Methods and Tools for Focusing and Prioritizing the Testing Effort

Dario Di Nucci
Vrije Universiteit Brussel, Brussels, Belgium
dario.di.nucci@vub.be

***Abstract*—Software testing is essential for any software development process, representing an extremely expensive activity. Despite its importance recent studies showed that developers rarely test their application and most programming sessions end without any test execution. Indeed, new methods and tools able to better allocating the developers effort are needed to increment the system reliability and to reduce the testing costs. In this work we focus on three activities able to optimize testing activities, specifically, bug prediction, test case prioritization, and energy leaks detection. Indeed, despite the effort devoted in the last decades by the research community led to interesting results, we highlight some aspects that might be improved and propose empirical investigations and novel approaches. Finally, we provide a set of open issues that should be addressed by the research community in the future.**

***Index Terms*—Testing; Bug Prediction; Test Case Prioritization; Energy Efficiency**

I. Context

Software testing is widely recognized as an essential part of any software development process, representing, however, an extremely expensive activity [2]. Despite its importance, recent studies [3] showed that developers rarely test their application and most programming sessions end without any test execution. Thus, the available resources should be allocated effectively upon the portions of the source code that are more likely to contain bugs.

In this context, *bug prediction* [4] is a powerful technique that allow to predict which software components more likely contain bugs and need to be tested more extensively. In the last decade, several techniques were developed. They can be roughly classified into two families, based on the information they exploit to discriminate between "buggy" and "not buggy" code components. The first set of techniques exploits *product metrics* (*i.e.,* metrics capturing intrinsic characteristics of the code components, like their size and complexity) [5], while the second one focuses on *process metrics* (*i.e.,* metrics capturing specific aspects of the development process, like the frequency of changes performed to code components) [5]. While some studies highlighted the superiority of these latter with respect to the *product metric based* techniques [5] there is a general consensus on the fact that no technique is the best in all contexts [5]. For this reason, the research community is investigating under which circumstances and during which coding activities developers tend to introduce more bugs [6].

Bug prediction models rely on machine learning classifiers and their choice strongly influences the accuracy of predictions [7]. Panichella *et al.* [8] and Bowes *et al.* [9] demonstrated that the predictions of different classifiers are highly complementary despite similar prediction accuracy. Based on such findings, an emerging trend is the definition of prediction models which are able to combine multiple classifiers (*a.k.a., ensemble* techniques) and their application to bug prediction [8], [10]. Such models can be trained using a sufficient amount of labeled data coming from (i) the previous history of the same project where the model is applied to (*a.k.a., within-project* strategy) or (ii) other similar projects (*a.k.a., cross-project* strategy). Previous studies showed how within-project bug prediction models have higher capabilities than cross-project ones since they rely on data that better represents the characteristics of the source code elements of the project where the model have to be applied [11]. As a drawback, a within-project training strategy cannot often be adopted in practice since new projects might not have enough data to setup a bug prediction model [12]. As a consequence, the research community has started investigating ways to make cross-project bug prediction models more effective to allow a wider adoption of bug prediction models [13].

Another effective way to reduce the testing effort is applying regression testing optimization techniques [14], [15]. One of the known approaches, *test case prioritization* [16], aims at executing the available test cases in a specific order that increases the likelihood of revealing regression faults earlier. Thus, it is possible to reveal test cases that are unlucky to find faults, spending additional efforts in the maintenance of those that are more promising. Since fault detection capability is unknown before test execution, most of the proposed techniques use coverage criteria [14] as surrogates, with the idea that test cases with higher code coverage will have higher probability to reveal faults. Once a coverage criterion is chosen, search algorithms can be applied to find the ordering that maximize the selected criterion.

Finally, testing is not only related to the functional properties of a software system, but also on its non-functional ones. Among these, *energy efficiency* is becoming a major issue in modern software engineering, as applications performing their activities need to preserve battery life. The problem is even more evident in the context of mobile applications, where billions of customers rely on smartphones every day for social and emergency connectivity. Although the problem is mainly concerned with hardware efficiency, Hindle *et al.* [17] showed how even software may be the cause of energy leaks. Despite this, developers have limited knowledge of energy efficiency and the causes behind energy consumption [18]. Therefore, testing energy efficiency is particularly expensive and focusing on the software components that are more likely to consume energy would be reasonable. In the context of Android mobile applications, a set of new peculiar bad programming practices has been defined by Reimann *et al.* [19]. These Android-specific smells may threat several non-functional attributes of mobile applications, such as security, data integrity, and source code quality [19]. Moreover, as highlighted by Hetch *et al.* [20], they could also lead to performance issues.

