accuracy for feature selection.*

3.4.5 RQ4: Independent variables. The independent variable is represented by the same methods used in RQ2 (see Section 3.3.1) and RQ3. In RQ4, we use the labeled classes in RQ3 to select features.

3.4.6 RQ4: Dependent Variables. Our dependent variable is the accuracy for selecting features [20, 32]. We compare which features are selected on the same dataset when created by our different studied methods. The following metrics have been redefined for this RQ:

- True Positive(TP): The feature is selected in the actual repository and it is selected in the repository generated by a method.
- False Negative(FN): The feature is selected in the actual repository and it is not selected in the repository generated by a method.

- True Negative(TN): The feature is not selected in the actual repository and it is not selected in the repository generated by a method..
- False Positive(FP): The feature is not selected in the actual repository and it is selected in the repository generated by a method.

As features, to be selected, we used 17 well-defined product and project features that have been shown to be useful for defect prediction [8, 12]. Table 1 details the set of features.

Table 1. Defect prediction features.

Metric	Description
Size	Lines of code(LOC)
LOC Touched	Sum over revisions of LOC added + deleted
NR	Number of revisions
Nfix	Number of bug fixes
Nauth	Number of authors
LOC Added	Sum over revisions of LOC added
MAX LOC Added	Maximum over revisions of LOC added
AVG LOC Added	Average LOC added per revision
Churn	Sum over revisions of added - deleted LOC
Max Churn	Maximum churn over revisions
Average Churn	Average churn over revisions
Change Set Size	Number of files committed together
Max Change Set	Maximum change set size over revisions
Average Change Set	Average change set size over revisions
Age	Age of Release
Weighted Age	Age of Release weighted by LOC touched

3.5 RQ4: Measurement Procedure

For each project we compute the features in Table 1 as shown in Figure 5 and detailed in four steps.

(1) For each project, we begin by removing the last 50% of versions due to the fact that classes snore as described by Ahluwalia et al. [1].
(2) For each project P, we compute the features as described in Table 1 for each version-class pair.
(3) For each of the methods M, we combined their produced AV datasets with the version-class pair's defectiveness (as computed in RQ3), which we labeled as P_M_Complete.
(4) For each version R within a project, we created a dataset including all version-class pairs with versions 1 to R labeled P_M_R_Complete. This dataset uses the defectiveness computed by method M in RQ3.

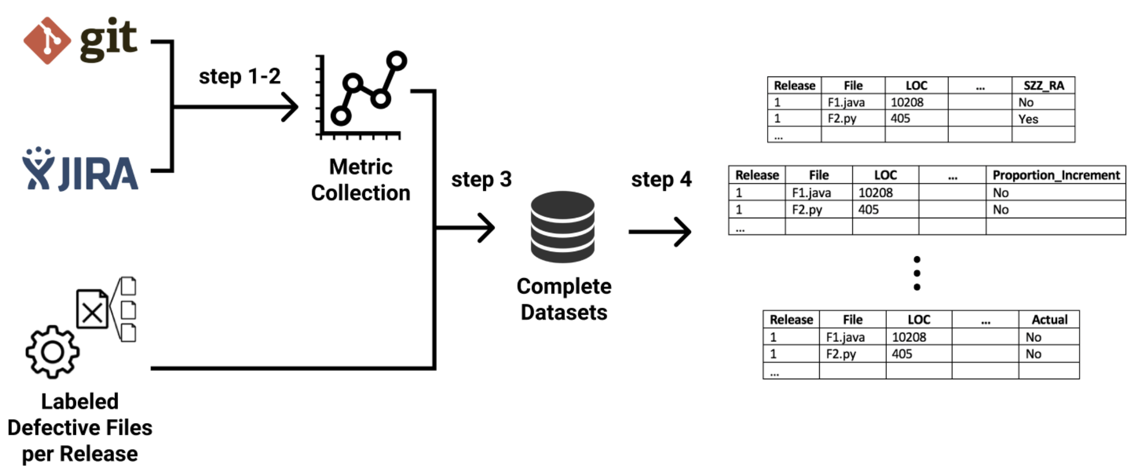

Fig. 5. The process of creating the Complete datasets for each project and method.

Afterwards, we analyze which features are selected. Figure 6 reports an overview of the approach used in this RQ to measure the accuracy of methods for accurate feature selection. This approach consists of three steps:

(1) For each dataset P, for each version R, we set the class defectiveness according to each method M, and we perform on P_M_R an Exhaustive Search Feature Selection[8] using Weka [32, 62]. This search technique performs an exhaustive search through the space of features subsets starting from the empty set of features. If two subsets have the same merit which are also the best merit encountered, then the technique favours the smaller subset. We used CfsSubsetEval[9] for the evaluation function which evaluates the worth of a subset of features by considering the individual predictive ability of each feature along with the degree of redundancy between them. Subsets of features that are highly correlated with the class while having low inter-correlation are preferred [20, 32].
(2) For each dataset P, for each version R, we set the class defectiveness according to the available actual/observed AVs and we perform, on P_Actual_R the Exhaustive Search Feature Selection using Weka and CfsSubsetEval as we did for the studied methods.
(3) For each P_M_R, we compare the features selected in P_M_R to the features selected in P_Actual_R.

3.5.1 Hypothesis testing. To test hypothesis H_{30}, we use the same statistical machinery used in RQ2 and RQ3. We perform Kruskal-Wallis and Dunn tests followed by the Scott-Knott ESD procedure [58]

4 RESULTS

4.1 RQ1: Are AVs available and consistent?

Figure 7 reports the distribution of 212 Apache projects having a specific proportion of defects with an unreliable AV (left side) or without the AV (right side). According to Figure 7, most of the projects have more than 25% of defect reports with no AV. We also measured the total number of closed defect reports linked with git commits in the 212 Apache projects, which resulted to be 125,860. Of these, 63,539 defect reports (51%) resulted in not having or having

[8]https://weka.sourceforge.io/doc.packages/attributeSelectionSearchMethods/
[9]https://weka.sourceforge.io/doc.dev/weka/attributeSelection/CfsSubsetEval.html

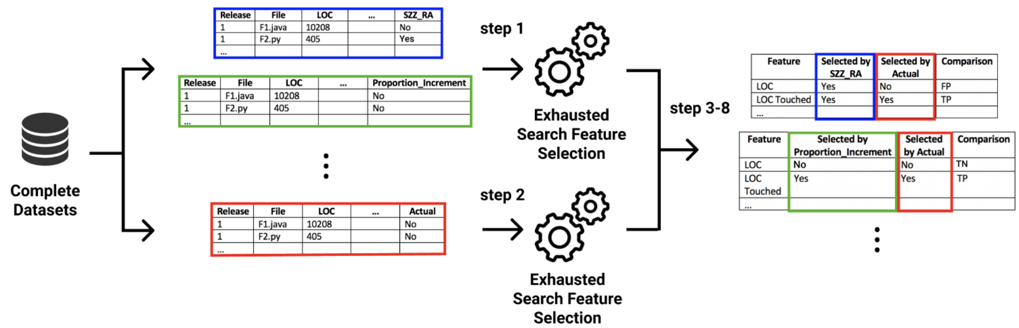

Fig. 6. The process to measure the accuracy of methods in leading to accurate feature selection.

