



# Software Analysis, Evolution, and Reengineering, and ICT Sustainability

Jeffrey Carver, Birgit Penzenstadler, and Alexander Serebrenik

**THIS ISSUE'S ARTICLE** reports on papers from the IEEE 25th International Conference on Software Analysis, Evolution, and Reengineering (SANER 18) and 5th International Conference on Information and Communications Technology for Sustainability (ICT4S 18). Feedback or suggestions are welcome. In addition, if you try or adopt any of the practices described in the article, please send Jeffrey Carver and the paper authors a note about your experiences.

## Examining Previous Results' Reliability (SANER)

This first group of papers comes from the Reproducibility Studies and Negative Results track at SANER. They make the point that researchers and practitioners must be careful about accepting the results from one study, no matter how well executed, because those results might not apply in all contexts. Replication studies are crucial to increase our confidence in reported results and their applicability in different situations. Negative results are also important because they indicate that the researchers have

tried their best and can't make an approach work, so others shouldn't waste time trying again.

"Re-evaluating Method-Level Bug Prediction," by Luca Pascarella and his colleagues, describes a replication of prior research on method-level bug prediction.<sup>1</sup> The authors applied the metrics used in the original study, including standard product metrics and process metrics, to a different set of 13 large open-source projects, including Ant, Eclipse JDT (Java development tools), and Lucene.

Pascarella and his colleagues' results showed that the previously published techniques were useful for predicting bugs in the current software release but not for predicting bugs in forthcoming releases. Random guessing was as good as the previous techniques. The overall takeaways are that each task needs an appropriate validation approach and that standard tenfold cross-validation might not provide confidence in future predictions' validity. Access this paper at [http://bit.ly/PD\\_2018\\_July\\_1](http://bit.ly/PD_2018_July_1).

"Detecting Code Smells Using Machine Learning Techniques: Are

We There Yet?," by Dario Di Nucci and his colleagues, presents a similar finding about a different software engineering phenomenon.<sup>2</sup> The authors examined earlier findings that machine-learning techniques built with cross-project data could predict code smells with up to 95 percent accuracy. In the previous research, the techniques tried to predict code smells in isolation; that is, they focused on the methods and classes affected by the smell, as opposed to those unaffected by the smell. But, in practice, those smelly methods and classes are often a small fraction of the overall number of methods and classes. Those methods and classes can often be affected by multiple smells simultaneously.

When Di Nucci and his colleagues investigated these two simplifying assumptions' effects on the overall results, they found that their results were up to 90 percent lower than those of the previous research. They also found that the techniques couldn't properly classify whether the code elements contained code smells. These results again highlight the importance of replicating

studies in realistic settings with valid assumptions. Access this paper at [http://bit.ly/PD\\_2018\\_July\\_2](http://bit.ly/PD_2018_July_2).

“Keep It Simple: Is Deep Learning Good for Linguistic Smell Detection?,” by Sarah Fakhoury and her colleagues, reports on whether deep-learning techniques, which have been found useful for image recognition, can also be useful for software engineering tasks such as identifying linguistic antipatterns.<sup>3</sup> This study compared convolutional neural networks (CNNs), a deep-learning technique, with the traditional machine-learning techniques implemented in the Linguistic Antipattern Detection (LAPD) tool.

The findings showed that LAPD, when properly tuned with methods such as Bayesian optimization, dramatically outperformed CNN. Furthermore, finding the optimal LAPD configuration took only minutes, compared to several days for CNN. These results suggest that you must take care when applying approaches from one domain to another. Research results should be validated in the context in which they'll be applied. Access this paper at [http://bit.ly/PD\\_2018\\_July\\_3](http://bit.ly/PD_2018_July_3).

### Automated Program Repair (SANER)

The next three papers focus on automated program repair techniques, which aim to relieve developers from fixing numerous, often trivial bugs. Even if these techniques can fix only a small fraction of all defects, the impact on developers' effort can greatly benefit companies.

“Mining StackOverflow for Program Repair,” by Xuliang Liu and Hao Zhong, discusses the SOFix tool, which can produce patches for previously unresolved situations.<sup>4</sup> While studying Stack Overflow, Liu and

Zhong identified several new automated program repair techniques—modifications that automated program repair tools should try when looking to fix a defect. For example, one technique replaces a variable with the invocation of a method with no parameters, mimicking when a developer confuses a variable's name with a method's name. Another template reverses the priority of two linked binary operators, making SOFix the only automated tool that can fix the Math 80 bug in the Defects4J benchmark (a standard dataset for automatic program repair taken from a number of open source systems; <https://github.com/rjust/defects4j>), by replacing `int j = 4 * n - 1` with `int j = 4 * (n - 1)`. Access this paper at [http://bit.ly/PD\\_2018\\_July\\_4](http://bit.ly/PD_2018_July_4).

