Regularity or Anomaly? On The Use of Anomaly Detection for Fine-Grained Just-in-Time Defect Prediction

Francesco Lomio,[1] Luca Pascarella,[2] Fabio Palomba,[3] Valentina Lenarduzzi[4]
[1]Tampere University — [2]Universitá 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 pushed by developers onto a shared repository. Most of the techniques proposed in literature 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 methods can be used as an alternative to supervised learning, given that these do not necessarily need labelled samples. We aim at assessing how anomaly detection methods 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. However, our results are negative because anomaly detection methods, taken alone, do not overcome the prediction performance of existing machine learning solutions.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**.

KEYWORDS

Defect Prediction, Anomaly Detection, Empirical Software Engineering, Software Evolution

ACM Reference Format:
Francesco Lomio,[1] Luca Pascarella,[2] Fabio Palomba,[3] Valentina Lenarduzzi[4]. 2022. Regularity or Anomaly? On The Use of Anomaly Detection for Fine-Grained Just-in-Time Defect Prediction. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (ICPC'22), May 16-17, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/xxx.xxx.xxx

1 INTRODUCTION

Throughout the software lifecycle, developers eventually introduce defects as a consequence of development and evolutionary activities [12]. With these circumstances being inevitable, the research community has proposed plenty of methods to support developers in identifying both defects and source code showing characteristics indicating its proneness to be defective, such as static analysis techniques, code smell detection methods, and more [17, 31, 33, 46]. One of the most timely practices to deal with software defects is called *fine-grained just-in-time defect prediction* [60]: this is a variation of the most well-known just-in-time defect prediction [38] that concerns with the adoption of (un)supervised learning mechanisms to locate defect-prone source code files at commit-level, namely while developers are pushing their changes onto a shared repository [66].

In this context, most of the existing work has studied just-in-time defect prediction under different perspectives: researchers have indeed heavily worked on (i) the definition of suitable predictors to use when feeding machine learning algorithms [18, 24, 36, 53], (ii) the optimization of the training strategies [64, 71, 90], (iii) the correct estimation of the nature of defects [41, 56, 76], and (iv) the configuration of machine learning algorithms [19, 81, 82]. More recently, Pascarella et al. [60] proposed to lower the granularity of just-in-time defect prediction so that the recommendations given can indicate the exact files within commits that are likely to be defective. In so doing, the authors have devised a supervised learning mechanism fed with a set of process metrics and whose performance was assessed in an empirical study that showed the potential usefulness of fine-grained just-in-time defect prediction.

While the extensive body of knowledge on both just-in-time defect prediction and its fine-grained variant has led to the definition of techniques able to achieve promising performance, the adoption of such techniques in practice is still threatened by usability concerns, e.g., lack of empirical investigations into the developer's ability to deal with defect prediction warnings [43]. Perhaps more importantly, only a few researchers have explicitly considered the *distribution* of defects in software projects: as a matter of fact, one of the causes of wrong predictions given by (fine-grained) just-in-time defect prediction models concerns with the fact that machine learning algorithms are often called to learn features from imbalanced environments [39], i.e., from data sets presenting only a few examples of defective source code entities. More specifically, recent studies [42, 50, 65, 88] showed that machine learning algorithms fed with imbalanced data might drastically lose their prediction capabilities and lead to a biased interpretation of the results.

Anomaly detection, a.k.a., outlier detection [13], 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 [1, 74], 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPC'22, May 16-17, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN xxx.xxx.xxx.xxx. . . $15.00
https://doi.org/10.1145/xxx.xxx.xxx

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 build on this line of research and propose the first experimental investigation into 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. 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. While the Local Outlier Factor provides slightly better F-Measure scores than supervised learning mechanisms, all the anomaly detection methods perform worse in terms of AUC-ROC. 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.

Structure of the paper. Section 2 reports the methodology employed to address the goal of our empirical study, while Section 3 presents the achieved results. In Section 4 we further discuss the main findings and provide the implications of our work for the research community. The threats to the validity of our results and our mitigation strategies are discussed in Section 5. The related literature is surveyed in Section 6, while Section 7 concludes the paper and overviews our future research agenda on the matter.

2 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. The *perspective* is that of researchers, who might be willing to assess the feasibility of using anomaly detection methods for defect prediction, as well as of practitioners, who might want to evaluate the feasibility of using those methods within their contexts. More specifically, we applied anomaly detection methods for the identification of defective files within commits of software projects and evaluated the resulting accuracy. Moreover, we compared the performance of anomaly detection methods against the state of the art, which in our case is represented by the adoption of machine learning models. Hence, we formulate our RQ:

RQ. *How do anomaly detection methods compare to existing supervised learning techniques?*

2.1 Context of the Study

The *context* of the study was composed of open-source software projects, and in particular by their change history information. In this respect, we exploited the *Technical Debt Dataset* [44], which is a curated collection of data coming from 32 Java projects mostly pertaining to the Apache Software Foundation ecosystem. Despite belonging to a single ecosystem, the projects of this dataset were originally selected by following the diversity guidelines introduced by Nagappan et al. [54], i.e., they were selected by addressing the representativeness of projects in terms of age, size, and domain, other than considering the Patton's "criterion sampling" [63], namely they are more than four years old, have more than 500 commits and 100 classes, and have more than 100 issues reported in their issue tracking system. As such, this dataset minimizes by design possible threats to external validity. To further verify the properties of this dataset, we have manually investigated the corresponding Github repositories and discovered that all of them adhere to a strict code of conduct citetourani2017code and regularly review source code to improve their quality processes [62]. This additional analysis further confirmed the suitability of the dataset.

In terms of commits, 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 [30], 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 [15] or code style (e.g., space versus tabular indentation of the code) [6, 91]. As pointed out by previous work [4, 32], those commits might negatively bias the interpretability of automated methods. For the sake of having a curated dataset, we decided to only retain 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. [4]. This filtering procedure removed 3,239 commits and, therefore, our analyses were based on a total amount of 61,081 code changes.