Method	Avg. Precision	Scott–Knott ESD rank
Simple	1.0	1st
Proportion Increment	0.90	2nd
Proportion Cold Start	0.89	2nd
Proportion Moving Window	0.81	3rd
SZZ_B+	0.26	4th
SZZ_B	0.25	4th
SZZ_U+	0.22	5th
SZZ_U	0.22	5th
SZZ_RA+	0.21	5th
SZZ_RA	0.20	5th

Table 2. Scott–Knott results for the *precision* values of hypothesis H_{10}

.

inconsistent AVs. Thus, we can claim that ***in most of defect reports, we cannot use the AVs and, hence, we often need an automated method for retrieving AVs.***

4.2 RQ2: Do methods have different accuracy for labeling affected versions?

Figure 8 reports the distribution, across 76 Apache projects, of Precision, Recall, F1, MCC, and Kappa, of different methods in labeling AV. According to Figure 8:

- All the Proportion methods have a higher Precision and composite accuracy (F1, MCC, and Kappa) than all SZZ methods.
- Simple has a higher Precision and composite accuracy (F1, MCC, and Kappa) than all SZZ methods.
- SZZ_U has the highest Recall than all other methods.
- SZZ_B+ has the highest Precision and the highest composite accuracy (F1, MCC, and Kappa) than any other SZZ method.
- The method with the highest precision is Simple. This is true by definition.
- There is no single dominant method among the Proportion methods. For instance, Proportion_Increment provides the highest Precision, F1 and Kappa and it dominates Proportion_ColdStart. Proportion_MovingWindow provides the highest Recall (among Proportion methods) and MCC.

Fig. 7. Distribution of 212 Apache projects having a specific proportion of defect reports with an unreliable AV (left side) or without the AV (right side).

Method	**Avg. Recall**	**Scott–Knott ESD rank**
SZZ_U+	0.97	1st
SZZ_U	0.97	1st
SZZ_RA+	0.94	2nd
SZZ_RA	0.91	3rd
SZZ_B+	0.89	4th
SZZ_B	0.85	5th
Proportion Moving Window	0.78	6th
Proportion Increment	0.73	7th
Proportion Cold Start	0.72	7th
Simple	0.30	8th

Table 3. Scott–Knott results for the *recall* values of hypothesis H_{10}

.

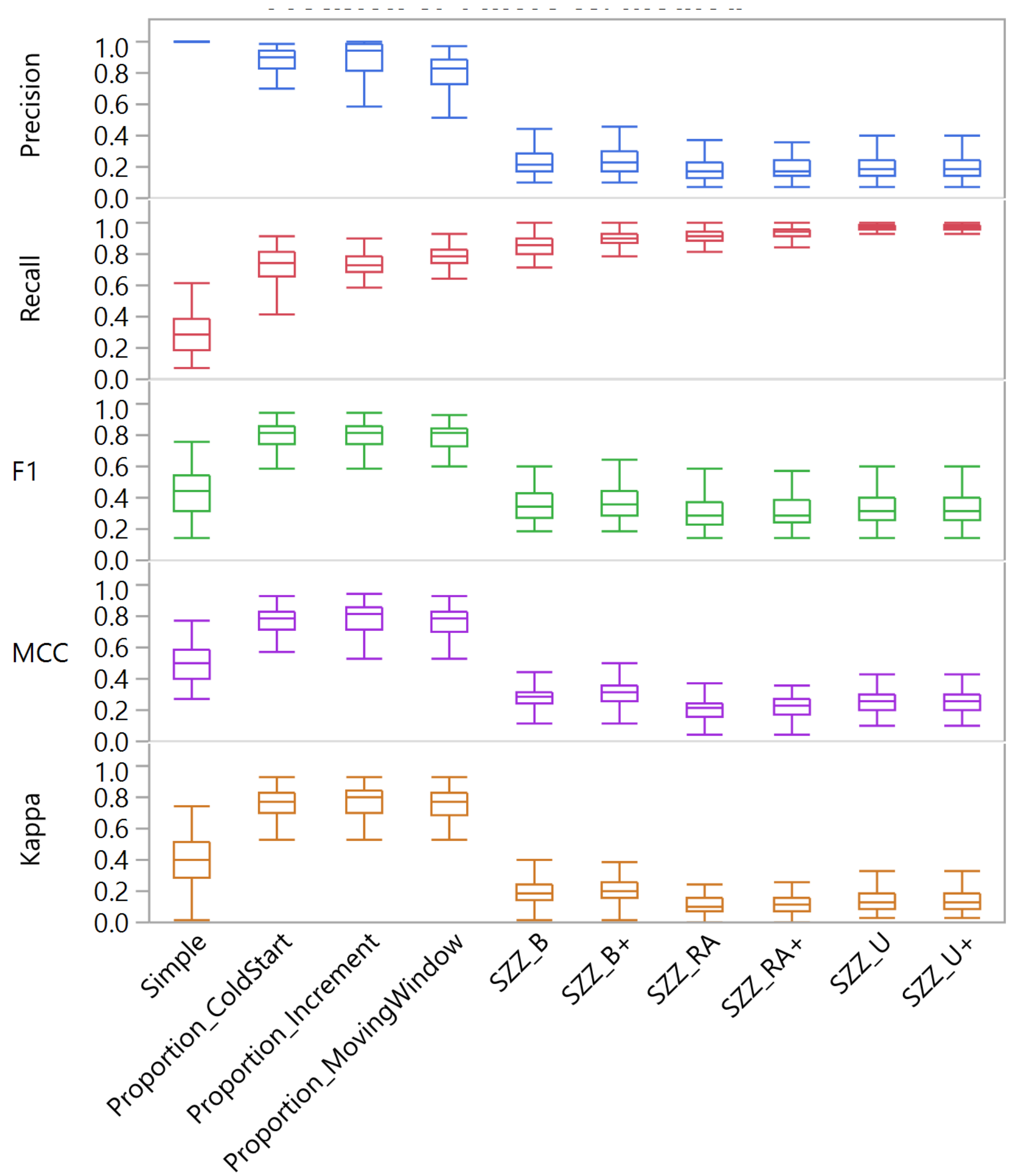

Fig. 8. Distribution, across 76 Apache projects, of Precision, Recall, F1, MCC, and Kappa, of different methods in labeling AV.

Our statistical results on the 76 Apache projects reveal that the differences between our studied methods in terms of our accuracy metrics are statistically significant for H_{10} (i.e., our Kruskall–Wallis and Dunn tests have $p-values < 0.05$). Therefore, our results reveal that the proportional methods have significantly better accuracy values compared to the studied SZZ based methods. [10] Tables 2, 3, and 4 show our Scott-Knott results for the precision, recall, and F1 metrics. According to our observations, the *Proportion Increment*, *Proportion Moving Window*, and *Proportion Cold Start* methods

[10] All of our statistical results are available in our online appendix: https://zenodo.org/deposit/3722782

Method	Avg. F1	Scott–Knott ESD rank
Proportion Increment	0.80	1st
Proportion Moving Window	0.80	1st
Proportion Cold Start	0.79	1st
Simple	0.44	2nd
SZZ_B+	0.39	3rd
SZZ_B	0.37	4th
SZZ_U+	0.35	4th
SZZ_U	0.35	4th
SZZ_RA+	0.32	5th
SZZ_RA	0.32	5th

Table 4. Scott–Knott results for the *f1* values of hypothesis H_{10}

.

Method	Avg. Precision	Scott–Knott ESD rank
Simple	1.0	1st
Proportion Increment	0.91	2nd
Proportion Cold Start	0.90	2nd
Proportion Moving Window	0.86	3rd
SZZ_B+	0.38	4th
SZZ_B	0.37	4th
SZZ_RA+	0.37	4th
SZZ_RA	0.37	4th
SZZ_U+	0.36	4th
SZZ_U	0.36	4th

Table 5. Scott–Knott results for the *precision* values of hypothesis H_{20}

.

are all in the 1st rank in terms of F1. Interestingly, even the *Simple* method (which fell in the 2nd rank, see Table 4) outperforms all the SZZ based methods, significantly.

