LEVERAGING DEFECTS LIFE-CYCLE FOR LABELING DEFECTIVE CLASSES

A Thesis
presented to
the Faculty of California Polytechnic State University,
San Luis Obispo

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Computer Science

by
Bailey Vandehei
December 2019
COMMITTEE MEMBERSHIP

TITLE: Leveraging Defects Life-Cycle for Labeling Defective Classes

AUTHOR: Bailey Vandehei

DATE SUBMITTED: December 2019

COMMITTEE CHAIR: Davide Falessi, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Alexander Dekhtyar, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Bruno DaSilva, Ph.D.
Professor of Computer Science
ABSTRACT
Leveraging Defects Life-Cycle for Labeling Defective Classes

Bailey Vandehei

Data from software repositories are a very useful asset to building different kinds of models and recommender systems aimed to support software developers. Specifically, the identification of likely defect-prone files (i.e., classes in Object-Oriented systems) helps in prioritizing, testing, and analysis activities. This work focuses on automated methods for labeling a class in a version as defective or not. The most used methods for automated class labeling belong to the SZZ family and fail in various circumstances. Thus, recent studies suggest the use of affect version (AV) as provided by developers and available in the issue tracker such as JIRA. However, in many circumstances, the AV might not be used because it is unavailable or inconsistent. The aim of this study is twofold: 1) to measure the AV availability and consistency in open-source projects, 2) to propose, evaluate, and compare to SZZ, a new method for labeling defective classes which is based on the idea that defects have a stable life-cycle in terms of proportion of versions needed to discover the defect 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 AV cannot be used in the majority (51%) of defects. Therefore, it is important to investigate automated methods for labeling defective classes. Results related to 76 open-source projects from the Apache ecosystem, featuring a total of about 6,250,000 classes that are affected by 60,000 defects and spread over 4,000 versions and 760,000 commits, show that the proposed method for labeling defective classes is, in average among projects and defects, more accurate, in terms of Precision, Kappa, F1 and MCC than all previously proposed SZZ methods. Moreover, the improvement in accuracy from combining SZZ with defects life-cycle information is statistically significant but practically irrelevant (<1%). Moreover, selecting features and observing correlations information are both, overall and in average, more accurate via defects’ life-cycle than any SZZ method.
I would like to thank Davide Falessi, my advisor, for his guidance and help throughout this process. The devotion and commitment Dr. Falessi offered was beyond helpful. This work could not have been completed without his support and direction.

In addition to my advisor, I would also like to thank the rest of my committee members: Alexander Dekhtyar and Bruno Da Silva, for challenging and encouraging me.

I would also like to thank Daniel Alencar da Costa who ran SZZ_B and SZZ_RA on the 76 selected projects in this work. His contribution made a significant impact towards our results.

In addition, I thank Max Dipenta who help guide the work in this thesis.

Finally, I would like to thank Cal Poly for providing me with a wonderful college experience and education. I have grown and learned so much throughout my time here.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>viii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>ix</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Context</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Key Concepts</td>
<td>1</td>
</tr>
<tr>
<td>1.3 Aim</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Structure</td>
<td>3</td>
</tr>
<tr>
<td>2 Background and Related Work</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Methodology in Defect Prediction</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Noise in Defect Prediction</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Analyzing Defect Introduction</td>
<td>6</td>
</tr>
<tr>
<td>2.4 Defect Prediction Datasets Selection</td>
<td>6</td>
</tr>
<tr>
<td>3 Study Design</td>
<td>8</td>
</tr>
<tr>
<td>3.1 RQ1: Is the AV Available and Consistent?</td>
<td>8</td>
</tr>
<tr>
<td>3.1.1 Independent Variables</td>
<td>8</td>
</tr>
<tr>
<td>3.1.2 Dependent Variables</td>
<td>8</td>
</tr>
<tr>
<td>3.1.3 Measurement Procedure</td>
<td>8</td>
</tr>
<tr>
<td>3.2 RQ2: Do Methods Have Different Accuracy in AV Labeling?</td>
<td>10</td>
</tr>
<tr>
<td>3.2.1 Independent Variables</td>
<td>11</td>
</tr>
<tr>
<td>3.2.2 Dependent Variables</td>
<td>14</td>
</tr>
<tr>
<td>3.2.3 Measurement Procedure</td>
<td>15</td>
</tr>
<tr>
<td>3.3 RQ3: Do Methods Have Different Accuracy in Classes Labeling?</td>
<td>18</td>
</tr>
<tr>
<td>3.3.1 Independent Variables</td>
<td>19</td>
</tr>
<tr>
<td>3.3.2 Dependent Variables</td>
<td>21</td>
</tr>
<tr>
<td>3.3.3 Measurement Procedure</td>
<td>22</td>
</tr>
<tr>
<td>3.4 RQ4: Do Methods Lead to Different Identification of Important Features?</td>
<td>22</td>
</tr>
<tr>
<td>3.4.1 Independent Variables</td>
<td>24</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Dependent Variables</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Measurement Procedure</td>
</tr>
<tr>
<td>3.4.3.1</td>
<td>Feature Selection</td>
</tr>
<tr>
<td>3.4.3.2</td>
<td>Correlation</td>
</tr>
<tr>
<td>4</td>
<td>Results</td>
</tr>
<tr>
<td>4.1</td>
<td>RQ1: Is the AV Available and Consistent?</td>
</tr>
<tr>
<td>4.2</td>
<td>RQ2: Do Methods Have Different Accuracy in AV Labeling?</td>
</tr>
<tr>
<td>4.3</td>
<td>RQ3: Do Methods Have Different Accuracy in Classes Labeling?</td>
</tr>
<tr>
<td>4.4</td>
<td>RQ4: Do Methods Lead to Different Identification of Important Features?</td>
</tr>
<tr>
<td>5</td>
<td>Replicability</td>
</tr>
<tr>
<td>5.1</td>
<td>RQ1: Is the AV Available and Consistent?</td>
</tr>
<tr>
<td>5.2</td>
<td>RQ2: Do Methods Have Different Accuracy in AV Labeling?</td>
</tr>
<tr>
<td>5.3</td>
<td>RQ3: Do Methods Have Different Accuracy in Classes Labeling?</td>
</tr>
<tr>
<td>5.4</td>
<td>RQ4: Do Methods Lead to Different Identification of Important Features?</td>
</tr>
<tr>
<td>6</td>
<td>Threats to Validity</td>
</tr>
<tr>
<td>6.1</td>
<td>Conclusion</td>
</tr>
<tr>
<td>6.2</td>
<td>Internal</td>
</tr>
<tr>
<td>6.3</td>
<td>Construct</td>
</tr>
<tr>
<td>6.4</td>
<td>External</td>
</tr>
<tr>
<td>7</td>
<td>Conclusion</td>
</tr>
<tr>
<td></td>
<td>BIBLIOGRAPHY</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-----------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>3.1</td>
<td>Defect prediction features.</td>
</tr>
<tr>
<td>4.1</td>
<td>Variation, in terms of standard deviation, of IV, OV, FV, and P across defects of 76 Apache projects.</td>
</tr>
<tr>
<td>4.2</td>
<td>Relative increment in the accuracy of labeling AV, by combining Simple with a SZZ method, over SZZ, in average across defects of 76 Apache projects.</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Example of the life-cycle of a defect: Introduction Version (IV), Opening Version (OV), Fixed Version (FV), and Affected Versions (AV). Note, versions 0.19 and 0.21 were only &quot;baselines&quot; and not &quot;user-intended&quot; versions and, hence, were excluded</td>
<td>2</td>
</tr>
<tr>
<td>3.1</td>
<td>General overview to compute labeled AV in RQ2 for each method.</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>General overview to compute defective files for each version within a project in RQ3 for each method.</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>General overview of the process of creating the Complete datasets for each project and method.</td>
<td>25</td>
</tr>
<tr>
<td>3.4</td>
<td>General overview of the process of feature selection for each version of each project.</td>
<td>26</td>
</tr>
<tr>
<td>3.5</td>
<td>General overview of the process of correlation in RQ4 computed for each project, version, method, and feature.</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Distribution of 212 Apache projects having a specific proportion of defects' issue reports with an unreliable AV (left side) or without the AV (right side).</td>
<td>30</td>
</tr>
<tr>
<td>4.2</td>
<td>Distribution, across 76 Apache projects, of Precision, Recall, F1, MCC, and Kappa, of different methods in labeling AV.</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>Distribution of value of IV, OV, FV, and P across defects of 76 Apache projects.</td>
<td>33</td>
</tr>
<tr>
<td>4.4</td>
<td>Distribution of standard deviation of IV, OV, FV, and P, across 76 Apache projects.</td>
<td>34</td>
</tr>
<tr>
<td>4.5</td>
<td>Distribution, across 76 Apache projects, of Precision, Recall, F1, MCC, and Kappa, of different methods in labeling defective classes.</td>
<td>36</td>
</tr>
<tr>
<td>4.6</td>
<td>Distribution among datasets of selection frequency of each feature.</td>
<td>38</td>
</tr>
<tr>
<td>4.7</td>
<td>Distribution, across versions and 76 Apache projects, of Precision, Recall, F1, MCC, and Kappa, of different methods in selecting features.</td>
<td>39</td>
</tr>
<tr>
<td>4.8</td>
<td>Distribution among datasets of correlation of each feature to class defectiveness.</td>
<td>41</td>
</tr>
<tr>
<td>4.9</td>
<td>Mean relative error of correlation, for each feature, of each method.</td>
<td>42</td>
</tr>
</tbody>
</table>
1.1 Context