2.2 Collecting Fine-Grained Information on Software Defects

To address our research question, we first needed to collect information about the defectiveness of the files within each of the commits of the dataset—in other words, we needed to collect reliable information on the dependent variable that we aim at classifying using anomaly detection and traditional machine learning models. To this aim, we employed a similar methodology as done by Pascarella et al. [60]. We first identified the so-called *defect-fixing commits*, namely, the set of code changes where developers touch one or more files to address the previously introduced defects: these were identified by looking at the information available on the Jira issue tracker of the projects[1] 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. To this purpose, we run the well-known SZZ algorithm [34], which exploits the `git blame` capabilities to estimate the lines of code of a file that induced a defect fixed in a defect-fixing commit. Specifically, for each commit that fixes an issue, SZZ returns a list of one or more commits in which diverse files are blamed as the source of the issue. However, the blamed commits might touch several other files that are not involved in the successive fix. As done in the reference work [60], we considered *defect-inducing* only those files

[1] It is worth recalling that Jira allows developers to annotate the commit ID that fixed a given defect explicitly.

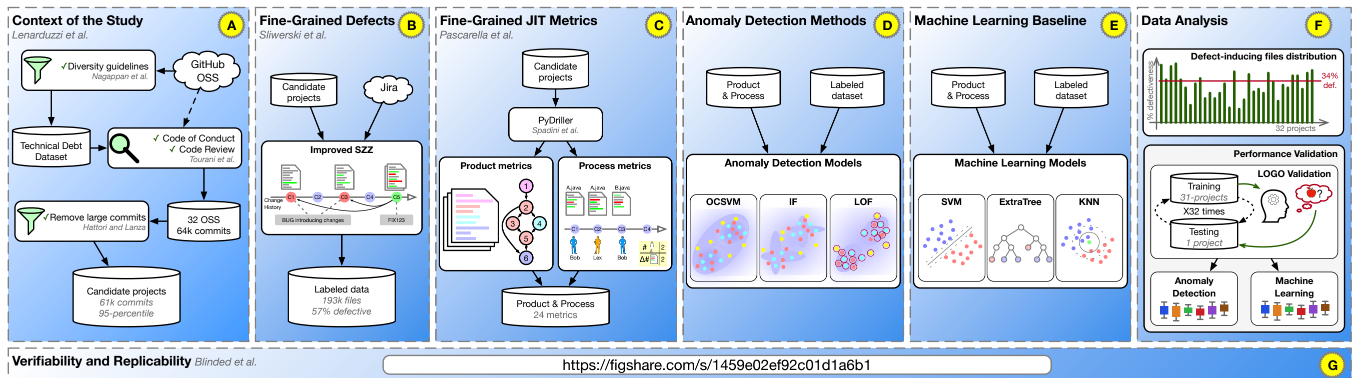

Figure 1: Overview of the methodology employed in our empirical study.

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 [75, 76]. To limit the amount of false positives given by the algorithm, we have applied some adjustments. In the first place, we ignored non-source code files belonging to defect-fixing commits (e.g., documentation or blob resources): in this way, we prevented SZZ from identifying inducing commits that are clearly not related to the defect. In second place, we filtered out the defect-inducing commits that appeared as merge commits—these do not report actual modifications. Last but not least, we 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 [56, 57]), as these were shown to underperform with respect to the original implementation proposed by Śliwerski et al. [34].

2.3 Collecting Fine-Grained Software Metrics

The second step of our data collection methodology was connected to the computation of software metrics that we could then use as independent variables of both anomaly detection and machine learning approaches. In particular, we required to gather a set of metrics for each file belonging to the commits of the considered projects. To this purpose, we considered exactly the same metrics as the baseline fine-grained model proposed by Pascarella et al. [60]. On the one hand, this choice allowed us to rely on an established set of independent variables previously validated in the context of fine-grained just-in-time defect prediction. On the other hand, it allowed a fair comparison between the results achieved by anomaly detection methods and those of machine learning models: indeed, we could replicate the previous study by Pascarella et al. [60], hence contrasting their results with the ones of anomaly detection and providing researchers with concrete indications on when and why one approach should be preferred to the other (and viceversa).

For the sake of space limitations, we report names and description of those metrics in our online appendix.[3] These metrics represent an adaptation of those originally proposed by Rahman et al. [70] and Kamei et al. [38]. With respect to the original works, the granularity of the metrics was lowered so that they could be computed on the files modified over the change history of software projects. Interestingly, the metrics cover various aspects of the development process such as: (1) the developers' expertise (e.g., the contribution frequency of a developer [38]), (2) the structure of changes (e.g., the number of changed lines in a commit [70]), (3) the evolution of the changes (e.g., the frequency of changes [70]), and (4) the dimensional footprint of a committed change (e.g., the relation between uncorrelated changes in a commit [79]). Hence, we could characterize commit' files under different perspectives—according to previous findings, a high metric diversity improves the capabilities of learning methods like the ones investigated in the context of our study [11, 18].

Pascarella et al. [60] provided a publicly accessible appendix containing the scripts used to compute the metrics. While we were able to correctly download the scripts, we needed to slightly adjust them in order to fix a runtime issue caused by an API that was no longer available. 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 permanently deleted. We made the updated tool publicly available in our online appendix [3].

2.4 Setting the Anomaly Detection Models

Once we had collected the dataset, we proceeded with the definition of the anomaly detection methods. 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 [13], other than being commonly used in multiple environments [3], including software maintenance and evolution [2, 58].

Particularly, the three methods are defined as follows:

OneClassSVM (OCSVM) [78]. This method is based on Support Vector Machine. Similarly, it learns a frontier which delimits the

initial observations. Any future observation will either lay in the frontier, therefore belonging to the same class as the original data (normal) or it will fall outside the frontier, being therefore classified as new, anomalous data. Unfortunately, *OneClassSVM* is prone to overfitting and, perhaps more importantly, the tuning of its hyperparameters can be challenging [13].

IsolationForest (IF) [48]. This is an *ensemble* technique based on the *Extremely Randomized Tree* model. In particular, it randomly selects a feature and randomly selects a split value between the maximum and minimum of the selected feature. The number of splitting required to isolate a sample equals the path length from the root to the final node of a tree. Since random partitioning produces noticeably shorter paths for anomalies, when a forest of random trees produces shorter path for particular samples, these are highly likely to be anomalies. By design, this method can handle high-dimensional data [13].