4.3 RQ3: Do methods have different accuracy for labeling defective classes?

Figure 9 reports the distribution, across 76 Apache projects, of Precision, Recall, F1, MCC, and Kappa, of different methods for labeling defective classes. According to Figure 9:

- All the proportional methods have a higher Precision and composite accuracy (F1, MCC, and Kappa) compared to all SZZ methods. Therefore, we can claim that **labeling classes using defects' life cycle information is in overall and, in average, more accurate than the studied SZZ methods.**
- SZZ_U has the highest Recall than all other methods.
- SZZ_B+ has a highest Precision and lower Recall than any other SZZ method.
- SZZ_B+ has a higher composite accuracy (F1, MCC, and Kappa) than Simple and any other SZZ method.
- **The Proportion_MovingWindow method dominates all methods on all composite accuracy (F1, MCC, and Kappa)**.

Our statistical results (i.e., Kruskal-Wallis and Dunn's tests) reveal that hypothesis H_{20} can be rejected. Therefore, our life cycle based methods for labeling defective classes outperform the studied SZZ based methods, significantly—in

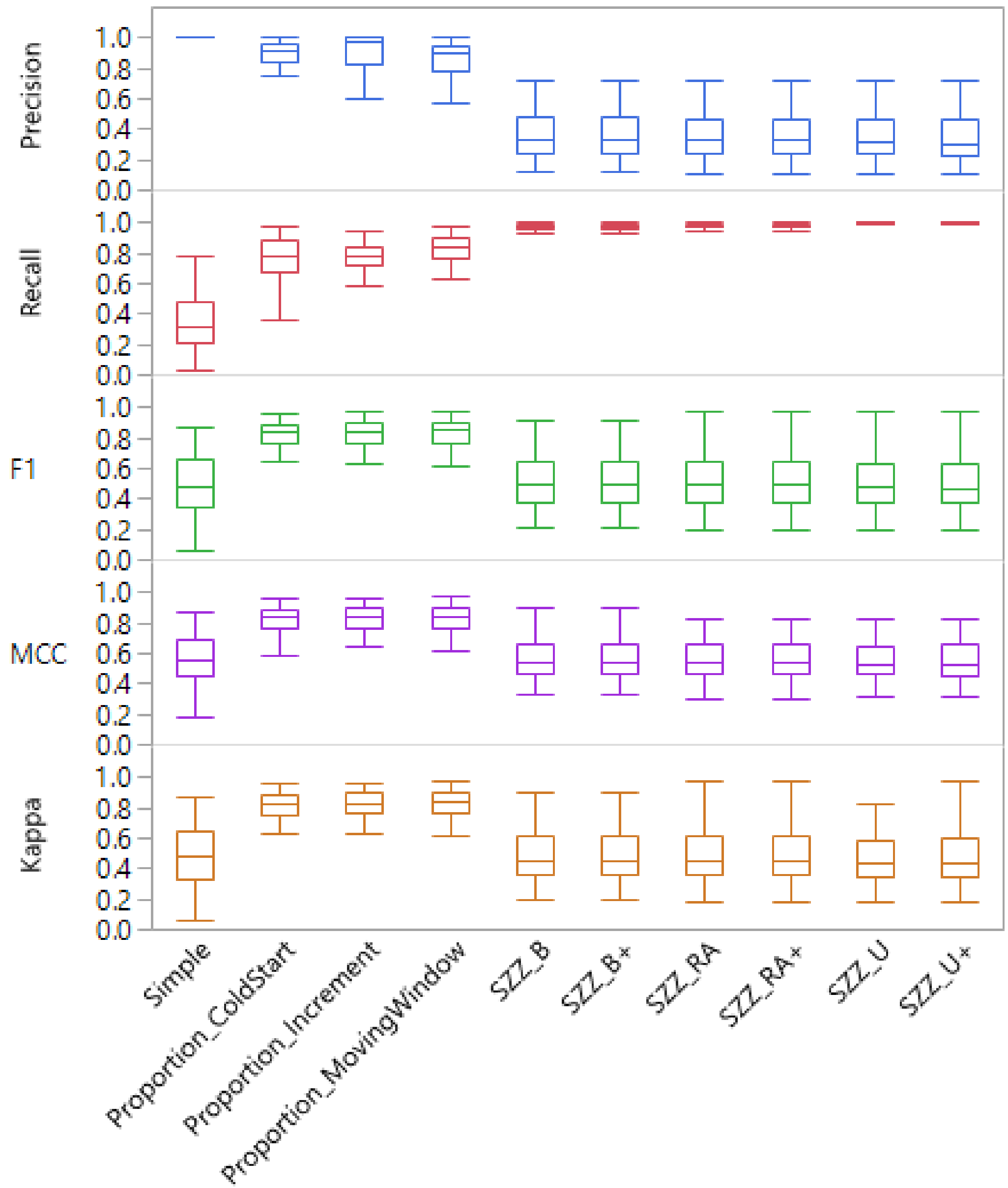

Fig. 9. Distribution, across 76 Apache projects, of Precision, Recall, F1, MCC, and Kappa, of different methods in labeling defective classes.

terms of our studied accuracy metrics, i.e., precision, recall, F1, kappa, and MCC. Tables 5, 6, and 7 show our results after performing the Scott–Knott ESD procedure. Indeed, we observe that all the proportional methods fall in the 1st rank of $F1$, whereas the SZZ based methods and the *Simple* method fall in the 2nd rank of $F1$, obtaining a significantly lower performance.

Method	Avg. Recall	Scott–Knott ESD rank
SZZ_U+	0.99	1st
SZZ_U	0.99	1st
SZZ_B+	0.97	2nd
SZZ_RA+	0.97	2nd
SZZ_RA	0.97	2nd
SZZ_B	0.96	3rd
Proportion Moving Window	0.82	4th
Proportion Increment	0.76	5th
Proportion Cold Start	0.76	5th
Simple	0.36	6th

Table 6. Scott–Knott results for the *recall* values of hypothesis H_{20}

.

Method	Avg. F1	Scott–Knott ESD rank
Proportion Moving Window	0.83	1st
Proportion Increment	0.82	1st
Proportion Cold Start	0.81	1st
SZZ_B+	0.52	2nd
SZZ_B	0.51	2nd
SZZ_RA	0.51	2nd
SZZ_RA+	0.51	2nd
SZZ_U+	0.51	2nd
SZZ_U	0.51	2nd
Simple	0.50	2nd

Table 7. Scott–Knott results for the $F1$ values of hypothesis H_{20}

.

4.4 RQ4: Do methods lead to selecting different features?

Fig. 10 reports the distribution among datasets of the actual selection frequency of each feature. Since the frequency of selection varies across features, then it is important to select the correct set of features.

Regarding the comparison of the features selected on a dataset produced by a method (i.e., where the AVs are those retrieved by a method) versus the features selected by using the actual/observed dataset (i.e., where the AV are those provided be developers), Figure 11 reports the distribution of a certain method to retrieve AVs (x-axis, across versions and 76 Apache projects) of Precision, Recall, F1, MCC, and Kappa for selecting features. According to Figure 11, the proportional methods have a higher accuracy (in all five metrics) compared to the studied SZZ methods. For example, according to 11 the proportional methods are the only methods having a perfect median Precision and Recall.

