SNORING: A NOISE IN DEFECT PREDICTION DATASETS

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

Aalok Ahluwalia

June 2019

COMMITTEE MEMBERSHIP

TITLE:	Snoring: A Noise in Defect Prediction Datasets
AUTHOR:	Aalok Ahluwalia
DATE SUBMITTED:	June 2019
COMMITTEE CHAIR:	Davide Falessi, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Alexander Dekhtyar, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Bruno da Silva, Ph.D. Professor of Computer Science

ABSTRACT

Snoring: A Noise in Defect Prediction Datasets

Aalok Ahluwalia

Defect prediction aims at identifying software artifacts that are likely to exhibit a defect. The main purpose of defect prediction is to reduce the cost of testing and code review, by letting developers focus on specific artifacts. Several researchers have worked on improving the accuracy of defect estimation models using techniques such as tuning, re-balancing, or feature selection. Ultimately, the reliability of a prediction model depends on the quality of the dataset. Therefore effort has been spent in identifying sources of noise in the datasets, and how to deal with them, including defect misclassification and defect origin. A key component of defect prediction approaches is the attribution of a defect to a projects release. Although developers might be able to attribute a defect to a specific release, in most cases a defect is attributed to the release after which the defect has been discovered. However, in many circumstances, it can happen that a defect is only discovered several releases after its introduction. This might introduce a bias in the dataset, i.e., treating the intermediate releases as defect-free and the latter as defect-prone. We call this phenomenon a "sleeping defect". We call "snoring" the phenomenon in which classes are affected by sleeping defects only, that would be treated as defect-free until the defect is discovered. In this work, we analyze, on data from more than 4,000 bugs and 600 releases of 20 open source projects from the Apache ecosystem for investigating: 1)the magnitude of the sleeping defects, 2) the magnitude of the snoring classes, 3)if snoring impacts the evaluation of classifiers, 4)if snoring impacts classifier accuracy, and 5)if removing the last releases of data is beneficial in reducing the negative impact of the snoring noise on classifiers accuracy. Our results show that, on average across projects: 1)most of the defects in a project slept for more than 19% of the existing releases, 2)the missing rate is more than 50% unless we remove more than 20% of the releases, 3) the relative error in measuring the classifier accuracy achieved by using a dataset with snoring is about 100% in all accuracy metrics other than AUC, 4) the presence of snoring decreases the accuracy in each of the 15 classifiers, in each of the 6 accuracy metrics. For instance, Recall, F1, Kappa and Matthews decreases by about 80%, and 5) re-

moving one release of data is better than removing no data in all accuracy metrics. For instance, Recall, F1, Kappa and Matthews increase by about 30%.

ACKNOWLEDGMENTS

Thanks to:

- My adviser, Davide Falessi, for his continued support in the design and development of this project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 Context

Defect prediction aims at identifying software artifacts that are likely to exhibit a defect [41, 62]. The main purpose of defect prediction is to reduce the cost of testing and code review, by letting developers focus on specific artifacts.

Several researchers have worked on improving the accuracy of defect estimation models using techniques such as tuning [20, 23, 60], re-balancing [1, 4], or feature selection [61]. In order to promote the usage and improvement of prediction models, researchers have provided means to create [21, 64], collect [14] and select [22, 43, 54] datasets of real defects.

Ultimately, the reliability of a prediction model depends on the quality of the dataset [36, 56]. Previous works have identified sources of noise in datasets, including defect misclassification [5, 25, 35, 49, 59] and defect origin, [53] and proposed solutions to deal with them.

1.2 Definitions

A key component of defect prediction is the attribution of a defect to a particular release in a project. A defect can only be attributed to a specific release once it has been discovered. However, this introduces an imprecision, which we define as a "sleeping defect". Let us consider a project with three releases, *r1*, *r2*, and *r3*. If a defect has been actually introduced in *r1* and only discovered in *r3*, then the presence of the defect would not be considered for *r1* and *r2*. Now, if an artifact, say a class, does not exhibit in *r1* and *r2* any other defect but the one in question, such a class will erroneously be treated, in the dataset, as defect free. In other words, this is a false negative (FN) in a defect dataset. We call the status of this class in *r1* and *r2* as "snoring", as the class contains a sleeping defect that produces noise in the dataset.

Table 1.1: An example of a project in which events (I = injected, N = nothing, and F = fixed) happen in four releases and three different classes.

	r1	r2	r3
C1	IF	I	F
C2	I	N	F
C3	II	F	F

To better understand the concept of sleeping defects and how many releases defects sleep, Table 1.1 reports a scenario of a project in which events related to defects (I = injected, N = nothing, and F = fixed) happen in three releases (columns) and impact three classes (rows). A defect is defined to be a post-release defect if it is fixed in a release after the one it has been injected. Thus, a defective class is a class having at least one post-release defect. For instance, in column 2, row 2 of Table 1.1, a defect in class C1 is injected and fixed in the same release, *r1*. Instead, columns 3 and 4, row 2 of Table 1.1 show that a defect is injected in *r2* and fixed in *r3*. Therefore, class C1 is defective in *r2*. For instance, the defect injected in C1 at *r2* (Table 1.1) sleeps for zero releases since it is fixed in the next release. Instead the defect injected in C2 at *r1* (Table 1.1) sleeps for one release.

The intuition of the paper is that, as defects are known only when they are discovered (and fixed), this affects the construction of defect datasets, especially the most recent releases of a software project.

Table 1.2: The post-release defectiveness of a class in a specific release as computed at *r2*.

Classes	r1
C1	ND
C2	ND
C3	D

Table 1.3: The post-release defectiveness of a class in a specific release as computed at *r3*.

Classes	r1	r2
C1	ND	D
C2	D	D
C3	D	D

To better illustrate this intuition, Table 1.2 reports the dataset created at *r2*, and Table 1.3 reports a dataset created at *r3*, based on the events described in Table

1.1. Each class in each release is marked as defective (D) or not defective (ND). It is important to note that the status of C2 in *r1* changes based on whether the dataset is created at the end of *r2* or *r3*. Specifically, if the dataset is created at the end of *r2*, then the injection is not discovered and hence the class C2 at *r1* is marked as not defective.

Table 1.4: The accuracy of the dataset maker performed in *r2*.

Classes	r1
C1	TN
C2	FN
C3	P

Table 1.5: The accuracy of the dataset maker performed in *r3*.

Classes	r1	r2
C1	TN	P
C2	P	P
C3	P	P

Table 1.4 and Table 1.5 report the accuracy of a specific class in a specific release according to when the dataset is created in terms of FN = the class is erroneously marked as not defective despite being defective, TN = the class is marked as not defective and is not defective and P = the class is marked as defective. Specifically, C2 at *r1* is a FN for a dataset created at the end of *r2* (Table 1.4) and a P for dataset created at *r3* (Table 1.5). Note that we only consider FN and not FP because the snoring noise only introduces FN, i.e., classes considered as defect-free while they should be defect-prone, and not the other way around.

1.3 Aim

Both Perez et al. [50] and Costa et al.[10] show that the time to fix a defect, i.e., sleeping, is on average about one year. Thus, we conclude that dataset creation will miss most defects on releases that are less than a year old. One possible approach aimed at identifying when a defect has been actually introduced in a software project is the SZZ algorithm [58]. 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 such a fix. In its improved version [32], SZZ enhances the simple annotation feature with heuristics such as excluding cosmetic changes and comments. Truly, SZZ is not perfect. While it has been adopted

for many purposes, including building just in time defect prediction models [34, 40], different works have identified its limitations [10, 51, 52]. For instance, SZZ cannot find the correct location of bugs that are fixed by adding code [10]. Nevertheless, even when one is able to correctly attribute a defect to a release (e.g. using SZZ or even manually), the problem of sleeping defects still persists because when a defect dastset is constructed, some of the defects in the recent releases might not have been discovered or fixed yet. As a result, some classes might be treated as defect-free.

The aim of this work is to investigate the snoring phenomenon and its impacts on defect prediction accuracy. Specifically, we investigate the following research questions:

- **RQ1**: **To what extent do defects sleep?** We are interested in measuring how many releases elapse between the injection and fix of defects. These are the sleeping defects that are unknown to the dataset maker.
- **RQ2**: **To what extent do classes snore?** A sleeping defect does not always produce one snoring class since 1) another defect exists and it is not snoring, i.e., the class is marked defective despite some defects are unknown, 2) a single defect can impact, make snoring, multiple classes. Thus, we are interested in measuring to what extent classes snore.
- **RQ3:To what extent does snoring impact the evaluation of classifiers accuracy?** Classifier accuracy can be evaluated on several independent variables such as their type [18] or the selection of their parameters [20, 60] . Since snoring impacts classifier accuracy, then snoring likely biases classifier evaluation, including the one in the present work. Since evaluations have noise in both the training and test sets, the two snoring effects could counterbalance each other, thus having an insignificant (combined) effect to classifiers accuracy. In other words, we want to investigate if evaluation studies need to remove snoring data as user of classifier do (RQ3).
- **RQ4**: **To what extent does snoring impact defect prediction accuracy?** In this research question, we are interested in measuring how much the presence of the snoring noise in defect prediction datasets impacts the accuracy of classifiers.

- **RQ5**: **To what extent is no data better than snoring data?** Since snoring data decreases prediction accuracy (see RQ3), we should remove noise from the data. Unfortunately, in a realistic prediction context, the only way to remove noise in the data is to avoid the use of potentially noisy data. At the time of prediction, more recent releases will contain more noise. We cannot wait for these releases to age and become less noisy. Therefore, in this research question, we investigate if the use of data potentially snoring is better than avoiding the use of this data. become no recent anymore to make the prediction.

1.4 Accuracy Metrics

The following accuracy metrics will be used to assess the accuracy of classifiers:

- **Precision**
- **Recall**
- **F1**
- Cohen's **Kappa** : A statistic that assesses the classifier's performance against random guessing. A measure of agreement between two raters classifying records into mutually exclusive categories.
- **AUC** (Area Under Curve)
- **Matthews** Correlation Coefficient : A correlation coefficient used to assess binary classifications.

1.5 Method and Results

Our empirical procedure consists in analyzing 20 Apache projects featuring a total of more than 4,000 bugs and 600 releases. We performed a 66/33 holdout and used 15 classifiers. Our results show that, in average across projects:

1. Most of the defects in a project slept for more than 19% of the existing releases.

2. The missing rate is more than 50% unless we remove more than 20% of the releases.

3. The relative error in measuring the classifiers' accuracy achieved by using a dataset with snoring is about 100% in all accuracy metrics other than AUC.

4. The presence of snoring decreases the accuracy in each of the 15 classifiers, in each of the 6 accuracy metrics. For instance, Recall, F1, Kappa and Matthews decrease by about 80%.

5. Removing one release of data is better than removing no data in all accuracy metrics. For instance, Recall, F1, Kappa and Matthews increases by about 30%.

1.6 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

2.1 Defect prediction

Menzies et al. [41] reported on the current results, limitations, and new approaches of defect prediction from static code features. They advise against the indiscriminate use of classifiers, suggesting to choose and customize the classifiers to the goal at hand.

Turhan et al. [62] 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.

