Regularity or Anomaly? On The Use of Anomaly Detection for Fine-Grained JIT Defect Prediction

Francesco Lomio,[1] Luca Pascarella,[2] Fabio Palomba,[3] Valentina Lenarduzzi[4]
[1]Tampere University — [2]Università della Svizzera Italiana — [3]University of Salerno — [4]University of Oulu
francesco.lomio@tuni.fi; luca.pascarella@usi.ch; fpalomba@unisa.it; valentina.lenarduzzi@oulu.fi

***Abstract*—Fine-grained just-in-time defect prediction aims at identifying likely defective files within new commits. Popular techniques are based on supervised learning, where machine learning algorithms are fed with historical data. One of the limitations of these techniques is concerned with the use of imbalanced data that only contain a few defective samples to enable a proper learning phase. To overcome this problem, recent work has shown that anomaly detection can be used as an alternative. With our study, we aim at assessing how anomaly detection can be employed for the problem of fine-grained just-in-time defect prediction. We conduct an empirical investigation on 32 open-source projects, designing and evaluating three anomaly detection methods for fine-grained just-in-time defect prediction. Our results do not show significant advantages that justify the benefit of anomaly detection over machine learning approaches.**

***Index Terms*—Defect Prediction, Anomaly Detection, Empirical Software Engineering, Software Evolution.**

I. Introduction

Throughout the software lifecycle, developers eventually introduce defects as a consequence of development activities. Therefore, researchers have proposed methods to identify source code showing characteristics indicating the proneness to be defective, such as static analysis techniques, code smell detection methods, and more (1). One of the most timely practices to deal with software defects is called *fine-grained just-in-time defect prediction* (2).

In this context, most of the existing work has studied just-in-time defect prediction under different perspectives: researchers have indeed worked on (i) the definition of suitable predictors to use when feeding machine learning algorithms (3), (ii) the optimization of the training strategies (4), (iii) the correct estimation of the nature of defects (5), and (iv) the configuration of machine learning algorithms (6). Recently, Pascarella et al. (2) proposed to lower the granularity of just-in-time defect prediction so that the recommendations given can indicate the files within commits that are likely to be defective.

Anomaly detection, a.k.a., outlier detection (7), is the field of machine learning that concerns with the identification of rare items that raise suspicions by differing significantly from the majority of the data. As recently shown (8), anomaly detection can represent a promising alternative to standard supervised machine learning models when it comes to defect prediction, especially because they do not necessarily require to be trained and, for this reason, do not risk to learn from imbalanced data sets. Indeed, in such a formulation defects are seen as rare events that appear during the development, while the task of defect predictors is to mine time series representing the evolution of source code properties over time and learn which of these properties deviate from the normal behavior, indicating the presence of defective source code.

In this paper, we propose the first investigation to evaluate the performance of anomaly detection methods for fine-grained just-in-time defect prediction. In particular, we conduct an empirical investigation involving 32 open-source projects with a total amount of 61,081 commits where the percentage of defective files is around 34%, overall (9). We experiment with three anomaly detection techniques for fine-grained just-in-time defect prediction, i.e., One Class SVM, Isolation Forest, and Local Outlier Factor. We assess these methods and compare them with three baseline fine-grained just-in-time supervised learning models. We report a negative result. However, we discover a complementarity between anomaly detection and machine learning methods, with the former having higher capabilities on unbalanced datasets and the latter on more defective projects.

II. Research Methodology

The *goal* of the empirical study was to assess the performance of anomaly detection methods when employed for the task of fine-grained just-in-time defect prediction, with the *purpose* of understanding how they compare with traditional approaches based on machine learning.

