Tampere University

FRANCESCO LOMIO

Machine Learning for Software Fault Detection

Issues and Possible Solutions

Tampere University Dissertations 613

Tampere University Dissertations 613

FRANCESCO LOMIO

Machine Learning for Software Fault Detection

Issues and Possible Solutions

ACADEMIC DISSERTATION
To be presented, with the permission of
the Faculty of Information Technology and Communication Sciences
of Tampere University,
for public discussion in the auditorium TB109
of Tietotalo, Korkeakoulunkatu 1, Tampere,
on 22nd June 2022, at 10.30.

ACADEMIC DISSERTATION
Tampere University,
Faculty of Information Technology and Communication Sciences
Finland

Responsible supervisor and Custos	Associate Professor Davide Taibi Tampere University Finland	
Pre-examiners	Professor Letizia Jaccheri Norwegian University of Science and Technology Norway	Professor Sandro Morasca University of Insubria Italy
Opponent	Professor Danilo Caivano University of Bari Italy	

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

Cover design: Roihu Inc.

ISBN 978-952-03-2430-8 (print)
ISBN 978-952-03-2431-5 (pdf)
ISSN 2489-9860 (print)
ISSN 2490-0028 (pdf)
http://urn.fi/URN:ISBN:978-952-03-2431-5

PunaMusta Oy – Yliopistopaino
Joensuu 2022

To my loved ones

ACKNOWLEDGEMENTS

The work presented in this thesis was carried out at Tampere University (Tampere University of Technology when the research first started), between the years 2018 and the year 2022. The first part of this work has been performed in the Machine Learning Group, where I first started my doctoral studies and where the basis for my research were laid, and it concluded in the CloudSEA group, where the works were conducted and the thesis took shape.

My greatest gratitude goes to my supervisor, Associate Professor Davide Taibi, for his excellent supervision, and for allowing me to transform what was supposed to be "just an application" into my main research topic. Similarly, my gratitude goes to Assistant Professor Valentina Lenarduzzi, for being one of my main coauthors, and for pushing me beyond my limits. Davide and Valentina have been my mentors from the moment I first started collaborating with them, teaching me all I currently know about this career and making me fall in love with my research. For all they have done, also on the personal level, I consider myself lucky to be able to call them friends. Grazie!

Thank you also to all the people who participated in one way or another in my academic career up to this moment, as without them this work would have not been possible.

I also want to thank all the members of my research group: thank you for making the moments in the office memorable, and for helping keep the morale high all the time. We used to jokingly call this group a family, and it feels exactly like it. Thank you!

My thanks also go to my former research group, the Machine Learning Group, its members, and my previous supervisor Heikki Huttunen. He was the first to give me a chance and allow me to start my doctoral studies. If it wasn't for him, I probably would not be writing my Ph.D. thesis today.

To all the people that I met during the past 5 years in Finland, thank you, as in

one way or another you have been able to make this experience a wonderful one, and for that, I will always be grateful. To Michelangelo and Sergio in particular, for all the support you have given me and for listening to me when I most needed it, thank you!

Thank you to my parents and my sister, for not really getting why I was coming all the way to Finland, but still always believing in me.

Lastly, but not less importantly, thank you Barbara for always being by my side and for keeping up with my ups and downs. If it wasn't for you I would never have been in this country, let alone pursuing a Ph.D., and for this, I will never thank you enough. All accomplishments I have made in the past years, would not have been possible without you, and are yours as much as they are mine. And I'm sure the ones in the future will be too.

Tampere, April 2022
Francesco Lomio

ABSTRACT

Over the past years, thanks to the availability of new technologies and advanced hardware, the research on artificial intelligence, more specifically machine and deep learning, has flourished. This newly found interest has led many researchers to start applying machine and deep learning techniques also in the field of software engineering, including in the domain of software quality.

In this thesis, we investigate the performance of machine learning models for the detection of software faults with a threefold purpose. First of all, we aim at establishing which are the most suitable models to use, secondly we aim at finding the common issues which prevent commonly used models from performing well in the detection of software faults. Finally, we propose possible solutions to these issues.

The analysis of the performance of the machine learning models highlighted two main issues: the unbalanced data, and the time dependency within the data. To address these issues, we tested multiple techniques: treating the faults as anomalies and artificially generating more samples for solving the unbalanced data problem; the use of deep learning models that take into account the history of each data sample to solve the time dependency issue.

We found that using oversampling techniques to balance the data, and using deep learning models specific for time series classification substantially improve the detection of software faults.

The results shed some light on the issues related to machine learning for the prediction of software faults. These results indicate a need to consider the time dependency of the data used in software quality, which needs more attention from researchers. Also, improving the detection performance of software faults could help the practitioners to improve the quality of their software.

In the future, more advanced deep learning models can be investigated. This includes the use of other metrics as predictors and the use of more advanced time series analysis tools for better taking into account the time dependency of the data.

TIIVISTELMÄ

Viime vuosina tekoälyn ja etenkin kone- ja syväoppimisen tutkimus on menestynyt osittain uusien teknologioiden ja laitteiston kehityksen vuoksi. Tutkimusalan uudelleen alkanut nousu on saanut monet tutkijat käyttämään kone- ja syväoppimismalleja sekä -tekniikoita ohjelmistotuotannon alalla, johon myös ohjelmiston laatu sisältyy.

Tässä väitöskirjassa tutkitaan ohjelmistovirheiden tunnistukseen tarkoitettujen koneoppimismallien suorituskykyä kolmelta kannalta. Ensin pyritään määrittämään parhaiten ongelmaan soveltuvat mallit. Toiseksi käytetyistä malleista etsitään ohjelmistovirheiden tunnistusta heikentäviä yhtäläisyyksiä. Lopuksi ehdotetaan mahdollisia ratkaisuja löydettyihin ongelmiin.

Koneoppimismallien suorituskyvyn analysointi paljasti kaksi pääongelmaa: datan epäsymmetrisyys ja aikariippuvuus. Näiden ratkaisemiseksi testattiin useita tekniikoita: ohjelmistovirheiden käsittely anomalioina, keinotekoisesti uusien näytteiden luominen datan epäsymmetrisyyden korjaamiseksi sekä jokaisen näytteen historian huomioivien syväoppimismallien kokeilu aikariippuvuusongelman ratkaisemiseksi.

Ohjelmistovirheet havaittiin merkittävästi paremmin käyttämällä dataa tasapainottavia ylinäytteistämistekniikoita sekä aikasarjaluokitteluun tarkoitettuja syväoppimismalleja. Tulokset tuovat selvyyttä ohjelmistovirheiden ennustamiseen koneoppimismenetelmillä liittyviin ongelmiin. Ne osoittavat, että ohjelmistojen laadun tarkkailussa käytettävän datan aikariippuvuus tulisi ottaa huomioon, mikä vaatii etenkin tutkijoiden huomiota. Lisäksi ohjelmistovirheiden tarkempi havaitseminen voisi auttaa ammatinharjoittajia parantamaan ohjelmistojen laatua.

Tulevaisuudessa tulisi tutkia kehittyneempien syväoppimismallien soveltamista. Tämä kattaa uusien metriikoiden sisällyttämisen ennustaviin malleihin, sekä kehittyneempien ja paremmin datan aikariippuvuuden huomioon ottavien aikasarjatyökalujen hyödyntämisen.