Fu et al. [20] 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. [60] 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) assess 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 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

Kim et al. [35] 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.

Tantithamthavorn et al. [59] 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.

Herzig et al. [25] reported that 39% of files marked as defective actually never had a bug. They discuss the impact of this misclassification on earlier studies and recommend manual data validation for future studies.

Rahman et al. [49] 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.

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.

2.3 Defect prediction datasets creation

Śliwerski et al. [58] 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 fix, when they have last been changed before such a fix.

Kim et al. [33] presented algorithms to automatically and accurately identify bug-introducing changes which improves on SZZ.

Da Costa et al. [9] 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. [45] 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 [14] 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. [53] investigated the complex phenomenon of bug introduction and bug fix. They show that less than 30% of bugs 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

Gousios and Spinellis [22] proposed the Alitheia Core analysis platform which preprocesses repository data into an intermediate format that allows researchers to provide custom analysis tools.

Rozenberg et al. [54] 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. Nagappan et al. [43] 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).

Falessi et al. [16] 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.

Chapter 3

STUDY DESIGN

3.1 RQ1: To what extent do defects sleep?

In this research question we are interested in investigating the extent of the sleeping phenomenon. The number of releases a bug sleeps is measured as the number of releases between the release when the bug is injected to the release when the bug is fixed.

3.1.1 Measurement Procedure

The study *context* consists of data from 20 open source projects from the Apache ecosystem. We focused on Apache[1] projects rather than random GitHub projects because the former have a higher quality of defect annotation and to avoid using toy projects [42]. We select the 20 projects that are managed in JIRA, versioned in Git, have at least 8 releases, have most of the commits related to Java code and have the highest proportion of bugs linked to commits in the source code. A bug is linked if it can be associated with some commit in the source code's commit log. Table 3.3 reports the details of the used twenty projects in terms of: releases, days, commits, and defects.

To compile a set of bugs, we query Jira for the ids of all issues with type "Bug" and status "Closed". We now must link this set of bugs with the commit log in the source code repository. For each bug, we walk over the commit log and at each commit we decide to associate a commit with a bug if its bug id is found in the commit message. In the case that there are multiple bug ids in the commit message, we associate the commit with the first bug mentioned in the message. We assume that a commit does not fix more than one bug.

We will use the tags on Git commits to identify the times and names of releases.

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

Table 3.1: Details of the used projects.

Project	Releases	Days	Commits	Defects
AVRO	46	3528	1770	114
CHUKWA	11	3638	849	7
FALCON	31	2450	2227	219
GIRAPH	10	2719	1096	123
IVY	17	4977	2973	133
OPENJPA	31	4671	4978	380
PROTON	51	2513	3929	53
SSHD	26	3652	1589	124
STORM	36	2634	9754	442
THRIFT	12	3926	5627	72
WHIRR	8	1788	569	20
ZEPPELIN	14	2067	4048	200
ZOOKEEPER	49	3923	1820	219
BOOKKEEPER	22	2880	2056	184
CONNECTORS	118	3311	4672	261
CRUNCH	17	2663	1055	132
SYNCOPE	48	3214	6320	296
TAJO	13	2407	2273	286
TEZ	34	2163	2661	559
TOMEE	22	4792	12135	277

Given a particular bug, we find the release that introduced such bug by using a mixed approach. If information about the "Affect Version" for a bug is available in Jira, then we consider the earliest Affect Version in Jira to be the release that introduced the bug. Otherwise, we use SZZ. 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 such a fix. We re-implemented the SZZ algorithm [58]. We then applied the SZZ algorithm in correspondence with each bug fix without information about the affect version. When applying SZZ, we ignored comments[2], indentation, white spaces, and documentation strings[3] as changes introducing defects. We tagged as defective the least recent change of the potential bug-introducing changes.

3.2 RQ2: To what extent do classes snore?

3.2.1 Dependent Variables

In this research question we are interested in investigating the extent of the snoring phenomenon. Specifically, for each class, we will have a measure of a class' defectiveness at different releases. Thus, there are 3 possible combinations for classes:

- True Positive(TP): A class is measured to be defective both at the time of the given release and at the time of investigation
- True Negative(TN): A class is measured to be non-defective both at the time of the given release and at the time of investigation
- False Negative(FN): A class is measured to be non-defective at a given time, but later found to contain a defect. This is an instance of a snoring class

Observe that there is no false positive, meaning a class can't be perceived to be defective at a given release and later on observed to be non-faulty. At each release we will calculate the missing rate. The missing rate is also called false negative rate,

[2]https://goo.gl/X8fHFc
[3]https://goo.gl/dNXb6N

Type II error [46] or 1 - Recall, and in our case it regards the proportion of defective classes that are not identified as defective.The missing rate can be expressed as

$$MissingRate = \frac{FN}{FN + TP} \tag{3.1}$$

3.2.2 Measurement Procedure

In order to minimize the effect of snoring on our measurements, we consider only the sub-dataset consisting of the first 5% of releases per project. Finally, since the missing rate likely depends on the time distance between defect introduction and its measurement, we measure how the missing rate of the 5% sub-dataset varies throughout different releases. Thus, we define Progress as the release in which the sub-dataset is measured divided by the total number of releases. For instance, progress is 100% when measuring the sub-dataset from the last release of the project.

For example, the class BookKeeper.java[4] in the Bookkeeper project is marked as non-defective in release 1 as measured at the end of the 5% subdataset, at the end of release 2. However, when measured after release 3, the class would be marked as defective at release 1 because a bug with JIRA id BOOKKEEPER-327[5] has been fixed during release 3.

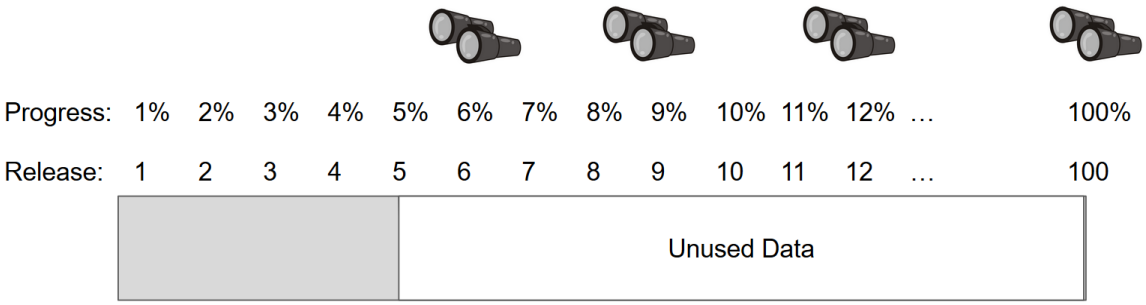

Figure 3.1: An example of progress for a project with 100 releases.

[4]https://github.com/apache/bookkeeper/blob/master/bookkeeper-server/src/main/java/org/apache/bookkeeper/client/BookKeeper.java
[5]https://issues.apache.org/jira/browse/BOOKKEEPER-327

3.3 RQ3:To what extent does snoring impact the evaluation of classifier accuracy?

Since snoring impacts the accuracy of classifier, snoring likely biases the evaluations of classifiers too, including the one in the present work. However, since evaluations have noise in both the training and test sets, then the two snoring effects could counterbalance each other thus providing an insignificant (combined) effect to classifiers accuracy. In this research question we propose the following null hypothesis: H10: There is no difference between the prediction accuracy evaluated with versus without snoring noise.

3.3.1 Independent Variables

The independent variable for this research question is the presence or absence of snoring noise in both train and testing set.

3.3.2 Dependent Variables

This research question has the following dependent variables.

1. **Relative bias**. We measured relative bias (RB) as the relative distance between the accuracy of a classifier trained on noise free datasets versus the accuracy of the same classifier on a the same dataset with noise. Specifically,

   $$RB = \frac{|accuracywithoutsnoring - accuracywithsnoring|}{accuracywithoutsnoring}.$$

2. **Selection accuracy**. We measured selection accuracy as the event when the best classifier, for a specific accuracy metric, and project, identified used a dataset with snoring versus no snoring.

3. **Ranking accuracy**. We measured ranking accuracy as the correlation between the ranking of classifiers, for a specific accuracy metric and project, identified using a dataset with snoring versus no snoring.

We measure accuracy according to the following metrics:

- True Positive(TP): The class is measured to be defective at the end of the project and is predicted to be defective.
- False Negative(FN): The class is measured to be defective at the end of the project and is predicted to be non-defective.
- True Negative(TN): The class is measured to be non-defective at the end of the project and is predicted to be non-defective.
- False Positive(FP): The class is measured to be non-defective at the end of the project and is predicted to be 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
- **AUC** (Area Under Curve)
- **Matthews** Correlation Coefficient

3.3.3 Measurement Procedure

To answer this research question, we emulate a scenario in which the complete dataset is used for classifier evaluation using a 66/33 ordered holdout approach. We use the first 66% of data to construct the training set and the next 33% of data to construct the test set. We compare the results of an evaluation biased by snoring with an evaluation using the same exact releases, but observed at a release much later in the future so that measurements are free of snoring.

3.3.3.1 Noise removal

Given a dataset **D**, we created **SD** by removing several last releases to avoid having our measurement itself be affected by snoring. To do this, we used the approach detailed in our previous paper [3]. Specifically we observed that in average among our 18 projects, the missing rate is more than 50% unless we remove more than 20% of the releases. To create a snoring free portion of our dataset, we removed several releases from the end so that the expect amount of snoring in the last considered release is 1%. For instance, the number of releases required to have snoring at 1% is 50% in the Storm dataset. Thus as a result, we neglected 18 of the 36 releases of the Storm project. Thus, we are confident that our ground truth is not affected by snoring. The subdataset of D without snoring is called *SD*.

3.3.3.2 Train and test sets

This step consists of two sub steps.

1. We split SD into a training and testing set by using the first 66% and the last 33% respectively. Those subdatsets are called *TrNS* and *TeNS*, respectively. The *NS* indicates the absence of snoring noise in the dataset.

2. We create a training set with snoring called *TrS*. To do so, we measure the bugginess of a class in a release as known in the last release of Tr; as it is realistically. Note that the bugginess of a class in a release in TrNS and TeNS is measured as known in the last release of D.

For example, in project Bookkeeper, the training set (TR) is constructed using the first 8 releases. Class BookieProtocol.java in releases 7 and 8 of TrS is marked as non-defective, but will become defective if observed after release 11 because of the bug BOOKKEEPER-1018[6]. Class BookieProtocol.java in releases 7 and 8 of TrNS is marked as defective.