II. Research Statement

Even though the effort devoted by the research community to focus and prioritizing the testing effort through the conduction of empirical studies and the definition of new approaches led to interesting results, in the context of our research we highlight some aspects that might be improved, summarized as follow:

Bug Prediction: although previous studies showed the potential of human-related factors in bug prediction, this information is not captured in state-of-the-art models. Indeed, the models based on process metrics exploit predictors based on (i) the number of developers working on a code component [21]; (ii) the analysis of change-proneness [22]; (iii) the entropy of changes [23], and (iv) the micro-interactions of developers [24]. Thus, despite the previously discussed finding, none of these models considers to what extent developers performing changes are focused or these changes are scattered.

Bug Prediction: traditional ensemble approaches miss the predictions of a large part of bugs that are correctly identified by a single classifier. Therefore, "ensemble decision-making strategies need to be enhanced to account for the success of individual classifiers in finding specific sets of bugs" [9]. Moreover, previous empirical studies on the use of ensemble techniques for bug prediction [25] have some critical limitations that could have impacted on the performances of classifiers thus possibly threatening the conclusions provided so far. These issue were related to (i) data quality, (ii) data preprocessing, (iii) data analysis, and (iv) limited size. Furthermore, these studies exposed an unclear relationship between local learning and ensemble classifiers and did not compare the performance achieved by cross-project models with respect to within-project ones.

Test Case Prioritization: we observed that the AUC metric used in the related literature represents a simplified version of the well-known hypervolume [26]. This metric is already used in many-objective optimization and we argue that it could be used to condense not only a single cumulative code coverage criteria (as done by previous AUC metrics used in literature) but also multiple testing criteria, such as the test case execution cost or further coverage criteria (*e.g.,* branch and past-fault coverage). This scalar value could be used as fitness function in a search-based algorithm.

Energy Efficiency Testing: little knowledge is available in literature on the potential impact on energy consumption of the Android-specific code smells defined by Reimann *et al.* [19]. These smells are detectable through static analysis, hence they could be used to efficiently detect energy leaks. Unfortunately, while the impact of these smells on energy consumption has been theoretically supposed by Reimann *et al.* [19], there exists only a few empirical evidence on it. For this reason, we aim at conducting a large empirical study to analyze the impact of the Android-specific smells on the energy consumption of mobile apps.

Based on these observations, our research have the goal to address the following high-level research questions:

- **RQ$_1$**: *To what extent developer's scattering metrics are able to improve a bug prediction model based on state-of-the-art metrics?*
- **RQ$_2$**: *To what extent a technique able to select a classifier based on the characteristics of the class is able to outperform state-of-the-art classifiers?*
- **RQ$_3$**: *What are the cost-effectiveness, the efficiency, and scalability of a genetic algorithm based on the hypervolume, compared to state-of-the-art test case prioritization techniques?*
- **RQ$_4$**: *To what extent Android-specific code smells can be used to focus energy testing of mobile apps?*

The final goal is to provide developers new approaches and tools, able to (i) focus their effort on bug-prone components, (ii) prioritize test cases to find faults earlier, and (iii) quickly detect energy flaws in mobile apps.

III. Research Results

The research conducted so far achieved the results reported in the following:

Bug Prediction: A Developer Centered Bug Prediction Model. To answer RQ$_1$, we define two metrics [27], [28], namely the *developer's structural and semantic scattering*. The first assesses how "structurally far" in the software project the code components modified by a developer in a given time period are. The second measure is instead meant to capture how much spread in terms of implemented responsibilities the code components modified by a developer in a given time period are. The conjecture behind the proposed metrics is that high levels of these metrics make developers more error-prone. To verify this conjecture, we build two predictors exploiting the proposed metrics and we use them in a bug prediction model. Firstly, we conduct a case study on 26 software systems and compare our model with respect to four models based on state-of-the-art metrics. Secondly, we devise and discuss the results of an hybrid bug prediction model, based on the best combination of predictors exploited by the five prediction models experimented. We find that the model based on the developer's scattering metrics performs better than the baseline approaches, demonstrating their superiority in correctly predicting buggy classes. By combining the eleven predictors exploited by the five prediction models subject of our study we obtain a boost of prediction accuracy up to +5% with respect to the best performing model and +9% with respect to the best combination of baseline predictors. Also, the top five "hybrid" prediction models include at least one of the predictors proposed in the work and the best model includes both.