CONTENTS

List of Figures

List of Tables

ABBREVIATIONS

DL	Deep Learning
ExtraTrees	Extremely Randomized Trees
FCN	Fully Convolutional Network
IF	Isolation Forest
KNN	KNearestNeighbors
LOF	Local Outlier Factor
ML	Machine Learning
NVD	National Vulnerability Database
OCSVM	OneClass Support Vector Machine
ResNet	Residual Network
ROC-AUC	Area Under the Receiver Operating Characteristics
SAT	Static Analysis Tool
SMOTE	Synthetic Minority Oversampling Technique
SQ	SonarQube
SVM	Support Vector Machine
VIF	Variable Inflation Factor

ORIGINAL PUBLICATIONS

Publication I V. Lenarduzzi, F. Lomio, H. Huttunen, and D. Taibi, "Are sonarqube rules inducing bugs?", *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020. DOI: 10.1109/SANER48275.2020.9054821.

Publication II F. Lomio, E. Iannone, A. De Lucia, F. Palomba, and V. Lenarduzzi, "Just-in-time software vulnerability detection: Are we there yet?", *Journal of Systems and Software*, 2022. DOI: 10.1016/j.jss.2022.111283.

Publication III F. Lomio, D. M. Baselga, S. Moreschini, H. Huttunen, and D. Taibi, "Rare: A labeled dataset for cloud-native memory anomalies", *International Workshop on Machine-Learning Techniques for Software-Quality Evaluation*, 2020. DOI: 10.1145/3416505.3423560.

Publication IV F. Lomio, L. Pascarella, F. Palomba, and V. Lenarduzzi, "Regularity or anomaly? on the use of anomaly detection for fine-grained just-in-time defect prediction", 2022, Submitted.

Publication V N. Saarimäki, S. Moreschini, F. Lomio, R. Penaloza, and V. Lenarduzzi, "Towards a robust approach to analyze time-dependent data in software engineering", *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022, In press.

Publication VI F. Lomio, S. Moreschini, and V. Lenarduzzi, "A machine and deep learning analysis among sonarqube rules, product, and process metrics for faults prediction", *Empirical Software Engineering (EMSE)*, 2022, In press.

Author's contribution

The contribution of the author, Francesco Lomio, in each of the publication included in this thesis, is described below.

Publication I — The author took charge of planning, researching and executing the suitable techniques for the data analysis. He also took care of the interpretation of the results given by the machine learning models. The author participated in the writing process of the description of the data analysis part, outlining the pros and cons of the models used, as well as the implication in terms of results.

Publication II — The author participate in the overall conceptualization and planning of the study. The author took charge of planning, researching and executing the suitable techniques for the data analysis. He also took care of the interpretation of the results given by the machine learning models. The author participated in the overall writing process. The author is the lead author of the publication.

Publication III — The author participate in the overall conceptualization and planning of the study. The author took charge of planning, researching and executing the suitable techniques for the data analysis. The author participated in the overall writing process. The author is the lead author of the publication.

Publication IV — The author participate in the overall conceptualization and planning of the study. The author took charge of planning, researching and executing the suitable techniques for the data analysis. He also took care of the interpretation of the results given by the machine learning models. The author participated in the overall writing process. The author is the lead author of the publication.

Publication V — The author took charge of planning and researching the suitable techniques for the data analysis. He also took care of the conceptualization and planning of the study. The author participated in the writing process including but not limited to the discussion of the implication of the findings.

Publication VI The author participate in the overall conceptualization and planning of the study. The author took charge of planning, researching and executing the suitable techniques for the data analysis. He also took charge of the interpretation of the results given by the machine learning models. The author participated in the overall writing process. The author is the lead author of the publication.

1 INTRODUCTION

The software quality domain has been deeply investigated in the past decades, with new metrics and measures introduced and studied since the 60s. Traditionally, statistical techniques have been used for fault detection and prevention [1]. Although a conspicuous number of techniques and methodologies have been used for the detection of faults, it is still not clear which are the most suitable for software fault detection. Over the past years, partially thanks to the availability of new technologies and advanced hardware, the research on artificial intelligence, more specifically machine and deep learning, has flourished. These novel advancements in these techniques have found application in the most variety of fields, spanning from computer vision and image recognition, to data analysis and time series analysis.

As expected, many researchers started to apply machine learning and deep learning models and techniques also in the field of software engineering. Following this trend, we decided to investigate which are the techniques best suited for the software faults detection problem, in order to shed some light on which are the issues linked to the use of machine learning techniques for the detection of faults, and how these issues can be addressed.

More recently, in fact, there has been an increase in research interest on how to apply machine learning models to different aspects of software quality, for example on the detection of code smells [2]–[5], or software vulnerabilities [6]–[10]. Similarly, the interest in how to improve the prediction of faults has increased, for example, the granularity at which analyzing the software [11] and the type of features and metrics to calculate and take into account[12]–[14].

Besides the increase of interest in machine learning techniques, there has also been further interest in the application of more advanced, deep learning models. Multiple works have in fact started to investigate how deep learning models could benefit the prediction of software faults, with some initial promising results [15].

Although many works are studying the use of these newer models and techniques

for the purpose of software quality, there is still no generic answer on what are the techniques that are proven to correctly predict faults in software. Similarly, there is no clear answer on what are the issues related to the prediction of faults using machine and deep learning techniques and how these can be solved. As a matter of fact, most of the recent research on this topic focuses on trying different approaches for the analysis of software quality, being these the use of different techniques, or the use of different metrics.

For this reason, in this thesis, we aim at providing some additional insights on which models are suitable for the detection of faults, and more specifically what are the solution to some of the issues related to the use of machine learning models for software fault detection.

1.1 Goal and Research Questions

The goal of the thesis is to shed some light on the use of machine learning techniques for software fault detection. More specifically we aim at finding what are the machine learning techniques most suitable for the software faults detection and how their performance can be improved by solving the issues linked to the detection of faults. We have in fact seen that although many methodologies have been investigated for detecting software faults, there indeed are some issues that prevent them from properly detect faults. In this thesis we aim at finding those issues and propose some viable solutions to improve the software fault detection problem. Following are the research questions we are trying to answer to.

RQ1 - What machine learning techniques can be adopted for software fault detection?

The first research question aims at finding a set of machine learning models that are able to properly predict software faults. Given the multitude of machine learning models and techniques available today, it becomes increasingly difficult to discriminate which could bring an added value to the detection of software faults. One way to solve this is to train and test multiple machine learning models, based on commonly used underlying techniques (e.g., tree-based, boosting techniques), and verify how these perform in the fault detection task. By analyzing the performance of multiple models, we will also be able to discriminate what eventually the issues are and gain some perspective on how to improve the detection.

RQ2 - What issues prevent the proper use of machine learning for software fault detection?

The second research question we are trying to answer to relates to the identification of additional issues with fault detection in the case of machine learning models, related to external causes. By external causes, we mean issues related not to the models themselves, but mainly to the data available. After answering the first research question, we are in fact confident as to which machine learning models perform best for software fault detection. We are also aware though that these results will not be perfect and that the data plays a huge role in the performance of machine learners. For this reason, we expected to identify which of these issues are and outline a way to approach them.

RQ3 - How can the issues linked to fault detection be solved?

The last research question is on finding solutions to the software fault detection issues found in the previous stages of this research work. For this reason, we will implement multiple solutions trying to tackle the issues from different angles and analyze which of the proposed solution better solves the fault detection issues. After having selected and analyzed suitable solutions, we summarize the finding trying to answer what is the best combination of techniques to use for software fault detection.