[6]https://issues.apache.org/jira/browse/BOOKKEEPER-1018

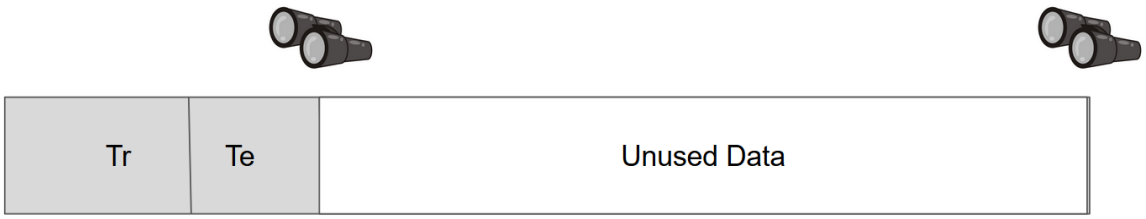

Figure 3.2: Measure defective status of classes once at the end of SD and at D

3.3.3.3 Predictor metrics computation

We use 17 well-defined product and project metrics that have been shown to be useful for defect prediction [11, 15]. The used metrics are detailed in Table 3.2. We note that TrNS and TrS share all predictor metrics and values, and differ only by the measurement of buginess.

3.3.3.4 Accuracy measurement

This step consists of two sub steps.

1. Feature Selection. We filter the predictor variables presented above by using correlation-based featured subset selection [24].

2. Classifiers. As classifiers we use the 14 used in previous related paper [30]:

   - Decision Stump : A single level decision tree performing classification based on entropy [28].
   - Decision Table : Two major parts : schema, the set of features included in the table, and a body, labelled instances defined by features in the schema. Given unlabelled instance, try matching instance to record in the table. [37]
   - IBk : K-nearest neighbors classifier run with k = 1 [2].
   - J48 : Generates a pruned C4.5 decision tree [48].
   - JRip : A propositional rule learner, Repeated Incremental Pruning to Produce Error Reduction (RIPPER) [8].

Table 3.2: 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

- KStar : Instance-based classifier using some similarity function. Uses an entropy-based distance function [7].
- Naive Bayes - Classifies records using estimator classes and applying Bayes theorem[29]
- Naive Bayes Updateable - An instance of the Naive Bayes classifier with different weight initial values and constraints[29]
- OneR - 1R classifier using the minimum-error attribute for prediction [27].
- PART - Uses seperate-and-conquer, building partial C4.5 decision trees and turning the best leaf into a rule [19].

- Random Forest - Ensemble learning creating a collection of decision trees. Random trees correct for overfitting[6].
- REPTree - Fast decision tree learner. Builds decision tree using information gain and variance, and prunes using reduced-error pruning [39].
- SMO : John Platt's sequential minimal optimization algorithm for training a support vector classifier [47]

3. Tuning. Instead of tuning each single classifiers, we used AutoWEKA [38], an automated approach for classifier selection and hyperparameter optimization. We run AutoWEKA for two hours on each training set. Thus we refer to *AutoWEKA* as the classifier and parameter selected by AutoWEKA for the specific training set.

4. Test. We train each of the 14 classifiers, plus the AutoWEKA classifier, on TrNS and TrS. We test the trained classifiers on TeNS.

3.3.4 Analysis Procedure

The hypotheses of this and the following research questions are tested using the Kruskal-Wallis test [57] which is similar to the more famous Anova test, but does not require any assumptions about the distribution of the data. Because the non-parametric Kruskal-Wallis test is less powerful than the Anova test, it is more prone to not rejecting hypotheses when they actually ought to be rejected. On the other hand, when rejecting a hypothesis, it is more reliable than Anova. Moreover, the Kruskal-Wallis test is particularly recommended when the compared distributions are not independent. In our case, the distributions are computed over the same projects and releases and hence are not independent.

3.4 RQ4: To what extent does snoring impact defect prediction accuracy?

Given that there are snoring classes, noisy data points exist. The aim of this research question is to measure the impact of snoring classes on the accuracy of defect prediction. In this research question, we propose the following null hypothesis: H10:

There is no difference between the prediction accuracy achieved with versus without snoring.

3.4.1 Independent Variables

The independent variable for this research question is the presence or absence of snoring noise in the training set.

3.4.2 Dependent Variables

The dependent variable is the accuracy in defect prediction as in RQ1.

3.4.3 Measurement procedure

Our procedure is very similar to the one of RQ1. Specifically, we created Tr with snoring by measuring it as observed by the end of Tr. We created Tr without snoring by measuring it as observed by the end of D. Both datasets Tr will be used to train the classifiers presented before and tested on the Te without testing, i.e., a Te measured as observed by the end of D.

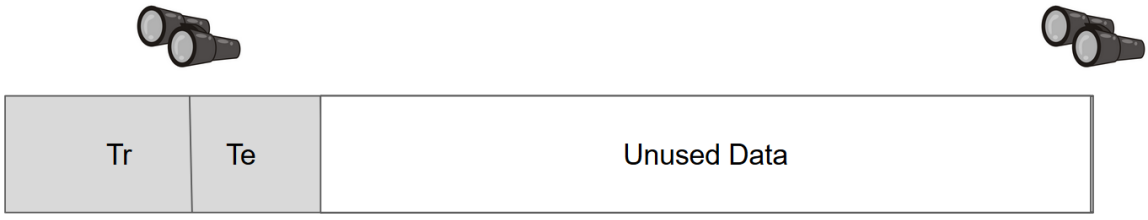

Figure 3.3: The defect status is computed after the last release in the test set and at the end of the project

3.5 RQ5: To what extent is no data better than snoring data?

Given that noisy data points do exist and impact the defect prediction task, this research question aims to determine if removing noisy data improves defect prediction. Since more recent releases are more likely to snore, removing them, may make the

model more accurate. Removing noisy data improves the quality of the data, but reduces the amount of data the model can train on. In this research question, we propose the following null hypothesis: H30: There is no difference between the prediction accuracy achieved by removing versus not removing the last release of data.

3.5.1 Independent Variables

The independent variable for this research question is the number of releases to eliminate.

3.5.2 Dependent Variables

This research question has the same dependent variables as RQ3.

3.5.3 Analysis Procedure

To answer this research question, we used the same training set affected by snoring used in RQ3. Afterwards, we remove 0,1,2,3, and 4 releases of data, we call these TrS0, TrS-1, TrS-2, TrS-3, TrS-4. Next, we use the same models in RQ3 and train them on the trimmed datasets. Finally, we test our models on a test set without noise TeNS (as in RQ3).

For example, removing one release from the training set of the Bookkeeper project, to form TrS-1, would eliminate all False defect-status measurements from release 8, the last release in the train dataset. In the example above, removing release 8 would remove classes that snore at that release such as ZkLedgerUnderreplicationManager.java [7].

[7] https://github.com/apache/bookkeeper/blob/master/bookkeeper-server/src/main/java/org/apache/bookkeeper/meta/ZkLedgerUnderreplicationManager.java

Table 3.3: Details of the used projects.

Project	Releases	Days	Commits	Defects
AVRO	46	3528	1770	114
CHUKWA	11	3638	849	7
FALCON	31	2450	2227	219
GIRAPH	10	2719	1096	123
IVY	17	4977	2973	133
OPENJPA	31	4671	4978	380
PROTON	51	2513	3929	53
SSHD	26	3652	1589	124
STORM	36	2634	9754	442
THRIFT	12	3926	5627	72
WHIRR	8	1788	569	20
ZEPPELIN	14	2067	4048	200
ZOOKEEPER	49	3923	1820	219
BOOKKEEPER	22	2880	2056	184
CONNECTORS	118	3311	4672	261
CRUNCH	17	2663	1055	132
SYNCOPE	48	3214	6320	296
TAJO	13	2407	2273	286
TEZ	34	2163	2661	559
TOMEE	22	4792	12135	277

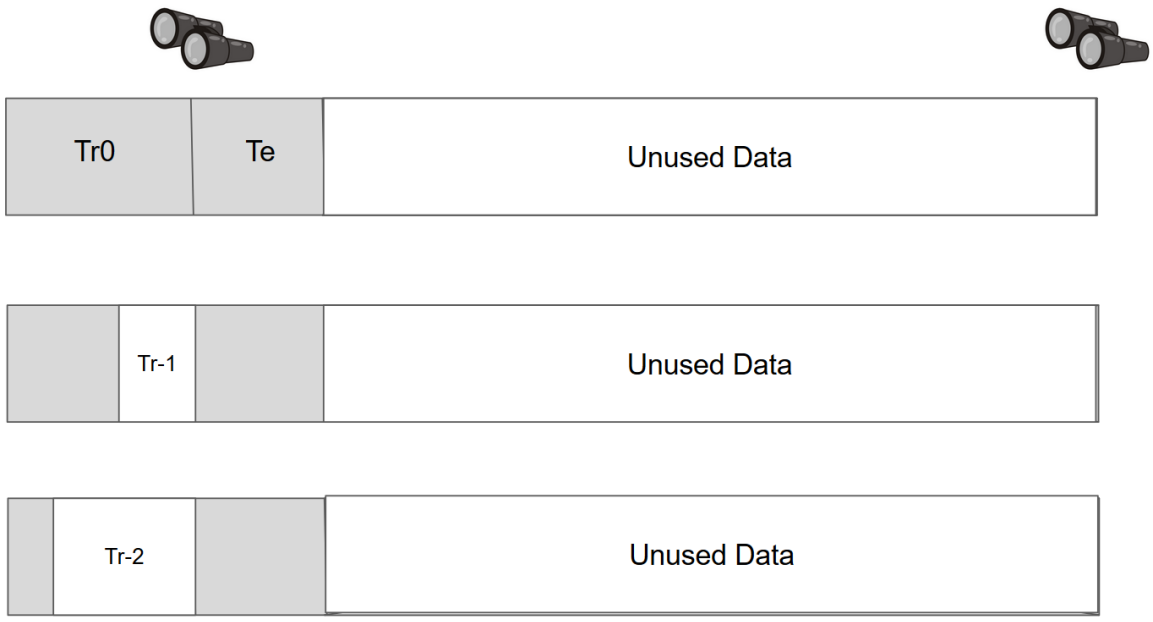

Figure 3.4: Trimming releases from Tr

Chapter 4

RESULTS

4.1 RQ1:To what extent do defects sleep?

Fig. 4.1 reports the distribution of the number of releases slept among different projects. According to Fig. 4.1, the mean bug sleeps for 7 releases.

Fig. 4.2 reports the distribution of the percentage of releases slept among different projects.

4.2 RQ2:To what extent do classes snore?

Fig. 4.3 reports the missing rate (y-axis) of the sub-dataset comprising the first 5% of the releases of a project, computed at different levels of progress in the project (x-axis). We note that the observed releases do no change, they are always the 5% of the dataset; the only variable is when the releases are observed. Fig. 4.4 better explains results of Fig. 4.3 by reporting the minimum progress required to have a specific missing rate in a specific project. Table 4.1 synthesizes Fig. 4.4 by reporting the median among datasets of the minimum percentage of releases that needs to be removed to achieve a specific missing rate.

4.3 RQ3:To what extent does snoring impact the evaluation of classifier accuracy?

Fig. 4.5 reports the distribution of accuracy, among classifiers and datasets, achieved with versus without snoring.

Table 4.2 reports the statistical test results on the difference in accuracy among classifiers and datasets, achieved with versus without snoring.