LocalOutlierFactor (LOF) [10]. This computes the local density of a given sample compared to its neighbors. The LOF density of an observation is given by the ratio of the average local density of its k-nearest neighbors, and its own local density. If the density is different than that their neighbors, it means that the sample analysed is an anomaly, else it is considered to be normal.

The anomaly detection methods were individually fed with the information collected on the subject systems. Hence, each method took as input a *NxM* dataset containing 193,800 rows (N) and 24 columns(M)—the rows reported all the files involved in all the 61,081 commits taken into account, while the columns the values of the 24 metrics considered in the study.

The target variable was set according to the outcome of the SZZ algorithm previously run. Hence, this was a binary variable in the set {"0", "1"}, where "0" indicates that a file was non-defective, "1" otherwise. It is worth remarking that the anomaly detection methods are unsupervised learning mechanisms and, as such, use the target variable *only* in the evaluation phase, i.e., only to assess how good they are in terms of classification performance.

The models were implemented using the `scikit-learn` library.[2] When setting their hyperparameters, we followed recent guidelines [83]: for *IsolationForest*, we used 100 trees as weak classifiers; for *OCSVM* we used the default parameters from the library; for *LOF* we used 20 as the number of neighbors parameter. The models were run on a server with an AMD Ryzen Threadripper PRO 3975WX with 32 cores and 252GB of RAM.

2.5 Setting the Machine Learning Models

To perform a fair comparison, we selected three machine learning techniques based on the same underlying algorithms used by the employed anomaly detection methods: we relied on *Support Vector Machine*, *Extremely Randomized Trees*, and *KNearestNeighbour*. By considering the supervised learning versions of the anomaly detection methods selected we were able to more precisely quantify the improvement (if any) given by anomaly detection to the problem of fine-grained just-in-time defect prediction. In a nutshell, these supervised learning algorithms can be described as follows:

Support Vector Machine (SVM) [16]. This statistical learner constructs the best hyper-plane out of the infinite possibilities, in a N-dimensional space (with N being the number of features) which can distinctly separate the data points. The best hyper-plane is the one having the maximum margin, which is the largest distance to the nearest training data points of any class).

Extremely Randomized Trees (ExtraTrees) [25]. It is based on *Random Forest* [9], but it adds another level of randomization. In particular, beside generating each weak classifier by randomly choosing a subsets of the features in the dataset, an *ExtraTrees* model randomizes also the way each node is split. In fact, instead of using the Gini impurity or the Information Gain to find the optimal cut-off for each node, this process is randomized. Hence, the *ExtraTrees* model is less computationally expensive than *Random Forest*, while retaining a higher generalization capability compared to the single decision trees.

KNearestNeighbour (KNN) [89]. This non-parametric model does not learn a generalized representation of the data, but rather computes the classification for each sample in the dataset. The classification is made as a majority vote: each sample is classified based on the class of the majority of its k nearest neighbours.

The machine learning classifiers used the same data as the one used for the anomaly detectors. In this case the binary variable representing the defectiveness of files within commits was used also during the training phase. The models were implemented using the `scikit-learn` library and run on the same server infrastructure. Their hyperparameters were tuned as the anomaly detection ones to ensure a fair comparison: for the *ExtraTrees* model we used 100 trees as weak classifiers; for *SVM* we used the default parameters from the library; for *KNN* we used 20 as the number of neighbors.

2.6 Data Analysis

After devising the objects of our empirical study, we defined the data analysis procedures to address our research question.

While assessing the performance of anomaly detection methods for fine-grained just-in-time defect prediction as well as their comparison with baseline machine learning models, we required to define a proper validation strategy. In this respect, we opted for a Leave One Group Out (LOGO) validation. As the name suggests, this validation method 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: in other words, we enforced the validation strategy to simulate a realistic use case scenario where we cannot predict the existence of defective files in a project by using information that might be connected to events that happened earlier in time. We further discuss this choice in Section 5.

More in general, the selection of the LOGO validation strategy was based on two observations. With respect to other strategies (e.g., Out-of-sample bootstrap validation), it is among the easiest to interpret [45], hence perfectly fitting the goal of our exploratory analysis: indeed, we aimed at establishing the feasibility of using anomaly detection for the problem of just-in-time defect prediction

[2]The `scikit-learn` library: https://scikit-learn.org/0.17/index.html.

and the adoption of this validation strategy allowed us to better understand its strengths and weaknesses. In the second place, the LOGO validation represents a good compromise between the bias and variance of estimates of defect prediction models [80], thus further strengthening its suitability for our case.

For each fold experimented during the validation, we assessed the capabilities of both anomaly detection and machine learning methods using a number of accuracy metrics. First, we computed precision and recall. However, as suggested by Powers [67], these two measures present some biases as they are mainly focused on positive examples and predictions and do not capture any information about the rates and kind of errors made. The contingency matrix (a.k.a. confusion matrix), and the related F-Measure overcome this issue. Moreover, we computed the Matthews Correlation Coefficient (MCC) to understand possible disagreement between actual values and predictions—the coefficient involves all the four quadrants of the contingency matrix. In addition, from the contingency matrix we retrieved the measure of *true negative rate* (TNR), which measures the percentage of negative sample correctly categorized as negative, *false positive rate* (FPR) which measures the percentage of negative sample misclassified as positive, and *false negative rate* (FNR), measuring the percentage of positive samples misclassified as negative. The measure of *true positive rate* is left out as equivalent to the recall.

Finally, we computed the Receiver Operating Characteristics (ROC), and the related Area Under the Receiver Operating Characteristic Curve (AUC). This gave us the probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one. In Section 3, we present and discuss the results using boxplots reporting the distribution of the accuracy metrics computed over the validated folds.

2.7 Verifiability and Replicability

To enable full verifiability and replicability, we made available in our online appendix[3] the complete raw data.

3 EMPIRICAL STUDY RESULTS

Table 1 shows descriptive statistics of the considered systems. As inferred, each commit contained an average of 2.95 files, and 1.01 turned out to be defective—this translates to 34%, overall.

In the context of our research question, we aimed at assessing the performance of anomaly detection methods for fine-grained just-in-time defect prediction when compared to the one of traditional machine learning approaches. Figures 2 and 3 depict box plots reporting the distribution of AUC-ROC and F-Measure values obtained by the experimented techniques over the considered dataset, respectively. In both figures, 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 these two performance metrics, while the full results are included in our replication package.