1.2 Thesis Contribution

The research that brought to this thesis involved multiple areas of software quality, and multiple techniques from the field of machine and deep learning. More specifically, this thesis contributes to increasing the research body on the use of machine and deep learning models for the prediction of software faults, with a minor contribution also to the detection of software vulnerabilities.

In this thesis work, in fact, besides defining what are the issues to take into account when using machine and deep learning models, we also thoroughly investigate some of the solutions that could be adopted, finding what are the ones that better improve the detection of the faults.

1.3 Thesis Structure

The remainder of this thesis is structured as follows. In Chapter 2, we present the background of the concept and techniques analyzed throughout the thesis, including the review of the state-of-the-art in the field of software fault detection, and the application of machine learning. In Chapter 3, we illustrate the methodology and rationale behind the multiple steps that led this work from the first analysis of the machine learning application to software fault prediction to the study and use of more complex techniques to tackle the problems linked with the fault detection. Chapter 4 describes the results obtained in the study of machine learning for software fault detection. These results are then thoroughly discussed and linked to the research questions in Chapter 5. Finally, Chapter 6 concludes the thesis. Besides these chapters, the peer-reviewed works that form the backbone of this work are reprinted and attached at the end of this thesis.

2 BACKGROUND AND RELATED WORKS

In this chapter, we will explore a detailed background of the concepts studied during this thesis work, alongside relevant literature already available on the topic and that has served as the basis for this work. More specifically, in Section 2.1 we will provide a definition of software quality and how it can be measured. We will also introduce the background of the research in the field of software faults prediction and software vulnerability prediction, in order to provide insights on what has already been studied. In Section 2.2, on the other hand, we will describe more in detail the techniques used throughout this thesis.

2.1 Software Quality

According to the ISO25010 quality model, a system quality is the *"degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value"* [16]. The quality of the software is related to both its static and dynamic properties, and it is divided according to the categorization in Figure 2.1. During this thesis, I mainly focused on the subcategories of Software Reliability and Software Security. More specifically, the focus was put on the detection of software faults.

A fault is defined as an abnormal condition, which causes the software to not performed as intended. A fault causes the software to perform unreliably. It is important to distinguish a fault from a defect. The latter is usually intended for something unexpected found after the software goes into production, deviating from the requirements. A fault is commonly used interchangeably with the term *bug*, even though this usually refers to any error in the software which can result in a failure.

2.1.1 Software Reliability

By reliability, it is indicated the ability of a system to perform specified functions under specified conditions for a specified amount of time. For a software to be

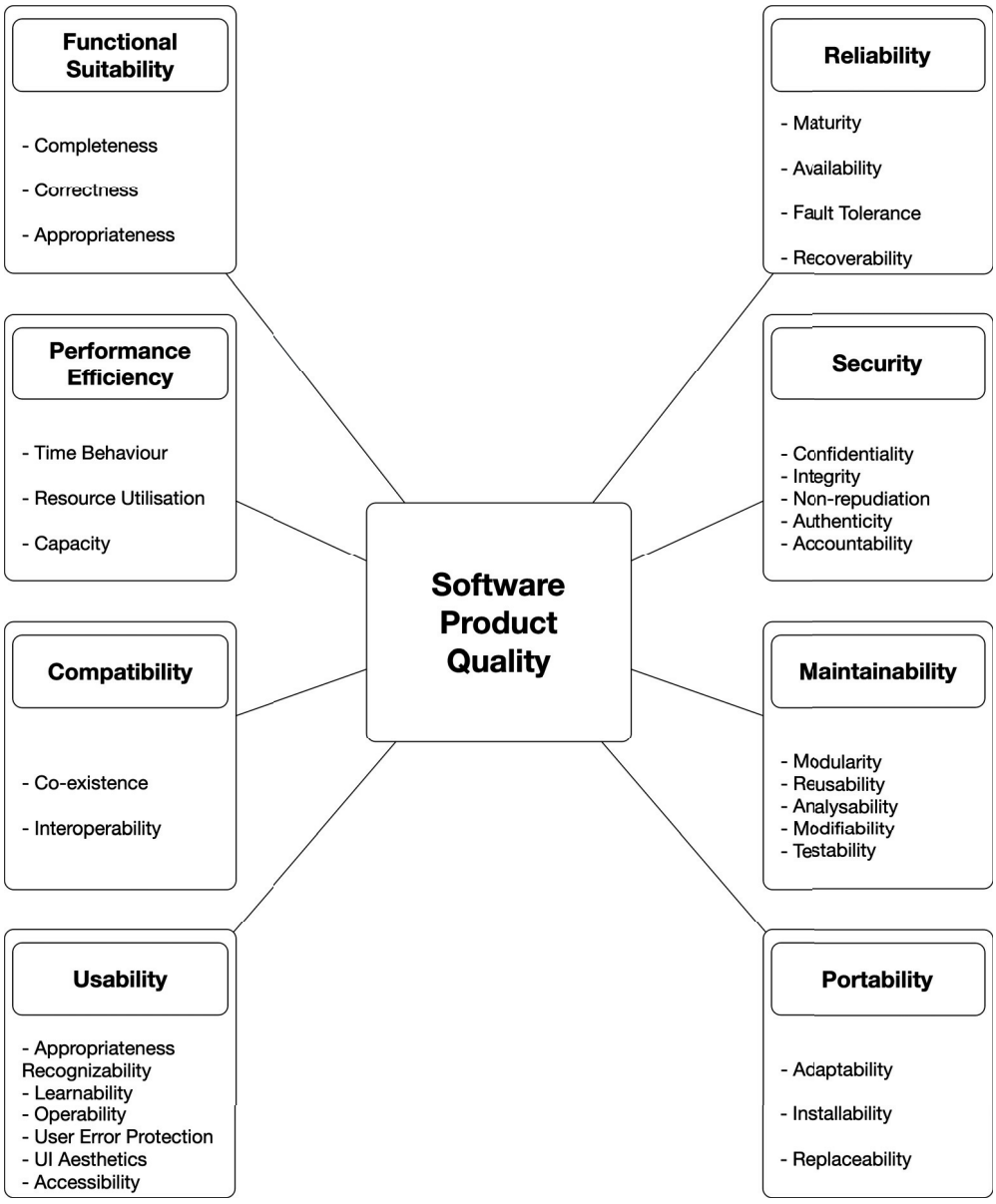


Figure 2.1 ISO25010 Software product quality categorization

reliable, it is essential to minimize the number of faults occurring, and in case it is not possible, to predict the fault in order to solve potential future problems.

Software fault prediction has been profoundly investigated in the past years, with many researchers focusing mainly on the improvement of the granularity of the [11], for instance, method or file [17]–[20], on the inclusion of features, e.g., code review [12], change context [21], as well as on the application of machine and deep learning techniques [22], [23].

In more recent years, the research shifted towards investigating principally the analysis of shorter-term defects as this fits, in fact, better the developers' needs [24]. Also, developers can immediately spot defects in the code by using short-term approaches [25].