Bug Prediction: Dynamic Selection of Classifiers in Bug Prediction. To answer RQ_2, we propose a novel adaptive prediction model, coined as ASCI (**A**daptive **S**election of **C**lass**I**fiers in bug prediction) [29], which dynamically recommends the classifier able to better predict the bug-proneness of a class, based on the structural characteristics of the class. To build and evaluate our approach, we provide a *differentiated* replication study in which not only we corroborate previous empirical research on the performances of ensemble classifiers for cross-project bug prediction, but also extend previous knowledge by assessing the extent to which local bug prediction [30] can benefit of the usage of ensemble techniques. The study has been conducted on a PROMISE dataset [31], [32] composed of 21 software systems, where we apply a number of corrections suggested by Shepperd *et al.* [33] to make it suitable for our purpose. We take into account several different ensemble techniques, belonging to six categories, measuring their performances using the two metrics recommended by previous work, *i.e.,* AUC-ROC and Matthew's Correlation Coefficient (MCC) [34], [35]. We find that, in the context of within-project bug prediction, the use of an ensemble classifier does not guarantee better prediction performances with respect to the best stand-alone classifier (*e.g.,* NAIVE BAYES). We confirm that the models based on VALIDATION AND VOTING are able to achieve slightly better results, but the obtained improvement is not statistically significant with respect to other ensemble techniques, such as RANDOM FOREST and ASCI. None of the cross-project models experimented is able to exceed 25% of MCC (on average), meaning that the problem of identifying buggy classes using external sources of information is still far from being solved. Furthermore, the use of ensemble techniques does not provide evident benefits with respect to stand-alone classifiers. Indeed, the models based on NAIVE BAYES or using it as weak learner are able to achieve the best performances. ASCI, instead, does not work properly in a cross-project context. Moreover, we find that local learning is often not able to improve the performances of bug prediction models. The only exception is represented by ASCI, which has better performances with respect to those achieved by global models. The statistical analysis, however, highlight how local and global models are mostly equivalent.

Test Case Prioritization: Hypervolume Genetic Algorithm for Test Case Prioritization. To answer RQ_3, we propose HGA (HYPERVOLUME-BASED GENETIC ALGORITHM) [36] and provide an extensive evaluation of Hypervolume-based and state-of-the-art approaches for solving the problem when dealing with up to five testing criteria. In particular, we carry out a case study to assess the *cost-effectiveness*, the *efficiency*, and the *selective pressure* capabilities of the various approaches. We compare `HGA` with respect to five state-of-the-art techniques: (i) a cost cognizant additional greedy algorithm [16], [37], (ii) `GA`, a single objective genetic algorithm based on an AUC metric [15], (iii) `NSGA-II`, a multi-objective search-based algorithm [38], (iv) `GDE3` [39], and (v) `MOEA/D-DE` [40]. Our results suggest that the solution (test ordering) produced by `HGA` is more cost-effective than the solution generated by `Additional Greedy`, `GA`, and the Pareto optimal solution achieved by `NSGA-II`. In terms of efficiency, `HGA` is much faster than `GA`, `NSGA-II`, and `Additional Greedy`, and its efficiency does not decrease as the size of the software program or the test suite increase. Moreover, `HGA` is not only more or equally effective than the state-of-the-art many-objective algorithms but it is also up to 3 times more efficient.