In “Using a Probabilistic Model to Predict Bug Fixes,” Mauricio Soto and Claire Le Goues analyze and classify an extensive body of bug-fixing commits (the 100 most recent from 500 of the most popular Java projects on GitHub).<sup>5</sup> They also present a probabilistic program repair approach based on the most frequently used repair templates. In an evaluation on a subset of the Defects4J benchmark, the probabilistic approach outperformed heuristic techniques in terms of time and quality. Access this paper at [http://bit.ly/PD\\_2018\\_July\\_5](http://bit.ly/PD_2018_July_5).

“Automatically Repairing Dependency-Related Build Breakage,” by Christian Macho and his colleagues, describes BuildMedic, a tool to repair Maven builds that break owing to dependency-related issues.<sup>6</sup> Evaluation of BuildMedic on 84 cases of dependency-related build breakage in 23 open source Java projects showed that it could automatically repair 45 of those broken builds. In addition, 36 percent of the time,

BuildMedic performed the same fix the human developer performed. Access this paper at [http://bit.ly/PD\\_2018\\_July\\_6](http://bit.ly/PD_2018_July_6).

### Sustainability (ICT4S)

In “An Empirical Evaluation of Database Software Features on Energy Consumption,” Sedef Akınlı Koçak and her colleagues argue that although software doesn't consume energy by itself, its characteristics determine the energy required by hardware resources.<sup>7</sup> They analyze the energy effects of different features of IBM DB2, a commonly used database product. Kocak and her colleagues executed a workload in preconfigured software with some features enabled or disabled and with different numbers of users.

Using three sets of green metrics, Kocak and her colleagues identified which parts of the software system consumed energy. For the CPU, the main energy consumer was the compression feature. However, when that feature interacted with other features, it lowered the overall energy consumption in several scenarios. I/O-intensive activities were the other source of high energy consumption.

The authors' findings suggest that you can mitigate conflicts among software system performance, functionality, and energy consumption (which are competing goals) by combining features that interact in an energy-efficient way. An analysis of feature interactions is worth the effort for optimizing overall energy consumption. Access this paper at [http://bit.ly/PD\\_2018\\_July\\_7](http://bit.ly/PD_2018_July_7).

In “Empirical Evaluation of the Energy Impact of Refactoring Code Smells,” Roberto Verdecchia and his colleagues explain how they applied JDeodorant, a code smell refactoring tool, to three

## ABOUT THE AUTHORS



**JEFFREY CARVER** is a professor in the University of Alabama's Department of Computer Science. Contact him at [carver@cs.ua.edu](mailto:carver@cs.ua.edu).




**BIRGIT PENZENSTADLER** is an assistant professor of software engineering at California State University, Long Beach. Contact her at [birgit.penzenstadler@csulb.edu](mailto:birgit.penzenstadler@csulb.edu).



**ALEXANDER SEREBRENIK** is an associate professor in Eindhoven University of Technology's Department of Mathematics and Computer Science. Contact him at [a.serebrenik@tue.nl](mailto:a.serebrenik@tue.nl).

5. M. Soto and C. Le Goues, "Using a Probabilistic Model to Predict Bug Fixes," *Proc. IEEE 25th Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER 18)*, 2018, pp. 221–231.
6. C. Macho, S. McIntosh, and M. Pinzger, "Automatically Repairing Dependency-Related Build Breakage," *Proc. IEEE 25th Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER 18)*, 2018, pp. 106–117.
7. S. Akinli Koçak et al., "An Empirical Evaluation of Database Software Features on Energy Consumption," *Proc. 5th Int'l Conf. Information and Communications Technology for Sustainability (ICT4S 18)*, 2018, pp. 1–19.
8. R. Verdecchia et al., "Empirical Evaluation of the Energy Impact of Refactoring Code Smells," *Proc. 5th Int'l Conf. Information and Communications Technology for Sustainability (ICT4S 18)*, 2018, pp. 365–383.

open source Java applications based on object-relational mapping (Cash-Manager, JTrack, and Spring Pet-Clinic).<sup>8</sup> JDeodorant currently can refactor only one code smell at a time, so the largest application was 14 KLOC. The key finding is that refactoring code smells can significantly improve software energy efficiency. Specifically, refactoring the Feature Envy and Long Methods smells improved energy efficiency by 49 percent. Furthermore, the authors found no correlation between established software metrics and energy consumption. Access this paper at [http://bit.ly/PD\\_2018\\_July\\_8](http://bit.ly/PD_2018_July_8). 

## References

1. L. Pascarella, F. Polomba, and A. Bacchelli, "Re-evaluating Method-Level Bug Prediction," *Proc. IEEE 25th Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER 18)*, 2018, pp. 592–601.
2. D. Di Nucci et al., "Detecting Code Smells Using Machine Learning Techniques: Are We There Yet?," *Proc. IEEE 25th Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER 18)*, 2018, pp. 612–621.
3. S. Fakhoury et al., "Keep It Simple: Is Deep Learning Good for Linguistic Smell Detection?," *Proc. IEEE 25th Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER 18)*, 2018, pp. 602–611.
4. X. Liu and H. Zhong, "Mining StackOverflow for Program Repair," *Proc. IEEE 25th Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER 18)*, 2018, pp. 118–129.

Read your subscriptions through the myCS publications portal at

<http://mycs.computer.org>