A significant advancement in the research of just-in-time defect prediction is given by the works of Kamei et al. [26], [27]. In these works, the authors proposed a just-in-time prediction model that is able to predict if a change would lead to a defect. The aim of this is to reduce the effort needed from the developers and reviewers. Notably, they used a logistic regression model, which takes into account multiple change measures (e.g., diffusion, size, and purpose), and obtained an accuracy of $\sim$ 68% and a recall of $\sim$ 64%. Pascarella et al. [11], more recently, complemented the results filtering only the files which are defect-prone and not the whole commit. This reduction in granularity could be justified since 42% of defective commits included both files that are changed without the introduction of defects and files that are changed introducing defects. Further results showed that almost 43% of the files that were changed introduced a defect, while the remaining were defect-free.

2.1.1.1 Important features for fault prediction

Regarding the factors used to predict bug-inducing changes, some researchers adopted change-based metrics [12], which include size [13], history of a change, the experience of the developer [13], or churn metrics [14]. A further study included the code review metrics as predictive feature [12]. Among the aspects investigated, one which is noteworthy is the decreasing effort needed for the diagnoses of a defect [11]. Many other software properties were included by the researcher, as for example structural properties [28], [29], historical properties [30], [31], and alternative metrics [32]–[34]. Among the factors considered to detail the software properties, the product and process metrics are among the most promising [33].

Other factors included are static analysis warnings [35], [36]. These were used to build just-in-time defect prediction models. The results of these works showed that the warnings could indeed improve the models' accuracy [35]. Also, among the results, it was shown that code metrics, as well as static analysis warnings, are correlated with bugs. Hence they both can improve the prediction accuracy [36].

Regarding the type of models adopted, most of them are supervised [37]–[39] and unsupervised models [40], [41]. These models take into account features like product (e.g., CK metrics [29]) or process metrics (e.g., entropy of the development process [42]).

2.1.1.2 Machine learning for software reliability

Machine learning models have been used to investigate fault prediction. The focus was on the role of the features (e.g., change size or changes history), which are used to represent a code change. These were used as predictors [11], [13], [43].

Machine learning models were also extensively applied in the detection of technical issues in the code, as in the case of code smells [2]–[5]. Although machine learning techniques have been used for the detection and classification of different code smell types [44], [45], few studies used machine learning techniques to examine static analysis tool rules, like SonarQube [23], [46], [47] or PMD [48].

Regarding defect prediction, Yang et al. [49] proposed a novel approach TLEL based on an ensemble learning technique. A bagging classifier was used in the inner layer as a basis for a Random Forest in the outer layer.

Other machine learning models were used to detect different type of code smell [2], evaluate their harmfulness [2], define their intensity [50], and classify them based on their perceived criticality [4]. Also, machine learning techniques were successfully used to classify code smells using different software metrics [51].

Among the factors that could influence the accuracy of machine learning models, we find the selection of training data [3]. The code smells in a dataset represent is, in fact, the minority class, with very few samples [52].

2.1.1.3 Static analysis tools for software faults detection

Concerning static analysis tool rules detection, SonarQube was the most investigated tool, with multiple works that focused on the link between the presence of its

rules and the fault-proneness [23], [46] or the change-proneness [47].

In all these cases, machine learning techniques were applied, and the results showed that 20% of faults would have been avoidable if the SonarQube-related issues had been solved [46]. Another interesting finding was that the real harmfulness of the SonarQube rules is low [23]. Other positive results were found on the application of SonarQube regarding the class change-proneness [47].

Machine learning models were also used to specify whether the SonarQube technical debt could be predicted using software metrics [53]. The results showed the impossibility of having valuable predictions. Another point of view that has benefited from the use of machine learning was the evaluation of the remediation effort calculated by SonarQube [54], [55]. The results of these works highlighted that the models overestimated the time needed to fix the Technical Debt-related issues.

2.1.1.4 Deep Learning for Software Reliability

One of the techniques that have become increasingly studied in the last years is Deep Learning. This has become popular in many domains [56] including computer vision [57] and natural language processing [58]. Also in software engineering, many works have recently adopted deep learning techniques [59]–[63]. Thanks to the promising results, Deep Learning models could be an exciting approach in software faults prediction. It could, in fact, improve the performance of just-in-time defect prediction.

The author in [15], for example, used deep learning to improve the logistic regression weaknesses when combining features to generate new ones. They considered 14 traditional change level features in order to predict bugs.

Many works are still investigating whether deep learning models could have better performance compared to machine learning models, in the task of just-in-time fault prediction [15], [64]–[66]. These studies showed promising results in the bug prediction accuracy compared with other approaches (32.22% increased in the number of bugs detected) [15]. The benefits could be seen especially for small datasets and in the feature selection [64]. Also, the prediction of the presence of bugs in classes from static source code metrics [65] benefitted from the use of deep learning.

Among the deep learning architecture used, some of the most adopted are the Long Short Term Memory (LSTM) [67], the Convolutional Neural Network (CNN) [68] and the Deep Belief Network [66].

2.1.2 Software Security

Besides the application of predictive models for software fault detection, also in the field of software security there has been a number of research work investigating the use of models for predicting software vulnerabilities. These works mainly focused on the identification of the best metrics and features to use for establishing the presence of vulnerabilities.

Many of the works involved the use of either structural metrics [29], or product metrics calculated on the source code(e.g., Lines of Code). Notably, metrics indicating complexity(e.g., Cyclomatic Complexity [69]) received particular attention. Shin et al. [6], [70], [71], regarding MOZILLA FIREFOX, demonstrated a strong positive correlation between the proneness to the vulnerabilities of a file, and the number of decisions present. Specifically, the prediction models—built using as predictors the complexity metrics—achieved greater precision when the predictions were restricted only to the most vulnerable files, suggesting that when the files are subject to many vulnerabilities, they have high complexity values. This result was also confirmed in other studies [10], [72]–[74]. Similarly, other studies demonstrated that the vulnerabilities are positively correlated with coupling and negatively correlated with cohesion metrics. This confirmed that code with poor quality, raises the risk of flaws introduction [72]. Neuhaus et al. [75] found a positive correlation between the vulnerabilities in C functions and the number of imports. This suggested that they could be useful in a vulnerability prediction model. To confirm this, they used a Support Vector Machine (SVM), trained on the number of past vulnerabilities on the imported C files in the context of MOZILLA FIREFOX. This approach led to a precision of 70%, reducing the recall to 45%. Nguyen and Tran [76], extracted a set of metrics from the Component Dependency Graphs (CDG) for the prediction of vulnerabilities in files written in C++ in JS ENGINE of FIREFOX. This led to improving the recall and overall accuracy when compared to other models built using only complexity metrics. Scandariato et al. [77] used a *bag-of-words* method [78], [79] for the extraction of the most frequent terms (i.e., words), being the first to investigate textural features. The bag-of-word was extracted from JAVA files and it was used for predicting vulnerabilities on 20 ANDROID apps. This approach showed a high prediction performance for within-projects (i.e., the predictions were made on files belonging to the same projects used for the training of the model), but did

not succeed for cross-project analysis (i.e., the prediction is made on files belonging to projects which were excluded from the training). This was further confirmed by Walden et al. [74]. Zhang et al. [9] further combined product metrics with the bag-of-words method. This allowed them to achieve a better F-measure compared to the prediction models used in [74]. Zimmermann et al. [73] considered yet other information, namely how vulnerabilities in Windows Vista are related to code churn (i.e., the rate of changes applied to binaries) and organizational metrics (e.g., the number of developers). This resulted in high precision but lower recall, similarly to the findings of further studies [6], [7], [10], [15]. Smith and Williams [80] tested two prediction models WordPress and WakkaWiki, using as predictors the warnings of possible SQL Injections. They found a positive correlation existing with many vulnerability types. Theisen and Williams [81] put together all the findings above in their study, which allowed them to identify the models using a combination of the metrics to be the best performing.