Identifying and reducing defects in software systems has paramount importance from both economic and dependability perspectives. For this reason, a significant amount of research effort has been spent trying to reduce defects by accurately estimating where they are [49]. Specifically, prediction models can support test resource allocation by predicting the existence of defects in a software artifact (e.g., a class). In other words, the identification of likely defect-prone files (i.e., classes in OO systems) helps in prioritizing testing and analysis activities.

Creating a defect prediction model implies the availability of a dataset upon which the model is being trained. Thus, researchers provided means to create [26, 73], collect [18] and select [27, 51, 61] datasets associating software defects to other metrics.

Ultimately, the accuracy of a prediction model depends on the quality of the underlying dataset [41, 63]. Therefore, effort has been spent in identifying sources of noise in the datasets and how to deal with it, including defect misclassification [5, 30, 40, 57, 67] and imprecise defect origin identification [60].

This work focuses on automated methods for dataset creation, i.e., for labeling a class in a version as defective or not.

1.2 Key Concepts

Fig. 1.1 illustrates the key concept and definitions of our study by using, as an example, the defect QPID-4462 \(^1\). The defect is first injected in the code, we called this version the introduction version (IV), i.e., 0.18 in Fig. 1.1. Afterward, a failure is experienced and a issue report is created describing what is wrong with the system behavior; we call this version the opening version (OV), i.e., 0.20 in Fig. 1.1. Then,

\(^1\)https://issues.apache.org/jira/browse/QPID-4462
Figure 1.1: Example of the life-cycle of a defect: Introduction Version (IV), Opening Version (OV), Fixed Version (FV), and Affected Versions (AV). Note, versions 0.19 and 0.21 were only ”baselines” and not ”user-intended” versions and, hence, were excluded

in a given software version, the defect is fixed; we call this version the fixing version (FV), i.e., 0.22 in Fig. 1.1. A version is affected by a defect if its users could have experienced the related failure. Thus, the affected versions (AV) are those in the range [IV, FV), i.e., 0.18 and 0.20 in Fig. 1.1. Note that IV is, by definition, the oldest among the AVs. Moreover, since this study focuses only on post-release defects, all defects have at least one AV.

The OV is available in all defects as it is being generated by the issue tracker upon ticket creation date. The FV is available in the defects where developers added the issue report’s ID in the comment of the commit fixing the defect. For instance commit 732ab160852f943cd847646861dd48370dd23ff3 is the last commit including [QPID-4462] in its comment. Since such a commit has been performed on 2013-03-31T21:51:49+00:00 then we know it has been performed after 0.20 and before 0.22.

The AV is a field in the defect’s issue report often filled by developers. If such a field is missing, we know that the version after the fix commit does not contain the defect and the one before the fix commit contains the defect, however we do not know when the defect originated, i.e., we do not know the IV unless specified by the developers. One possible method aimed at identifying IV is the SZZ algorithm [65]. However, several works have identified its limitations [13, 59, 60]. For instance, SZZ cannot determine the correct location of defects that were fixed by solely adding code [13] or defects of the regression type [53, 71]. Two recent SZZ studies [13, 66] suggest
considering the AV, as provided by the issue tracker system when creating defects datasets. However, no study investigated whether the AV can actually be used (i.e., it is available and consistent), nor how to estimate it when it cannot be used. The aim of this study is to cover this gap.

1.3 Aim

The goal of this paper is, first, to investigate the extent to which the AV-related information is available in open-source project, and whether it is reliable to be used to label defective classes and hence build defect datasets. Second, we propose, evaluate, and compare a new method for labeling defective classes which is based on the idea that defects, even of different projects, have a stable life-cycle. Specifically, our intuition is that defects share the same life-cycle in terms of proportion of number of versions between its fix and its discovery. More specifically, we address the following four research questions:

- **RQ1**: Is the AV Available and Consistent?
- **RQ2**: Do Methods Have Different Accuracy in AV Labeling?
- **RQ3**: Do Methods Have Different Accuracy in Classes Labeling?
- **RQ4**: Do Methods Lead to Different Identification of Important Features?

Results of our study, conducted on 212 open source Java projects from the Apache ecosystem indicated that the AV is available and consistent in 49% of the cases.