Indeed, our statistical tests reveal that hypothesis H_{30} (i.e., different methods have the same accuracy when selecting features) can be rejected. Therefore, we can claim that **retrieving AVs based on the defects' life cycle can lead to an overall, and on average, more accurate feature selection than the studied SZZ methods.**

Tables 8, 9, and 10 show the Scott-Knott results. Indeed, we observe that, in terms of $F1$, the proportional based methods fall in the 1st rank, whereas the SZZ based methods and the Simple method fall in the 2nd, obtaining a much lower average $F1$. All of the results for our statistical tests are available in our online appendix.[11]

[11] https://zenodo.org/deposit/3722782

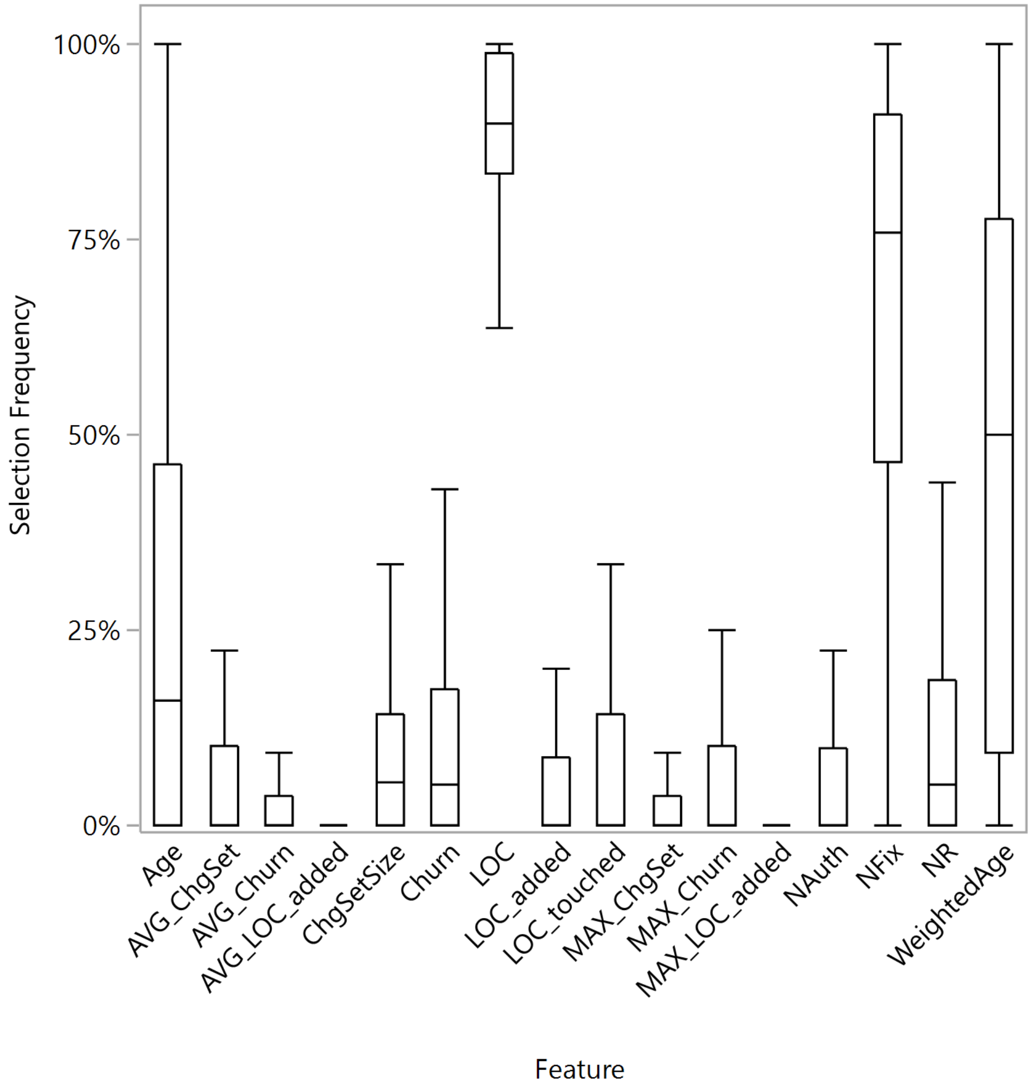

Fig. 10. Distribution among datasets of selection frequency of each feature.

5 DISCUSSION

This sections discuss our main results and their possible causes and implications.

5.1 RQ1 Are AVs available and consistent?

The main result of RQ1 is that most of defect reports in Apache projects do not provide AVs. However, according to Figure 7, the median project has most of the defects providing AV. This means that in projects having a higher number of defects there is a higher proportion of missing AVs compared to projects having a small number of defects. The main

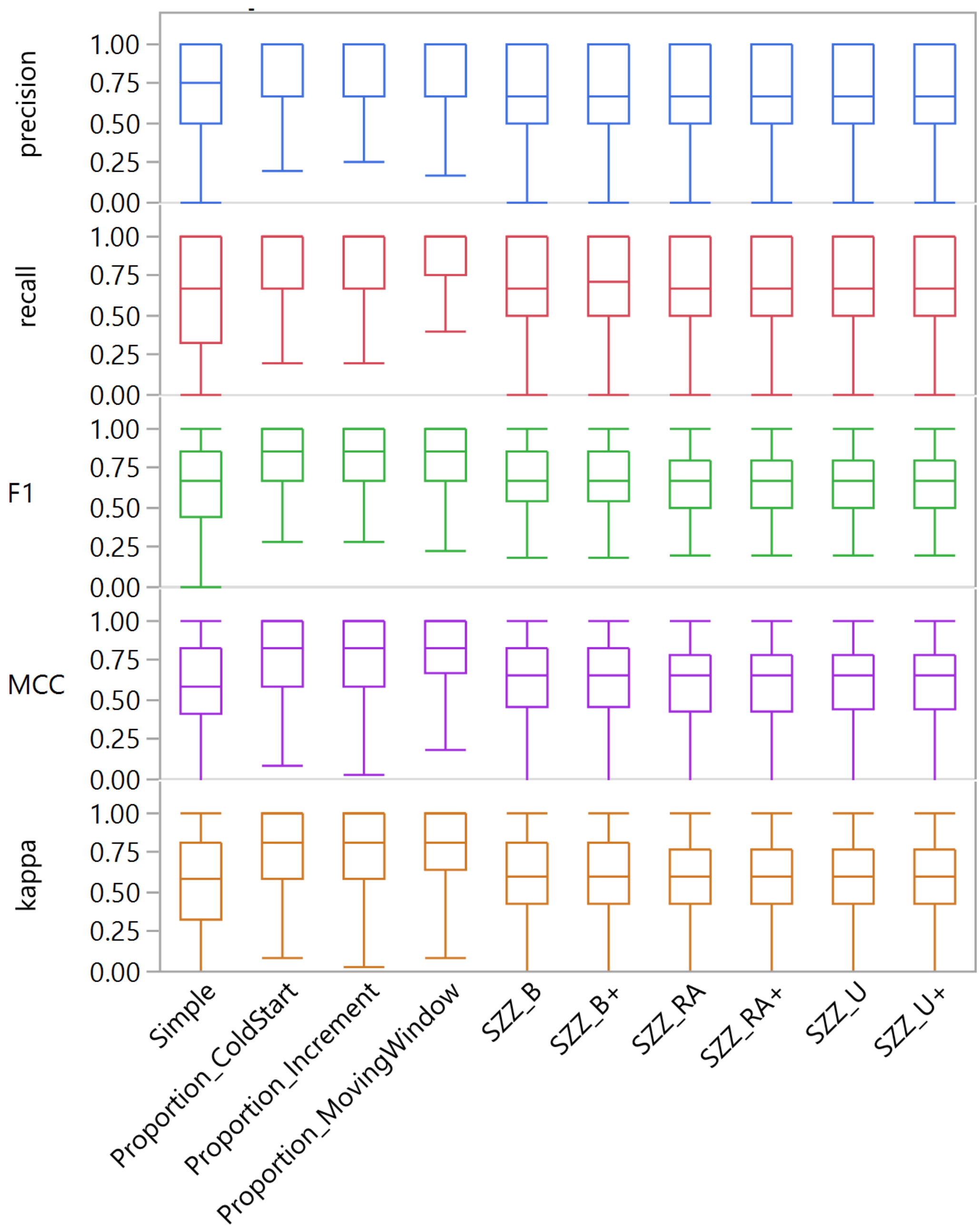

Fig. 11. Distribution, across versions and 76 Apache projects, of Precision, Recall, F1, MCC, and Kappa, of different methods in feature selection.

