Comparing fine-grained source code changes and code churn for bug prediction - A retrospective

Martin Pinzger
Department of Informatics Systems
Universität Klagenfurt
Klagenfurt, Austria
martin.pinzger@aau.at

Emanuel Giger
Zurich, Switzerland
gigerem@gmail.com

Harald C. Gall
Department of Informatics
University of Zurich
Zurich, Switzerland
gall@ifi.uzh.ch

DOI: 10.1145/3468744.3468751
https://doi.org/10.1145/3468744.3468751

ABSTRACT

More than two decades ago, researchers started to mine the data stored in software repositories to help software developers in making informed decisions for developing and testing software systems. Bug prediction was one of the most promising and popular research directions that uses the data stored in software repositories to predict the bug-proneness or number of bugs in source files. On that topic and as part of Emanuel's PhD studies, we submitted a paper with the title *Comparing fine-grained source code changes and code churn for bug prediction* [8] to the 8th Working Conference on Mining Software Engineering, held 2011 in beautiful Honolulu, Hawaii. Ten years later, it got selected as one of the finalists to receive the MSR 2021 Most Influential Paper Award. In the following, we provide a retrospective on our work, describing the road to publishing this paper, its impact in the field of bug prediction, and the road ahead.

ACM Reference Format:
Martin Pinzger, Emanuel Giger, and Harald C. Gall. 2021. Comparing fine-grained source code changes and code churn for bug prediction - A retrospective. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/1122445.1122456

1 SETTING THE SCENE

State-of-the-art approaches in bug prediction 10–15 years ago typically used traditional machine learning algorithms, such as linear regression, binary logistic regression, decision trees, or support vector machine, fed with code metrics and/or process metrics to train models to predict the bug-proneness of source files. Several such approaches have been proposed by researchers that combined different metrics and machine learning algorithms. Furthermore, studies such as by Moser *et al.* [15], found out that process metrics perform explicitly well. Regarding process metrics, Nagappan *et al.* [16] published a seminal work in 2005, in which they show that the relative code churn (sum of lines added, deleted and modified weighted by the total lines of code) is a significant indicator for buggy source files. This approach represented the state-of-the-art in bug prediction at that time.

A key ingredient for training prediction models is a large amount of cleansed data for training, testing, and validating the models. In our research group at the University of Zurich we had access to such data that we mined with the Evolizer platform [6]. Most important for our work has been the ChangeDistiller [5], a key component of Evolizer, that is capable of mining detailed information on code changes from versioning archives. In contrast to traditional line-based differencing, such as used for computing code churn, ChangeDistiller applies tree-based differencing to two versions of a source file to capture syntactical and some semantical information on code changes. For that it classifies the extracted changes according to its change type taxonomy that consists of more than 45 different changes types. For instance, it can capture changes that inserted, modified, or deleted an if-statement. This way it also filters formatting changes that can bias the code churn.

Motivated by previous work and by the availability of the more detailed information on code changes (in the following denoted as *SCC*), we hypothesized to improve existing prediction models, in particular the ones that use code churn (in the following denoted as *LM*) as a predictor. Using the data from 15 Eclipse plugins, we investigated the following three hypotheses:

H1: *SCC* does have a stronger correlation with the number of bugs than *LM*.
H2: *SCC* achieves better performance to classify source files into bug- and not bug-prone files than *LM*.
H3: *SCC* achieves better performance when predicting the number of bugs in source files than *LM*.

2 RUNNING THE EXPERIMENTS

As a first step, we used Evolizer and ChangeDistiller to prepare the dataset for our experiments. Figure 1 depicts an overview of these steps.

First, we used Evolizer to fetch and process all the log entries for each source file and to compute the *LM* for each file revision. Second, we used Evolizer to search the commit messages of the log entries for references to bug reports and compute the number of bugs per file revision. Third, we used ChangeDistiller to extract the code changes between pairs of subsequent revisions and counted the number of changes for each change type per file revision. Both, Evolizer and ChangeDistiller, store the results into the Evolizer database, that made it easy for us to compute the features for training the prediction models. Note, both tools are still publicly available, though maintenance for the Evolizer platform has stopped.