1.4 Structure

The remainder of this document is structured as it follows. Section 2 contains information about the background and related works. Section 3 features the study design. Section 4 details the results of the investigation. Section 5 is a discussion of results. Section 6 explains the threats to validity. Section 7 provides a conclusion to our findings.
Chapter 2

BACKGROUND AND RELATED WORK

Since our work proposes an approach to improve the construction of defect prediction datasets, we survey the related literature regarding methodologies in defect prediction, noise in defect prediction, defect introduction, and defect prediction datasets.

2.1 Methodology in Defect Prediction

Turhan et al. [70] proposed a practical defect prediction approach for companies that do not track defect related data. Specifically, they investigate the applicability of cross-company (CC) data for building localized defect predictors using static code features.

Menzies et al. [49] reported on the current results, limitations, and new approaches of defect prediction from static code features. They advise against the indiscriminate use of classifiers and, instead, suggest choosing and customizing the classifiers to the goal at hand.

Fu et al. [25] showed that tuning classifiers is simple and very effective; thus, it is no longer enough to just run a data miner and present the result without conducting a tuning optimization study.

Similarly, Tantithamthavorn et al. [68] showed that tuning yields substantial benefits in terms of performance improvement and stability, while incurring a manageable additional computational cost. Thus, tuning should be included in future defect prediction studies.

Bayley and Falessi [4] investigated the use and optimization of prediction intervals by automatically configuring Random Forest. Their results show that no single validation technique is always beneficial for tuning.

Agrawal and Menzies [1] reported and fixed an important systematic error in prior studies that ranked classifiers for software analytics. Those studies did not (a) as-
sess classifiers on multiple criteria and they did not (b) study how variations in the data affect the results. Their results show that (1) data pre-processing can be more important than the classifier choice, (2) ranking studies are incomplete without such pre-processing, and (3) SMOTUNED, a tuned implementation of John Platt’s sequential minimal optimization algorithm, is a promising candidate for pre-processing.

2.2 Noise in Defect Prediction

Bird et al. [5] 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. [40] measured the impact of noise on defect prediction models and provides guidelines for acceptable noise level. 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. [30] 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. [57] 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. [67] 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.
2.3 Analyzing Defect Introduction

Śliwerski et al. [65] proposed the first implementation of the SZZ algorithm, an algorithm for finding bug-inducing commits. 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’s fix, when they have last been changed before such a fix.

Kim et al. [38] presented algorithms to automatically and accurately identify bug-introducing changes which improves on SZZ.

da Costa et al. [13] proposed three criteria and evaluated five SZZ implementations. They conclude that current SZZ implementations still lack mechanisms to accurately identify bug-introducing changes.

Neto et al. [52] found that 19.9% of lines that are removed during a fix are related to refactorings and, therefore, their respective inducing changes are false positives.

Falessi and Moede [18] presented the Pilot Defects Prediction Dataset Maker (PDPDM), a desktop application for measuring metrics to use for defect prediction. PDPDM avoids the use of outdated datasets and it allows researchers and practitioners to create defect datasets without the need to write any lines of code.

Rodríguez-Pérez et al. [60] investigated the complex phenomenon of bug introduction and bug fix. They show that less than 30% of defects can be found using the algorithm based on the assumption that “a given bug was introduced by the lines of code that were modified to fix it”.

2.4 Defect Prediction Datasets Selection

Nagappan et al. [51] combine ideas from representativeness and diversity and introduce a measure called sample coverage, defined as 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 a research (space).

Gousios and Spinellis [27] 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. [61] 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. [20] 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.
3.1 RQ1: Is the AV Available and Consistent?

Two recent SZZ studies [13, 66] suggest considering the AV, as provided by the issue tracker system when creating defects datasets. However, how often do developers actually fill in this field? In this research question we do not have a formal hypothesis to investigate. We are interested in investigating the extent to which the AV is usable, i.e., it is available and consistent.

3.1.1 Independent Variables

The independent variable is the defect.

3.1.2 Dependent Variables

The dependent variable is the percentage of available, and available & consistent AV. An AV is available if provided in the JIRA ticket by the developer. An AV is consistent if not after the OV date. The rationale is that the defect is existing at least the day the related ticket has been created.

3.1.3 Measurement Procedure

We follow a procedure consisting of the following nine steps:

1. We retrieve the JIRA and Git URL of all existing Apache projects\(^1\). We focused on Apache projects rather than random GitHub projects because the formers have a higher quality of defect annotation and to avoid using toy projects [50].

\(^1\)https://people.apache.org/phonebook.html
2. We filter out projects not managed in JIRA or not versioned in Git. This leads to 212 projects.

3. As recently done by Borg et al. [45], for each project, we count the number of issue reports by performing the following query to the JIRA repository: $\text{Type} == \text{"defect" AND (status == \text{"Closed" OR status == \text{"Resolved"}) AND Resolution == \text{"Fixed"}}$. 

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 industry as production defects, i.e., defects that were present in software projects used by users. 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.

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: $\text{Affect Version} \neq \text{Null}$. Thus, each issue report is tagged as available or unavailable.

7. For each issue report, we check its AV consistency, i.e., if $\text{IV} \leq \text{OV}$.

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.
3.2 RQ2: Do Methods Have Different Accuracy in AV Labeling?

As mentioned in the previous research question, it is not required developers list the AV for an issue report. Therefore, it may often be left blank. In this case, we must find an automated method to label the AV when it is not listed or incorrectly listed by developers, i.e., not available or not consistent. This is the motivation of this research question.

One way to determine AV is using an SZZ algorithm to find the oldest defect-introducing commit, then labeling each version between the oldest defect-introducing commit and the fix commit (excluded) as an AV. However, we know the defect’s ticket creation date and fix date. With this information, we know that the AV were at least the versions between the creation date and the fix. In addition, our intuition is that defects have a stable life-cycle in terms of the number of versions to find and to fix a defect. We hope to exploit this stability to aid the labelling of AV.

This research question investigates which methods is most accurate in labeling versions defective or not at the defect level. In the within-project across-version defect-level context, a specific version is labeled to be defective or not. In other words, the predicted variable is the defectiveness of a version for a given defect. Each method provides a labeled IV for a given defect. The defectiveness is True if the
version is before the defect’s FV and greater than or equal to the defect’s IV labeled by the method.

In this research question, we propose the following two hypothesis:

• $H_{10}$: different methods have the same accuracy in AV labeling.

• $H_{20}$: combining defects’ information with SZZ methods does not increase the accuracy of SZZ methods in AV labeling.

### 3.2.1 Independent Variables

Our independent variable is the method used to identify the IV, i.e., to label version as affected or not by a defect. We have a total of 10 methods which can be categorized based on the family, the use or not of historical data, and the combination with a simple heuristic. In this work we analyze four families of methods:

1. **Simple**: It simply assumes that the IV corresponds to OV. The rationale is that, by definition, all versions from OV to FV (excluded) are AV; however, versions before OV can be AV too. Therefore, we expect this heuristic to achieve a 100% Precision and a low Recall. Specifically, this heuristic would identify 0.20 as IV in Fig. 1.1; therefore, it would miss 0.18 (false negative) and would correctly identify 0.20 (true positives) as AV.