Method	Avg. Precision	Scott–Knott ESD rank
Proportion Moving Window	0.84	1st
Proportion Cold Start	0.83	1st
Proportion Increment	0.82	1nd
Simple	0.74	2nd
SZZ_B+	0.70	3th
SZZ_B	0.70	3th
SZZ_U+	0.69	3th
SZZ_U	0.69	3th
SZZ_RA	0.69	3th
SZZ_RA+	0.68	3th

Table 8. Scott–Knott results for the *precision* values of hypothesis H_{30}

.

Method	Avg. Recall	Scott–Knott ESD rank
Proportion Moving Window	0.84	1st
Proportion Cold Start	0.82	1st
Proportion Increment	0.81	1st
SZZ_B+	0.71	2nd
SZZ_B	0.71	2nd
SZZ_RA	0.70	2nd
SZZ_U	0.70	2nd
SZZ_RA+	0.70	2nd
SZZ_U+	0.70	2nd
Simple	0.61	3rd

Table 9. Scott–Knott results for the *recall* values of hypothesis H_{30}

Method	Avg. F1	Scott–Knott ESD rank
Proportion Moving Window	0.83	1st
Proportion Cold Start	0.81	1st
Proportion Increment	0.80	1st
SZZ_B+	0.38	2nd
SZZ_B	0.37	2nd
SZZ_RA	0.37	2nd
SZZ_U	0.36	2nd
SZZ_U+	0.36	2nd
SZZ_RA+	0.37	2nd
Simple	1.0	2nd

Table 10. Scott–Knott results for the $F1$ values of hypothesis H_{30}

.

implications of RQ1 is that relying on the available AVs means neglecting most of the defects. Therefore, effort should be spent in estimating AV (hence the importance of our work).

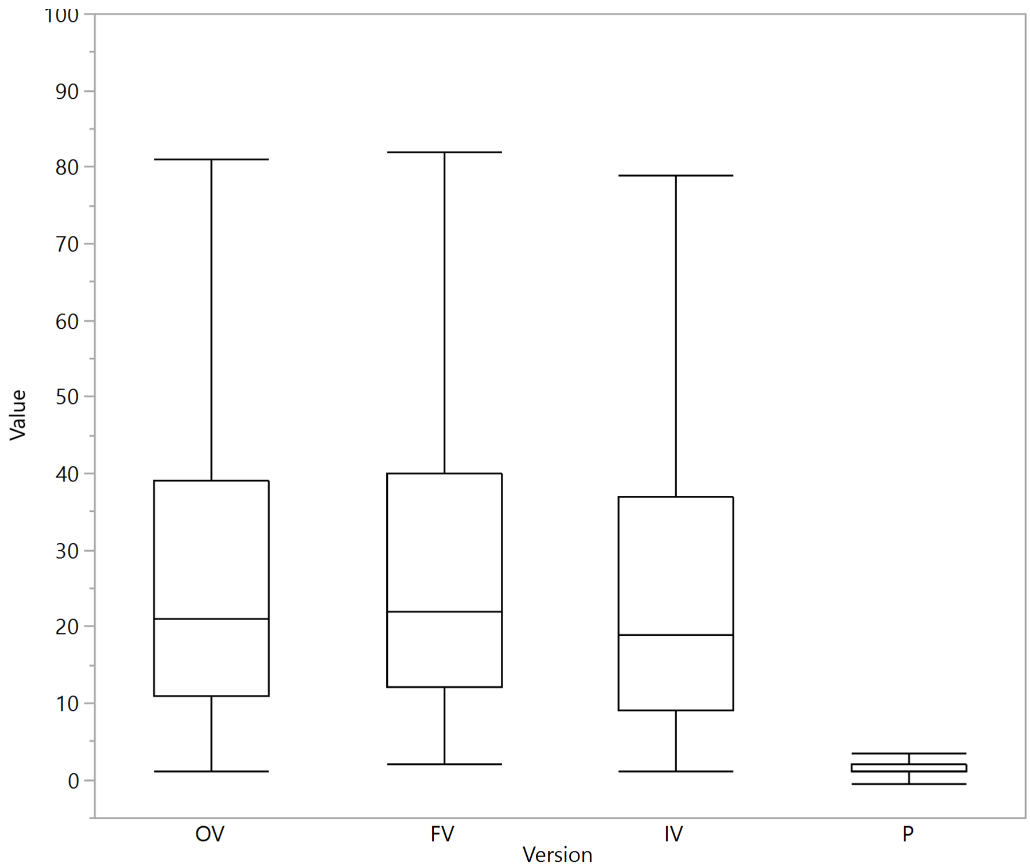

Fig. 12. Distribution of values of IV, OV, FV, and P across defects of 76 Apache projects.

Table 11. Variation, in terms of standard deviation, of IV, OV, FV, and P across defects of 76 Apache projects.

Version	STDV
IV	38.36
OV	40.17
FV	41.85
P	5.43

5.2 RQ2 Do methods have different accuracy for labeling affected versions?

The main result of RQ2 is that all proportional methods have a higher Precision and composite accuracy (F1, MCC, and Kappa) than all SZZ methods. One of the possible reasons for the high accuracy achieved by the proportional methods is that P is substantially stable across projects (i.e., Proportion_ColdStart) and more stable within the same project (i.e., Proportion_Increment and Proportion_MovingWindow). Figure 12 reports the distribution of values of IV, OV, FV, and P across defects of different projects. Table 11 reports the variation, in terms of standard deviation, of IV, OV, FV, and P in case it is computed across different projects. According to both Figure 12 and Table 11, P is substantially stable across defects of different projects especially when compared to IV, OV and FV.

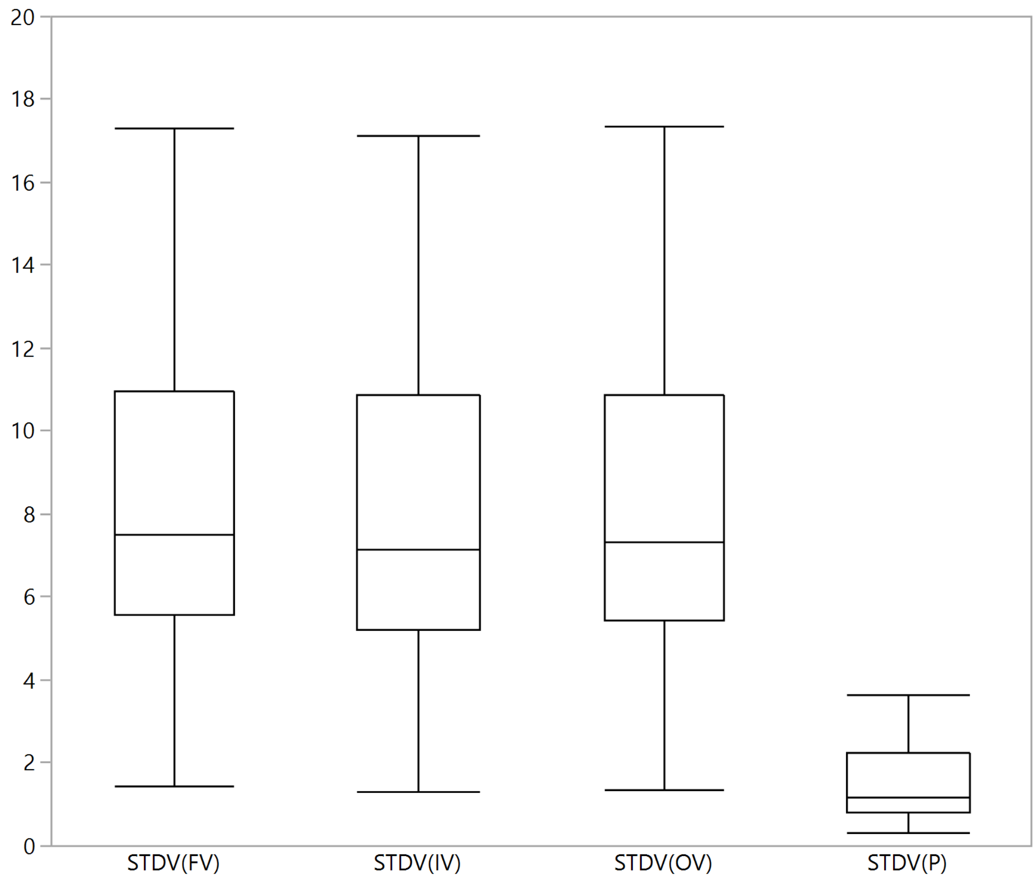

Fig. 13. Distribution of standard deviation of IV, OV, FV, and P, as computed within the same project, across 76 Apache projects.