Fig. 4.6 reports the distribution of bias, among classifiers and datasets. Table 4.3

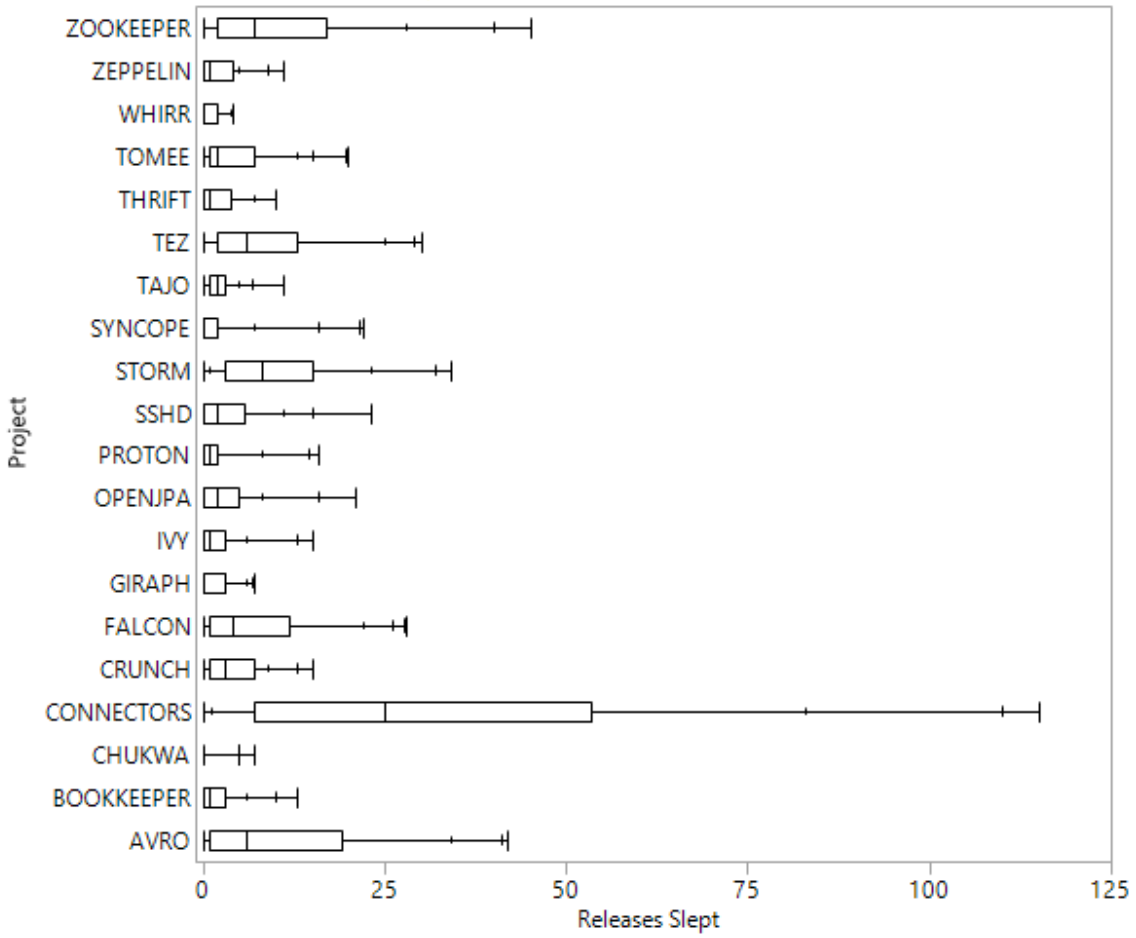

Figure 4.1: Distribution of the number of releases slept by post-release defects.

Table 4.1: Median among datasets of the minimum percentage of releases (Progress) that needs to be removed to achieve a specific missing rate.

Missing Rate	Progress
0	0.71
0.01	0.71
0.05	0.62
0.1	0.49
0.25	0.3
0.5	0.2

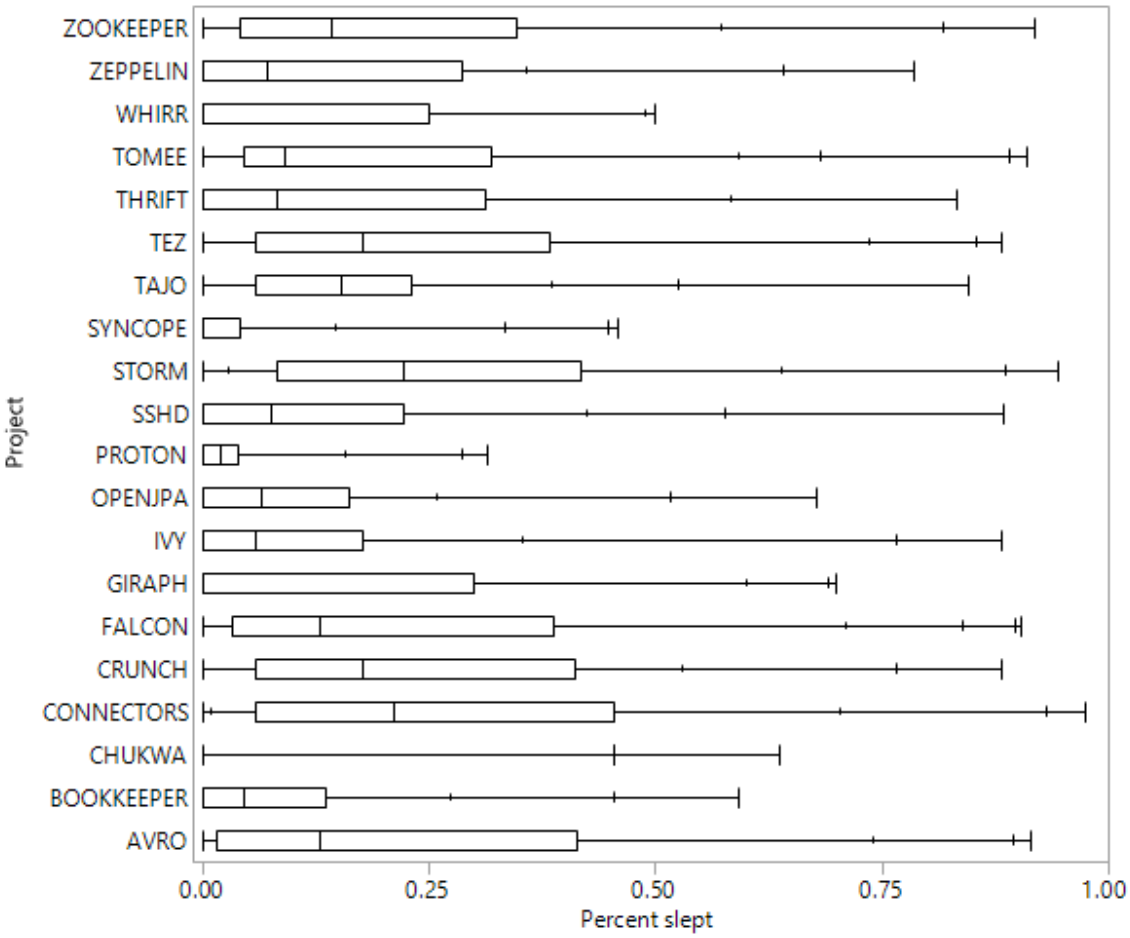

Figure 4.2: Distribution of the number of releases slept by post-release defects.

Table 4.2: Statistical test results on the difference in accuracy, among classifiers and datasets, achieved with versus without snoring.

Metric	chi-square	p-value
Precision	183.3573	<0.0001
Recall	0.0138	0.9064
F1	115.6406	<0.0001
Kappa	91.6156	<0.0001
Matthews	80.3604	<0.0001
AUC	9.1938	0.0024

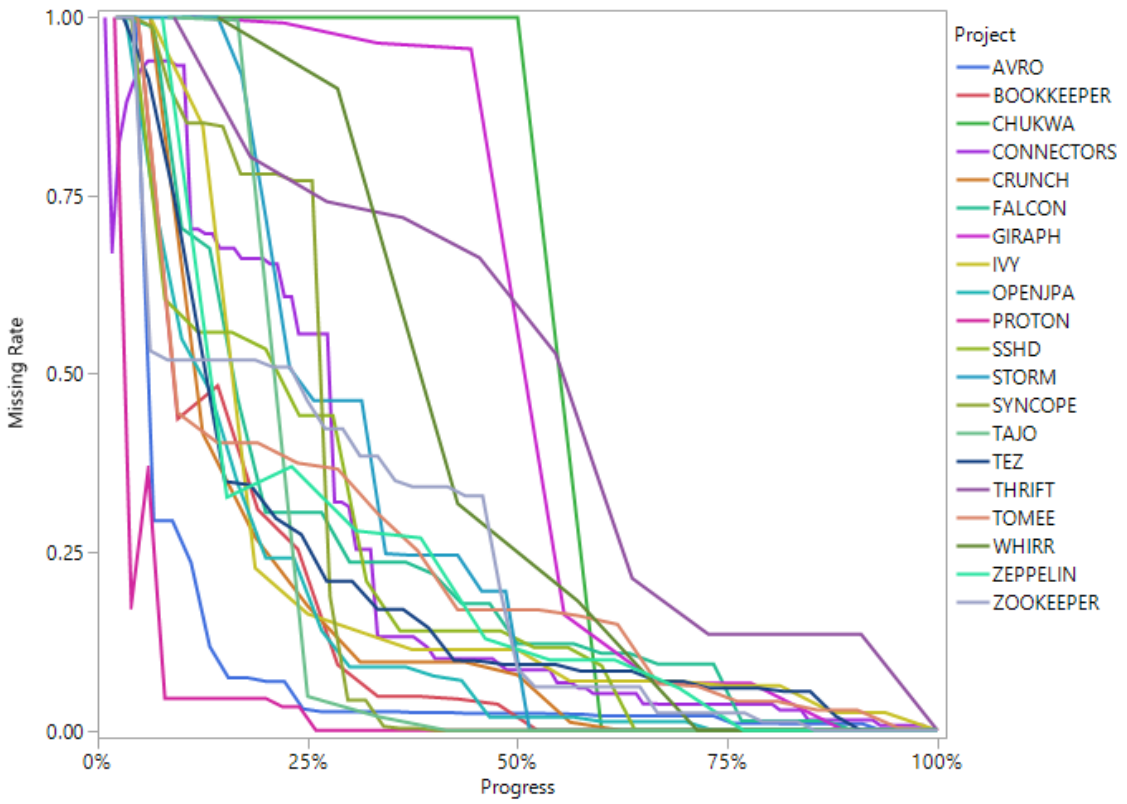

Figure 4.3: Defect-prone classes missing rate for the first 5% releases, observed at different levels of progress in a project.

Table 4.3: Average relative bias (%), among classifiers and datasets, in specific metrics.

Precision	Recall	F1	Auc	Kappa	Matthews
0.98	1.11	0.96	0.11	1.21	0.92

summarizes Fig. 4.6 by reporting the average bias,among classifiers and datasets, in specific metrics.

Table 4.4 reports the cases in which the evaluation affected by snoring was able to select the actual best classifier.

Table 4.5 reports the correlation between the ranking of classifiers provided by the evaluation affected by snoring and the actual ranking of classifiers.

Table 4.4: Cases in which the evaluation affected by snoring was able to select the best classifier.