2. **SZZ**: As previously discussed in Section 2.3, 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 [65] to determine when the defect has been introduced, and we assume as AV all versions between the IV and the FV (excluded). In the example in Fig. 1.1, **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 AV 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) \textbf{SZZ}_U: We rely on an open implementation of SZZ by Borg et al. \cite{6} and we set depth to one. This SZZ implementation does not discard cosmetic changes (since it supports all programming languages), however it uses Jaccard distances to map moving lines. In the example in Fig. 1.1, \textbf{SZZ}_U identified one defect-introducing commits dated 2012-05-18T20:54:25 which falls into version 0.16 labeled as the IV. Therefore, the AV 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) \textbf{SZZ}_RA: We use a refactoring-aware SZZ algorithm implemented by Da Costa \cite{13}. This algorithm can track defect-introducing commits and filters out changes due to refactoring. However, this implementation only analyzes java files, so the defect-introducing commits for non-java files are determined by \textbf{SZZ}_U. In the example in Fig. 1.1, \textbf{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 AV 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).

3. \textbf{Proportion}. It assumes a stable proportion ($P$), among defects of the same project, between the number of 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; these numbers can be 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 issue reports where AV is not available and 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 Fig. 1.1, the $P_{Increment}$, computed as the average $P$ among defects in versions 1 to 15, is 1.7775. Therefore, $IV = 16 - (16 - 15) \times 1.7775$ 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 does not make sense. Therefore, we want to utilize the average $P$ values of other projects. Recall we analyzed 76 projects. For each project, we computed the average $P$ across all defects within the project. Let's label each of these $P_{PROJECT}$ where $PROJECT$ is the project’s ID. Then for each project, we took the median of the $P_{PROJECT}$ values among all other 75 projects to use as the $P_{ColdStart}$. Using the example in Fig. 1.1, the indexes of the 0.18, 0.20, and 0.22 are 14, 15, and 16 respectively. The $P_{ColdStart}$, computed as the median of the 75 other projects' $P_{PROJECT}$, is 1.8089. Therefore, $IV = 16 - (16 - 15) \times 1.8089$ 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 fix date. For each defect within a project, we used the average $P$ among the last 1% of fixed defects. Using the example in Fig. 1.1, the $P_{MovingWindow}$ is computed as the average $P$ among the last 1% of defects. There are 1192 defects in this project. Therefore, there is about 12 defects in 1%. The average $P$ among the last 12 fixed defects is 2.167. Therefore, $IV = 16 - (16 - 15) \times 2.167$ 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.

4. +: All the SZZ can be merged with simple. Therefore, for each SZZ method, we created $SZZ_X+$ 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 method. The rationale is that if Simple labels an AV as defective, then the AV is actually defective.

To illustrate how this works, we will use a new example, WICKET-4071\(^2\). This issue report’s AV are versions 1.4.6, 1.4.7, 1.4.8, 1.4.19, 1.4.10, and 1.5-M1. Its 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).

### 3.2.2 Dependent Variables

Our dependent variable is the accuracy in labeling versions of a project as affected, or not, by a defect. Note, we do not use Area Under the Receiver Operating Characteristic Curve since there is no threshold we can vary. We simply have binary classifications, i.e., true and false. 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:** \[\frac{TP}{TP + FN}\]

\(^2\)https://issues.apache.org/jira/browse/WICKET-4071
**F1**: \( \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \)

- **Cohen’s Kappa**: A statistic that assesses the classifier’s performance against random guessing [10]. \( Kappa = \frac{\text{Observed} - \text{Expected}}{1 - \text{Expected}} \) 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. \( \frac{TP + FP}{TP + TN + FP + FN} \times \frac{TP + FN}{TP + TN + FP + FN} \)
    * \( P_{No} \): Probability of negative agreement. \( \frac{TN + FP}{TP + TN + FP + FN} \times \frac{TN + FN}{TP + TN + FP + FN} \)

- **Matthews Correlation Coefficient**: \( \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \)

**3.2.3 Measurement Procedure**

We began by selecting the projects with the highest percent of available and consistent AV. We selected projects with at least 100 defects that were linked with git and contained available and consistent AV. Then, we filtered out projects with less than 6 versions. Lastly, we filtered out projects where the percent of available and consistent AV is greater than 50% of AV issues. This left us with 76 projects. For each project we followed the steps below. See Fig. 3.1 for an overview of this process.

1. We retrieved the project’s versions and version dates by JIRA. We numbered these versions beginning with the oldest version as version 1.

2. We used the defects whose issue reports’ AV were found to be available and consistent in RQ1. For each defect, we determined the IV (version of the first AV labeled by JIRA), OV (version of the ticket creation), FV (fix version), and 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 \( P_{\text{ColdStart}} \), i.e., the median P of all other projects.

iii. For each defect, we computed the IV as \( IV = (FV - OV) * P_{\text{ColdStart}} \). 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 \( P_{\text{Increment}} \) as the average P among defects fixed in versions 1 to R-1.

ii. We used the \( P_{\text{ColdStart}} \) for \( P_{\text{Increment}} \) values containing less than 5 defects in the average.
iii. For each defect in each version, we computed the IV as $IV = (FV - OV) \times P_{Increment}$. 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. Recall, the defects are ordered by 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 $IV = (FV - OV) \times P_{MovingWindow}$. 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.

(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.
3.3 RQ3: Do Methods Have Different Accuracy in Classes Labeling?

Defect prediction is important to reduce the cost of testing and code review, by letting developers focus on specific artifacts. However, in order to create datasets for defect prediction, we must determine which classes in versions are defective. The AV in issue trackers can be used as the ground truth for defectiveness of classes in versions. One can determine the defectiveness by (1) computing which classes were touched by a defect’s fix, then (2) using the defect’s issue report’s AV to label each of the classes in each of the versions in the AV as defective. However, if the AV is not available, we must use an automated method for determining the AV to then compute the defectiveness of version-class pairs. This is the motivation for this research question.

It is important to understand which methods provides the most accurate defect prediction dataset. In the within-project across-version class-level context, a specific class in a specific version is labeled to be defective or not. In other words, the predicted variable is the defectiveness of a class in a given version. This defectiveness is True if there is at least one defect that affects that class in that version, i.e. if that class will be changed to fix a defect.

In this research question, we propose the following two hypotheses:

- $H_{30}$: different methods have the same accuracy in class labeling.
• H$_{40}$: combining defects’ information with SZZ methods does not increase the accuracy of SZZ methods in class labeling.

### 3.3.1 Independent Variables

The independent variables are the same methods presented in RQ2 (see Section 3.2.1), however in this research question, each method labels classes in versions rather than versions per defect. A class in a version of a project is labeled as defective if there is at least one defect fix touching that file and if the version is labeled as affected by the method for that defect.

To show the difference in labeling from the previous research question we will use the example in Fig 1.1. This defect’s fix commit touched the class qpid/java/broker/src/main/java/org/apache/qpid/server/security/auth/manager/SimpleLDAP AuthenticationManager.java. It is important to note, defect QPID-4476, also touched this file in its fix commit.