The main implications of RQ2 to practitioners is that SZZ should be used over proportional methods only in cases where higher Recall values are preferred over Precision, F1, MCC, and Kappa values. As for researchers, the results of RQ2 highlights exciting possibilities for future work in the area of defect prediction. For example, the framework proposed by da Costa et al. [7] to evaluate SZZ implementations can be enhanced with the AVs retrieved by our proportional methods.

5.3 RQ3 Do methods have different accuracy for labeling defective classes?

The main results of RQ3 is that Proportion_MovingWindow method dominates all methods on all composite accuracy metrics (i.e., F1, MCC, and Kappa). This results is likely due to the fact that P is more stable within the same project than across projects. Figure 13 reports the distribution of standard deviation of IV, OV, FV, and P, across 76 Apache projects. According to Figure 13 the STDV is much higher across projects than within the same project. Specifically, the median STDV of P computed within the same project is less than 2 (Figure 13) whereas the one across projects is about 5 (Table 11). In conclusion, the high stability reported in Figure 7, Table 11 and Figure 13 shows that **the proportion of number of versions between its discovery and its fix is more stable within the same project than across different projects.**

The main implications of RQ3 to practitioners and researchers is to use SZZ over Proportion methods only in case they are interested in Recall over Precision, and composite accuracy metrics.

5.4 RQ4 Do methods lead to selecting different features?

The main results of RQ4 is that the proportional methods have a higher accuracy (in all five metrics) than all SZZ methods. Moreover, by observing Figure 11 we note that the accuracy of SZZ methods can reach even a negative value of kappa. This means that a random selection of features is more accurate than a selection based on a dataset produced by an SZZ method.

The main implications of RQ4 results to practitioners and researchers is to prefer using the Proportion over SZZ methods when performing feature selection.

5.5 Merged results

RQ2, RQ3, and RQ4 share several results including that all proportion methods have a higher Precision and composite accuracy than all SZZ methods.

The major differences among RQ2 and RQ3 results is in that SZZ_B+ has a higher composite accuracy than Simple and any other SZZ method. One possible reason is that, in RQ3, a class can be affected by multiple defects and therefore methods which miss defects can still perform accurately (see discussion in Section 3.3.2). By comparing Figure 9 to Figure 8 we observe that, all methods are more accurate for labeling classes (RQ3) than AVs (RQ2) on all accuracy metrics. Specifically, by comparing the median accuracy (across methods and datasets), we observe an increase in labeling classes over AVs of 13% in Precision, 5% in Recall, 16% in F1, 27% in MCC and 39% in Kappa. It is interesting to note that the increase is higher in composite accuracy metrics than in atomic metrics. Again, we believe that one of the possible reasons is that, in RQ3, a class can be affected by multiple defects and, therefore, methods which miss defects can still perform accurately.

By comparing RQ4 to RQ2 and RQ3 we observe that there is less variation among accuracy of methods in RQ4 than in RQ2 or RQ3. In other words, the choice of the methods to retrieve AV has less impact on feature selection (RQ4) than on class labeling (RQ3). However, in RQ2 and RQ3 the proportional methods performed better than the other methods in four and three metrics, respectively, whereas in five metrics in RQ4. In other words, the superiority of the proportional methods is clearer in feature selection (RQ4) than in class labeling (RQ3). Another major difference between RQ4 to RQ2 and RQ3 is that the distribution of accuracy are much wider in RQ4 than in RQ2 or RQ3. For instance, when the methods are inaccurate they are extremely less accurate in RQ4 than in RQ2 or RQ3. In other words, a medium amount of inaccuracy in class labeling leaded to a big amount of inaccuracy in feature selection. For example, the lowest score of Proportion_MovingWindow in feature selection in Kappa, F1, Precision and Recall is less than 0.25 in RQ4 but higher than 0.6 in RQ3. Similarly the lowest scores of SZZ methods are even negative in case of Kappa for feature selection (RQ4) and higher than 0.2 in RQ3.

6 THREATS TO VALIDITY

In this section, we report the threats to validity related to our study. The description is organized by threat type, i.e., Conclusion, Internal, Construct, and External.

6.1 Conclusion

Conclusion validity concerns issues that affect the ability to draw accurate conclusions about the observed relationships between the independent and dependent variables [63].

We tested all hypotheses with non-parametric tests (e.g., Kruskal–Wallis) which are prone to type-2 error, i.e,. not rejecting a false hypothesis. We have been able to reject the hypotheses in most of the cases; therefore, the likelihood of a type-2 error is low. Moreover, the alternative would have been using parametric tests (e.g., ANOVA) which are prone to type-1 error, i.e., rejecting a true hypothesis, which in our context is less desirable than type-2 error. Also, we acknowledge that our proposed methods (i.e., independent variables) do not represent an exhaustive list of methods that could have been implemented (for example, one could use machine learning to optimize the proportions used in the ColdStart method). However, our proposed methods are a simple and effective baseline to start with (as shown by our obtained results).

6.2 Internal

Internal validity is concerned with the influences that can affect the independent variables with respect to causality [63]. A threat to *internal validity* is the lack of ground truth for class defectiveness, which could have been underestimated in our measurements. In other words, the AVs provided by developers might be inaccurate due to human error. Nevertheless, we would argue that this is a common threat in most of empirical research in the area of software engineering [26].

6.3 Construct

Construct validity is concerned with the degree to which our measurements indeed reflect what we claim to measure [63].

In our study, we compare our proposed life cycle methods with the SZZ approaches. We are aware that the output of SZZ are defect-introducing changes and not affected versions. For example, although SZZ may output three distinct bug-introducing changes (which we may interpret as three distinct affected versions), we do not investigate the dependency between these defect-introducing changes. For instance, a defect may only be present when all the three defect-introducing changes are present. Therefore, a version that contains only one of the defect-introducing changes may not be, in actuality, an affected version. Nevertheless, our assumptions are aligned with prior work, which has considered every defect-introducing change as indeed defect-introducing [7] and, therefore, can be interpreted as incurring an affected version.

In addition, we use Precision, Recall, F1-Score, Matthews Correlation Coefficient, and Cohen's Kappa to measure the accuracy for labeling defectiveness in RQ2 and RQ3. Although we do not use the *Area Under the Curve* (AUC) metric, which is a threshold-free metric [59], our methods do not output probabilities. Therefore, our evaluations are not impacted by threshold choices.

6.4 External

External validity is concerned with the extent to which the research elements (subjects, artifacts, etc.) are representative of actual elements [63].

This study used a large set of datasets and hence could be deemed of high generalization compared to similar studies. Of course, our results cannot be generalized by projects that would significantly differ from the settings used in this present study.