Project	Precision	Recall	F1	Kappa	Matthews	AUC
BOOKKEEPER	No	Yes	No	No	No	Yes
CHUKWA	Yes	Yes	Yes	Yes	Yes	Yes
CONNECTORS	Yes	Yes	Yes	Yes	Yes	No
CRUNCH	Yes	Yes	Yes	Yes	Yes	No
FALCON	No	Yes	No	No	No	Yes
GIRAPH	Yes	Yes	Yes	Yes	Yes	No
IVY	No	No	Yes	Yes	No	Yes
OPENJPA	No	No	No	No	No	No
PROTON	Yes	No	No	No	No	Yes
SSHD	No	Yes	No	No	No	Yes
STORM	Yes	Yes	No	No	No	Yes
SYNCOPE	No	No	Yes	Yes	Yes	No
TAJO	No	No	No	No	No	No
TEZ	No	Yes	No	No	No	Yes
TOMEE	Yes	Yes	Yes	Yes	Yes	No
WHIRR	No	Yes	No	No	No	Yes
ZEPPELIN	No	No	No	No	No	No
ZOOKEEPER	No	Yes	No	No	No	Yes

Table 4.5: Correlation between the ranking of classifiers provided by the evaluation affected by snoring and the actual ranking of classifiers.

Project	AUC	F1	Kappa	Matthews	Precision	Recall
BOOKKEEPER	0.99	-0.10	-0.10	0.15	0.16	0.88
CHUKWA	0.07	0.46	0.29	0.38	0.24	0.54
CONNECTORS	0.00	0.00	0.00	0.00	0.00	0.00
CRUNCH	0.00	0.00	0.00	0.00	0.00	0.00
FALCON	0.81	0.23	0.23	0.26	0.06	0.36
GIRAPH	0.67	0.73	0.73	0.68	0.73	0.00
IVY	0.95	0.62	0.66	0.70	0.77	0.34
OPENJPA	0.85	-0.02	0.01	0.22	0.55	0.38
PROTON	0.92	0.30	0.30	0.06	0.49	0.22
SSHD	0.86	-0.17	-0.17	-0.16	0.21	0.11
STORM	0.96	0.70	0.71	0.85	0.67	0.91
SYNCOPE	-0.44	-0.25	0.01	-0.03	-0.35	-0.01
TAJO	0.01	-0.38	-0.14	0.13	0.91	0.04
TEZ	0.25	-0.22	-0.22	-0.25	-0.06	-0.20
TOMEE	0.00	0.00	0.00	0.00	0.00	0.00
WHIRR	0.91	0.62	0.39	0.47	0.69	0.58
ZEPPELIN	0.12	-0.83	-0.84	-0.92	0.29	-0.41
ZOOKEEPER	0.20	0.28	0.37	0.37	0.13	0.14

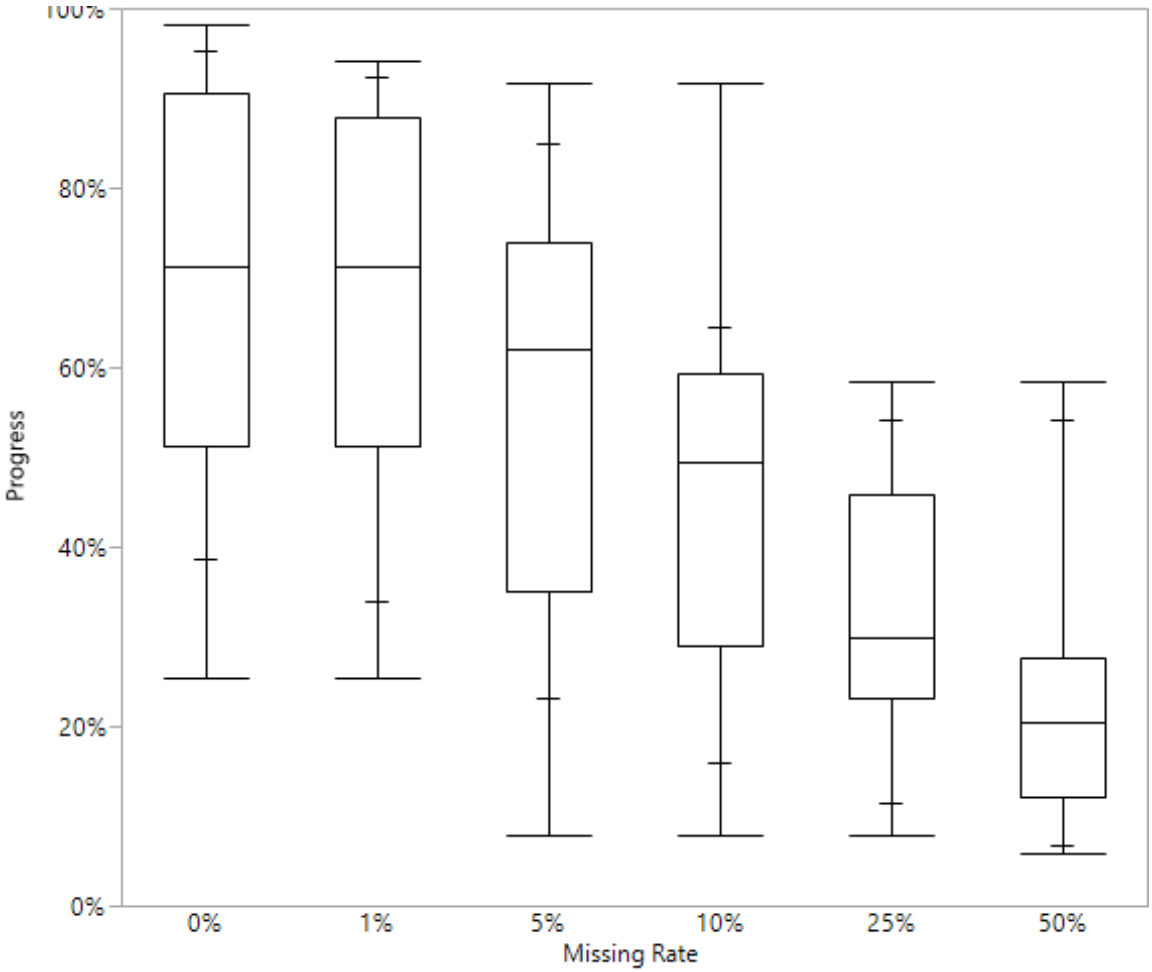

Figure 4.4: Distribution, among datasets, of percentage of releases (Progress) that needs to be removed to achieve a specific missing rate.

4.4 RQ4:To what extent does snoring impact defect prediction accuracy?

Fig. 4.7 reports the distribution,of accuracy, among datasets, of different classifiers (x-axis), in the cases of with, or without, the snoring noise.

Table 4.7 reports the statistical test results on the difference in accuracy, in each classifier, in the cases of with, or without, the snoring noise.

Table 4.6: Relative loss (%) in accuracy with the presence of the snoring noise, in average across classifiers and datasets.

Precision	Recall	F1	Kappa	Matthews	AUC
-0.31	-0.83	-0.77	-0.80	-0.73	-0.26

Table 4.7: Statistical test results on the difference in accuracy, in each classifier, in the cases of with, or without, the snoring noise.

Metric	chi-square	p-value
Precision	31.5075	<0.0001
Recall	227.2857	<0.0001
F1	197.7716	<0.0001
Kappa	176.4255	<0.0001
Matthews	168.3359	<0.0001
AUC	287.8487	<0.0001

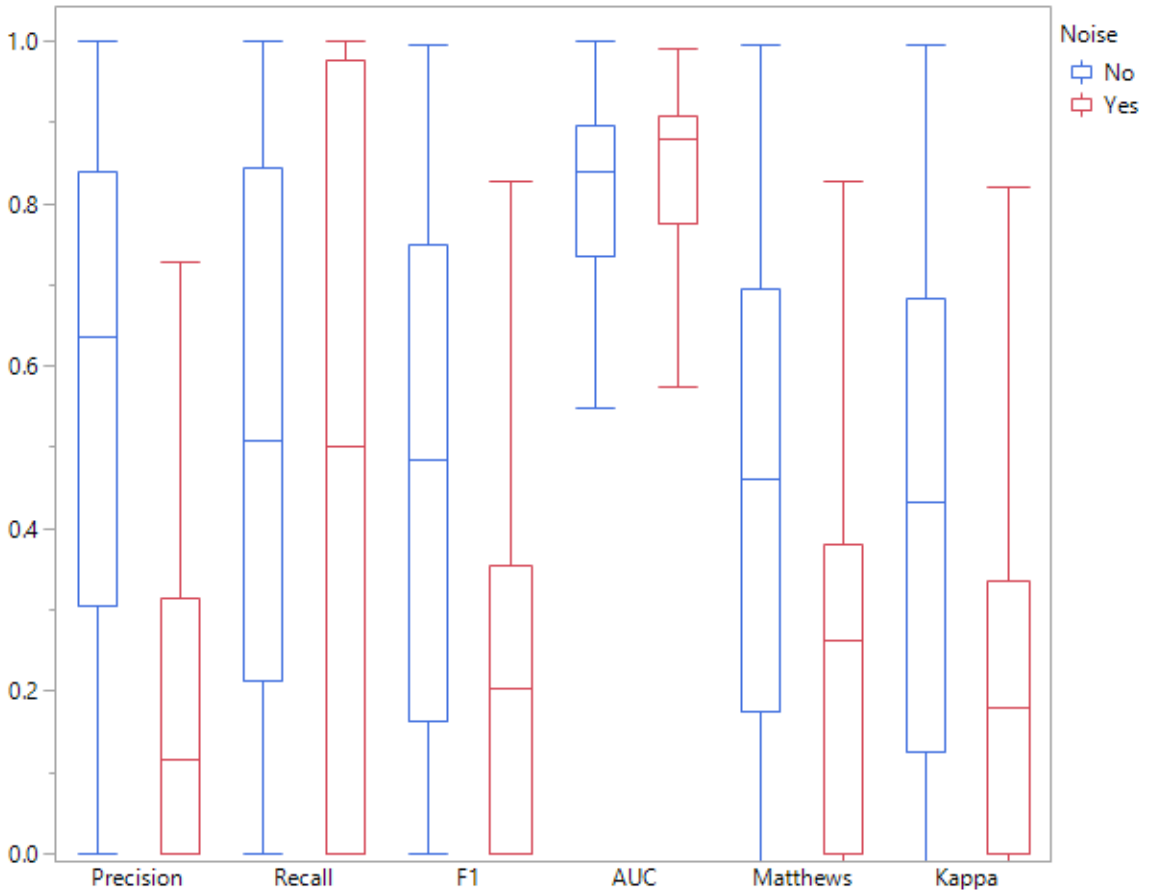

Figure 4.5: Distribution of accuracy, among classifiers and datasets, achieved with versus without snoring.

4.5 RQ5:To what extent is no data better than snoring data in supporting accurate defect prediction?

Fig. 4.8 reports the distribution among datasets and classifiers, of different accuracy metrics (x-axis), when removing of a specific number of last releases data (color). Table 4.8 summarizes Fig. 4.8 by reporting the relative gain (%), in average among datasets and classifiers, by removing a specific number of releases of data, in average across classifiers and datasets.