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.

[3] https://figshare.com/s/1459e02ef92c01d1a6b1

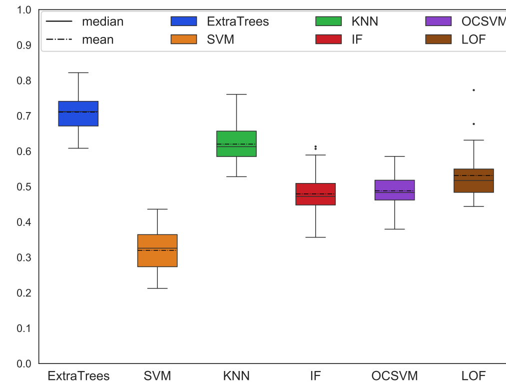

Figure 2: AUC comparison comparison among anomaly detection and supervised learning models for the filtered dataset

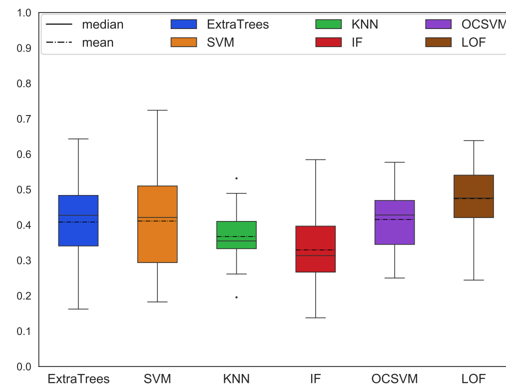

Figure 3: F-measure comparison comparison among anomaly detection and supervised learning models for the filtered dataset

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 [19, 82]: 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.

Table 1: Defect-inducing files distribution

Project	#Commits	Files (tot)	Files (max)	Files (average)	Defected		
					#	average	%
accumulo	5,236	16,240	21	3.10	7,748	1.48	48%
ambari	5,887	25,750	21	4.37	9,427	1.60	37%
atlas	1,623	6,761	21	4.17	3,175	1.96	47%
aurora	1,831	8,786	21	4.80	4,293	2.35	49%
batik	2,144	7,885	21	3.68	2,920	1.36	37%
cocoon	6,063	17,391	21	2.87	5,386	0.89	31%
commons-bcel	1,059	2,853	21	2.69	521	0.49	18%
commons-beanutils	787	2,043	20	2.60	694	0.88	34%
commons-cli	479	1,205	21	2.52	454	0.95	38%
commons-codec	1,276	2,749	20	2.15	639	0.50	23%
commons-collections	2,410	7,308	21	3.03	1,974	0.82	27%
commons-configuration	2,171	5,145	21	2.37	1,727	0.80	34%
commons-daemon	114	280	16	2.46	45	0.39	16%
commons-dbcp	1,381	3,341	21	2.42	1,436	1.04	43%
commons-dbutils	314	753	19	2.40	114	0.36	15%
commons-digester	1,307	2,631	21	2.01	571	0.44	22%
commons-exec	295	712	20	2.41	290	0.98	41%
commons-fileupload	481	1,018	18	2.12	282	0.59	28%
commons-io	1,836	4,024	21	2.19	1,220	0.66	30%
commons-jelly	916	3,052	21	3.33	837	0.91	27%
commons-jexl	1,051	2,837	21	2.70	1,201	1.14	42%
commons-jxpath	363	1,142	21	3.15	403	1.11	35%
commons-net	1,494	3267	20	2.19	687	0.46	21%
commons-ognl	418	755	21	1.81	247	0.59	33%
commons-validator	806	1,729	19	2.15	540	0.67	31%
commons-vfs	1,634	4,310	21	2.64	1,740	1.06	40%
felix	8,462	24,220	21	2.86	9,658	1.14	40%
httpcomponents-client	2,419	8,559	21	3.54	2,560	1.06	30%
httpcomponents-core	2,658	10,648	21	4.01	3,365	1.27	32%
mina-sshd	1,399	6,400	21	4.57	1,914	1.37	30%
santuario-java	1,485	5,171	21	3.48	2,177	1.47	42%
zookeeper	1,282	4,835	21	3.77	2,140	1.67	44%
Overall	**61,081**	**193,800**	-	**2.95**	**70,385**	**1.01**	-

A different discussion can be delineated when considering the F-Measure. Indeed, here we observed a much balanced situation where the anomaly detection methods have comparable results with respect to the baselines. In addition, the LOF method has a slightly higher median F-Measure than all the other experimented models. This metric measures the harmonic mean of precision and recall and provides a clearer indication of the true positive defective files that the techniques can detect. Based on the results achieved, it seems that LOF improves on the baselines and might find more defects. The Mann-Whitney test, however, did not reveal a statistical significance in the differences observed.

More in general, we observed that the F-Measure values are pretty low for all the approaches, as all of them do not go beyond a median 0.5. This indicates that the problem of identifying defective files within commits is still far from being solved. The other metrics considered—included in our online appendix—revealed similar results, with the anomaly detection methods better in some of them and worst in others.

By summing all up, we could conclude that anomaly detection methods behave similarly, if not worse, than traditional machine learning approaches. As such, we could not confirm the promising preliminary results achieved by researchers that applied anomaly detection to higher-level defect prediction problems [1, 74]. Nonetheless, this might be explained by the peculiarities of the dataset considered. As already noticed, there exist some variability in terms of defects: it is reasonable to believe that this variability may lead anomaly detection to work better on certain projects, i.e., those characterized by a significantly lower amount of defects, and worse in others, those having a high number of defective files. These observations are investigated in Section 4, in an effort of making further sense of the results achieved as well as a potential orthogonality between anomaly detection and machine learning models that might pave the way for novel, context-dependent techniques that may recommend which technique should be used based on the project to analyze.

Main results

When evaluating anomaly detection methods over the entire dataset, we observed that these perform similarly, if not worse than the baseline machine learners.

