2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)

Snoring: a Noise in Defect Prediction Datasets

Aalok Ahluwalia
California Polytechnic State University
San Luis Obispo, California, USA
ahluwali@calpoly.edu

Davide Falessi
California Polytechnic State University
San Luis Obispo, California, USA
dfalessi@calpoly.edu

Massimiliano Di Penta
University of Sannio
Benevento, Italy
dipenta@unisannio.it

***Abstract*—In order to develop and train defect prediction models, researchers rely on datasets in which a defect is often attributed to a release where the defect itself is 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 as "sleeping defects". We call "snoring" the phenomenon where classes are affected by sleeping defects only, that would be treated as defect-free until the defect is discovered. In this paper we analyze, on data from 282 releases of six open source projects from the Apache ecosystem, the magnitude of the sleeping defects and of the snoring classes. Our results indicate that 1) on all projects, most of the defects in a project slept for more than 20% of the existing releases, and 2) in the majority of the projects the missing rate is more than 25% even if we remove the last 50% of releases.**

***Index Terms*—Defect prediction, Fix-inducing changes, Dataset bias**

I. Introduction

Defect prediction aims at identifying software artifacts that are likely to exhibit a defect [16], [31]. The main purpose of defect prediction is to reduce the cost of testing and code review, by letting developers focus on specific artifacts only.

Several researchers have worked on improving the accuracy of defect estimation models using techniques such as tuning [9], [6], [29], re-balancing [2], [1], or feature selection [30]. In order to promote usage and improvements of prediction models, researchers provided means to create [7], [32], collect [5] and select [8], [25], [18] datasets of real defects.

Ultimately, the reliability of a prediction model depends on the quality of the dataset [14], [26]. Therefore effort has been spent in identifying sources of noise in the datasets, and how to deal with it, including defect misclassification [12], [10], [20], [28], [3] and defect origin [23].

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, this introduces an imprecision, which we define as "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 consequence is that 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 be treated, in the dataset, as defect free, while it should not be. 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".

Both Perez et al. [21] and Costa et al.[4] show that the defect fixing time, 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 [27]. 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 [13] 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 [11], [15], different works have identified its limitations [4], [22], [24]. For instance, SZZ cannot find the correct location of bugs that are fixed by adding code [4].

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. This is because, if one builds a dataset using recent releases of a software project, some of its defects might not have been discovered or fixed yet and, as a result, some classes might be treated as defect-free.

With all due limitations of SZZ, in this paper, we perform an early investigation of the magnitude of the "defect sleeping" and, consequently, of the "class snoring" problem in defect datasets. More specifically, by considering 282 releases from six Java open source projects belonging to the Apache ecosystem, we address the following research questions:

- **RQ1**: **To what extent do defects sleep?** We are interested in measuring how many releases elapse within 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.

2574-3864/19/$31.00 ©2019 IEEE
DOI 10.1109/MSR.2019.00019

TABLE I
DETAILS OF THE USED PROJECTS.

Name	Releases	Days	Commits	Defects
abdera	10	1422	167	13
jackrabbit	87	1907	5431	426
lucene	85	4322	11454	361
shindig	22	1743	2003	100
stratos	53	982	8429	104
tuscany	25	1354	4388	126

II. STUDY DESIGN

The *goal* of this study is to investigate the phenomenon of sleeping defects, i.e., of defects being attributed to releases subsequent to those in which they have been introduced, and the extent to which this affects defect prediction datasets because of the presence of snoring classes, i.e., classes that are considered as defect-free while they should not be.

The study *context* consists of data from six open source projects from the Apache ecosystem. We focused on Apache[1] projects rather than random GitHub projects because the formers have a higher quality of defect annotation and to avoid using toy projects [17]. We select projects that are managed in JIRA, versioned in Git, having at least 10 releases, and have most of the commits related to Java code. To avoid the misclassification noise, we added to the list of projects Lucene and Jackrabbit. This because, according to Herzig et al. [10], we can analyze defects that we know are correctly classified. Then, to avoid noise in our measurement, we excluded defects and releases not analyzed in Herzig et al. [10]. Table I reports the details of the used six projects in terms of: releases, days, commits, and defective classes.

In the following, we explain how we address our two research questions formulated in Section I.