1. **Simple**: Simple would correctly classify qpid/java/broker/src/main/java/org/apache/qpid/server/security/auth/manager/SimpleLDAP AuthenticationManager.java in version 0.20 as defective (true positives). However, it would miss this class in 0.18 (false negative).

2. **SZZ**:
   - (a) **SZZ_Basic**: SZZ_B would correctly classify qpid/java/broker/src/main/java/org/apache/qpid/server/security/auth/manager/SimpleLDAP AuthenticationManager.java in versions 0.18 and 0.20 as defective (true positives), receiving 100% accuracy for this class.
   - (b) **SZZ_U**: SZZ_U would correctly classify qpid/java/broker/src/main/java/org/apache/qpid/server/security/auth/manager/SimpleLDAP AuthenticationManager.java in versions 0.18 and 0.20 as defective (true positives). However, it would incorrectly classify this class as defective in versions 0.16 (false positives).
   - (c) **SZZ_RA**: SZZ_RA would correctly classify qpid/java/broker/src/main/java/ org/apache/qpid/server/security/auth/manager/SimpleLDAP
AuthenticationManager.java in versions 0.18 and 0.20 as defective (true positives). However, it would incorrectly classify this class as defective in versions 0.16 (false positives).

3. Proportion:

(a) **Proportion_ColdStart**: Proportion_ColdStart would correctly classify qpid/java/broker/src/main/java/org/apache/qpid/server/security/auth/manager/SimpleLDAPAuthenticationManager.java would be labeled as defective in version 0.20 for defect QPID-4462. However, QPID-4476 also touched this class in its fix commit. The IV for QPID-4476 for this method is 0.18. Therefore, the class would be identified as defective in versions 0.18 and 0.20 (true positives), giving 100% accuracy for this class.

(b) **Proportion_Incremental**: Proportion_Incremental would correctly classify qpid/java/broker/src/main/java/org/apache/qpid/server/security/auth/manager/SimpleLDAPAuthenticationManager.java would be labeled as defective in version 0.20 for defect QPID-4462. However, QPID-4476 also touched this class in its fix commit. The IV for QPID-4476 for this method is 0.18. Therefore, the class would be identified as defective in versions 0.18 and 0.20 (true positives), giving 100% accuracy for this class.

(c) **Proportion_MovingWindow**: Proportion_MovingWindow would correctly classify qpid/java/broker/src/main/java/org/apache/qpid/server/security/auth/manager/SimpleLDAPAuthenticationManager.java in versions 0.18 and 0.20 as defective (true positives), giving 100% accuracy.

4. +: To illustrate the + method, we will use the defect WICKET-4071 example again. This defect fix commit touches wicket/src/main/java/org/apache/wicket/markup/parser/filter/OpenCloseTagExpander.java. Simple would classify this class in 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 this class in versions 1.4.10 and 1.5-M1 as defective (true positives) and miss this class in versions 1.4.6, 1.4.7, 1.4.8, and 1.4.19 (false negatives). However, SZZ_B+ would classify this class in versions 1.4.8, 1.4.19, 1.4.10, and 1.5-M1 as defective (true positives) and would miss this class in versions 1.4.6
and 1.4.7 (false negatives).

### 3.3.2 Dependent Variables

The dependent variables are the same presented in RQ2 (see Section 3.2.2) with the only difference that the unit upon which the accuracy is computed is the defectiveness of a class in a version. Recall, that a version is labeled as defective or not for a given defect in RQ2. If various defects touch the same class in their fix commit and classify the same version, there may be overlap in the version-class defect classification. In other words, a single defect may classify the version-class pair as non-defective. However, another defect may classify the version-class pair as defective. RQ3 aggregates these classifications together. If at least one defect labeled the version-class pair as defective, then it is labeled as defective for RQ3. This is demonstrated in Fig. 3.2 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 this research question and RQ2, let’s consider the case of methods A, B, and C, and a class that in a version was actually affected by three defects. Suppose that A was able to identify that the class was affected by one defect, B by three defects, C by 4 defects. In this example, all three methods correctly identify the class in the version as defectiveness and therefore they all result with perfect accuracy. However, method B results more accurate than methods A and C in RQ2.

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 be 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.
3.3.3 Measurement Procedure

For each project we followed the steps below. See Fig. 3.2 for an overview of this process.

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 same defect’s fix commit touched the class. Otherwise, the version-class pair was labeled as not defective.

3. We determined the actual defectiveness of each version-class pair. To do so, we labeled each version-class pair as defective if the version of the pair was labeled as an AV of at least one defect by JIRA developers and the same defect’s fix commit touched the class. Otherwise, the version-class pair was labeled as not defective.

4. For each method, we compared the classifications to the actual classification, and computed the TP, FN, TN, FP, Precision, Recall, F1, Matthews, and Kappa across the project.

3.4 RQ4: Do Methods Lead to Different Identification of Important Features?

In this research question, we investigate the accuracy of methods in identifying important features affecting class defectiveness. This research question is the application of RQ5 of [22] in the context of class defectiveness. Specifically, [22] reports that it is important to understand the most important metrics for identifying class defectiveness, developers can avoid the pitfalls that show high association with the appearance of defective classes.

There are various ways to interpret which features are most important in determining class defectiveness. First, we can analyze which features are most commonly
selected. Classifier can run quicker and more efficiently by performing feature selection algorithms and reducing the dimensionality of the data before running the classifier on it. A common feature selection algorithm is exhausted search which searches among all possible subsets of the features which combination is best in determining the classification [17]. However, feature selection also requires an evaluation method to evaluate the subset under examination [17]. There are various ways to do so, however one possibility is to use correlation such that values of the subsets are correlate highly with the class value and contain low correlation with each other [17].

Second, we can investigate the correlation between a feature and class defectiveness. In regards to the previous research questions, we use the AV as the ground truth for determining class defectiveness. However, as mentioned previously, the AV is not always available. Therefore, an alternative method must be used to compute class defectiveness. If a different method is used to compute AV when the AV is not available in the issue report, does this affect which features are most important in determining class defectiveness? This is the motivation for this research question.

In this research question, we propose the following four hypothesis:

- **H_{50}**: the selection frequency varies among features.
- **H_{60}**: the correlation varies among features.
- **H_{70}**: different methods have the same accuracy in identifying important features.
  - **H_{70F}**: different methods have the same accuracy in selecting features.
  - **H_{70C}**: different methods have the same accuracy in correlating features.
- **H_{80}**: combining defects’ information with SZZ methods does not increase the accuracy over SZZ methods in identifying important features.
  - **H_{80F}**: combining defects’ information with SZZ methods does not increase the accuracy over SZZ methods in selecting features.
  - **H_{80C}**: combining defects’ information with SZZ methods does not increase the accuracy over SZZ methods in correlating features.
### 3.4.1 Independent Variables

Again, the independent Variables are the same methods presented in RQ2 (see Section 3.2.1).