4 DISCUSSION AND FURTHER ANALYSES

While the results achieved on the full dataset revealed that anomaly detection methods rarely outperform the traditional approaches based on machine learning algorithms, it is worth combining these findings with the data on the distribution of defective files within commits. By definition, anomaly detectors are supposed to give their best in cases where a few amount of data follows anomalous patterns that strongly differ from a general, regular trend observable in a dataset. Given these characteristics, 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

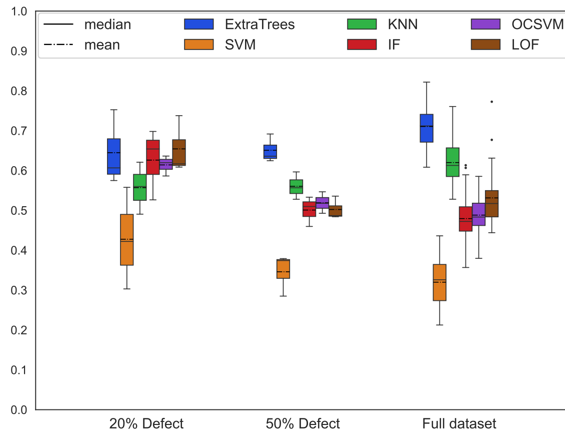

Figure 4: AUC-ROC comparison among anomaly detection and supervised learning models on the filtered datasets.

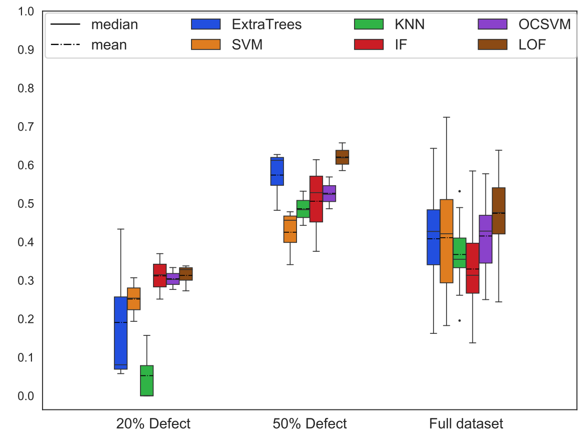

Figure 5: F-Measure comparison among anomaly detection and supervised learning models on the filtered datasets.

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% in the extreme case.

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.

The performance were assessed using the same indicators as our research question. Figures 4 and 5 depict the results: to ease their interpretability, the figures report the box plots of the AUC-ROC and F-Measure values, respectively, 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 3.

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. Also in this case we observed a balanced situation in terms of F-Measure, with LOF producing the best results, even though the differences with the baselines were not statistically significant. All in all, 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 [19, 85], 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.

Take-away message: There is a complementarity between anomaly detection and machine learning techniques. Such a complementarity might be potentially exploited to devise novel, context-dependent fine-grained just-in-time defect predictors.

5 THREATS TO VALIDITY

Construct Validity. Threats in this category might have been connected to the dataset exploited as well as the measurements performed to compute the additional data required. We first based our investigation on the *Technical Debt Dataset* [44]: while most of the systems come from the Apache Software Foundation ecosystem, these were selected using well-established guidelines that ensured representativeness [54]. In our work, we also additionally verified the suitability of the dataset by conducting manual analysis.

As for the set of factors employed to characterize files committed during the history of the considered projects, we were not interested in establishing new indicators: as such, we relied on a set of metrics that were previously used in the context of fine-grained just-in-time defect prediction [60]. To label defective files, instead, we adopted the SZZ algorithm [34]. We are aware that our results might have been affected by the erroneous identification, since the algorithm performance has been criticized in the last years [75]. Nonetheless, we adopted some precautions to mitigate the risk of false positives, e.g., we ignored non-source files belonging to defect-fixing commits. Furthermore, we employed the original version of the algorithms based on the recommendations recently provided [76].

Internal Validity. In our empirical study, we selected and experimented with a number of anomaly detection and machine learning models in an effort of better understanding their strengths and weaknesses. Of course, the setting up of those approaches might have biased our results. In so doing, we followed a well-established process through which we addressed possible issues due to multicollinearity and missing hyper-parameter configuration. We recognize, however, that other statistical or machine learning techniques might have yielded similar or even better performance. A larger analysis is part of our future research agenda.

Conclusion Validity. When interpreting the results of the different techniques experimented, we computed a number of evaluation metrics that could provide us with a broad overview of their performance. Perhaps more importantly, we considered the time-sensitivity of code commits and, for this reason, adopted the Leave One Group Out (LOGO) validation strategy. Its usage allowed us not to split commits coming from the same system over training and test sets. We are aware that different results might have been obtained if different validation strategies would have been implemented: in this sense, replications of our study would be desirable to increase the scientific understanding of anomaly detection for fine-grained just-in-time defect prediction.

External Validity. Our study analyzed a large number of open-source projects. While the large-scale nature of the study mitigated potential threats to the generalizability of the results, we still recognize that different results could be obtained on systems written in different programming languages and following different quality assurance mechanisms. To stimulate researchers to replicate our study in different contexts, we made publicly available all scripts and datasets used in our empirical investigation on the matter.

6 RELATED WORK

Software defect prediction is one of the most active research areas in software engineering. Researchers focused on several aspects, from the detection at change- or component-level until a lower granularity such as method or file [7, 40, 51, 68]. Many studies proposed different approaches for defect prediction. The most adopted approaches are based on supervised [26, 28, 35] and unsupervised models [23, 47]. These models consider features such as product (e.g., CK metrics [14]) or process features (e.g., entropy of the development process [29]). For a full overview of defect prediction research, the reader may refer to recent literature reviews [28, 46].

Only two papers approached defects as anomalies, applying anomaly detection techniques to predict them: these were based on univariate and multivariate Gaussian distribution [55] or supervised anomaly detection model [2].

Just-in-time defect prediction refers to a class of techniques that can anticipate the identification of defects at commit-time [22, 38, 86], rather than predicting defects using a long-term approach that, instead, shows prediction results at release-time [28, 84]. In the recent past, researchers pushed for a shorter-term analysis of defects since this better fits the developers' needs [62]: indeed, these models allow developers to promptly react when changes are still fresh in their minds [87]. For example, Mockus and Weiss [52] 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 [52]. Later on, Madeyski and Kawalerowicz [49] elaborated on *continuous* defect prediction, developing a public dataset containing tools for experimenting with defect prediction techniques.