A. RQ1: Do defects sleep?

In this research question we are interested in investigating the extent of the sleeping phenomenon, thus we observe the number of releases defects sleep. To better understand the concept of sleeping defects and how many releases defects sleep, Table II reports a scenario of a project where events related to defects (I = injected, N = noting, 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 II, a defect in class C1 is injected and fixed in the same release *r1*. Instead, column 3, row 2 of Table II shows that a defect is injected in *r2* and it is fixed in *r3*. Therefore, class C1 is defective in *r2*. For instance, the defect injected in C1 at *r2* (Table II) sleeps for zero releases since it is fixed in the next release. Instead the defect injected in C2 at *r1* (Table II) sleeps for one release. In order to identify when a defect has been introduced in the project, we re-implemented the SZZ algorithm [27]. We then applied the SZZ algorithm in correspondence of each bug fix. 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.

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

TABLE II
AN EXAMPLE OF PROJECT WHERE EVENTS (I = INJECTED, N = NOTING, 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

B. RQ2: Do classes snore?

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

TABLE III
THE POST-RELEASE DEFECTIVENESS OF A CLASS IN A SPECIFIC RELEASE AS COMPUTED AT *r2*.

Classes	r1
C1	ND
C2	ND
C3	D

TABLE IV
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 III reports the dataset created at *r2*, and Table IV reports a dataset created at *r3*, based on the events described in Table II. 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 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 V
THE ACCURACY OF THE DATASET MAKER PERFORMED IN *r2*.

Classes	r1
C1	TN
C2	FN
C3	P

TABLE VI
THE ACCURACY OF THE DATASET MAKER PERFORMED IN *r3*.

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

Table V and Table VI 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 it is not defective and P = the class is marked as defective. Specifically, C2 at *r1* is a FN for dataset created at *r2* (Table V) and a P for dataset created at *r3* (Table VI). Note that we only consider FN and not FP because the

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

snoring noise only introduces FN, i.e., classes considered as defect-free while they should be defect-prone, and not the other way around.

In this research question we are interested in investigating the extent of the snoring phenomenon, thus we measure the missing rate: FN/(FN+TP). The missing rate is also called false negative rate, Type I error [19] or 1 - Recall, and in our case it regards the proportion of defective classes that are not identified as defective. 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.

C. Threats to validity

A relevant threat to the validity of this work is the lack of ground truth for class defectiveness. Specifically, our measurement likely underestimates the actual snoring phenomenon. There could be future defect fixes that make classes currently identified as defect-free as defect-prone; this would, in turn, increase the number of releases spent by defects and the missing rate we measured. Intuitively, this threat is more likely to exist for recent releases than for older releases. In order to face this threat:

1) We consider only the first 5% of the releases of a project. This 5% is a reasonable trade-off between having a data that is representative (i.e., not too small and an outlier), and being old.
2) While discussing results, we treat the measured missing rate as a lower-bound of the actual missing rate, i.e., the measured noise is likely lower than the actual.

A further threat to validity, that is common in this type of studies, is the possibility that the actual defect change is antecedent to the identified change, i.e. the most recent change is not the bug introducing change. We note that this, as the previous, threat has a conservative impact, i.e., the actual noise is likely higher than the measured noise.

III. Results

This section reports results addressing our two research questions.

A. RQ1: Do defects sleep?

Fig. 1 reports the distribution of the number of releases slept among different projects. These results must be interpreted by considering the number of releases of a project (see Table I), i.e., it is obvious that projects with a few releases have defects with few releases slept. According to Table I and Fig. 1, on all projects, most of the defects in a project slept for more than 20% of the existing releases. To better understand the phenomenon, we manually looked at some extreme cases. The

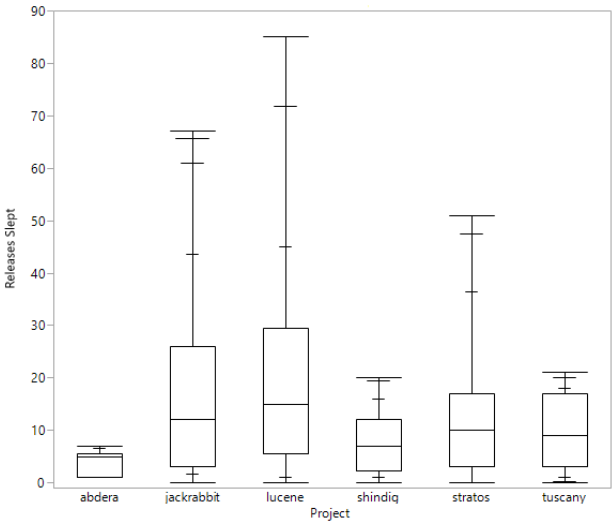

Fig. 1. Distribution of the number of releases slept by post-release defects.

defect sleeping the most is LUCENE-3672[4]. 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[5]. This defect slept 51 (out of 53) releases and was caused by using the wrong method.