MSR '21, May 23–24, 2021, Madrid, Spain
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

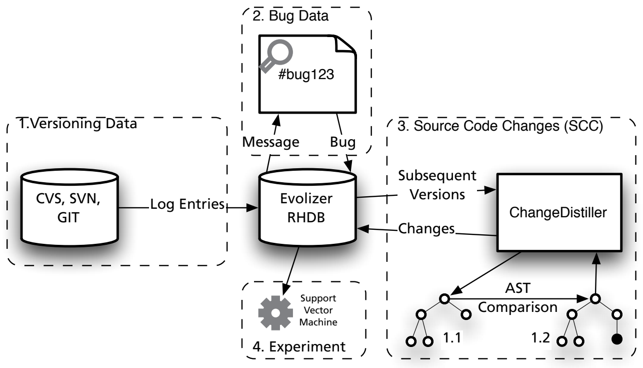

Figure 1: Overview of the data extraction process.

Following good practices for conducting empirical studies, we first had a look at the descriptive statistics and distribution of our data. We soon found out that several change types occurred very rarely, therefore, we decided to aggregate the change types to 9 change type categories with the hope to obtain more balanced numbers. Next, we investigated whether the change type categories exhibit a stronger correlation with the number of bugs than *LM*. Computing the Spearman correlation with the numbers summed up for each file over the full observation period, we found that this is true for the aggregated number of changes (*SCC*) leading us to accept hypothesis H1. Note, this result was a strong indicator that the number of changes might contain more valuable information to predict the bug-proneness of source files and even the number of bugs than *LM*.

Next, we investigated hypothesis H2 by computing prediction models for each project, once with *LM* and once with *SCC* using binary logistic regression and 10-fold cross validation. The results in terms of precision, recall, and AUC all show that the models, except for the Eclipse Help plugin, trained with *SCC* outperformed the models trained with *LM* with a median AUC_{SCC} of 0.90 vs. a median AUC_{LM} of 0.85. We also found that neither considering the single change type categories nor considering multiple other classifiers did help to improve the AUC values. Finally, we investigated hypothesis H3 by fitting an asymptotic model to estimate the number of bugs based on the number of changes, again once using *LM* and once using *SCC*. For each project, the model trained with *SCC* showed a higher R^2 value confirming our assumption that *SCC* has more explanatory power compared to *LM*. For more details, we refer the reader to our original paper [8].

Summing up the results, the key take away message from this paper is: *SCC contains more detailed information on code changes therefore has more explanatory power than code churn for predicting the bug-proneness of source files.* Consequently, future studies should use *SCC* instead of code churn. Whether they did that and why not is discussed next.

3 WHAT IS THE IMPACT?

Addressing the question from the previous section, we had a look at the citations for our paper on google scholar. We found that our work received a constant number of citations each year with peaks in the years 2012–2015. Looking at a sample of these works, we found that our work mainly inspired and impacted research on: 1) using detailed information on code changes to improve program repair *e.g.*, [14], patch propagation [12] and merge support, [2]; 2) further improving bug/defect prediction, *e.g.*, [1, 3, 7, 13, 17]; 3) being careful when choosing a machine learning technique and checking the assumptions/prerequisites for using that technique, *e.g.*, [9, 10]; and 4) diverse other studies, such as on the repetitiveness of code changes [18], antipatterns in source code [19], plagiarism detection [11], and taint analysis [20].

Regarding our question whether these studies favored fine-grained changes over code churn, we found no clear answer. While several studies follow or support the key take away message of our paper, such as [2, 12, 14], several other studies, such as [13, 17], also cited our work to motivate the usage of code churn. The reason seems to be that code churn is a measure that can be easily obtained while the extraction of fine-grained code changes needs more sophisticated techniques, such as a parser, that needs to be provided for each programming language.

4 MOVING FORWARD

In our opinion, the most promising road ahead is the research on techniques and tools that use detailed information on code changes to automate some of the recurring development activities, such as automatically fixing (simple) bugs or regression test selection, or to support software developers, for instance, in understanding changes and their impact. Key for walking this research road is the availability of detailed information on code changes. Since the introduction of ChangeDistiller [5], several other approaches for extracting detailed information on code changes have been introduced. Most notable is the GumTree framework [4] and various improvements of it that support other programming languages. However, there is still lots of room for improving these tools, for instance to support the variety of technologies that are used to develop modern software systems, such as cloud native applications. These are some of the topics, that we are currently working on and that will keep us busy for the next couple of years.