Regarding the selection of the models to use for vulnerability detection, a set of different machine learning models have been used, namely Naïve Bayes [6], [8], [9], [76], [77], [81], Support Vector Machines (SVM) [7], [8], [73], [75]–[77], [82], Decision Trees [6], [9], [10], [77], [81], and Random Forests [6], [8]–[10], [74], [77], [81]. Among these, Random Forests reached higher precision in multiple contexts. On the other hand, Naïve Bayes achieved the highest recall values (fewer false negative rate). Similar models have been used in similar tasks, like exploitability [83] and defect prediction [71], [84].

Many of the studies presented focused on the prediction of vulnerabilities at the source code file level [6], [9], [10], [70], [72], [74], [76], [77], [80]. The prediction models, therefore, suggest if a specific file is vulnerable or not. In these cases, the effort of the developers can be invested in the inspection and testing of the vulnerable files. A similar concept is applied for prediction models focusing on binary files [8], [73], [81], [82]. These use machine code produced by a compiler. Sultana et al. [85] described a predicition model that works on Java, while Neuhaus et al. [75] developed a tool named Vulture, that predicts the vulnerabilities on C/C++ functions. Perl et al. [7] showed a method that allows obtaining the commit that contributed to the vulnerability using 66 C/C++ open-source projects. For this task, they relied on the `git blame` command to find the commits that last modified the lines deleted in a commit that fixed a known vulnerability. The most blamed commits were therefore

labeled as a vulnerability-contributing commits. Finally, a Support Vector Machine model was used on the dataset, achieving greater accuracy than equivalent static analysis tools.

2.1.3 Static Analysis

Static analysis indicates the analysis of software performed without executing the code. The static analysis of software is usually performed using automatic tools, which allow us to detect potential source code quality issues. Among the most adopted static analysis tools, we find Sonarqube, Checkstyle, Findbugs, and PMD. In this thesis and the related cases studied, we focused our attention on SonarQube, which is among the most commonly adopted open-source static code analysis tools both in academia [1], [86] and in industry [87]. SonarQube can be used either via the online sonarcloud.io platform or as a local execution on a private server.

Among the capabilities of SonarQube, we found the ability to calculate multiple metrics, like the code complexity and the number of lines of code. It also verifies the compliance of the code with respect to a specific set of "coding rules", which are defined for most of the common development languages. When a coding rule is violated by the source code, or when a rule exceeds a predefined threshold, SonarQube records an "issue". Among the rules included in Sonarqube, we find Reliability, Maintainability, and Security rules.

Reliability rules called "bugs" in SonarQube, generate issues (code violations) that "represent something wrong in the code" which will likely be reflected in a bug. We also find "Code smells", defined as "maintainability-related issues" which decrease the readability of the code and its modifiability. It is important to note that the "code smells" defined in SonarQube do not represent the more commonly known code smells defined by Fowler et al. [88]. They represent in fact a separate set of rules. More specifically, Fowler et al. [88] consider code smells as "surface indication that usually corresponds to a deeper problem in the system" as they can indicate different problems (e.g., bugs, maintenance effort, and code readability). The code smells defined by SonarQube, on the other hand, only refer to maintenance issues. Also, only 4 out of the 22 smells proposed by Fowler et al. are considered in the "Code Smells" rules by SonarQube (Duplicated Code, Long Method, Large Class, and Long Parameter List). It is also important to notice that SonarQube classifies the rules into five *severity* levels: Blocker, Critical, Major, Minor, and Info.

2.1.4 Software Metrics

In order to analyze a software and assess its reliability and security, it is necessary to record some measurements. These measures can be used to evaluate the overall quality of software and how new tools and methods affect it. In general, the metrics that can be collected for the assessment of the quality of the software can be either related to the *process* of the software or related to the *product* itself. More specifically, *process metrics* are the ones that allow us to evaluate the development process. On the other hand, *product metrics* are the metrics used to quantify the internal attributes of a software. According to the book Software Engineering [89], product metrics can be either dynamic or static. The first can be used to measure the efficiency and reliability of the system. The second ones can be used to measure the "complexity, understandability, and maintainability" of the software. In this thesis, we have used a different set of metrics as variables used to predict software faults. More specifically, we used the metrics calculated by SonarQube shown in Table 2.1, the product and process metrics showed in Table 2.3, and the metrics proposed by Rahman et al. [90] and Kamei et al. [26] showed in Table 2.2.

Table 2.1 SonarQube Software Metrics (PUBLICATION VI)

Metric	Description
	Size
NC	Number of classes (including nested classes, interfaces, enums and annotations).
NF	Number of files.
LL	Number of physical lines (number of carriage returns).
NCLOC	Also known as Effective Lines of Code (eLOC). Number of physical lines that contain at least one character which is neither a whitespace nor a tabulation nor part of a comment.
NCI	Number of Java classes and Java interfaces
MPI	Missing package-info.java file (used to generate package-level documentation)
P	Number of packages
STT	Number of statements.
NOF	Number of functions. Depending on the language, a function is either a function or a method or a paragraph.
NOC	Number of lines containing either comment or commented-out code. Non-significant comment lines (empty comment lines, comment lines containing only special characters, etc.) do not increase the number of comment lines."
NOCD	Density of comment lines = Comment lines / (Lines of code + Comment lines) * 100
	Complexity
COM	It is the Cyclomatic Complexity calculated based on the number of paths through the code. Whenever the control flow of a function splits, the complexity counter gets incremented by one. Each function has a minimum complexity of 1. This calculation varies slightly by language because keywords and functionalities do.
CCOM	Complexity average by class
FC	Complexity average by method
COGC	How hard it is to understand the code's control flow.
PDC	Number of package dependency cycles
	Test coverage
COV	It is a mix of Line coverage and Condition coverage. Its goal is to provide an even more accurate answer to the following question: How much of the source code has been covered by the unit tests?
LTC	Number of lines of code which could be covered by unit tests (for example, blank lines or full comments lines are not considered as lines to cover).
LC	On a given line of code, Line coverage simply answers the following question: Has this line of code been executed during the execution of the unit tests?
UL	Number of lines of code which are not covered by unit tests.
	Duplication
DL	Number of lines involved in duplications
DB	Number of duplicated blocks of lines.
DF	Number of files involved in duplications.
DLD	= (duplicated lines ÷ lines) * 100

Table 2.2 Rahman [90] and Kamei [26] Software Metrics (PUBLICATION VI)

	Metric	Description
Rahman et al. [90]	COMM	The cumulative number of changes.
	ADEV	The cumulative number of active developers.
	DDEV	The cumulative number of distinct developers .
	ADD	The normalized number of lines of code added.
	DEL	The normalized number of lines of code removed.
	OWN	The value indicating whether the owner of the file does the commit.
	MINOR	The number of contributors who contributed less than 5% .
	SCTR	The number of packages modified by the committer.
	NADEV	The number of active developers who changed any of the files involved in the commits where the given file has been modified.
	NDDEV	The number of distinct developers who changed any of the files involved in the commits where the given file has been modified.
	NCOMM	The number of commits where the given file has been involved.
	NSCTR	The number of different packages touched by the developer.
	OEXP	The percentage of code lines authored by a given developer in the whole project.
	EXP	The mean of the experience of all developers across the whole project.
Kamei et al. [26]	ND	The number of directories involved.
	ENTROPY	The distribution of the modified code across each given file.
	LA	Ten number of lines of code added to the given file (absolute number of the ADD metric).
	LD	The number of lines of code removed from the given file (absolute number of the DEL metric).
	LT	The number of lines of code in the given file in the considered commit before the change.
	AGE	The average time span between the last and the current change.
	NUC	The number of times the file has been modified alone up to considered commit.
	CEXP	The number of commits performed on the given file by the committer up to the considered commit.
	REXP	The number of commits performed on the given file by the committer in the last month.
	SEXP	The number of commits performed by a given developer in the considered package that contains the given file.