To better characterize the prediction problem, researchers such as Sliwerski et al. [34], Eyolfson et al. [21], and Rahman and Devanbu [69] investigated the common causes that conceal behind a defective change. They all concluded by reporting correlations between the developer's experience and the defectiveness of changes. However, also other factors may come into play: Sliwerski et al. [34] and Rosen et al. [77] confirmed that commit size affects the defectiveness of changes, i.e., large commits that involve several files are more prone to be defective. Moreover, defective changes are characterized by higher entropy than non-defective ones [73].

To enlarge the body of knowledge on the properties characterizing defective changes, researchers searched for several other software properties, like structural [5, 14], historical [20, 27], and alternative [8, 59, 61] metrics. Despite the variety of the investigated software properties, the two top promising categories are still product and process metrics, as confirmed by Pascarella et al. [61].

Significant milestones for just-in-time defect prediction are represented by the works made by Kamei et al. [37, 38]. They proposed a just-in-time prediction model to predict whether or not a change will lead to a defect with the aim of reducing developers and reviewers' effort. In particular, they applied logistic regression considering different change measures such as diffusion, size, and purpose, obtaining an average accuracy of 68% and an average recall of 64%. More recently, Pascarella et al. [60] complemented their results considering the attributes necessary to filter only those files that are defect-prone. The reduced granularity is justified by the fact that 42% of defective commits are partially defective, i.e., composed of both files that are changed without introducing defects and files that are changed introducing defects. Furthermore, in almost 43% of the changed files a defect is introduced, while the remaining files are defect-free. Our work can be complementary with respect to

the papers on just-in-time defect prediction and its fine-grained version, as it aims at assessing how well can anomaly detection methods be used to identify defective files within commits.

As a final note, it is worth remarking that, despite other studies showed promising results in applying alternative techniques, such as cached history [72], deep learning [86], and textual analysis [61], they are not in our main focus. With this study we aim to evaluate the effectiveness of using anomaly detection as a defect prediction tool rather than improve current defect prediction techniques.

7 CONCLUSION

In this paper, we proposed an investigation into the capabilities of anomaly detection methods for fine-grained just-in-time defect prediction. We analyzed 32 open-source projects with a total amount of 61,081 commits that contain 70,385 defective files resulting in 36% of defectiveness, overall.

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.

Our future research agenda includes a larger-scale replication of the study but, more importantly, the investigation on intelligent approaches that can exploit contextual information to select the technique to use for predicting defects in a project. Finally, we aim at conducting an *in-vivo* replication to assess our preliminary results based on developer's feedback.

REFERENCES

[1] Petar Afric, Lucija Sikic, Adrian Satja Kurdija, and Marin Silic. 2020. REPD: Source code defect prediction as anomaly detection. *Journal of Systems and Software* 168 (2020), 110641.
[2] Petar Afric, Lucija Sikic, Adrian Satja Kurdija, and Marin Silic. 2020. REPD: Source code defect prediction as anomaly detection. *Journal of Systems and Software* 168 (2020), 110641.
[3] Shikha Agrawal and Jitendra Agrawal. 2015. Survey on anomaly detection using data mining techniques. *Procedia Computer Science* 60 (2015).
[4] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I Maletic. 2008. What's a typical commit? a characterization of open source software repositories. In *Int.conference on program comprehension*. 182–191.
[5] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. 1996. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. on Software Eng.* 22, 10 (1996), 751–761.
[6] Jennifer Bauer, Janet Siegmund, Norman Peitek, Johannes C Hofmeister, and Sven Apel. 2019. Indentation: simply a matter of style or support for program comprehension?. In *Int. Conference on Program Comprehension*. 154–164.
[7] N. Bettenburg, M. Nagappan, and A. E. Hassan. 2012. Think locally, act globally: Improving defect and effort prediction models. In *Working Conference on Mining Software Repositories*. 60–69.
[8] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 2011. Don't touch my code! Examining the effects of ownership on software quality. In *European conference on Foundations of software engineering*. 4–14.
[9] Leo Breiman. 2001. Random forests. *Machine learning* 45 (2001).
[10] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. 2000. LOF: identifying density-based local outliers. In *Int.conference on Management of data*. 93–104.
[11] Bora Caglayan, Ayse Bener, and Stefan Koch. 2009. Merits of using repository metrics in defect prediction for open source projects. In *Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*. 31–36.
[12] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. 2019. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software* 152 (2019), 165–181.
[13] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. *ACM computing surveys (CSUR)* 41, 3 (2009), 1–58.
[14] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Trans. on Software Eng.* 20 (1994).
[15] Jorge Colazo and Yulin Fang. 2009. Impact of license choice on open source software development activity. *Journal of the American Society for Information Science and Technology* 60, 5 (2009), 997–1011.
[16] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine learning* 20, 3 (1995), 273–297.
[17] Elder Vicente de Paulo Sobrinho, Andrea De Lucia, and Marcelo de Almeida Maia. 2018. A systematic literature review on bad smells—5 W's: which, when, what, who, where. *IEEE Trans. on Software Eng.* (2018).
[18] Dario Di Nucci, Fabio Palomba, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia. 2017. A developer centered bug prediction model. *IEEE Trans. on Software Eng.* 44, 1 (2017), 5–24.
[19] Dario Di Nucci, Fabio Palomba, Rocco Oliveto, and Andrea De Lucia. 2017. Dynamic selection of classifiers in bug prediction: An adaptive method. *Transactions on Emerging Topics in Computational Intelligence* 1, 3 (2017), 202–212.
[20] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2012. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering* 17, 4 (2012), 531–577.
[21] Jon Eyolfson, Lin Tan, and Patrick Lam. 2011. Do time of day and developer experience affect commit bugginess?. In *Working Conference on Mining Software Repositories*. 153–162.
[22] Yuanrui Fan, Xin Xia, Daniel Alencar Da Costa, David Lo, Ahmed E Hassan, and Shanping Li. 2019. The impact of changes mislabeled by szz on just-in-time defect prediction. *IEEE Trans. on Software Eng.* (2019).
[23] Wei Fu and Tim Menzies. 2017. Revisiting Unsupervised Learning for Defect Prediction. In *Joint Meeting on Foundations of Software Engineering*. 72–83.
[24] Kehan Gao, Taghi M Khoshgoftaar, Huanjing Wang, and Naeem Seliya. 2011. Choosing software metrics for defect prediction: an investigation on feature selection techniques. *Software: Practice and Experience* 41, 5 (2011), 579–606.
[25] Pierre Geurts, Damien Ernst, and Louis Wehenkel. 2006. Extremely randomized trees. *Machine Learning* 63, 1 (4 2006), 3–42.
[26] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. 2000. Predicting fault incidence using software change history. *IEEE Trans. on Software Eng.* 26, 7 (2000), 653–661.
[27] Todd L Graves, Alan F Karr, James S Marron, and Harvey Siy. 2000. Predicting fault incidence using software change history. *IEEE Trans. on Software Eng.* 26, 7 (2000), 653–661.
[28] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. 2012. A Systematic Literature Review on Fault Prediction Performance in Software Engineering. *IEEE Trans. on Software Eng.* 38 (2012).
[29] Ahmed E Hassan. 2009. Predicting faults using the complexity of code changes. In *Int.conference on software engineering*. 78–88.
[30] Lile P Hattori and Michele Lanza. 2008. On the nature of commits. In *Int.Conference on Automated Software Engineering-Workshops*. 63–71.
[31] Sarah Heckman and Laurie Williams. 2011. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology* 53, 4 (2011), 363–387.
[32] Kim Herzig, Sascha Just, and Andreas Zeller. 2016. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering* 21, 2 (2016).
[33] Seyedrebvar Hosseini, Burak Turhan, and Dimuthu Gunarathna. 2017. A systematic literature review and meta-analysis on cross project defect prediction. *IEEE Trans. on Software Eng.* 45, 2 (2017), 111–147.
[34] and A. Zeller J. Śliwerski, T. Zimmermann. 2005. When Do Changes Induce Fixes?. In *Int.Workshop on Mining Software Repositories*. 1–5.
[35] Xiao-Yuan Jing, Shi Ying, Zhi-Wu Zhang, Shan-Shan Wu, and Jin Liu. 2014. Dictionary Learning Based Software Defect Prediction. In *Int.Conference on Software Engineering*. 414–423.
[36] Marian Jureczko and Lech Madeyski. 2011. A review of process metrics in defect prediction studies. *Metody Informatyki Stosowanej* 5 (2011).
[37] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E Hassan. 2016. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering* 21 (2016).
[38] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2012. A large-scale empirical study of just-in-time quality assurance. *IEEE Trans. on Software Eng.* 39 (2012).
[39] Taghi M Khoshgoftaar, Kehan Gao, and Naeem Seliya. 2010. Attribute selection and imbalanced data: Problems in software defect prediction. In *Int.conference on tools with artificial intelligence*, Vol. 1. 137–144.
[40] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. 2011. Dealing with Noise in Defect Prediction. In *Int.Conference on Software Engineering*. 481–490.
[41] Sunghun Kim, Thomas Zimmermann, Kai Pan, E James Jr, et al. [n. d.]. Automatic identification of bug-introducing changes. In *Int.conference on automated software engineering (ASE'06)*. 81–90.
[42] Bartosz Krawczyk. 2016. Learning from imbalanced data: open challenges and future directions. *Progress in Artificial Intelligence* 5 (2016).
[43] Michele Lanza, Andrea Mocci, and Luca Ponzanelli. 2016. The tragedy of defect prediction, prince of empirical software engineering research. *IEEE Software* 33,