B. RQ2: Do classes snore?

Fig. 2 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. Table VII better explains results of Fig. 2 by reporting the minimum progress required to have a specific missing rate in a specific project. For instance, according to row one column two of Table VII, in the project *abdera* the missing rate is equal to 50% at 40% of progress. This means that only by removing the last 40% of the releases we have a dataset with more TP than FN. Thus, according to Table VII, for the majority of the projects: 1) the missing rate is more than 50% unless we remove more than 28% of the releases; 2) the missing rate is more than 5% unless we remove more than 74% of the releases; 3) the missing rate is not null unless we remove more than 86% of the releases; and 4) the missing rate is more than 25% even if we remove 50% of releases.

IV. Conclusions

In this paper, we investigated the phenomenon of "defect snoring", i.e., the extent to which defect datasets — used among others for defect prediction — ignore some defects because they have not been fixed yet. Results of our study, conducted on data from 282 releases of five Java open source

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

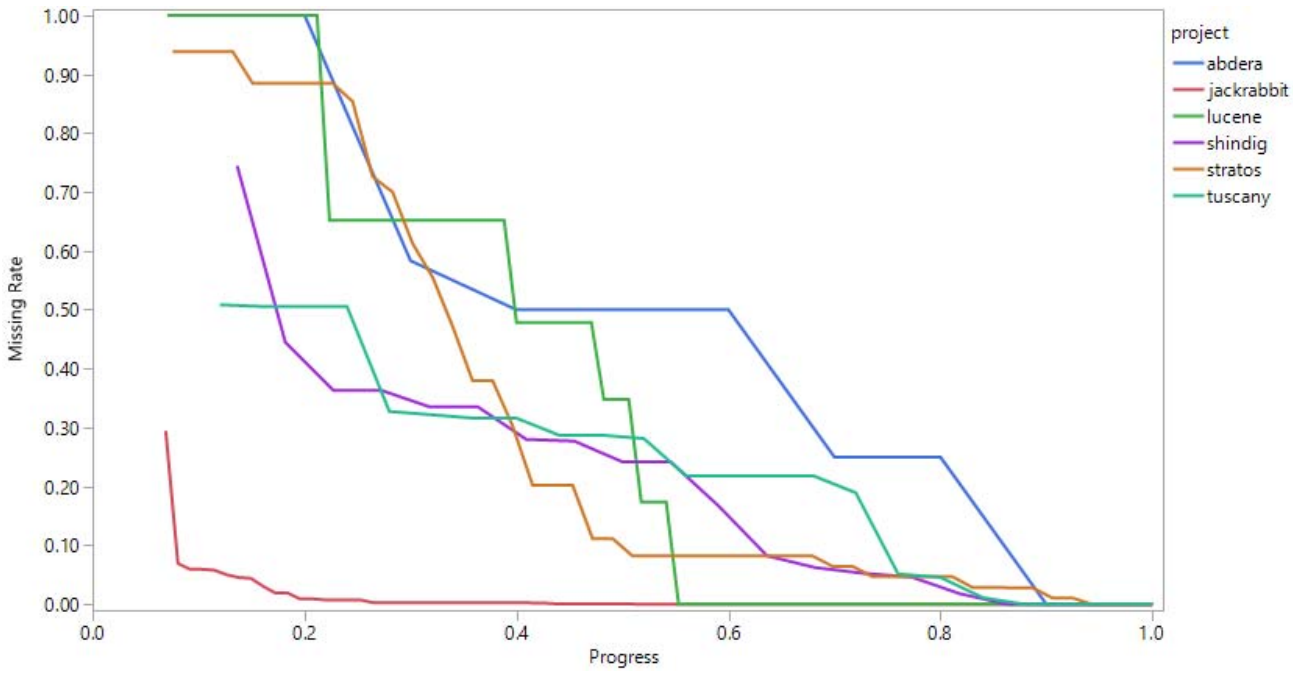

Fig. 2. Defect-prone classes missing rate for the first 5% releases, observed at different levels of progress in a project.

TABLE VII
MINIMUM REQUIRED PROGRESS TO HAVE A SPECIFIC MISSING RATE IN A SPECIFIC PROJECT.