Hence, we formulate our RQ: *How do anomaly detection methods compare to existing supervised learning techniques?*

Context of the Study. We exploited the *Technical Debt Dataset* (10), which is a set of data coming from 32 Java projects mostly pertaining to the Apache Software Foundation project. The projects summed up to 64,320 commits. Nevertheless, we observed that some of them were unreasonably large, e.g., up to 4,715 files modified within a single commit. As reported by Hattori and Lanza (11), this is a typical situation in open source, where some commits pervasively modify the status of the repository as a consequence of changes connected to licenses (12) or code style (e.g., space versus tabular indentation of the code) (13). As pointed out by previous work (14), those commits might negatively bias the interpretability of automated methods. Therefore, we only retained the commits that appeared in the 95-percentile of the distribution of files in all commits—the threshold was selected based on the recommendations given by Alabi et al. (14). This filtering procedure removed 3,239 commits and, therefore, our analyses were based on a total amount of 61,081 code changes.

Collecting Fine-Grained Information on Software Defects. We employed a similar methodology as done by Pascarella et al. (2). We first identified the so-called *defect-fixing commits* by looking at the information available on the JIRA issue tracker of the projects. and by mining commits whose messages explicitly report a fix operation. The union of the commits identified using the two procedures represented our final set of defect-fixing commits and contained the full list of files that we needed to trace back to the point in time in which they were made defective. As done in the reference work (2), we considered *defect-inducing* only those files which modified lines are the actual source of defects and discarded the remaining files whose changes are not involved in fixes. We are aware of the criticisms made with respect to the accuracy of SZZ (5). To limit the amount of false positives given by the algorithm, we have applied some adjustments. In the first place, we (i) ignored non-source code files belonging to defect-fixing commits, (ii) filtered out the defect-inducing commits that appeared as merge commits, and (iii) carefully considered the literature on the adoption of SZZ and decided not to opt for variants of the algorithms that take into account specific conditions (e.g., appearance of refactoring operations (15)).

Collecting Fine-Grained Software Metrics. We considered the same metrics as the baseline fine-grained model proposed by Pascarella et al. (2). This choice allowed us to (i) rely on an established set of independent variables previously validated in the context of fine-grained just-in-time defect prediction and (ii) have a fair comparison between the results achieved through anomaly detection methods and those of machine learning models. For the sake of space limitations, we report those metrics in our online appendix (9). Although Pascarella et al.(2) provided a publicly accessible appendix containing the scripts used to compute the metrics, we needed to slightly adjust them to fix a runtime issue caused by an outdated API. The revised tool systematically collected the metrics for each file of each commit belonging to the considered projects. In so doing, the tool (1) started collecting new metrics as soon as a new file F_i was added to a repository, (2) updated the metrics of F_i whenever a commit modified it, (3) kept track of possible file renaming by relying on the GIT internal rename heuristic and subsequently updating the name of F_i, and (4) removed F_i in the case it was deleted.

Setting the Anomaly Detection Models. In the context of our study, we focused on three models such as *OneClassSVM*, *IsolationForest*, and *LocalOutlierFactor*. The reason behind the selection of these methods was twofold. On the one hand, each of them is based on a different class of algorithms, hence allowing us to provide a wider overview of how anomaly detection can be applied to the problem of fine-grained just-in-time defect prediction. On the other hand, these methods are among the most stable and reliable ones (7), other than being commonly used in multiple environments (16), including software maintenance and evolution (17).

Data Analysis. We opted for a Leave One Group Out (LOGO) validation strategy that trains n models, with n the number of groups (projects in our case) in the data. For each fold, $n - 1$ groups are used for training, and 1 for testing. For this work, we used 31 projects at the time as training set and 1 project as test set. This process was repeated 32 times, so that all the projects in the dataset were in the test set exactly once. It is important to highlight that doing this, the commit of a project cannot be split between train and test set. This constraint avoided possible bias due to the time-sensitive nature of code commits. With respect to other strategies (e.g., Out-of-sample bootstrap validation), it is among the easiest to interpret (18), hence perfectly fitting the goal of our exploratory analysis. Moreover, the LOGO validation represents a good compromise between the bias and variance of estimates of defect prediction models (19), thus further strengthening its suitability for our case.