Energy Efficiency Testing: On the Impact of Code Smells on the Energy Consumption of Mobile Applications. To answer RQ_4, we propose two novel tools and a large empirical study. ADOCTOR [41], a novel code smell detector that identifies 15 Android-specific code smells. The tool exploits the Abstract Syntax Tree of the source code and navigates it by applying detection rules based on the exact definitions of the smells provided by Reimann *et al.* [19]. We experiment ADOCTOR against the source code of 18 Android apps and compare the set of candidate code smells given by the tool with a manually-built oracle. At the same time, we develop PETRA [42], [43], a novel tool for extracting the energy profile of mobile applications specific for Android OS. We evaluate PETRA on 54 mobile applications belonging to a publicly available dataset. We compare the energy measurements provided by PETrA against the actual energy consumption computed using the Monsoon hardware toolkit[1] for the same apps and using the same hardware/software setting. Finally, we provide a deeper investigation to determine (i) to what extent code smells affecting source code methods of mobile applications influence energy efficiency and (ii) whether refactoring operations applied to remove them directly improve the energy efficiency of refactored methods. In particular, our investigation focuses on nine method-level code smells specifically defined for mobile applications by Reimann *et al.* [19] in the context of 60 Android apps. To the best of our knowledge, this is up to date the largest study aimed at practically investigating the actual impact of such code smells on energy consumption and quantifying the extent to which refactoring code smells is beneficial for improving energy efficiency. Our *coarse-grained* preliminary shows that the presence of code smells can result in a strong increment of the energy consumption. Moreover, we find that methods affected by more smells consume more than methods not affected by design flaws. Our analysis reveals that there exist four *energy-smells*, *i.e., Leaking Thread*, *Member Ignoring Method*, *Slow Loop*, and *Internal Setter*, which significantly impact the energy consumption of methods in a mobile app. Refactoring code smells has a key role in

[1]http://www.msoon.com/LabEquipment/PowerMonitor/

improving the energy efficiency of source code methods and should be applied by mobile developers.

Besides the contributions described above, we provide two further common contributions:

Large-scale Empirical Studies. All the studies conducted in our research have been conducted on large sets of software systems to ensure the generalizability of the findings.

Publicly Available Tools and Replication Packages. Tools, scripts, and datasets, needed to perform the analyses, are publicly available as tools or online replication packages[2,3].

IV. Challenges and Open Issues

Despite the effort devoted by the research community and the advances discussed in this work, reducing the testing effort still present a number of open issues and challenges that should be addressed in the future.

Challenge #1: Bug Prediction in the Wild. Bug prediction should be spread in industrial contexts. To reach this goal, new challenges arise:

Catch Hard Bugs. In the upcoming years more effort should be devoted in performing user studies with developers aimed at evaluating the real usefulness of the suggestions provided by the different bug prediction models [44]. Researchers should investigate the ability of bug prediction models in predicting bugs that are hard to catch for humans.

Investigate the Cause behind Scattering. The role of developer-related factors in the bug prediction field is still a partially explored area. A deeper investigation of the factors causing scattering to developers, and negatively impacting their ability of dealing with code change tasks is needed. Hence, user studies should be performed to analyze the bad practices that lead to introduction of bugs. New metrics able to capture these behaviors should be introduced.

Towards Agile and Continuous Integration. Despite just-in-time bug prediction models allow to produce fine-grained recommendations to developers, there are still many challenges [45], [46] to be addressed with the aim of bringing more effective and stable models applicable during continuous integration.

Bug Prediction for Resource Scheduling. Developers' scattering metrics have shown to be effective in predicting the bugginess of code components. Based on this observation, a next step would be to propose efficient scheduling able to minimize the developers' scattering thus reducing the number of bugs.

Challenge #2: Unified Algorithm for Regression Testing Optimization Problems. Test Case Prioritization is only one of the optimization problem related to regression testing. In the last decade for these problems different algorithms were proposed. The use of HGA [36] as unified algorithm for regression testing optimization problems, such as Test Suite Minimization and Test Case Selection, poses new challenges in terms of cost-effectiveness, efficiency, and scalability. Moreover, considering our previous results, we can assert that not all the testing criteria are equally important. Thus, more investigation on their fault discovering capabilities is needed.

[2]https://figshare.com/authors/Dario_Di_Nucci/3088926
[3]http://www.sesa.unisa.it/landfill

Challenge #3: Design and development of new tools for improving energy efficiency. The poor knowledge of developers regarding energy consumption issues is one of the main obstacles that prevent the diffusion of energy optimized mobile applications [18]. For this reason, a new generation of code quality-checkers and refactoring tools is needed. In this context, the automatic refactoring of energy greedy code components and the automatic generation of test cases able to discover energy leaks are part of our future agenda.