Missing rate	abdera	jackrabbit	lucene	tuscany	stratos	shindig
0.50	0.40	0.07	0.40	0.28	0.34	0.18
0.25	0.70	0.08	0.52	0.56	0.42	0.50
0.15	0.90	0.08	0.55	0.76	0.47	0.64
0.10	0.90	0.08	0.55	0.76	0.51	0.64
0.05	0.90	0.14	0.55	0.80	0.74	0.77
0.01	0.90	0.20	0.55	0.88	0.94	0.86
0.00	0.90	0.52	0.55	0.88	0.94	0.86

project, show that 1) on all projects, most of the defects in a project slept for more than 20% of the existing releases, and 2) in the majority of the projects the missing rate is more than 25% even if we remove the last 50% of releases. In other words, datasets used for training actual prediction models or for evaluating their adoption are noisy, as we do not know the impact of future defects fixes to the defectiveness of class in past releases.

In the absence of a ground truth on class defectiveness, we should be particularly careful when measuring the snoring noise as it likely impacts the measurement itself. This can be particularly important for inactive projects, where possible new defects would not be fixed (or not even reported). In agreement with previous studies [18] our suggestion is to use projects that are active, and with a high number of releases, discarding the most recent ones for which there is not enough information yet to categorize classes into defect-prone and defect-free.

However, if on the one side removing recent releases reduces noise, on the other side this would also reduce the size of the dataset, and make it possibly outdated. Therefore, we envision a plan to investigate the trade-off between size and noise [20]. Specifically, work-in-progress aims at: (i) measuring the impact of snoring, and the interaction factor with size, on the accuracy of prediction models; (ii) exploring and validating techniques to identify and remove noise due to snoring classes; and (iii) exploring and validating classifiers using as input a defectiveness confidence level rather than a binary metric (defective or not).

REFERENCES

[1] A. Agrawal and T. 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] S. Bayley and D. Falessi. Optimizing prediction intervals by tuning random forest via meta-validation. *CoRR*, abs/1801.07194, 2018.

[3] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. 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.

[4] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. 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.

[5] D. Falessi and M. J. 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.

[6] W. Fu, T. Menzies, and X. Shen. Tuning for software analytics: Is it really necessary? *Information & Software Technology*, 76:135–146, 2016.

[7] K. Gao, T. M. Khoshgoftaar, H. Wang, and N. Seliya. Choosing software metrics for defect prediction: an investigation on feature selection techniques. *Softw., Pract. Exper.*, 41(5):579–606, 2011.

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

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

[10] K. Herzig, S. Just, and A. 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.

[11] S. Kim, E. J. W. Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Software Eng.*, 34(2):181–196, 2008.

[12] S. Kim, H. Zhang, R. Wu, and L. 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.

[13] S. Kim, T. Zimmermann, K. Pan, and E. J. W. 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.

[14] P. S. Kochhar, Y. Tian, and D. 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.

[15] S. McIntosh and Y. 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.

[16] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Basar Bener. Defect prediction from static code features: current results, limitations, new approaches. *Autom. Softw. Eng.*, 17(4):375–407, 2010.

[17] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan. Curating github for engineered software projects. *Empirical Softw. Engg.*, 22(6):3219–3253, Dec. 2017.

[18] M. Nagappan, T. Zimmermann, and C. 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.

[19] R. Peck and J. L. Devore. *Statistics: The exploration & analysis of data*. Cengage Learning, 2011.

[20] F. Rahman, D. Posnett, I. Herraiz, and P. 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.

[21] G. Rodriguez-Perez, G. Robles, and J. 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.

[22] G. Rodríguez-Pérez, G. Robles, and J. 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.

[23] G. Rodríguez-Pérez, A. Zaidman, A. Serebrenik, G. Robles, and J. 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.

[24] G. Rodríguez-Pérez, A. Zaidman, A. Serebrenik, G. Robles, and J. 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.

[25] D. Rozenberg, I. Beschastnikh, F. Kosmale, V. Poser, H. Becker, M. Palyart, and G. 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.

[26] M. Shepperd, Q. Song, Z. Sun, and C. Mair. Data quality: Some comments on the nasa software defect datasets. *IEEE Trans. Softw. Eng.*, 39(9):1208–1215, Sept. 2013.

[27] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.

[28] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. 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.

[29] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. 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.

[30] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. The impact of automated parameter optimization on defect prediction models. *CoRR*, abs/1801.10270, 2018.

[31] B. Turhan, T. Menzies, A. Basar Bener, and J. 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.

[32] T. Zimmermann, R. Premraj, and A. 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.