Table 2.3 Product and Process Software Metrics (PUBLICATION II)

Metric	Description
Added Lines	Number of lines added
Deleted Lines	Number of lines removed
Added Methods	Number of added functions/methods
Deleted Methods	Number of deleted functions/methods
Modified Methods	Number of modified functions/methods
Added Conditions	Number of added conditional expressions
Removed Conditions	Number of removed conditional expressions
Added Method Calls	The number of added function or method call in the commit.
Removed Method Calls	Number of removed function or method call
Added Assignments	Number of added assignments
Removed Assignments	Number of removed assignments
Mean Days Since Creation	Mean number of days from the creation of each modified file
Mean of Past Changes	Mean number of previous changes
Past Different Authors	Number of distinct authors that modified the files.
Author Past Contributions	Number of commits done by the author
Author Past Contributions Ratio	*Author Past Contributions* divided by the total number of commits made
Author 30-days Past Contributions	Number of commits done by the author in past 30 days
Author 30-days Past Contributions Ratio	*Author 30-days Past Contributions* divided by the total number of commits made 30 days before the commit date
Author Workload	Amount of work that an author has invested in a 30-days time window
Days After Creation	Number of days from the project's repository creation
Fix	If the goal of the commit was fix an issue or a defect
Touched Files	Number of files modified in the commit
Entropy of Changes	Distribution of changes across each modified file
Number of Hunks	Number of continuous blocks of changes
LOC	Lines of Code, including comment lines
SLOC	Source Lines of Cod (LOC excluding comment)
WMC	Weighted Methods per Class
CBO	Coupling Between Object
RFC	Response For a Class, i.e., the number of methods that can potentially be called by other classes
DIT	Depth of Inheritance
NOC	Number of Children
LCOM1	Lack of Cohesion of Methods version 1
LCOM2	Lack of Cohesion of Methods version 2
Files Term(s) Frequency	The count of each word appearing in the modified JAVA files
Patches Term(s) Frequency	Number of times in which the words appearing in the patches were changed (added or removed)

2.2 Techniques

In this section we introduce the techniques used for this thesis work, introducing the background of the machine and deep learning models used, alongside the other statistical analysis tools.

2.2.1 Machine Learning

In this section, we will describe the Machine Learning techniques adopted in this work and the motivations for adopting them. Since in this thesis work we are trying to detect faults in software, all the models used are specifically made for classification. For this purpose, we compared multiple machine learning models throughout this work. Among these, we used a generalized linear model: Logistic Regression [91], tree based classifier: Decision Tree [92], *ensemble classifiers*: Bagging [93], Random Forest [94], Extremely Randomized Trees [95], AdaBoost [96], Gradient Boosting [97] and XGBoost [98], an optimized implementation of Gradient Boosting. We also include classical models like Support Vector Machine (SVM) [99], and KNearestNeighbors (KNN) [100].

The use of multiple machine learning models is justified by the fact that each of these models has different performances despite the data is identical. Besides the difference in underlying techniques, this is also due to their bias and variance. The bias represents the attention given by the model to the training data - the higher the bias, the less the attention. The variance, on the other hand, does the opposite - a higher variance corresponds to greater attention to the training data, which in turn results in the model overfitting the data. This leads to poor generalization capabilities. For this reason, to find what is a suitable model for the detection of software faults, we need to analyze the performance of multiple models to find the best trade-off between bias and variance [101].

Following is an in-depth description of the machine learning models used.

Logistic Regression This is one of the simplest models in Machine Learning. Compared to linear regression, which gives as output a numerical value, the Logistic Regression is used to predict the category in which a sample belongs. More specifically, a binary Logistic Regression model calculates what is the probability of a data sample belonging to either of the two classes (0 or 1). This is done by using

independent variables. Once the probabilities are known, these are used to specify which is the class to which the data sample is more likely to belong. Simlarly to the other linear classifiers, Logistic Regression projects the P-dimensional input $\mathbf{x}$ into a scalar using the dot product of the learned weight vector $\mathbf{w}$ and the input sample: $\mathbf{w} \cdot \mathbf{x} + w_0$, where $w_0 \in \mathbb{R}$ is the intercept. To have a result interpretable as a class membership probability—a number between 0 and 1— the projected scalar is passed by the Logistic Regression through the logistic function (sigmoid). The sigmoid function returns an output value between 0 and 1, for any given input x . The logistic function is defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

Finally, the class probability of a sample $\mathbf{x} \in \mathbb{R}^P$ is modeled as

$$Pr(c = 1 \mid \mathbf{x}) = \frac{1}{1 + e^{-(\mathbf{w} \cdot \mathbf{x} + w_0)}}.$$

The Logistic Regression is trained using the maximum likelihood: the model's parameters are estimated in a way that maximizes the likelihood of observing the inputs with respect to the parameters $\mathbf{w}$ and w_0. We chose to use this model as a baseline due to its simplicity and its easy implementation: by requiring few computational resources, it is easy to implement and fast to train. Moreover, it doesn't need the inputs to be scaled nor it needs to be tuned.

Support Vector Machine (SVM). This is a statistical model that constructs the best hyper-plane out of the infinite possibilities in a N-dimensional space—with N being the number of features. The best hyper-plane is capable of distinctly separating the data points, having the maximum margin (namely, the largest distance found to the nearest training data points of any class).

KNearestNeighbors(KNN). This is a non-parametric technique that classifies the samples using the dataset alone (i.e., without building a model). The classification is made as a majority vote, i.e., based on the class of the majority of its k nearest neighbors data points.

Decision Tree. One of the most frequently used models in Machine Learning is a *Decision Tree* classifier. The tree structure is characterized by multiple nodes: the *root node* and the *internal nodes*, which represent the inputs, and a series of *leaves*, which correspond to the outputs. All these nodes are connected via branches. A specific

path through the branches leads to an output. More specifically, given the input in the root node, this is elaborated and passed to the following nodes via the branches following an if-then-else diagram. The process is iterated for all the inputs until the leaves of the tree, representing the output. To create the best structure, we used the *GINI impurity*. It calculates how many inputs would be misclassified if assigned to a random class. This algorithm has a higher computational efficiency than, for example, *information gain*. This, together with the simplicity of the decision tree model, allowed us to have an easy-to-use classifier.