V. Lessons Learnt

In my opinion a Ph.D is not only another step in the educational ladder but a path composed of the people that you meet, the bad days after failures, the pros and cons of academic world, and last but not the least the research goals that you achieve.

Lesson #1: "Research is Fun!" There is nothing better than solving a cutting edge problem applying new methodologies and prototypes. I think that the choice of a research topic is a very difficult step. If you do not find that your research is interesting and fun, nobody will.

Lesson #2: Cope with Failures. Failure is part of research life. Rejections are part of the research life and are needed for improving. Thus the only way for achieving great results is cope with failures. Most of the papers that are part of my dissertation were rejected at least one time (*e.g.,* 6 rejections for 15 submission).

Lesson #3: Interact with People. Researchers are people and most of them are willing to share their knowledge. Talk with them, try to create collaborations, learn from their mistakes.

Lesson #4: Enjoy this World. Looking to the timeline of history, you could think that you are in the wrong period for doing research (*e.g.,* Alan Turing's H-index is only 39). I think that in the human history there has never been a period in which it is so easy to travel and meet new people that are in love with your research interests.

Acknowledgments

The author is very grateful to his advisor Andrea De Lucia for all the help and support received during his PhD.

References

[1] D. Di Nucci, "Methods and tools for focusing and prioritizing the testing effort," Ph.D. dissertation, Università degli Studi di Salerno, 2018. [Online]. Available: https://www.researchgate.net/publication/324007859_Methods_and_Tools_for_Focusing_and_Prioritizing_the_Testing_Effort

[2] B. Beizer, *Software testing techniques*. Dreamtech Press, 2003.

[3] M. Beller, G. Georgios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman, "Developer testing in the ide: Patterns, beliefs, and behavior," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2017.

[4] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Applied Soft Computing*, vol. 27, pp. 504–518, 2015.

[5] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4, p. 531–577, 2012.

[6] J. Sliwerski, T. Zimmermann, and A. Zeller, "Don't program on fridays! how to locate fix-inducing changes," in *7th Workshop Software Reengineering*, 2005.

[7] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *37th International Conference on Software Engineering*. IEEE, 2015, pp. 789–800.

[8] A. Panichella, R. Oliveto, and A. D. Lucia, "Cross-project defect prediction models: L'union fait la force," in *IEEE CSMR-WCRE*, 2014, pp. 164–173.

[9] D. Bowes, T. Hall, and J. Petrić, "Software defect prediction: do different classifiers find the same defects?" *Software Quality Journal*, pp. 1–28, 2017.

[10] J. Petrić, D. Bowes, T. Hall, B. Christianson, and N. Baddoo, "Building an ensemble for software defect prediction based on diversity selection," in *10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2016, p. 46.

[11] B. Turhan, T. Menzies, A. B. Bener, and J. S. D. Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Software Engineering*, vol. 14, no. 5, pp. 540–578, 2009.

[12] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *7th ACM SIGSOFT ESEC/FSE*. ACM, 2009, pp. 91–100.

[13] S. Herbold, A. Trautsch, and J. Grabowski, "A comparative study to benchmark cross-project defect prediction approaches," *IEEE Transactions on Software Engineering*, 2017.

[14] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[15] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[16] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[17] A. Hindle, "Green mining: A methodology of relating software change and configuration to power consumption," *Empirical Software Engineering*, vol. 20, no. 2, pp. 374–409, Apr. 2015.

[18] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, "What do programmers know about software energy consumption?" *IEEE Software*, vol. 33, no. 3, pp. 83–89, 2016.

[19] J. Reimann, M. Brylski, and U. Aßmann, "A tool-supported quality smell catalogue for android developers," 2014.

[20] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of android code smells," in *International Conference on Mobile Software Engineering and Systems*. ACM, 2016, pp. 59–69.

[21] R. Bell, T. Ostrand, and E. Weyuker, "The limited impact of individual developer data on software defect prediction," *Empirical Software Engineering*, vol. 18, no. 3, pp. 478–505, 2013.