III. RESULTS AND FURTHER ANALYSES

Figures 1 depicts box plots reporting the distribution of AUC-ROC obtained value. The first three box plots refer to the machine learning approaches, while the last three to the anomaly detection ones. Note that for the sake of space limitation, we only report the AUC-ROC, while the full results are included in our replication package (9). Looking at the results, we could first observe that the anomaly detection methods, i.e., IF, OSCVM, and LOF have a similar AUC-ROC, with values around 0.5. This indicates an overall low ability of these methods to distinguish between defective and non-defective files. The machine learning models, i.e., EXTRATREES, SVM, and KNN, were instead generally more performing. This is particularly true when considering EXTRATREES: this is the classifier that reached the best AUC-ROC values (median=0.71). The only exception concerned the performance of SVM, which were notably lower with respect to all other experimented methods.

On the one hand, our findings confirm that the choice of classifiers can significantly influence the performance of defect prediction models: with respect to previous work, we show that this is true also in the case of fine-grained just-in-time defect prediction. On the other hand, the relatively low values achieved by the experimented anomaly detection methods seem to highlight a negative result: these are not only unable to provide benefits in terms of AUC-ROC, but also have significantly lower performance with respect to traditional machine learning models. This was confirmed by the statistical tests we performed. In particular, we run the Mann-Whitney and Cliff's d tests to compare the mean of the distributions of AUC-ROC values and discovered that all the anomaly detection methods perform statistically worse than EXTRATREES and KNN, with large effect sizes. Results could not confirm the promising preliminary results achieved by researchers that applied anomaly detection to higher-level defect prediction problems. Nonetheless, this might be explained by the peculiarities of the dataset considered as well as the variability in terms of defects.

Discussion. Our results do not really come as a surprise: in a real-case scenario, there exist projects having different levels of defectiveness and, thus, it is reasonable to believe that anomaly detection methods might work well when considering projects with a low defectiveness, suffering instead in the opposite case. Our empirical study depicts a typical scenario observable in the wild: some projects reach up to 49% of defective files during their change history, others have instead a notably lower defectiveness, i.e., 15%.

To investigate the above mentioned conjecture and further understand the capabilities of anomaly detection for fine-grained just-in-time defect prediction, we conducted an additional analysis aiming at assessing its performance when considering datasets of various levels of defectiveness. Specifically, we re-executed the methodology, but this time on two smaller datasets: (1) the first composed of the three projects having the lowest percentage of defective files, i.e., COMMONDS-DBUTILS, COMMONS-DAEMON, and COMMONS-BCEL; (2) the second composed of the three projects with the highest amount of defective files, i.e., AURORA, ACCUMULO, and ATLAS. In this way, we could actually consider two extreme cases and verify if the performance of anomaly detection are higher than the baselines in the first case and lower in the second.

In both cases, the LOGO validation was performed so that two projects formed the training set and the remaining one the test set—as previously done, the validation was repeated three times to have each project in the test set exactly once. Figures 2 depicts the results: to ease their interpretability, the figure reports the box plots of the AUC-ROC values related to the performance of anomaly detection methods and machine learning models on (1) the dataset only composed of projects with low defectiveness; (2) the dataset only composed of projects with high defectiveness; and (3) the full dataset, i.e., the results presented in Section III. The additional analysis confirmed our intuition. The anomaly detection methods experimented, and in particular LOF, overcome the baseline machine learning models when considering the low-defectiveness dataset. In terms of F-Measure, the differences are pretty evident (up to 23%) as also confirmed by the statistical tests that reported them to be statistically significant with a large effect size. Interestingly, also the performance of the EXTRATREES learner—which was the best on the full dataset—decreased significantly, showing its limitations when dealing with unbalanced data. To make a more practical sense to the discussion, we also computed the cumulative number of absolute defects identified by the various techniques over the two smaller datasets—the full results are available in our online appendix. We could observe that the number of actual defects identified by the anomaly detection techniques increases with the hardness of finding those defects, i.e., with the decrease of the total number of defects. For instance, LOF was able to identify 521 defects more than EXTRATREES on the low-defectiveness dataset. On the contrary, the machine learning models are confirmed to be generally better on the high-defectiveness dataset. This is particularly relevant when it turns to the assessment of the AUC-ROC, where the EXTRATREES model is significantly better than all other baselines. These findings allow us to report a complementarity between anomaly detection and machine learning approaches that would be worth investigating further in the future. We believe that improvements in the field of fine-grained just-in-time defect prediction might be reached by means of combinations of multiple approaches: while some previous work focused on ensemble machine learning (20), our findings suggest that additional, potential enhancements may revolve around the definition of context-dependent ensembles of supervised and unsupervised learning mechanisms like machine learning and anomaly detection.