Random Forest. One of the problems that affect Decision trees models, is their tendency to overfit. This is due to the fact that they cannot properly generalize the data. For this reason, among the models tested we included the *Random Forest*. This is an ensemble model, which uses a set of weak classifiers to solve the assigned task. The week classifiers used in this case are decision trees. Each of the decision trees is trained on a separate subset of the data and, in order to reduce the correlation between the trees, a subset of the features of each sample is used. A sample can be used in multiple decision trees. Using a subset of the features is particularly useful in cases where many features are available, as the risk of them being correlated increases. The classification result given by the decision trees is then averaged to obtain a single output.

Bagging. Similar to the Random Forest, the Bagging classifier relies on an arbitrary number of decision trees. These are built using samples belonging to a subset of the original dataset. Compared to the Random Forest classifier, the split point is decided differently. While for the Random Forest the decision trees are split based on a random subset of the variables, in the Bagging algorithm the full set of variables is considered, and a split is made in a way to minimize the error. This leads to structural similarities between the trees, which in turn does not solve the overfitting issues of the individual decision trees. The inclusion of this model was due to a allow better comparison with newer and more performing models.

Extremely Randomized Trees. This model adds another layer of randomization to the Random Forest. The *Extremely Randomized Trees* (ExtraTrees) model, in fact, randomizes the optimal split in each node, besides randomly splitting the data for each of the individual trees. For this reason, the splitting rule is decided based on the best split obtained. While the generalization capabilities are higher, this model allow also for faster computation compared to the Random Forest.-

AdaBoost. Among the ensemble algorithms used, we also included the ones on *boosting*. Among models we found AdaBoost. This generates individual decision trees sequentially. Moreover, each sample of the training data gets assigned a weight, which is updated during the training. After creating the first tree, the algorithm assigns a higher weight to the misclassified samples and creates another decision tree. The generation of the decision trees, as well as the adjusting of the weights, continues until the model can no longer improve its accuracy.

Gradient Boosting. Similarly to AdaBoost, we also included the *Gradient Boosting* algorithm. Compared to AdaBoost, the trees are grown one at a time in a way that minimizes the loss. This process continues until the loss function can no longer be improved.

XGBoost. The downside of the Gradient Boost model is its computational inefficiency. To address this we also considered XGBoost, which is a better-performing implementation of the Gradient Boosting algorithm. It allows in fact for faster computation and easier parallelization. Therefore, the algorithm performs better and can be easily scaled to bigger datasets.

2.2.2 Deep Learning

Although machine learning models seem to become progressively better at solving the multiple tasks for which they are employed, they still suffer from the structural limitation that can't be easily overcome. More specifically, machine learning models see a degradation in their performances when dealing with very large datasets (high number of samples) and with high dimensional data (high number of features). These problems can be overcome by employing deep learning models. These, thanks to their *deep* structure, can learn finer details from a large amount of data that allows to better deal with the multitude of information available.

For the purpose of this thesis work, we focused our attention on deep learning models for time series classification. Our data depends, in fact, on the time (i.e., a commit likely depends on the previous commits). For this reason we used two deep neural networks, based on 1-dimensional convolutional neural networks (1D-CNN), the Fully Convolutional Network (FCN) [102] and Residual Network (ResNet) [102]. These two specific models have proven to be among the best performing models in the tasks of time series classification[103]. They have also

proven to be the best performing our previous work on the classification of time series data for autonomous surface recognition [104].

Following is a more specific description of the models adopted:

Residual Network. The first deep learning model used is a residual network (ResNet) [102]. Among the many different types of ResNet developed, the one we used is composed of 11 layers, of which nine are convolutional. Between the convolutional layers, it has some shortcut connection which allows the network to learn the residual [105]. In this way, the network can be trained more efficiently, as there is a direct flow of the gradient through the connections. Also, the connections help in reducing the *vanishing gradient effect*, which prevents deeper neural networks from training correctly. In this work, we employed the ResNet shown in [103]. It consists of three residual blocks, each composed of three 1-dimensional convolutional layers alternated to pooling layers, and their output is added to the input of the residual block. The last residual block is followed by a global average pooling (GAP) layer [106] in place of the fully connected layer. The GAP layer recognizes the features maps of the convolutional layers as a category confidence map. Also, the GAP reduces the number of parameters to be trained in the network, making it more lightweight and reducing the risk of overfitting, when compared to the fully connected layer.

Fully Convolutional Neural Network. The second method used is a fully convolutional neural network (FCN) [102]. Compared to the ResNet, this network does not present any pooling layer, which keeps the dimension of the time series unchanged throughout the convolutions. As for the ResNet, after the convolutions, the features are passed to a global average pooling (GAP) layer. The FCN architecture was originally proposed for semantic segmentation [107]. Its name derives from the fact that the last layer of this network is another convolutional layer instead of a classical fully connected layer. In this work, we used the architecture proposed by Wang et al. [102], which uses the original FCN as a feature extractor, and a softmax layer to predict the labels. More specifically, the FCN used in this work is adopted from [103]. This implementation consists of three convolutional blocks, each composed of a 1-dimensional convolutional layer and by a batch normalization layer [108]. It uses a rectified linear unit (ReLU) [109] activation function. The output of the last convolutional block is fed to the GAP layer, fully connected to a traditional softmax for the time series classification. This model has proven to be

on par with the state-of-the-art models in time series classification in previous works on time series classification [102]. Moreover, it is smaller than the ResNet, which would make the FCN model more computationally efficient.

2.2.3 Time Series

While working on this thesis, we realized that many of the problems we were facing with the data could be partially due to the time dependency of the data (PUBLICATION V). If we consider for example commit data, each commit will depend on the commits before. If this is not taken into account, we could lose precious information. For this reason, the commit data could actually be considered as time series data. More generally speaking, a time series assumes that the current state of a system depends on the states of the last n time points (this n is often called the *lag*), and tries to understand this dependency [110]. This is also true in SE; specifically, if we talk about data from version control systems (e.g., commit information). Each commit, in fact, depends *at least* on the previous status of the system, and therefore on the previous commits. It might also be the case in which a commit depends on other external factors, which therefore need to be analyzed further (multivariate analysis).

The analysis of this dependency from timed data has important applications in different domains; most notably in finance [111], [112] and weather analysis [113], where the goal is to *forecast* [114] the future behavior of a variable based on its previous performance (and that of other relevant variables). The fundamental technique used in these domains is statistical *time series analysis*: robust statistical methods designed for describing time dependencies accurately, considering also the presence of noise and natural variations.

One of the most important concepts in time series analysis methods is the *stationarity* of the data. Stationarity indicates that the statistical properties of the time series, like its mean and variance, are the same regardless of the time of observation. This makes the series more predictable. For example, series which constantly grow or have a seasonal behavior are not stationary. When the data is non-stationary, it is often possible to remove trends (and regain stationarity) by differencing [115]. Data stationarity can usually be verified using statistical tests like *Augmented Dickey-Fuller* (ADF) [116], [117], or Phillips-Perron test [118].

Another useful concept is the one of *autocorrelation*, which presents the correla-

tion between a variable at time t, i.e. $Y(t)$, and its previous (lagged) value $Y(t-k)$. As with normal correlation, the value of autocorrelation varies from -1 to 1. In an autocorrelation plot, the values are visualized for several lag values. Similarly, the *partial autocorrelation* also measures the correlation between $Y(t)$ and $Y(t-k)$ but the effect of the values between $Y(t)$ and $Y(t-k)$ are removed.

2.2.4 Anomaly Detection

Among the other problems found during the work that brought to this thesis, we could find the unbalanced data problem. This is due to the fact that faults are (or should be) a rarity. The issue of unbalanced data often leads to the misclassification of faulty instances.