6 (2016), 102–105.

[44] Valentina Lenarduzzi, Nyyti Saarimäki, and Davide Taibi. 2019. The Technical Debt Dataset. In *Conference on PREdictive Models and data analycs In Software Engineering*. 2 – 11.

[45] Benjamin Letham, Cynthia Rudin, Tyler H McCormick, David Madigan, et al. 2015. Interpretable classifiers using rules and bayesian analysis: Building a better stroke prediction model. *Annals of Applied Statistics* 9 (2015).

[46] Ning Li, Martin Shepperd, and Yuchen Guo. 2020. A systematic review of unsupervised learning techniques for software defect prediction. *Information and Software Technology* 122 (2020), 106287.

[47] Weiwei Li, Wenzhou Zhang, Xiuyi Jia, and Zhiqiu Huang. 2020. Effort-Aware semi-Supervised just-in-Time defect prediction. *Information and Software Technology* 126 (2020), 106364.

[48] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. 2008. Isolation forest. In *Int.Conference on Data Mining*. 413–422.

[49] Lech Madeyski and Marcin Kawalerowicz. 2017. Continuous defect prediction: the idea and a related dataset. In *Int.Conference on Mining Software Repositories*. 515–518.

[50] Zaheed Mahmood, David Bowes, Peter CR Lane, and Tracy Hall. 2015. What is the impact of imbalance on software defect prediction performance?. In *Int.Conference on Predictive Models and Data Analytics in Software Engineering*. 1–4.

[51] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. 2010. Defect Prediction from Static Code Features: Current Results, Limitations, New Approaches. *Automated Software Engg.* (Dec. 2010), 375–407.

[52] Audris Mockus and David M Weiss. 2000. Predicting risk of software changes. *Bell Labs Technical Journal* 5, 2 (2000), 169–180.

[53] 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 *Int.conference on Software engineering*. 181–190.

[54] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. 2013. Diversity in software engineering research. In *Joint meeting on foundations of software engineering*. 466–476.

[55] Kamrun Nahar Neela, Syed Asif Ali, Amit Seal Ami, and Alim Ul Gias. 2017. Modeling Software Defects as Anomalies: A Case Study on Promise Repository. *JSW* 12, 10 (2017), 759–772.

[56] Edmilson Campos Neto, Daniel Alencar Da Costa, and Uirá Kulesza. 2018. The impact of refactoring changes on the szz algorithm: An empirical study. In *Int.Conference on Software Analysis, Evolution and Reengineering*. 380–390.

[57] Edmilson Campos Neto, Daniel Alencar da Costa, and Uirá Kulesza. 2019. Revisiting and improving szz implementations. In *Int.Symposium on Empirical Software Engineering and Measurement*. 1–12.