### 3.4.2 Dependent Variables

The importance of a feature for classifying class defectiveness can be identified in two ways: 1) if it is selected to be used by the classifiers [29], 2) if it is highly correlated with the predicted variable [73].

As features we use 17 well-defined product and project metrics that have been shown to be useful for defect prediction [14, 19]. The used features are detailed in Table 3.1.

Regarding feature selection, we compare the features selected by using a dataset developed by a method versus the features selected by using a dataset of the actual methods. Since each feature has a binary value (selected/un-selected) then we can use the confusion metrics already used in RQ2 and RQ3: Precision, Recall, F1, MCC and Kappa.

Regarding correlation, we want to compare the correlation measured on a dataset developed by a method versus the correlation measured on the actual dataset. Therefore we used the standard mean relative error [12]:

$$\frac{|(\text{LabeledByMethod} - \text{Actual})|}{\text{Actual}}$$
Table 3.1: Defect prediction features.

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

Figure 3.3: General overview of the process of creating the Complete datasets for each project and method.
3.4.3 Measurement Procedure

We began by computing the metrics for each project as shown in Fig. 3.3.

1. For each project, we begin by removing the last 50% of versions due to the fact that classes snore as described in [3].

2. For each project P, we compute the metrics as described in Table 3.1 for each version-class pair.

3. For each of the methods M, we combined the datasets with the version-class pair’s defectiveness 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.

3.4.3.1 Feature Selection

First we begin by analyzing which features were selected by the method and compare this to the methods that should have been selected, i.e., the features selected in P_Actual.R.Complete. This is shown in Fig. 3.4.

1. For each dataset, P.M.R.Complete, we perform an Exhaustive Search Feature Selection using Weka. We used CfsSubsetEval for the evaluation function which
selects subsets of features based on their high correlation with the predicted variable and low correlation with each other.

2. For each dataset, P_Actual_R_Complete, we perform an Exhaustive Search Feature Selection using Weka and CfsSubsetEval as the evaluation method.

3. For each P_M_R_Complete dataset in RQ4, we compare the features selected to the features selected for P_Actual_R_Complete dataset.

4. If a feature was selected in P_Actual_R_Complete and P_M_R_Complete, it is marked as a true positive.

5. If a feature was not selected in P_Actual_R_Complete and not selected in P_M_R_Complete, it is marked as a true negative.

6. If a feature was selected in P_Actual_R_Complete and not selected in P_M_R_Complete, it is marked as a false negative.

7. If a feature was not selected in P_Actual_R_Complete and selected in P_M_R_Complete, it is marked as a false positive.

8. We then computed Precision, Recall, F1, Mathews Correlation, Kappa for each project, method and version.

9. Lastly, we compute the percentage of times a feature is selected across all P_Actual_R_Complete datasets.

3.4.3.2 Correlation

Next, we computed the strength of the monotonic relationship between the feature and the classification. These steps are demonstrated in Fig. 3.5

1. For each feature F, in each P_M_R_Complete dataset in RQ4, we computed the Spearman’s Correlation between the feature and the defectiveness computed by the method. We labeled this P_M_R_F_SC.
2. For each feature F, in each P_Actual_R_Complete dataset in RQ4, we computed the Spearman’s Correlation between the feature the actual defectiveness. We labeled this P_Actual_R_F_SC.

3. We then computed the relative disagreement between P_M_R_F_SC and P_Actual_R_F_SC where 
\[
\text{Rel.Disagreement} = \frac{|P_M_R_F_SC - P_Actual_R_F_SC|}{P_Actual_R_F_SC}.
\]

Figure 3.5: General overview of the process of correlation in RQ4 computed for each project, version, method, and feature.
4.1 RQ1: Is the AV Available and Consistent?

Fig 4.1 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 Fig 4.1 most of the projects have more than 25% of defects’ issue reports with no AV. Moreover, the total number of closed defects linked with git in the 212 Apache projects is 125,860. Of these, 63,539 defects’ issue reports (51%) resulted in not having or having an unreliable AV. Thus, we can claim that in most of the defects’ issue reports, we can’t use the AV and hence we frequently need an automated method for labeling classes.

4.2 RQ2: Do Methods Have Different Accuracy in AV Labeling?

Fig 4.2 reports the distribution, across 76 Apache projects, of Precision, Recall, F1, MCC, and Kappa, of different methods in labeling AV. According to Fig 4.2:

- The Proportion methods have a higher Precision and composite accuracy (F1, MCC, and Kappa) than all SZZ methods. Therefore, we can claim that labeling AV via defects’ life-cycle information is overall and in average more accurate than any SZZ method.

- 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.
Figure 4.1: Distribution of 212 Apache projects having a specific proportion of defects’ issue reports with an unreliable AV (left side) or without the AV (right side).
• 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.

Statistical results on the 76 Apache projects show that there is a significant difference in the accuracy of methods in labeling AV. Specifically, the p-value of the Wilcoxon test resulted less than 0.001 for Precision, Recall, F1, MCC, and Kappa. Therefore, we can reject $H_{10}$ in all five cases.

One of the possible reasons for the high accuracy achieved by the Proportion method is that P is pretty stable across projects (i.e., Proportion_ColdStart) and even more stable within the same project (i.e., Proportion_Increment and Proportion_MovingWindow). Fig 4.3 reports the distribution of value of IV, OV, FV, and P across defects of different projects. Table 4.1 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 Fig 4.3 and Table 4.1, P is pretty stable across defects of different projects especially when compared to IV, OV and FV. Fig 4.4 reports the distribution of standard deviation of IV, OV, FV, and P, across 76 Apache projects. According to Fig 4.4 the STDV is much higher across projects than within the same project. Specifically, the median STDV within projects is only 2 (Fig 4.4) whereas the one across projects is about 5 (Table 4.1). In conclusion, the high stability reported in Fig 4.1, Table 4.1 and Fig 4.4 shows that defects shares the same life-cycle in terms of proportion of number of versions between its fix and its discovery, especially within the same project.

Table 4.2 reports the relative increment of combing a SZZ method with the knowledge of the defects’ information (i.e., SZZ,x vs. SZZ,x+). According to Table 4.2 the increments are particularly high in the SZZ,B and in the MCC accuracy metric (+14%). Despite this low increment, statistical results on the 76 Apache projects show that there is a significant difference in the accuracy of SZZ methods with the addition of defects’ information over that SZZ method. Specifically, the p-value of the paired Wilcoxon Signed Rank test resulted less than 0.01 for Precision, Recall, F1, MCC, and Kappa in all three SZZ methods. Therefore, we can reject $H_{20}$ in all 15 cases.
Figure 4.2: Distribution, across 76 Apache projects, of Precision, Recall, F1, MCC, and Kappa, of different methods in labeling AV.
Figure 4.3: Distribution of value of IV, OV, FV, and P across defects of 76 Apache projects.

Table 4.1: Variation, in terms of standard deviation, of IV, OV, FV, and P across defects of 76 Apache projects.