REFERENCES

[1] HD Arora and Talat Parveen. 2019. Computation of Various Entropy Measures for Anticipating Bugs in Open-Source Software. In *Software Engineering*. Springer, 235–247.

[2] Caius Brindescu, Iftekhar Ahmed, Rafael Leano, and Anita Sarma. 2020. Planning for untangling: Predicting the difficulty of merge conflicts. In *Proceedgins of the International Conference on Software Engineering*. IEEE, 801–811.

[3] Nélio Cacho, Thiago César, Thomas Filipe, Eliezio Soares, Arthur Cassio, Rafael Souza, Israel Garcia, Eiji Adachi Barbosa, and Alessandro Garcia. 2014. Trading robustness for maintainability: an empirical study of evolving c# programs. In *Proceedings of the International Conference on Software Engineering*. IEEE, 584–595.

[4] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-Grained and Accurate Source Code Differencing. In *Proceedings of the International Conference on Automated Software Engineering*. ACM, 313–324.

[5] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald C. Gall. 2007. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering* 33, 11 (2007), 725–743.

[6] Harald C. Gall, Beat Fluri, and Martin Pinzger. 2009. Change Analysis with Evolizer and ChangeDistiller. *IEEE Software* 26, 1 (2009), 26–33.

[7] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C Gall. 2012. Method-level bug prediction. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*. IEEE, 171–180.

[8] Emanuel Giger, Martin Pinzger, and Harald C. Gall. 2011. Comparing Fine-Grained Source Code Changes and Code Churn for Bug Prediction. In *Proceedings of the Working Conference on Mining Software Repositories*. ACM, 83–92.

[9] Tihana Galinac Grbac, Goran Mausa, and Bojana Dalbelo Basic. 2013. Stability of Software Defect Prediction in Relation to Levels of Data Imbalance.. In *Proceedings of the Workshop on Software Quality Analysis, Monitoring, Improvement and Applications*. CEUR-WS.org, 1–10.

[10] Hadi Hemmati, Sarah Nadi, Olga Baysal, Oleksii Kononenko, Wei Wang, Reid Holmes, and Michael W Godfrey. 2013. The msr cookbook: Mining a decade of research. In *Proceedings for the Working Conference on Mining Software Repositories*. IEEE, 343–352.

[11] Vedran Ljubovic and Enil Pajic. 2020. Plagiarism Detection in Computer Programming Using Feature Extraction from Ultra-Fine-Grained Repositories. *IEEE Access* (2020).

[12] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. 2020. Spider: Enabling fast patch propagation in related software repositories. In *Proceedings of the Symposium on Security and Privacy*. IEEE, 1562–1579.

[13] Lech Madeyski and Marian Jureczko. 2015. Which process metrics can significantly improve defect prediction models? An empirical study. *Software Quality Journal* 23, 3 (2015), 393–422.

[14] Matias Martinez and Martin Monperrus. 2015. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering* 20, 1 (2015), 176–205.

[15] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. 2008. A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction. In *Proceedings of the International Conference on Software Engineering*. ACM, 181–190.

[16] Nachiappan Nagappan and Thomas Ball. 2005. Use of Relative Code Churn Measures to Predict System Defect Density. In *Proceedings of the International Conference on Software Engineering*. ACM, 284–292.

[17] Malanga Kennedy Ndenga, Ivaylo Ganchev, Jean Mehat, Franklin Wabwoba, and Herman Akdag. 2019. Performance and cost-effectiveness of change burst metrics in predicting software faults. *Knowledge and Information Systems* 60, 1 (2019), 275–302.

[18] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, and Hridesh Rajan. 2013. A study of repetitiveness of code changes in software evolution. In *Proceedings of the International Conference on Automated Software Engineering*. IEEE, 180–190.

[19] Daniele Romano, Paulius Raila, Martin Pinzger, and Foutse Khomh. 2012. Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes. In *Proceedings of the Working Conference on Reverse Engineering*. IEEE, 437–446.

[20] Tingyu Song, Xiaohong Li, Zhiyong Feng, and Guangquan Xu. 2019. Inferring Patterns for Taint-Style Vulnerabilities With Security Patches. *IEEE Access* 7 (2019), 52339–52349.