Fig. 4.9 reports the Accuracy achieved by removing or not the last release of data, in each classifier.

Table 4.9 reports the statistical test results on the difference in accuracy, in the

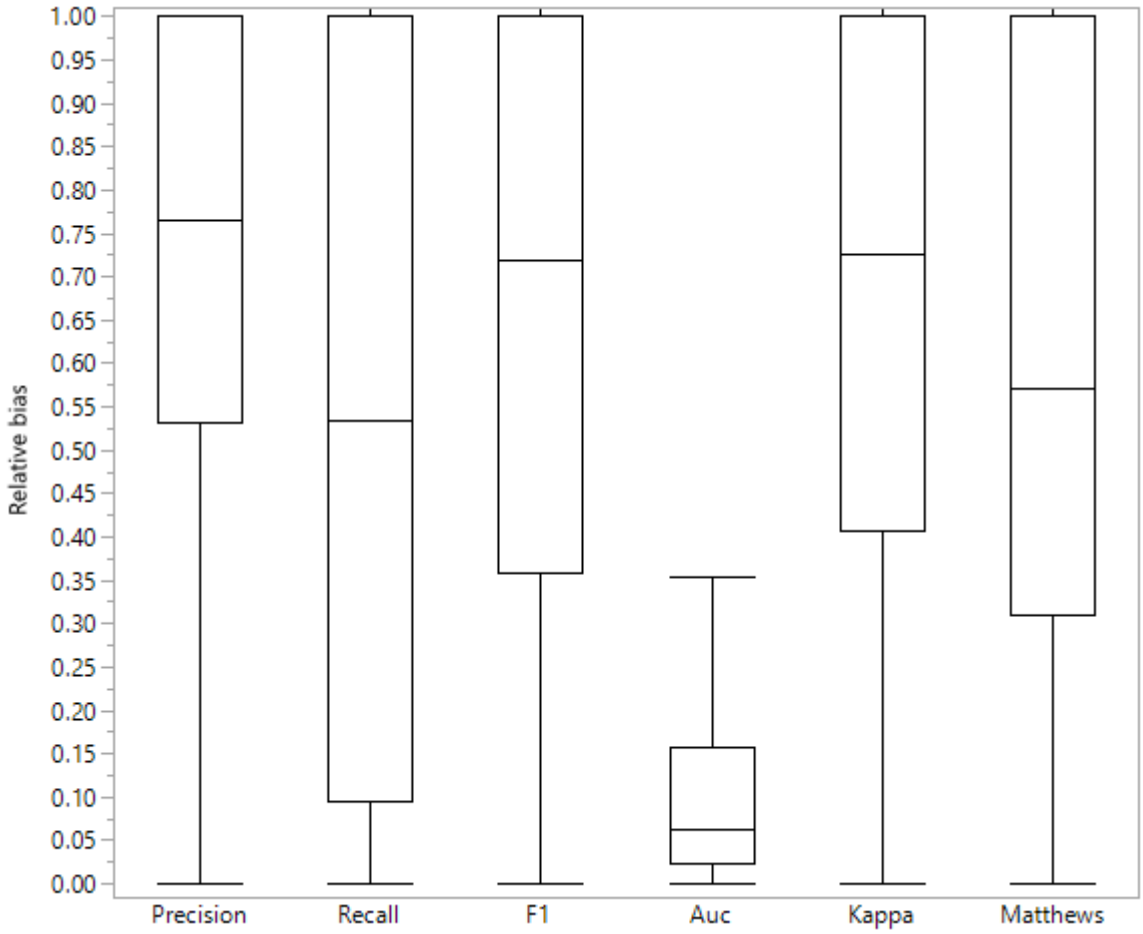

Figure 4.6: Distribution of relative bias, among classifiers and datasets.

cases of removing, or not, the last release data.

Table 4.8: Relative gain (%) in accuracy achieved by removing a specific number of releases of data, in average across classifiers and datasets.

Removed Releases	Precision	Recall	F1	Kappa	Matthews	AUC
1	0.17	0.41	0.33	0.39	0.32	0.03
2	-0.01	0.67	0.37	0.44	0.31	0.03
3	-0.27	0.45	0.07	-0.03	-0.12	0.02
4	-0.39	0.29	-0.09	-0.18	-0.25	0.00

Table 4.9: Statistical test results on the difference in accuracy, in the cases of removing, or not, the last release data.

Metric	chi-squre	p-value
Precision	3.4805	0.0621
Recall	9.0386	0.0026
F1	7.3716	0.0066
Kappa	5.6568	0.0174
Matthews	4.2771	0.0386
AUC	6.2638	0.0123

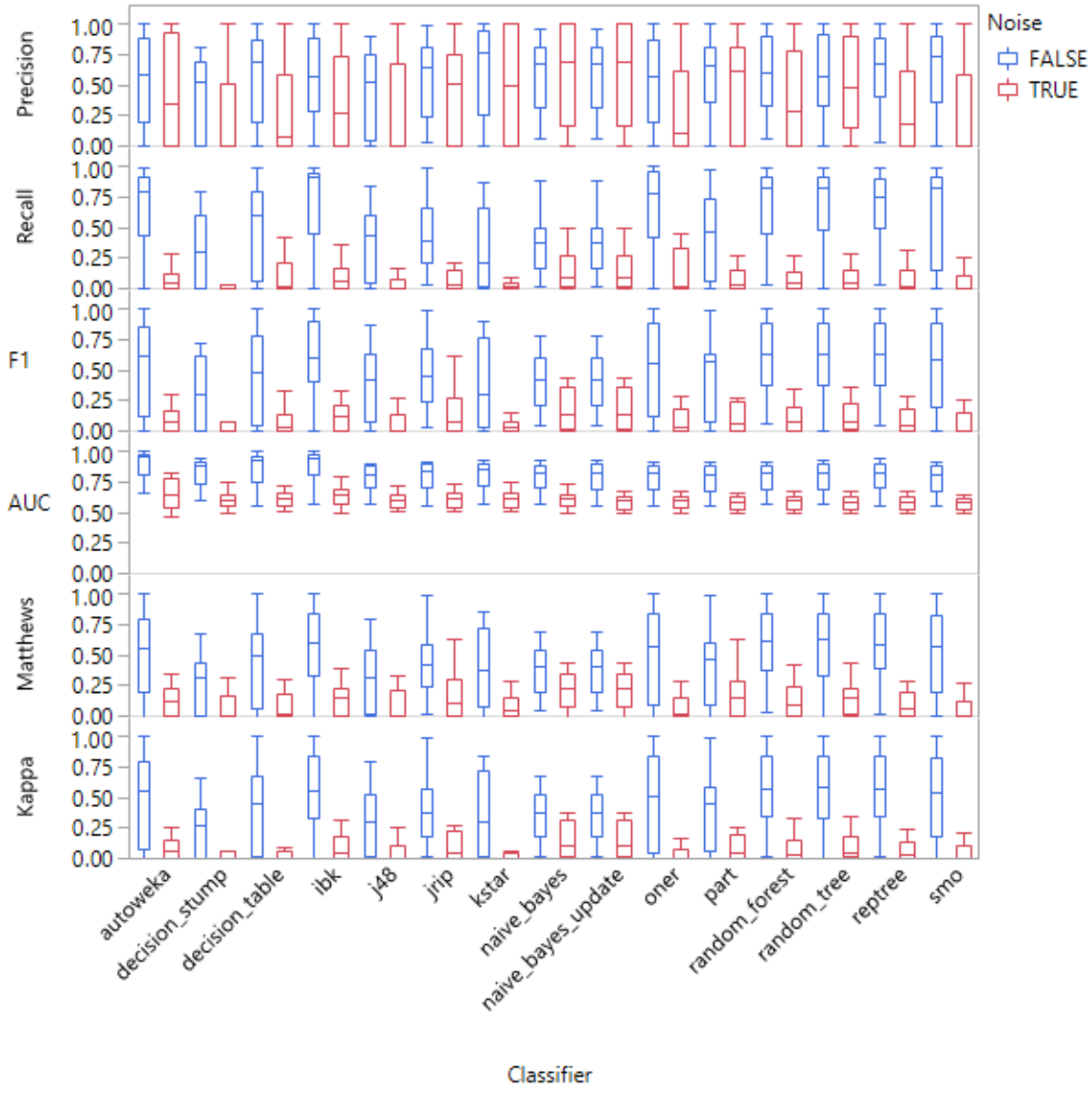

Figure 4.7: Distribution of accuracy, among datasets, of different classifiers (x-axis), in the cases of with, or without, the snoring noise.

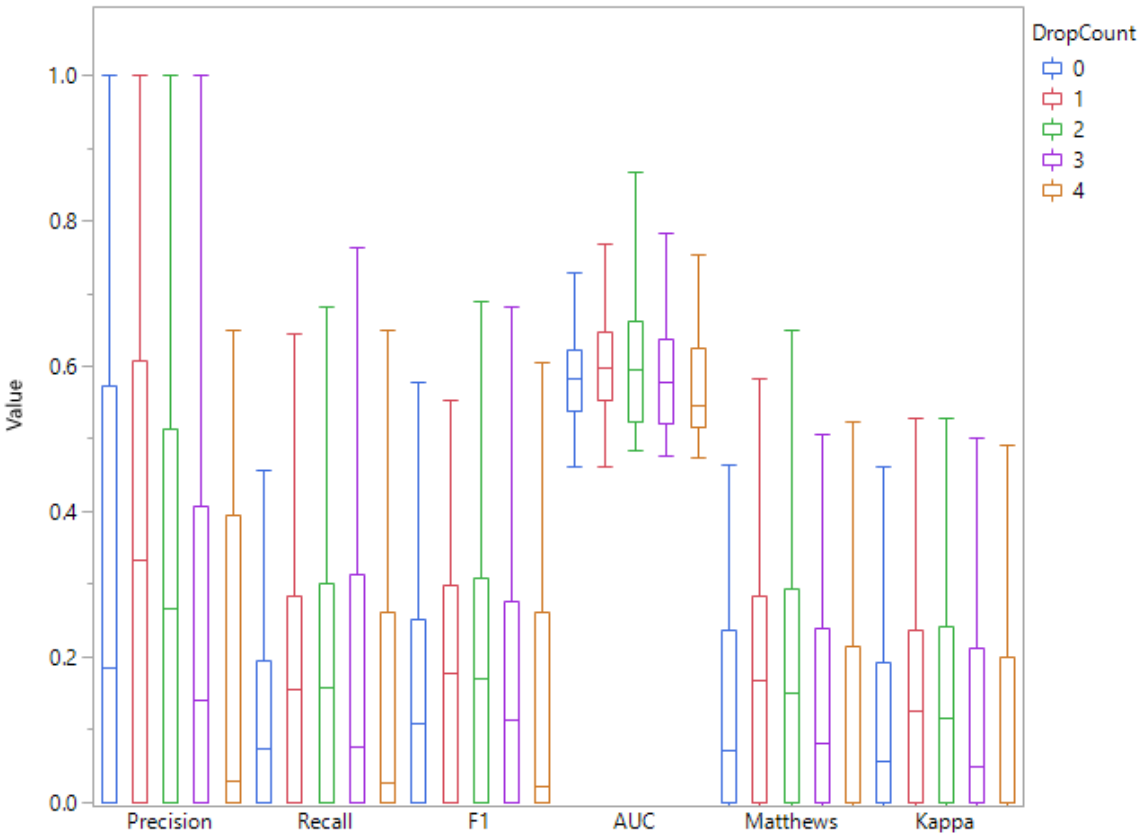

Figure 4.8: Distribution of accuracy, among datasets, in the cases of removal of the last releases of data.