[58] Stefano Dalla Palma, Majid Mohammadi, Dario Di Nucci, and Damian A Tamburri. 2020. 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*. 31–36.

[59] Fabio Palomba, Marco Zanoni, Francesca Arcelli Fontana, Andrea De Lucia, and Rocco Oliveto. 2017. Toward a smell-aware bug prediction model. *IEEE Trans. on Software Eng.* 45, 2 (2017), 194–218.

[60] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. 2019. Fine-grained just-in-time defect prediction. *Journal of Systems and Software* 150 (2019), 22–36.

[61] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. 2020. On the performance of method-level bug prediction: A negative result. *Journal of Systems and Software* 161 (2020).

[62] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. 2018. Information needs in contemporary code review. *ACM on Human-Computer Interaction* 2, CSCW (2018), 1–27.

[63] Michael Patton. 2002. *Qualitative Evaluation and Research Methods*. Sage, Newbury Park.

[64] Fabiano Pecorelli and Dario Di Nucci. 2021. Adaptive selection of classifiers for bug prediction: A large-scale empirical analysis of its performances and a benchmark study. *Science of Computer Programming* 205 (2021).

[65] Fabiano Pecorelli, Dario Di Nucci, Coen De Roover, and Andrea De Lucia. 2019. On the role of data balancing for machine learning-based code smell detection. In *Int.workshop on machine learning techniques for software quality evaluation*. 19–24.

[66] Reinhold Plösch, Harald Gruber, Christian Körner, and Matthias Saft. 2010. A method for continuous code quality management using static analysis. In *Int.Conference on the Quality of Information and Communications Technology*. 370–375.

[67] D. M. W. Powers. 2011. Evaluation: From precision, recall and f-measure to roc., informedness, markedness & correlation. *Journal of Machine Learning Technologies* 2, 1 (2011), 37–63.

[68] Lutz Prechelt and Alexander Pepper. 2014. Why software repositories are not used for defect-insertion circumstance analysis more often: A case study. *Information and Software Technology* 56, 10 (2014).

[69] Foyzur Rahman and Premkumar Devanbu. 2011. Ownership, experience and defects: a fine-grained study of authorship. In *Int.Conference on Software Engineering*. 491–500.

[70] Foyzur Rahman and Premkumar Devanbu. 2013. How, and why, process metrics are better. In *Int.Conference on Software Engineering*. 432–441.

[71] Foyzur Rahman, Daryl Posnett, and Premkumar Devanbu. 2012. Recalling the" imprecision" of cross-project defect prediction. In *Int.Symposium on the Foundations of Software Engineering*. 1–11.

[72] Foyzur Rahman, Daryl Posnett, Abram Hindle, Earl Barr, and Premkumar Devanbu. 2011. BugCache for inspections: hit or miss?. In *European conference on Foundations of software engineering*. 322–331.

[73] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "naturalness" of buggy code. In *Int.Conference on Software Engineering*. 428–439.

[74] Daniel Rodriguez, Israel Herraiz, Rachel Harrison, Javier Dolado, and José C Riquelme. 2014. Preliminary comparison of techniques for dealing with imbalance in software defect prediction. In *Int.Conference on Evaluation and Assessment in Software Engineering*. 1–10.

[75] G. Rodríguez-Pérez, G. Robles, and J. González-Barahona. 2018. 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* 99 (2018), 164–176.

[76] Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. 2021. Evaluating SZZ Implementations Through a Developer-informed Oracle. In *Int.Conference on Software Engineering*. to appear.

[77] Christoffer Rosen, Ben Grawi, and Emad Shihab. 2015. Commit guru: analytics and risk prediction of software commits. In *Joint meeting on foundations of software engineering*. 966–969.

[78] Bernhard Schölkopf, John C Platt, John Shawe-Taylor, Alex J Smola, and Robert C Williamson. 2001. Estimating the support of a high-dimensional distribution. *Neural computation* 13, 7 (2001), 1443–1471.

[79] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. 2015. Online defect prediction for imbalanced data. In *Int.Conference on Software Engineering*, Vol. 2. IEEE, 99–108.

[80] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. An empirical comparison of model validation techniques for defect prediction models. *IEEE Trans. on Software Eng.* 43, 1 (2016), 1–18.

[81] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2018. The impact of automated parameter optimization on defect prediction models. *IEEE Trans. on Software Eng.* 45, 7 (2018), 683–711.

[82] Stephen W Thomas, Meiyappan Nagappan, Dorothea Blostein, and Ahmed E Hassan. 2013. The impact of classifier configuration and classifier combination on bug localization. *IEEE Trans. on Software Eng.* 39, 10 (2013), 1427–1443.

[83] Kim Phuc Tran, Truong Thu Huong, et al. 2017. Data driven hyperparameter optimization of one-class support vector machines for anomaly detection in wireless sensor networks. In *Int.Conference on Advanced Technologies for Communications (ATC)*. 6–10.

[84] Romi Satria Wahono. 2015. A systematic literature review of software defect prediction. *Journal of Software Engineering* 1 (2015).

[85] Xinli Yang, David Lo, Xin Xia, and Jianling Sun. 2017. TLEL: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software Technology* 87 (2017), 206–220.

[86] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep learning for just-in-time defect prediction. In *Int.Conference on Software Quality, Reliability and Security*. 17–26.

[87] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. 2016. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *Int.symposium on foundations of software engineering*. 157–168.

[88] Xiao Yu, Jin Liu, Zijiang Yang, Xiangyang Jia, Qi Ling, and Sizhe Ye. 2017. Learning from imbalanced data for predicting the number of software defects. In *Int.Symposium on Software Reliability Engineering (ISSRE)*. 78–89.

[89] Zhongheng Zhang. 2016. Introduction to machine learning: k-nearest neighbors. *Annals of translational medicine* 4, 11 (2016).

[90] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. 2009. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *European software engineering conference on The foundations of software engineering*. 91–100.

[91] Weiqin Zou, Jifeng Xuan, Xiaoyuan Xie, Zhenyu Chen, and Baowen Xu. 2019. How does code style inconsistency affect pull request integration? An exploratory study on 117 GitHub projects. *Empirical Software Engineering* 24, 6 (2019), 3871–3903.