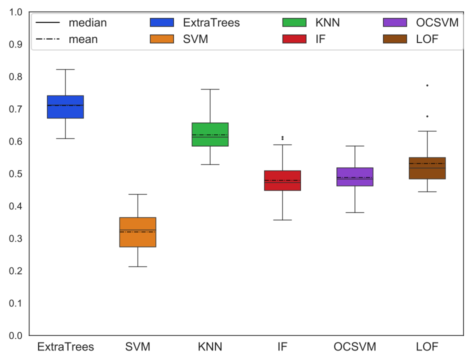

Fig. 1. AUC-ROC comparison comparison among anomaly detection and supervised learning models for the filtered dataset

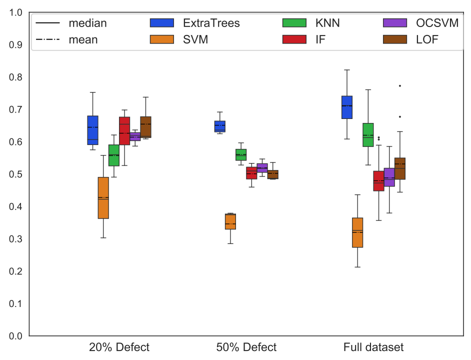

Fig. 2. AUC-ROC comparison among anomaly detection and supervised learning models on the filtered datasets.

IV. RELATED WORK

Although software defect prediction is one of the most active research areas in software engineering, due to space constraints, we focus only on works relevant to our subject while for a full overview of defect prediction research, we point to a recent literature review (1). Just-in-time defect prediction refers to a class of techniques that can anticipate the identification of defects at commit-time (21), rather than predicting defects using a long-term approach that, instead, shows prediction results at release-time (22). In the recent past, researchers pushed for a shorter-term analysis of defects since this better fits the developers' needs (23): indeed, these models allow developers to promptly react when changes are still fresh in their minds (24). For example, Mockus and Weiss (25) experimented with a preliminary model able to predict failure probabilities at commit-level: taking the properties of the change done by a developer as input, the model reported prompt feedback that made developers more able to operate on the code (25). Later on, Madeyski and Kawalerowicz (26) elaborated on *continuous* defect prediction, developing a public dataset containing tools for experimenting with defect prediction techniques. Only two papers approached defects as anomalies, applying anomaly detection techniques as univariate and multivariate Gaussian distribution (27) or supervised anomaly detection (28).

V. Conclusion

We proposed an investigation into the capabilities of anomaly detection methods for fine-grained just-in-time defect prediction. While the results achieved showed that anomaly detection performed similarly to machine learning models, we observed that the level of defectiveness of projects might influence the capabilities of anomaly detection methods. After a deeper investigation, we found a complementarity between anomaly detection and machine learning techniques that might be further exploited to create context-dependent predictors.

References

[1] N. Li, M. Shepperd, and Y. Guo, "A systematic review of unsupervised learning techniques for software defect prediction," *Information and Software Tech.*, 2020.

[2] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *Journal of Systems and Software*, vol. 150, pp. 22–36, 2019.

[3] D. Di Nucci and et al., "A developer centered bug prediction model," *IEEE Trans. on Software Eng.*, 2017.