[22] W. P. Raimund Moser and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *30th International Conference on Software Engineering*, 2008, pp. 181–190.

[23] A. E. Hassan, "Predicting faults using the complexity of code changes," in *31st International Conference on Software Engineering*. IEEE, 2009, pp. 78–88.

[24] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In, "Developer micro interaction metrics for software defect prediction," *IEEE Transactions on Software Engineering*, vol. 42, no. 11, pp. 1015–1035, 2016.

[25] Y. Zhang, D. Lo, X. Xia, and J. Sun, "An empirical study of classifier combination for cross-project defect prediction," in *IEEE Annual Computer Software and Applications Conference*, vol. 2. IEEE, 2015, pp. 264–269.

[26] A. Auger, J. Bader, D. Brockhoff, and E. Zitzler, "Theory of the hypervolume indicator: optimal μ-distributions and the choice of the reference point," in *SIGEVO workshop on Foundations of Genetic Algorithms*. ACM, 2009, pp. 87–102.

[27] D. Di Nucci, F. Palomba, S. Siravo, G. Bavota, R. Oliveto, and A. De Lucia, "On the role of developer's scattered changes in bug prediction," in *2015 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2015, pp. 241–250.

[28] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia, "A developer centered bug prediction model," *IEEE Transactions on Software Engineering*, vol. 44, no. 1, pp. 5–24, 2018.

[29] D. Di Nucci, F. Palomba, R. Oliveto, and A. De Lucia, "Dynamic selection of classifiers in bug prediction: An adaptive method," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 1, no. 3, pp. 202–212, 2017.

[30] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann, "Local versus global lessons for defect prediction and effort estimation," *IEEE Transactions on software engineering*, vol. 39, no. 6, pp. 822–834, 2013.

[31] "The promise repository of empirical software engineering data," 2015.

[32] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *6th International Conference on Predictive Models in Software Engineering*. ACM, 2010, pp. 9:1–9:10.

[33] M. Shepperd, Q. Song, Z. Sun, and C. Mair, "Data quality: Some comments on the nasa software defect datasets," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1208–1215, 2013.

[34] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "Developing fault-prediction models: What the research can show industry," *IEEE Software*, vol. 28, no. 6, pp. 96–99, 2011.

[35] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "Automated parameter optimization of classification techniques for defect prediction models," in *38th International Conference on Software Engineering*, 2016, pp. 321–332.

[36] D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "Hypervolume-based search for test case prioritization," in *Symposium on Search-Based Software Engineering*, ser. Lecture Notes in Computer Science. Springer, 2015, vol. 9275, pp. 157–172.

[37] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *International Symposium on Software Testing and Analysis*. ACM, 2007, pp. 140–150.

[38] M. G. Epitropakis, S. Yoo, M. Harman, and E. K. Burke, "Empirical evaluation of pareto efficient multi-objective regression test case prioritisation," in *2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 234–245.

[39] S. Kukkonen and J. Lampinen, "Gde3: The third evolution step of generalized differential evolution," in *2005 IEEE Congress on Evolutionary Computation*. IEEE, 2005, pp. 443–450.

[40] H. Li and Q. Zhang, "Multiobjective optimization problems with complicated pareto sets, moea/d and nsga-ii," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 2, pp. 284–302, 2009.

[41] F. Palomba, D. D. Nucci, A. Panichella, A. Zaidman, and A. D. Lucia, "Lightweight detection of android-specific code smells: The adoctor project," in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*, 2017, pp. 487–491.

[42] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. D. Lucia, "Software-based energy profiling of android apps: Simple, efficient and reliable?" in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*, 2017, pp. 103–114.

[43] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. De Lucia, "Petra: a software-based tool for estimating the energy profile of android applications," in *39th International Conference on Software Engineering Companion*. IEEE, 2017, pp. 3–6.

[44] M. Lanza, A. Mocci, and L. Ponzanelli, "The tragedy of defect prediction, prince of empirical software engineering research," *IEEE Software*, vol. 33, no. 6, pp. 102–105, 2016.

[45] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," *IEEE Transactions on Software Engineering*, 2017.

[46] 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 and Software Technology*, 2018.