Finally, in order to promote reproducible research, all datasets, results and scripts for this paper are available in our online appendix [12].

7 CONCLUSION

In this paper we first measured the AV availability and consistency in open-source projects, and then evaluated a new method for retrieving AVs, i.e., the origin of a defect, which is based on the idea that defects have a stable life cycle in terms of proportion of number of versions required to discover and to fix the defect. Results related to 212 open-source projects from the Apache ecosystem, featuring a total of about 125,000 defects, show that the AVs cannot be used in the majority (51%) of defect reports. Therefore, it is important to develop automated methods to retrieve AVs. Results related to 76 open-source projects from the Apache ecosystem, featuring a total of about 6,250,000 classes that are are affected by 60,000 defects and spread over 4,000 versions and 760,000 commits, show that our proposed methods are, on average, more accurate when compared to previously proposed and state-of-art SZZ based methods, for retrieving AVs. Our results suggest that our proposed methods are also better than SZZ based methods for labeling classes as defective and for developing defects repositories to perform feature selection. In conclusion, our proposed methods are a valid automated alternative to SZZ for estimating the origin of a defect and hence for building defects repository for defect prediction endeavours.

Future studies include:

- **Analyzing other bug-introducing commits in SZZ methods**. In our research, we selected the earliest possible bug-introducing commit returned by SZZ to be the IV for a defect. Future work will focus on how selecting later bug-introducing commits affects the accuracy in labeling classes in versions as defective or not.
- **Analyzing the role of reporting affect versions to developers**. In our study, we only analyzed whether AV were available and consistent. Future work will focus on why and how developers report AV; how do developers determine AV? Do developers find reporting AV important?
- **Replication in context of JIT**. Just In Time (JIT) prediction models, where the predicted variable is the defectiveness of a commit, have become sufficiently robust that they are now incorporated into the development cycle of some companies[38]. Therefore, it is important to investigate the accuracy of proportion in the context of JIT models.
- **Finer combination of Proportion and SZZ methods**. In this work we have combined SZZ and proportion method by simply tagging a version as defective if it came after the ticket creation and not tagged by SZZ. More finer combination are possible including the use of ML; i.e., the dataset to evaluate and use ML models can be created by ML models.
- **Use a finer P**. In this work, we simply used the proportion of versions to find and to fix a defect to determine P which is then used to label AV and classes. However, there is room for improvement in calculating P. For example, P can be improved using Linear Regression. In addition to the version information, the number of days can also be used.

REFERENCES

[1] Aalok Ahluwalia, Davide Falessi, and Massimiliano Di Penta. 2019. Snoring: a Noise in Defect Prediction Datasets. In *Proceedings of the 16th International Conference on Mining Software Repositories.*

[12] https://zenodo.org/deposit/3722782

[2] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. 1996. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering* 22, 10 (1996), 751–761.

[3] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. 2009. Fair and Balanced?: Bias in Bug-fix Datasets *(ESEC/FSE '09)*. ACM, New York, NY, USA, 121–130. https://doi.org/10.1145/1595696.1595716

[4] Markus Borg, Oscar Svensson, Kristian Berg, and Daniel Hansson. 2019. SZZ Unleashed: An Open Implementation of the SZZ Algorithm - Featuring Example Usage in a Study of Just-in-Time Bug Prediction for the Jenkins Project. *CoRR* abs/1903.01742 (2019). arXiv:1903.01742 http://arxiv.org/abs/1903.01742

[5] Tse-Hsun Chen, Meiyappan Nagappan, Emad Shihab, and Ahmed E. Hassan. 2014. An empirical study of dormant bugs. *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014* (2014). https://doi.org/10.1145/2597073.2597108

[6] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[7] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E. Hassan. 2017. A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes. *IEEE Trans. Software Eng.* 43, 7 (2017), 641–657.

[8] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2012. Evaluating Defect Prediction Approaches: A Benchmark and an Extensive Comparison. *Empirical Softw. Engg.* 17, 4-5 (Aug. 2012), 531–577. https://doi.org/10.1007/s10664-011-9173-9

[9] Olive Jean Dunn. 1964. Multiple comparisons using rank sums. *Technometrics* 6, 3 (1964), 241–252.

[10] Jon Eyolfson, Lin Tan, and Patrick Lam. 2011. Do time of day and developer experience affect commit bugginess?. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 153–162.

[11] Davide Falessi and Max Jason Moede. 2018. Facilitating feasibility analysis: the pilot defects prediction dataset maker. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics, SWAN@ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 5, 2018.* 15–18.

[12] Davide Falessi, Barbara Russo, and Kathleen Mullen. 2017. What if I Had No Smells? *(ESEM '17)*. IEEE Press, Piscataway, NJ, USA, 78–84. https://doi.org/10.1109/ESEM.2017.14

[13] Davide Falessi, Wyatt Smith, and Alexander Serebrenik. 2017. STRESS: A Semi-automated, Fully Replicabile Approach for Project Selection *(ESEM '17)*. IEEE Press, Piscataway, NJ, USA, 151–156.

[14] Yuanrui Fan, Xin Xia, Daniel Alencar da Costa, David Lo, Ahmed E Hassan, and Shanping Li. 2019. The Impact of Changes Mislabeled by SZZ on Just-in-Time Defect Prediction. *IEEE Transactions on Software Engineering* (2019).

[15] Wei Fu, Tim Menzies, and Xipeng Shen. 2016. Tuning for software analytics: Is it really necessary? *Information & Software Technology* 76 (2016), 135–146.

[16] Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. 2014. An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 172–181.

[17] Kehan Gao, Taghi M. Khoshgoftaar, Huanjing Wang, and Naeem Seliya. 2011. Choosing software metrics for defect prediction: an investigation on feature selection techniques. *Softw., Pract. Exper.* 41, 5 (2011), 579–606.

[18] Baljinder Ghotra, Shane McIntosh, and Ahmed E Hassan. 2017. A large-scale study of the impact of feature selection techniques on defect classification models. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 146–157.

[19] Georgios Gousios and Diomidis Spinellis. 2014. Conducting quantitative software engineering studies with Alitheia Core. *Empirical Software Engineering* 19, 4 (2014), 885–925.

[20] M. A. Hall. 1998. *Correlation-based Feature Subset Selection for Machine Learning.* Ph.D. Dissertation. University of Waikato, Hamilton, New Zealand.

[21] Maurice Howard Halstead et al. 1977. *Elements of software science*. Vol. 7. Elsevier New York.

[22] Kim Herzig, Sascha Just, and Andreas Zeller. 2013. It's not a bug, it's a feature: how misclassification impacts bug prediction *(ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 392–401. http://dl.acm.org/citation.cfm?id=2486788.2486840

[23] Sture Holm. 1979. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics* (1979), 65–70.

[24] Tian Jiang, Lin Tan, and Sunghun Kim. 2013. Personalized defect prediction. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Ieee, 279–289.

[25] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E Hassan. 2016. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering* 21, 5 (2016), 2072–2106.

[26] Yasutaka Kamei and Emad Shihab. 2016. Defect prediction: Accomplishments and future challenges. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 5. IEEE, 33–45.

[27] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2012. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2012), 757–773.

[28] Sunghun Kim, E James Whitehead Jr, and Yi Zhang. 2008. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering* 34, 2 (2008), 181–196.

[29] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. 2011. Dealing with Noise in Defect Prediction *(ICSE '11)*. ACM, New York, NY, USA, 481–490.

[30] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. 2006. Automatic Identification of Bug-Introducing Changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE '06)*. IEEE Computer Society, Washington, DC, USA, 81–90. https://doi.org/10.1109/ASE.2006.23

[31] Pavneet Singh Kochhar, Yuan Tian, and David Lo. 2014. Potential Biases in Bug Localization: Do They Matter? *(ASE '14)*. ACM, New York, NY, USA, 803–814.