[4] F. Pecorelli and D. Di Nucci, "Adaptive selection of classifiers for bug prediction: A large-scale empirical analysis of its performances and a benchmark study," *Science of Computer Programming*, vol. 205, 2021.

[5] G. Rosa and et al., "Evaluating szz implementations through a developer-informed oracle," in *Int.Conference on Software Engineering*, 2021.

[6] C. Tantithamthavorn and et al., "The impact of automated parameter optimization on defect prediction models," *IEEE Trans. on Software Eng.*, vol. 45, 2018.

[7] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys*, 2009.

[8] P. Afric, L. Sikic, A. S. Kurdija, and M. Silic, "Repd: Source code defect prediction as anomaly detection," *Journal of Systems and Software*, vol. 168, 2020.

[9] "https://figshare.com/s/1459e02ef92c01d1a6b1."

[10] V. Lenarduzzi, N. Saarimäki, and D. Taibi, "The technical debt dataset," in *Conference on PREdictive Models and data analycs In Software Engineering*, 2019, pp. 2 – 11.

[11] L. P. Hattori and M. Lanza, "On the nature of commits," in *Int.Conference on Automated Software Engineering-Workshops*, 2008, pp. 63–71.

[12] J. Colazo and Y. Fang, "Impact of license choice on open source software development activity," *Journal of the American Society for Information Science and Technology*, vol. 60, no. 5, pp. 997–1011, 2009.

[13] J. Bauer and et al., "Indentation: simply a matter of style or support for program comprehension?" in *Int. Conference on Program Comprehension*, 2019.

[14] A. Alali, H. Kagdi, and J. I. Maletic, "What's a typical commit? a characterization of open source software repositories," in *Int.conference on program comprehension*, 2008, pp. 182–191.

[15] E. C. Neto, D. A. da Costa, and U. Kulesza, "Revisiting and improving szz implementations," in *Symposium on Empirical Software Engineering and Measurement*, 2019.

[16] S. Agrawal and J. Agrawal, "Survey on anomaly detection using data mining techniques," *Procedia Computer Science*, vol. 60, 2015.

[17] S. D. Palma and et al., "Singling the odd ones out: a novelty detection approach to find defects in infrastructure as code," in *Int.Workshop on Machine-Learning Techniques for Software-Quality Evaluation*, 2020, pp. 31–36.

[18] B. Letham and et al., "Interpretable classifiers using rules and bayesian analysis: Building a better stroke prediction model," *Annals of Applied Statistics*, vol. 9, 2015.

[19] C. Tantithamthavorn and et al., "An empirical comparison of model validation techniques for defect prediction models," *IEEE Trans. on Software Eng.*, 2016.

[20] D. Di Nucci, F. Palomba, R. Oliveto, and A. De Lucia, "Dynamic selection of classifiers in bug prediction: An adaptive method," *Transactions on Emerging Topics in Computational Intelligence*, 2017.

[21] Y. Fan and et al., "The impact of changes mislabeled by szz on just-in-time defect prediction," *IEEE Trans. on Software Eng.*, 2019.

[22] R. S. Wahono, "A systematic literature review of software defect prediction," *Journal of Software Eng.*, 2015.

[23] L. Pascarella and et al., "Information needs in contemporary code review," *ACM on Human-Computer Interaction*, vol. 2, no. CSCW, pp. 1–27, 2018.

[24] Y. Yang and et al., "Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models," in *Int.symposium on foundations of software engineering*, 2016, pp. 157–168.

[25] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, 2000.

[26] L. Madeyski and M. Kawalerowicz, "Continuous defect prediction: the idea and a related dataset," in *Int.Conference on Mining Software Repositories*, 2017, pp. 515–518.

[27] K. N. Neela and et al., "Modeling software defects as anomalies: A case study on promise repository." *JSW*, 2017.

[28] P. Afric and et al., "Repd: Source code defect prediction as anomaly detection," *Journal of Systems and Software*, vol. 168, 2020.