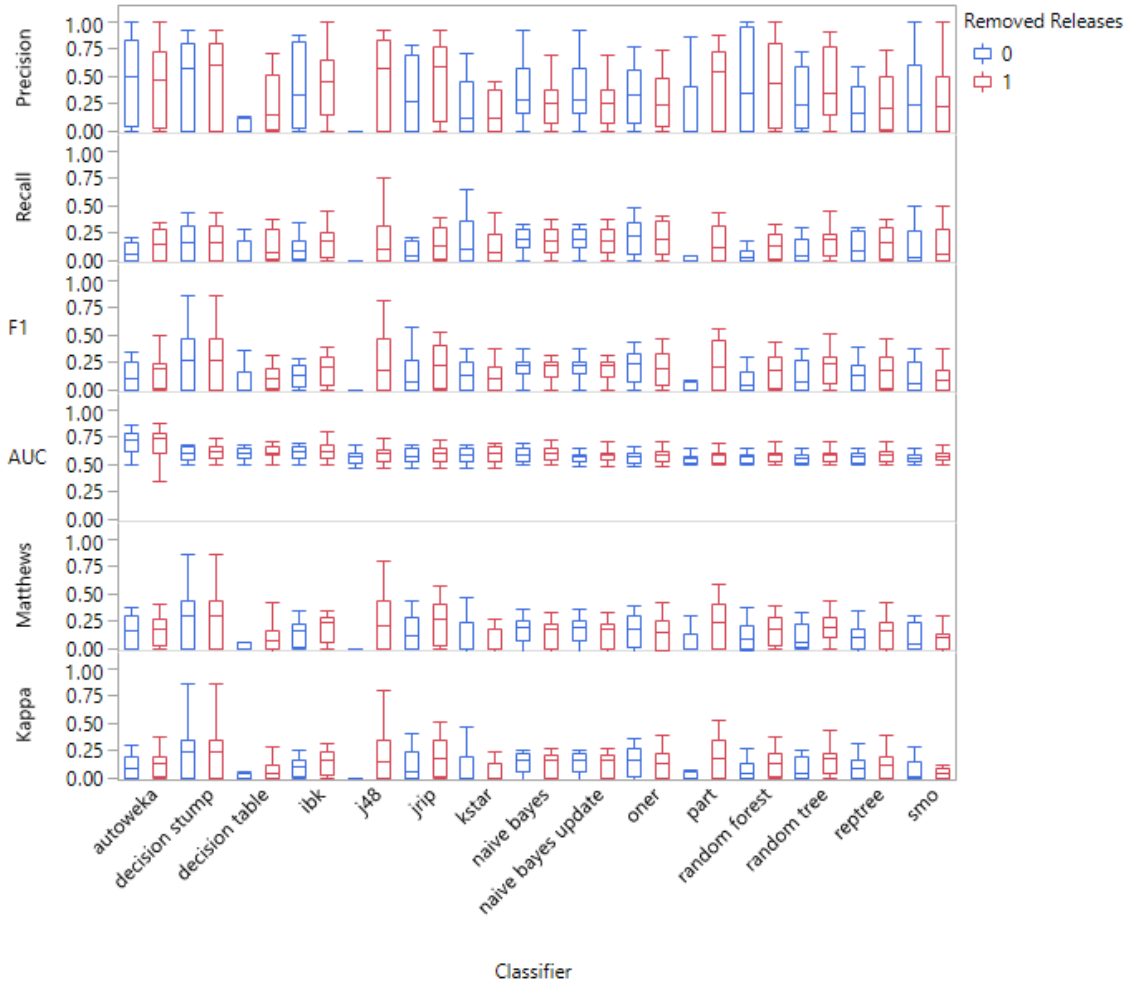

Figure 4.9: Accuracy in each classifier achieved by removing, or not, the last release data.

Chapter 5

DISCUSSION

5.1 RQ1:To what extent do defects sleep?

According to Fig. 4.1, the mean bug sleeps for 7 releases. According to Fig. 4.2 on all projects, most of the defects in a project slept for more than 19% of the existing releases. Moreover, the average bug sleeps a number of releases that varies in the range [4%, 41%], among projects. To better understand the phenomenon, we manually looked at some extreme cases. The defect sleeping the most is LUCENE-3672[1]. This defect slept for 84 releases and was caused by a change in the imported libraries. The defect who slept the most compared to the number of releases of a project is STRATOS-1653[2]. This defect slept 51 (out of 53) releases and was caused by using the wrong method.

5.2 RQ2:To what extent do classes snore?

According to Table 4.1, for the majority of the projects: 1) the missing rate is more than 50% unless we remove more than 20% of the releases; 2) the missing rate is more than 5% unless we remove more than 62% of the releases; 3) the missing rate is not null unless we remove more than 71% of the releases; and 4) the missing rate is more than 25% even if we remove 30% of releases.

5.3 RQ3:To what extent do snoring impact the evaluation of classifiers accuracy?

According to Table 4.4, the evaluation affected by snoring was able to select the best classifier in only in 45% (49 out of 108) cases. Specifically, the best classifier was always correctly identified in the CHUKWA project, and never in the OPENJPA,

[1]https://issues.apache.org/jira/browse/LUCENE-3672
[2]https://issues.apache.org/jira/browse/STRATOS-1653

TAJO, and ZEPPELIN projects. The best classifier was not particularly hard or easy to identify for a specific metric.

According to Table 4.3, the error in accuracy achieved by using a biased evaluation is about 100% in all accuracy metrics other than AUC.

According to Table 4.5, the correlation is strong (>0.75) in only 18% (19 out of 108) of the cases. Specifically, classifiers were easy to rank according to AUC.

According to Fig. 4.7, the presence of snoring makes evaluations underestimating Precision, F1, Matthews and Kappa.

According to Table 4.2, we can reject H30 in five out of six accuracy metrics cases. Specifically, the use of snoring makes evaluations differ from real values in terms of Precision, F1, Matthews and Kappa.

5.4 RQ4:To what extent do snoring impact defect prediction accuracy?

According to Fig. 4.7, the presence of snoring decreases the accuracy in each of the 15 classifiers, in each of the 6 accuracy metrics. We note that the difference is smaller in the case of the Precision accuracy metric. According to Table 4.7, we can reject H10 in all six accuracy metrics cases.

5.5 RQ5:To what extent is no data better than snoring data in supporting accurate defect prediction?

According to Table 4.8:

1. **Removing one release is better than removing no release in all accuracy metrics**.
2. Removing three or four releases is worse than removing one or two releases in all accuracy metrics.
3. Removing two releases is better than removing one release only in terms of Recall. In other words, removing one release is better than removing two releases

in Precision and all the combined accuracy metrics.

4. The gain in removing releases is particularly small in terms of AUC.

5. Removing more than one release reduces Precision.

According to Fig. 4.9, removing one release is better than removing no release of data for all classifiers and all accuracy metrics. According to Table 4.9, we can reject H20 in all six accuracy metrics.

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 [63].

We tested all hypotheses with nonparametric 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 [63].

Results of RQ3 and RQ4 are related to the specific set of predictor variables and classifiers used. It could be that results would differ by using other classifiers or predictor variables. However our sets are large and related to the state of the art.

6.3 Construct

Construct validity regards the ability to generalize the results of an experiment to the theory behind the experiment [63].

Results of RQ2 are related to the specific bugs injected in the first 5% of releases. Similarly, RQ3 and RQ4 are related to a specific 66 / 33 split in training and test sets. It could be that different splits would lead to different results. Similarly, it could be that a bug injected after the 5% release would sleep and hence cause classes to snore much more. In order to address this threat, we performed some sensitivity analyses and results show that the presented results would not vary by slightly changing those split choices.

In this study we do not use any sampling technique. Our preliminary results, report in progress, show that the use of sampling does not increase prediction accuracy.

6.4 External

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

This study used only 20 datasets and hence could be deemed of low generalization compared to studies using tens or hundreds of datasets. However, as stated by Nagappan et al. [44], "*more is not necessarily better.*" We preferred to test our hypotheses on datasets in which we were confident quality is high and that are close to industry. Moreover, our datasets are large if we considered the number of releases: 616 and bugs: 4101.

Finally, in order to promote replicability, all datasets and scripts for this paper are available[1].

[1] https://github.com/AalokAhluwalia/SnoringPublic

Chapter 7

CONCLUSION

In this work, we analyze, more than 4,000 bugs and 600 releases over 20 open source projects from the Apache ecosystem to understand, 1) the magnitude of the sleeping defects, 2) the magnitude of the snoring classes, 3) if snoring impact a classifier's evaluation, 4) if snoring impact a classifier's accuracy, and 5) if removing the last releases of data is beneficial in reducing the negative impact of the snoring noise on classifiers accuracy.

Our results show that, in average across projects:

1. Most of the defects in a project slept for more than 19% of the existing releases.
2. The missing rate is more than 50% unless we remove more than 20% of the releases.
3. The relative error in measuring the classifiers' accuracy achieved by using a dataset with snoring is about 100% in all accuracy metrics other than AUC.
4. The presence of snoring decreases the accuracy in each of the 15 classifiers, in each of the 6 accuracy metrics. For instance, Recall, F1, Kappa and Matthews decrease by about 80%.
5. Removing one release of data is better than removing no data in all accuracy metrics. For instance, Recall, F1, Kappa and Matthews increases by about 30%.

In terms of future works, we plan to extend this work by:

1. Fine-grained snoring removal: In RQ4 we removed snoring at the level of a release, thus treating all classes in that release as snoring. Since in RQ4 we observed that removing more than two releases of data negatively impacts classifier accuracy, then when removing an entire release of data we are removing useful information together with noisy information. We envision techniques able

to identify snoring at the level of the class or datapoints. The identification of snoring datapoints will likely require the use of classifiers and this could create threats to validity. Specifically, there could be a relation between the estimator of the dataset with the estimator that uses the dataset. For instance, an estimator A could result more accurate than another one B because the dataset, upon which the comparison between A and B is performed, has been estimated using a third dataset with C, and A and C share commonalities, e.g., both use the number of elapsed days across releases to estimate the defectiveness of a release.

2. Combine noise removal techniques: There are several types of noise currently known in defects datasets, including snoring and misclassification [25, 59]. However, there is no combined approach able to remove all snoring types from a dataset.

3. Distribution as inputs: We envision a model that accepts, as an input, a distribution of values, rather than a point-value [55]. In other words, since we cannot be sure about the defectiveness of a class in the training set, then intuitively the defectiveness of a class should be modeled by a confidence interval rather than a binary measure.