[32] Masanari Kondo, Cor-Paul Bezemer, Yasutaka Kamei, Ahmed E Hassan, and Osamu Mizuno. 2019. The impact of feature reduction techniques on defect prediction models. *Empirical Software Engineering* 24, 4 (2019), 1925–1963.

[33] William H Kruskal and W Allen Wallis. 1952. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association* 47, 260 (1952), 583–621.

[34] Guillermo Macbeth, Eugenia Razumiejczyk, and Rubén Daniel Ledesma. 2011. Cliff's Delta Calculator: A non-parametric effect size program for two groups of observations. *Universitas Psychologica* 10, 2 (2011), 545–555.

[35] Janaki T Madhavan and E James Whitehead Jr. 2007. Predicting buggy changes inside an integrated development environment. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*. 36–40.

[36] Borg Markus, Svensson Oscar, Berg Kristian, and Hansson Daniel. 2019. SZZ Unleashed: An Open Implementation of the SZZ Algorithm - Featuring Example Usage in a Study of Just-in-Time Bug Prediction for the Jenkins Project. (2019).

[37] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[38] Shane McIntosh and Yasutaka Kamei. 2018. Are Fix-Inducing Changes a Moving Target? A Longitudinal Case Study of Just-In-Time Defect Prediction. *IEEE Trans. Software Eng.* 44, 5 (2018), 412–428.

[39] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for Engineered Software Projects. *Empirical Softw. Engg.* 22, 6 (Dec. 2017), 3219–3253. https://doi.org/10.1007/s10664-017-9512-6

[40] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. 2013. Diversity in Software Engineering Research *(ESEC/FSE 2013)*. ACM, New York, NY, USA, 466–476.

[41] Jaechang Nam, Wei Fu, Sunghun Kim, Tim Menzies, and Lin Tan. 2017. Heterogeneous defect prediction. *IEEE Transactions on Software Engineering* 44, 9 (2017), 874–896.

[42] Jaechang Nam and Sunghun Kim. 2015. Clami: Defect prediction on unlabeled datasets (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 452–463.

[43] Edmilson Campos Neto, Daniel Alencar da Costa, and Uirá Kulesza. 2018. The impact of refactoring changes on the SZZ algorithm: An empirical study. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*. 380–390. https://doi.org/10.1109/SANER.2018.8330225

[44] Annibale Panichella, Carol V Alexandru, Sebastiano Panichella, Alberto Bacchelli, and Harald C Gall. 2016. A search-based training algorithm for cost-aware defect prediction. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. 1077–1084.

[45] Foyzur Rahman and Premkumar Devanbu. 2013. How, and why, process metrics are better. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 432–441.

[46] Foyzur Rahman, Daryl Posnett, Israel Herraiz, and Premkumar Devanbu. 2013. Sample Size vs. Bias in Defect Prediction *(ESEC/FSE 2013)*. ACM, New York, NY, USA, 147–157.

[47] Izabela A Rodenhuis-Zybert, Jan Wilschut, and Jolanda M Smit. 2010. Dengue virus life cycle: viral and host factors modulating infectivity. *Cellular and molecular life sciences* 67, 16 (2010), 2773–2786.

[48] Gema Rodríguez-Pérez, Gregorio Robles, and Jesús M. González-Barahona. 2018. Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the SZZ algorithm. *Information & Software Technology* 99 (2018), 164–176.

[49] Gema Rodríguez-Pérez, Gregorio Robles, Alexander Serebrenik, Andy Zaidman, Daniel M Germán, and Jesus M Gonzalez-Barahona. [n.d.]. How bugs are born: a model to identify how bugs are introduced in software components. *Empirical Software Engineering* ([n. d.]), 1–47.

[50] Gema Rodríguez-Pérez, Andy Zaidman, Alexander Serebrenik, Gregorio Robles, and Jesús M. González-Barahona. 2018. What if a Bug Has a Different Origin?: Making Sense of Bugs Without an Explicit Bug Introducing Change *(ESEM '18)*. ACM, New York, NY, USA, Article 52, 4 pages. https://doi.org/10.1145/3239235.3267436

[51] Daniel Rozenberg, Ivan Beschastnikh, Fabian Kosmale, Valerie Poser, Heiko Becker, Marc Palyart, and Gail C. Murphy. 2016. Comparing repositories visually with repograms. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*. 109–120.

[52] Martin Shepperd, Qinbao Song, Zhongbin Sun, and Carolyn Mair. 2013. Data Quality: Some Comments on the NASA Software Defect Datasets. *IEEE Trans. Softw. Eng.* 39, 9 (Sept. 2013), 1208–1215.

[53] Martin Shepperd, Qinbao Song, Zhongbin Sun, and Carolyn Mair. 2013. Data quality: Some comments on the nasa software defect datasets. *IEEE Transactions on Software Engineering* 39, 9 (2013), 1208–1215.

[54] Shivkumar Shivaji, E James Whitehead Jr, Ram Akella, and Sunghun Kim. 2009. Reducing features to improve bug prediction. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 600–604.

[55] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When Do Changes Induce Fixes? *(MSR '05)*. ACM, New York, NY, USA, 1–5. https://doi.org/10.1145/1082983.1083147

[56] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, Akinori Ihara, and Kenichi Matsumoto. 2015. The Impact of Mislabelling on the Performance and Interpretation of Defect Prediction Models *(ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 812–823. http://dl.acm.org/citation.cfm?id=2818754.2818852

[57] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. 2016. Automated parameter optimization of classification techniques for defect prediction models. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 321–332.

[58] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering* 43, 1 (2016), 1–18.

[59] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering* 43, 1 (2016), 1–18.

[60] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. 2018. The Impact of Automated Parameter Optimization on Defect Prediction Models. *CoRR* abs/1801.10270 (2018). arXiv:1801.10270 http://arxiv.org/abs/1801.10270

[61] Kapil Vaswani and Abhik Roychoudhury. 2010. Approach for root causing regression bugs. US Patent App. 12/469,850.

[62] Ian H. Witten, Eibe Frank, and Mark A. Hall. 2011. *Data Mining: Practical Machine Learning Tools and Techniques* (3rd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[63] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. 2012. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated.

[64] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 17–26.

[65] Xiaoxing Yang, Ke Tang, and Xin Yao. 2012. A learning-to-rank algorithm for constructing defect prediction models. In *International Conference on Intelligent Data Engineering and Automated Learning*. Springer, 167–175.

[66] Xiaoxing Yang, Ke Tang, and Xin Yao. 2014. A learning-to-rank approach to software defect prediction. *IEEE Transactions on Reliability* 64, 1 (2014), 234–246.

[67] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. 2016. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 157–168.

[68] Suraj Yatish, Jirayus Jiarpakdee, Patanamon Thongtanunam, and Chakkrit Tantithamthavorn. 2019. Mining Software Defects: Should We Consider Affected Releases? (2019), 654–665. https://doi.org/10.1109/ICSE.2019.00075

[69] Xiao Yu, Kwabena Ebo Bennin, Jin Liu, Jacky Wai Keung, Xiaofei Yin, and Zhou Xu. 2019. An Empirical Study of Learning to Rank Techniques for Effort-Aware Defect Prediction. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 298–309.

[70] Xiao Yu, Jin Liu, Jacky Wai Keung, Qing Li, Kwabena Ebo Bennin, Zhou Xu, Junping Wang, and Xiaohui Cui. 2019. Improving Ranking-Oriented Defect Prediction Using a Cost-Sensitive Ranking SVM. *IEEE Transactions on Reliability* (2019).

[71] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting Defects for Eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering (PROMISE '07)*. IEEE Computer Society, Washington, DC, USA, 9–.