<table>
<thead>
<tr>
<th>Version</th>
<th>STDV</th>
</tr>
</thead>
<tbody>
<tr>
<td>IV</td>
<td>38.36</td>
</tr>
<tr>
<td>OV</td>
<td>40.17</td>
</tr>
<tr>
<td>FV</td>
<td>41.85</td>
</tr>
<tr>
<td>P</td>
<td>5.43</td>
</tr>
</tbody>
</table>
Figure 4.4: Distribution of standard deviation of IV, OV, FV, and P, across 76 Apache projects.

Table 4.2: Relative increment in the accuracy of labeling AV, by combining Simple with a SZZ method, over SZZ, in average across defects of 76 Apache projects.

<table>
<thead>
<tr>
<th></th>
<th>SZZ_B</th>
<th>SZZ_B+</th>
<th>Gain on SZZ_B</th>
<th>SZZ_RA</th>
<th>SZZ_RA+</th>
<th>Gain on SZZ_RA</th>
<th>SZZ_U</th>
<th>SZZ_U+</th>
<th>Gain on SZZ_U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision</td>
<td>0.252</td>
<td>0.263</td>
<td>4%</td>
<td>0.206</td>
<td>0.21</td>
<td>2%</td>
<td>0.223</td>
<td>0.224</td>
<td>0%</td>
</tr>
<tr>
<td>Recall</td>
<td>0.848</td>
<td>0.894</td>
<td>5%</td>
<td>0.915</td>
<td>0.938</td>
<td>2%</td>
<td>0.97</td>
<td>0.971</td>
<td>0%</td>
</tr>
<tr>
<td>F1</td>
<td>0.369</td>
<td>0.387</td>
<td>5%</td>
<td>0.318</td>
<td>0.326</td>
<td>2%</td>
<td>0.345</td>
<td>0.345</td>
<td>0%</td>
</tr>
<tr>
<td>MCC</td>
<td>0.296</td>
<td>0.343</td>
<td>14%</td>
<td>0.213</td>
<td>0.229</td>
<td>7%</td>
<td>0.271</td>
<td>0.272</td>
<td>0%</td>
</tr>
<tr>
<td>Kappa</td>
<td>0.199</td>
<td>0.222</td>
<td>10%</td>
<td>0.122</td>
<td>0.131</td>
<td>7%</td>
<td>0.157</td>
<td>0.157</td>
<td>0%</td>
</tr>
</tbody>
</table>
4.3 RQ3: Do Methods Have Different Accuracy in Classes Labeling?

Fig 4.5 reports the distribution, across 76 Apache projects, of Precision, Recall, F1, MCC, and Kappa, of different methods in labeling defective classes. According to Fig 4.5:

- As in RQ2, the Proportion methods have a higher Precision and composite accuracy (F1, MCC, and Kappa) than all SZZ methods. Therefore, we can claim that labeling classes via defects’ life-cycle information is overall and in average more accurate than any SZZ method.

- As in RQ2, SZZ_U has the highest Recall than all other methods.

- As in RQ2, SZZ_B+ has a highest Precision and lower Recall than any other SZZ method.

- Differently from RQ2, SZZ_B+ has a higher composite accuracy (F1, MCC, and Kappa) than Simple and any other SZZ method.

- Differently than in RQ2, the Proportion_MovingWindow method dominates all methods on all composite accuracy (F1, MCC, and Kappa).

Regarding comparing the accuracy in class labeling of SZZ methods with defects’ information over that SZZ method (i.e., SZZx+ vs. SZZx), our results show less than 1% of improvement. Statistical results on the 76 Apache projects show that there is a significant difference in the accuracy of SZZ method with defects’ information over that SZZ method. Specifically, the p-value of the paired Wilcoxon Signed Rank test resulted less than 0.01 for Precision, Recall, F1, MCC, and Kappa in all SZZ methods. Therefore, we can reject H_40 in all 15 cases.

Moreover, by comparing Fig 4.5 to Fig 4.2 we observe that, all methods are more accurate in labeling classes than AV on all accuracy metrics. Specifically, by comparing the median accuracy (across methods and datasets) we see an increase in labeling classes over AV 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.
Figure 4.5: Distribution, across 76 Apache projects, of Precision, Recall, F1, MCC, and Kappa, of different methods in labeling defective classes.
Statistical results show that there is a significant difference in the accuracy of methods in labeling classes. Specifically, the p-value of the Wilcoxon test resulted less than 0.01 for Precision, Recall, F1, MCC, and Kappa. Therefore, we can reject $H_{30}$ in all five cases.

4.4 RQ4: Do Methods Lead to Different Identification of Important Features?

Regarding feature selection, Fig. 4.6 reports the distribution among datasets of selection frequency of each feature. Statistical results show that there is a significant difference in the accuracy of methods in labeling classes. Specifically, the p-value of the Wilcoxon test resulted less than 0.01 and therefore we can reject $H_{50}$ and claim that there is a difference among features in the selection frequency.

Regarding the comparison of the features selected by using a dataset developed by a method versus the features selected by using a dataset of the actual methods, Figure 4.7 reports the distribution, across versions and 76 Apache projects, of Precision, Recall, F1, MCC, and Kappa in selecting features.

According to Fig 4.7:

- the Proportion methods have a higher accuracy (in all five metrics) than all SZZ methods. Specifically, according to 4.7 the proportion methods are the only methods having a perfect median Precision and Recall. Therefore, we can claim that selecting features via defects’ life-cycle information is overall and in average more accurate than any SZZ method.

- Differently from both RQ2 and RQ3, there is not much difference in the accuracy of different SZZ methods.

Statistical results show that there is a significant difference in the accuracy of methods in selecting features. Specifically, the p-value of the Wilcoxon test resulted less than 0.01 for Precision, Recall, F1, MCC, and Kappa. Therefore, we can reject $H_{70F}$ in all 5 cases.
Figure 4.6: Distribution among datasets of selection frequency of each feature.
Figure 4.7: Distribution, across versions and 76 Apache projects, of Precision, Recall, F1, MCC, and Kappa, of different methods in selecting features.
Regarding comparing the accuracy in selecting features of SZZ methods with defects' information over that SZZ method (i.e., SZZx+ vs. SZZx), our results show less than 1% of improvement. Statistical results on the 76 Apache projects show that there is a significant difference in the accuracy of SZZ method with defects information over that SZZ method. Specifically, the p-value of the paired Wilcoxon Signed Rank test resulted less than 0.01 for Precision, Recall, F1, MCC, and Kappa in all SZZ methods other than Kappa between SZZ_B vs. SZZ_B+. Therefore, we can reject H_{S0F} in 14 out of 15 cases.

Regarding correlation accuracy Fig. 4.8 reports the distribution among datasets of correlation of each feature to class defectiveness. Statistical results show that there is a significant difference in the accuracy of methods in labeling classes. Specifically, the p-value of the Wilcoxon test resulted less than 0.01 and therefore we can reject H_{60} and claim that there is a difference among features in the correlation.