4. Replication in context of JIT : Just In Time (JIT) prediction models have become sufficiently robust that they are now incorporated into the development cycle of some companies[40]. This investigation can be replicated on JIT models to understand the effect snoring has on the accuracy of these models.

BIBLIOGRAPHY

[1] Amritanshu Agrawal and Tim Menzies. Is "better data" better than "better data miners"?: on the benefits of tuning SMOTE for defect prediction. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1050–1061, 2018.

[2] D. Aha and D. Kibler. Instance-based learning algorithms. *Machine Learning*, 6:37–66, 1991.

[3] Aalok Ahluwalia, Davide Falessi, and Massimiliano Di Penta. Snoring: a noise in defect prediction datasets. In *Proceedings of the 16th International Conference on Mining Software Repositories*, 2019.

[4] Sean Bayley and Davide Falessi. Optimizing prediction intervals by tuning random forest via meta-validation. *CoRR*, abs/1801.07194, 2018. URL `http://arxiv.org/abs/1801.07194`.

[5] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced?: Bias in bug-fix datasets. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 121–130, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-001-2. doi: 10.1145/1595696.1595716. URL `http://doi.acm.org/10.1145/1595696.1595716`.

[6] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[7] John G. Cleary and Leonard E. Trigg. K*: An instance-based learner using an entropic distance measure. In *12th International Conference on Machine Learning*, pages 108–114, 1995.

[8] William W. Cohen. Fast effective rule induction. In *Twelfth International Conference on Machine Learning*, pages 115–123. Morgan Kaufmann, 1995.

[9] Daniel Alencar Da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E Hassan. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, 43(7):641–657, 2016.

[10] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E. Hassan. A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. *IEEE Trans. Software Eng.*, 43(7):641–657, 2017.

[11] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Softw. Engg.*, 17(4-5):531–577, August 2012. ISSN 1382-3256. doi: 10.1007/s10664-011-9173-9. URL `http://dx.doi.org/10.1007/s10664-011-9173-9`.

[12] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Softw. Engg.*, 17(4-5):531–577, August 2012. ISSN 1382-3256. doi: 10.1007/s10664-011-9173-9. URL `http://dx.doi.org/10.1007/s10664-011-9173-9`.

[13] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Softw. Engg.*, 17(4-5):531–577, August 2012. ISSN 1382-3256. doi: 10.1007/s10664-011-9173-9. URL `http://dx.doi.org/10.1007/s10664-011-9173-9`.

[14] Davide Falessi and Max Jason Moede. 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*, pages 15–18, 2018.

[15] Davide Falessi, Barbara Russo, and Kathleen Mullen. What if i had no smells? In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical*

Software Engineering and Measurement, ESEM '17, pages 78–84, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5090-4039-1. doi: 10.1109/ESEM.2017.14. URL https://doi.org/10.1109/ESEM.2017.14.

[16] Davide Falessi, Wyatt Smith, and Alexander Serebrenik. STRESS: A semi-automated, fully replicable approach for project selection. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017*, pages 151–156, 2017. doi: 10.1109/ESEM.2017.22. URL https://doi.org/10.1109/ESEM.2017.22.

[17] Davide Falessi, Wyatt Smith, and Alexander Serebrenik. Stress: A semi-automated, fully replicabile approach for project selection. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '17, pages 151–156, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5090-4039-1.

[18] Davide Falessi, Likhita Narayana, Jennifer Fong Thai, and Burak Turhan. Preserving order of data when validating defect prediction models. *CoRR*, abs/1809.01510, 2018. URL http://arxiv.org/abs/1809.01510.

[19] Eibe Frank and Ian H. Witten. Generating accurate rule sets without global optimization. In J. Shavlik, editor, *Fifteenth International Conference on Machine Learning*, pages 144–151. Morgan Kaufmann, 1998.

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

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

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

[23] Robert J. Grissom and John J. Kim. *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, 2nd edition edition, 2005.

[24] M. A. Hall. *Correlation-based Feature Subset Selection for Machine Learning*. PhD thesis, University of Waikato, Hamilton, New Zealand, 1998.

[25] Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 392–401, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL `http://dl.acm.org/citation.cfm?id=2486788.2486840`.

[26] Tin Kam Ho. Random decision forests. *Proceedings of 3rd International Conference on Document Analysis and Recognition*, 1995. doi: 10.1109/icdar.1995.598994.

[27] R.C. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11:63–91, 1993.

[28] Wayne Iba and Pat Langley. Induction of one-level decision trees. *Machine Learning Proceedings 1992*, page 233240, 1992. doi: 10.1016/b978-1-55860-247-2.50035-8.

[29] George H. John and Pat Langley. Estimating continuous distributions in bayesian classifiers. In *Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 338–345, San Mateo, 1995. Morgan Kaufmann.

[30] Leif Jonsson, Markus Borg, David Broman, Kristian Sandahl, Sigrid Eldh, and Per Runeson. Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts. *Empirical Software Engineering*, 21(4): 1533–1578, 2016. doi: 10.1007/s10664-015-9401-9. URL `https://doi.org/10.1007/s10664-015-9401-9`.

[31] Yungbum Jung, Hakjoo Oh, and Kwangkeun Yi. Identifying static analysis techniques for finding non-fix hunks in fix revisions. In *Proceedings of the ACM First International Workshop on Data-intensive Software Management and Mining*, DSMM '09, pages 13–18, New York, NY, USA, 2009. ACM.

ISBN 978-1-60558-810-0. doi: 10.1145/1651309.1651313. URL http://doi.acm.org/10.1145/1651309.1651313.

[32] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Whitehead Jr. Automatic identification of bug-introducing changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*, pages 81–90, 2006.

[33] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. Automatic identification of bug-introducing changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2579-2. doi: 10.1109/ASE.2006.23. URL http://dx.doi.org/10.1109/ASE.2006.23.

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

[35] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. Dealing with noise in defect prediction. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 481–490, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0.

[36] Pavneet Singh Kochhar, Yuan Tian, and David Lo. Potential biases in bug localization: Do they matter? In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 803–814, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3013-8.

[37] Ron Kohavi. The power of decision tables. In *8th European Conference on Machine Learning*, pages 174–189. Springer, 1995.

[38] Lars Kotthoff, Chris Thornton, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *Journal of Machine Learning Research*, 18(25):1–5, 2017. URL http://jmlr.org/papers/v18/16-261.html.

[39] Yishay Mansour. Pessimistic decision tree pruning based on tree size. In *14th International Conference on Machine Learning*, pages 195–201, 1997.

[40] Shane McIntosh and Yasutaka Kamei. Are fix-inducing changes a moving target? A longitudinal case study of just-in-time defect prediction. *IEEE Trans. Software Eng.*, 44(5):412–428, 2018.

[41] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayse Basar Bener. Defect prediction from static code features: current results, limitations, new approaches. *Autom. Softw. Eng.*, 17(4):375–407, 2010.

[42] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Softw. Engg.*, 22 (6):3219–3253, December 2017. ISSN 1382-3256. doi: 10.1007/s10664-017-9512-6. URL `https://doi.org/10.1007/s10664-017-9512-6`.

[43] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. Diversity in software engineering research. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 466–476, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2237-9.

[44] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. Diversity in software engineering research. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pages 466–476. ACM, 2013.

[45] Edmilson Campos Neto, Daniel Alencar da Costa, and Uirá Kulesza. 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*, pages 380–390, 2018. doi: 10.1109/SANER.2018.8330225. URL `https://doi.org/10.1109/SANER.2018.8330225`.

[46] Roxy Peck and Jay L Devore. *Statistics: The exploration & analysis of data*. Cengage Learning, 2011.

[47] J. Platt. Fast training of support vector machines using sequential minimal optimization. In B. Schoelkopf, C. Burges, and A. Smola, editors, *Advances*

in Kernel Methods - Support Vector Learning. MIT Press, 1998. URL `http://research.microsoft.com/~jplatt/smo.html`.

[48] Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[49] Foyzur Rahman, Daryl Posnett, Israel Herraiz, and Premkumar Devanbu. Sample size vs. bias in defect prediction. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 147–157, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2237-9.

[50] Gema Rodriguez-Perez, Gregorio Robles, and Jesus M. Gonzalez-Barahona. How much time did it take to notify a bug?: Two case studies: Elasticsearch and nova. In *Proceedings of the 8th Workshop on Emerging Trends in Software Metrics*, WETSoM '17, pages 29–35, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-2807-2. doi: 10.1109/WETSoM.2017..6. URL `https://doi.org/10.1109/WETSoM.2017..6`.

[51] Gema Rodríguez-Pérez, Gregorio Robles, and Jesús M. González-Barahona. 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*, 99:164–176, 2018.

[52] Gema Rodríguez-Pérez, Andy Zaidman, Alexander Serebrenik, Gregorio Robles, and Jesús M. González-Barahona. What if a bug has a different origin?: making sense of bugs without an explicit bug introducing change. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2018, Oulu, Finland, October 11-12, 2018*, pages 52:1–52:4, 2018.

[53] Gema Rodríguez-Pérez, Andy Zaidman, Alexander Serebrenik, Gregorio Robles, and Jesús M. González-Barahona. What if a bug has a different origin?: Making sense of bugs without an explicit bug introducing change. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '18, pages 52:1–52:4, New York, NY,

USA, 2018. ACM. ISBN 978-1-4503-5823-1. doi: 10.1145/3239235.3267436. URL http://doi.acm.org/10.1145/3239235.3267436.

[54] Daniel Rozenberg, Ivan Beschastnikh, Fabian Kosmale, Valerie Poser, Heiko Becker, Marc Palyart, and Gail C. Murphy. 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*, pages 109–120, 2016.

[55] Mehrdad Sabetzadeh, Davide Falessi, Lionel Briand, and Stefano Di Alesio. A goal-based approach for qualification of new technologies: Foundations, tool support, and industrial validation. *Reliability Engineering & System Safety*, 119:52–66, 2013.

[56] Martin Shepperd, Qinbao Song, Zhongbin Sun, and Carolyn Mair. Data quality: Some comments on the nasa software defect datasets. *IEEE Trans. Softw. Eng.*, 39(9):1208–1215, September 2013. ISSN 0098-5589. doi: 10.1109/TSE.2013.11. URL http://dx.doi.org/10.1109/TSE.2013.11.

[57] Sidney Siegel. Nonparametric statistics for the behavioral sciences. 1956.

[58] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM. ISBN 1-59593-123-6. doi: 10.1145/1082983.1083147. URL http://doi.acm.org/10.1145/1082983.1083147.

[59] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, Akinori Ihara, and Kenichi Matsumoto. The impact of mislabelling on the performance and interpretation of defect prediction models. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 812–823, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. URL http://dl.acm.org/citation.cfm?id=2818754.2818852.

[60] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. 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*, pages 321–332, 2016.

[61] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. The impact of automated parameter optimization on defect prediction models. *CoRR*, abs/1801.10270, 2018. URL `http://arxiv.org/abs/1801.10270`.

[62] Burak Turhan, Tim Menzies, Ayse Basar Bener, and Justin S. Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14(5):540–578, 2009.

[63] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012. ISBN 3642290434, 9783642290435.

[64] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pages 9–, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2954-2.