Fig. 4.9 reports the mean relative error, for each feature, of each method. According to Fig 4.9:

- the Proportion methods have a higher accuracy than all SZZ methods. Specifically, the proportion methods have an error that is about 25% of the SZZ methods’ error. Therefore, we can claim that observing correlations via defects’ life-cycle information is overall and in average more accurate than any SZZ method.

- Differently from both RQ2 and RQ3, the Simple method is more accurate than all SZZ methods. Specifically, the Simple method has an error that is about half of the SZZ methods’ error.

Statistical results show that there is a significant difference in the accuracy of methods in correlating features. Specifically, the p-value of the Wilcoxon test resulted less than 0.01. Therefore, we can reject H_{70C} and claim that different methods have different accuracy in correlating features.

Regarding comparing the accuracy of methods in correlating features of SZZ methods with defects’ information over that SZZ method (i.e., SZZ_x vs. SZZ_x+), our results show less than 1% of improvement. Statistical results on the 76 Apache projects
Figure 4.8: Distribution among datasets of correlation of each feature to class defectiveness.
Figure 4.9: Mean relative error of correlation, for each feature, of each method.
show that there is a significant difference in the accuracy of SZZ method with defects’ information over that SZZ method. Specifically, the p-value of the paired Wilcoxon Signed Rank test resulted less than 0.01 in all SZZ methods. Therefore, we can reject $H_{SOC}$ in all three cases.
Chapter 5

REPLICABILITY

In order to allow researchers to replicate this study we make available our scripts, the raw input data, and the processed data. Specifically, this can be found at https://gitlab.com/Bvandehei/affectversions. Note, not all files mentioned in previous sections exist in the repository due to lack of space, however, the scripts to create the files are there. Below, are steps to replicate each research question. Please note, files may need to be moved to different folders or file paths in the scripts may need to be changed for some scripts.

5.1 RQ1: Is the AV Available and Consistent?

1. A list of projects by JIRA ID and their corresponding Git URLs were found manually. This file is called ”ProjectsAndURLsEditted.csv”.

2. Next, compile and run ”QualityModel_Plus_BugInfo.java”. This will generate a file called ”QualityModel.csv” which contains various information about the AV of each project. This also generates various files for the versions and defects of each project which are used in later RQs.

3. Order ”QualityModel.csv” by the last column in ascending order, called ”OrderedQualityModel.csv”.

4. Compile and run ”GetTopProjects.java”. This returns a file containing the top projects with at least 100 defects, greater than 50% of available and consistent AV, and at least 6 versions, i.e. the 76 projects to use in the remaining RQs.

5.2 RQ2: Do Methods Have Different Accuracy in AV Labeling?

1. Run SZZ Unleashed using the bash script in U-SZZ. This will create a list of bug-fixing and bug-introducing commits for each project.
2. Run SZZ\_B and SZZ\_RA. These can be found at https://github.com/danielcalencar. The results of running these two can be found in folders ”bszzresults” and ”raszzresults” respectively.

3. Compile and run ”NewRetrievalMethodsReduced.java”. This will create files containing the defectiveness classification of versions per defect, the accuracy statistics of each method, and information regarding the ”P” values for the methods.

5.3 RQ3: Do Methods Have Different Accuracy in Classes Labeling?

1. Compile and run ”ComputeMetrics.java” for each project passing in the project name and git repository as arguments.

2. Compile and run ”RQ3.java” to create the Complete files.

3. Compile and run ”RQ3RA.java” to create the new SZZ\_RA Complete files that merge results for SZZ\_U for non-java files.

4. Compile and run ”RQ3Statistics.java” to retrieve the accuracy statistics for RQ3.

5.4 RQ4: Do Methods Lead to Different Identification of Important Features?

1. Compile and run ”RQ4FS.java” for a list of features selected for each project, method, version.

2. Compile and run ”RQ4FSStat.java” to retrieve the accuracy statistics for RQ4 Feature Selection and Feature Selection Frequency.

3. Compile and run ”RQ4SC.java” to the relative disagreement between Spearman’s Correlation of a method and feature verse actual and feature.
Chapter 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 regards issues that affect the ability to draw accurate conclusions about relations between the treatments and the outcome of an experiment [72].

We tested all hypotheses with non-parametric tests (e.g., Spearman) 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.

6.2 Internal

Internal validity regards the influences that can affect the independent variables with respect to causality [72]. 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 AV provided by developers might be inaccurate.

Results of RQ4 are related to the specific set of used features. It could be that results would differ by using other features. However our feature sets are large and related to the state of the art.

Results of RQ4 utilized the Spearman’s Rank correlation. This correlation may not be the best tool to use due to the fact that the class defectiveness is a binary variable (i.e. 0 or 1), and therefore, may result in lower correlation than alternative
6.3 Construct

Construct validity regards the ability to generalize the results of an experiment to the theory behind the experiment [72].

We use Precision, Recall, F1-Score, Matthews Correlation Coefficient, and Cohen’s Kappa to measure the accuracy in labeling defectiveness in RQ2 and RQ3. The same metrics are used RQ4 to measure accuracy of selecting features. Statistical tests, specifically the p-value of the Wilcoxon test, is used to ensure conclusions are robust.

6.4 External

External validity regards the extent to which the research elements (subjects, artifacts, etc.) are representative of actual elements [72].

This study used a large set of datasets and hence could be deemed of high generalization compared to similar studies.

Finally, in order to promote replicability, all datasets and scripts for this paper are available\(^1\).

\(^1\)https://gitlab.com/Bvandehei/affectversions
Chapter 7

CONCLUSION

This work focuses on automated methods for labeling a class in a version as defective or not. In this study we measure the AV availability and consistency in open-source projects, and we propose, evaluate, and compare to SZZ, a new method for labeling defective classes which is based on the idea that defects have a stable life-cycle in terms of proportion of defects needed to be discovered and to be fixed. Results related to 212 open-source projects from the Apache ecosystem, featuring a total of about 125,000 issue reports, show that the AV cannot be used in the majority (51%) of issue reports. Therefore, it is important to investigate automated methods for labeling defective classes. 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 the proposed method for labeling defective classes is, in average among projects and defects, more accurate, in terms of Precision, Kappa, F1, and MCC than all previously proposed SZZ methods. Moreover, the improvement in accuracy from combining SZZ with defects’ life-cycle information is statistically significant but practically irrelevant (<1%). Moreover, observing correlations via defects’ life-cycle information is overall and in average more accurate than any SZZ method. Moreover, selecting features and observing correlations information are both, overall and in average, significantly more accurate via defects’ life-cycle than any SZZ method. 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 accuracy of affect versions labeled by JIRA developers.** In our study, we only analyzed whether AV were available and consistent. Future work will focus on how accurate the AV entered by developers actual are.
do developers determine AV? Do they find labeling AV important?

- **Replication in context of JIT.** Just In Time (JIT) prediction models, here 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[47]. 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.
BIBLIOGRAPHY