As it is described in the book *Identification of Outliers* ([119]), an anomaly is an observation that deviates from others in such a significant way that arises suspicion that it was generated by a different mechanism. Anomaly detection, in short, consists in finding patterns in data that don't follow the expected behavior ([120]). Due to the anomalous data being strictly linked to a specific domain, it is very hard to define in a unique way what is an anomaly and what it is not: a small deviation from normal can be something dangerous (body temperature in the medical field) or something totally normal (fluctuation of the value of a currency in the forex market). For this reason, the detection of anomalies has been studied in different research areas, and many techniques have been developed specifically for certain applications. One of the most classical examples that come to mind is the application of anomaly detection in credit card fraud: a strange, anomalous, transaction given a history of the transaction, could indicate that the credit card has been stolen ([121]). Moreover, we could think about anomalous patterns in data traffic in a computer network, which could mean that there is a hacking event in progress ([122]). Also, an anomaly in an MRI image could indicate the presence of a tumor ([123]), or a strange evolution of sensor data from a spacecraft could indicate a faulty component ([124]). Anomalies can be divided into three categories:

- *point anomalies*, where a single instance can be considered anomalous compared to the rest of the data;
- *contextual anomalies*, when an instance is anomalous in a specific instance but not otherwise ([125]). These have been specifically studied in time-series

data ([126], [127]), and in spacial data ([128]);

- *collective anomalies*, when a collection of data instances is anomalous with respect to the rest of the dataset. This type of anomalies have been particularly studied for sequence data ([129], [130]), graph data ([131]), and spatial data ([132]).

In order to detect anomalies, we can use models that operate in the following three modes, based on the availability and type of labels:

- *supervised*, which assumes that the data has been labeled fully (both normal and anomalous data). In this case, the model used is a normal classifier. Unfortunately, the anomalous instances are much fewer than the normal data, hence the dataset is extremely unbalanced. This can cause issues in the training of the models ([133]). Moreover, it is very difficult to be able to label all the possible anomaly scenarios for a given problem;
- *semisupervised*, which assumes that only the normal data has been labeled. Compared to supervised models, this avoids the problem related to unbalanced data, as the model needs to only learn to recognize what is normal. In the case of spacecraft fault detection ([124]), it would have been quite difficult to model all the anomaly scenarios. Therefore the typical approach is to build a model for the normal class, and test it to find anomalies in the test set;
- *unsupervised*, which does not require any sort of labeled data, therefore making the models more widely usable. The main assumption for unsupervised models is that the normal instances are far more frequent than the anomalous ones.

Following is a description of the anomaly detectors that have been used throughout the works for this thesis.

OneClassSVM (OCSVM) [134]. This method is based on a Support Vector Machine. Similarly, it learns a frontier that 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 [120].

IsolationForest (IF) [135]. 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 paths for particular samples, these are highly likely to be anomalies. By design, this method can handle high-dimensional data [120].

LocalOutlierFactor (LOF) [136]. This computes the local density of a given sample compared to its neighbors. The LOF density of 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 of their neighbors, it means that the sample analyzed is an anomaly, else it is considered to be normal.

3 RESEARCH METHODOLOGY

In this chapter, we present the structure of the thesis work, the rationale behind the analysis done alongside the data and the techniques used in the work. More specifically, Section 3.1 shows the structure and the logic of the whole thesis work, from the initial idea to the development of the research questions and goal achieved. Section 3.2 describes the data used in the work, more specifically focusing on the datasets used and the metrics investigated during the studies. Section 3.3 describes the techniques used in the different steps of the study, including the machine learning and deep learning models already described in Chapter 2, alongside other data analysis and preprocessing techniques.

3.1 Research Structure

The purpose of this thesis is that of discussing and provide useful approaches for detecting software faults utilizing advanced machine learning techniques. To achieve this goal, we followed the steps presented in Figure 3.1.

Machine Learning for Fault Detection: We first analyzed a set of Machine Learning models for the detection of faults. In particular, this approach has been used for the detection of faults at the commit level (Publication I). Then, we evaluated the performance of the set of Machine Learning models to select the best performing ones and to use them for identifying the most informative set of metrics.

While performing the machine learning analysis, we noticed two main issues:

1. **Unbalanced Data:** Software faults are (or should be) a rarity. For this reason, the data available for software faults are heavily unbalanced. The issue of unbalanced data often leads to the misclassification of faulty instances as the Machine Learning classifiers do not have enough data of both types of samples to properly generalize.

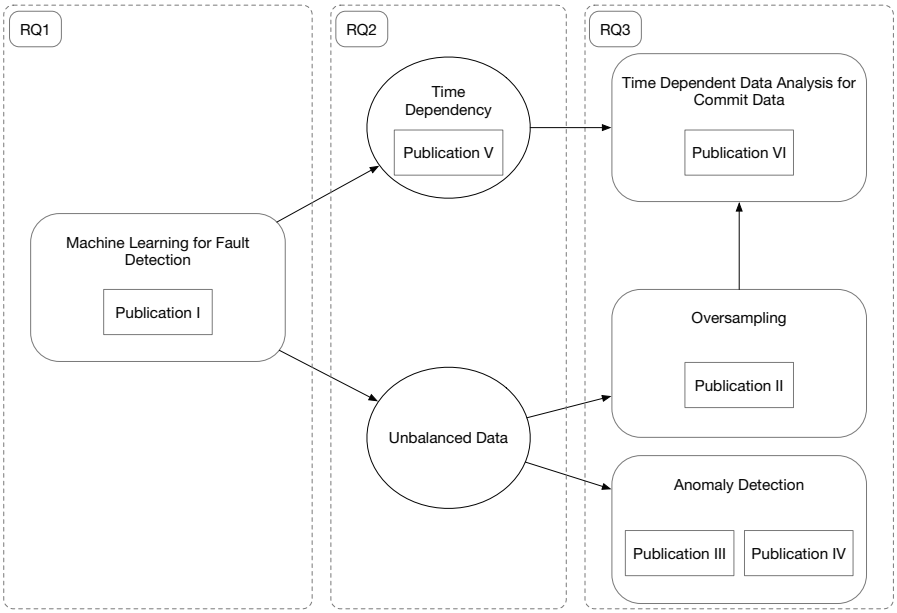


Figure 3.1 Methodology followed in the thesis

2. **Time Dependency**: The nature of the data is sequential and time dependent, and therefore it could be appropriate to take into account this dependency in the analysis, as it can provide additional information. We further investigated the time dependency of the commits using time series analysis methodologies, since it should be taken into account (Publication V).

To address the two aforementioned issues, we took the following steps:

1. Regarding the *unbalanced data* issue, we tackled the task in two way:
    - **Oversampling**: We applied oversampling techniques (i.e. SMOTE) to artificially generate additional samples of the minority class. This approach was used for vulnerability detection (Publication II).
    - **Anomaly Detection**: We firstly generated a dataset containing anomalies in a cloud-native system (Publication III) and tested a basic classifier to gain experience with anomalies and their detection in a traditional domain. We then brought what we learned on anomalies to the software quality domain, and applied some anomaly detection models for comparing their performances with classical Machine Learning classifiers (Publication IV).

2. On the subject of the *time dependency* of commit data, we included this information in our analysis by using Deep Learning classifiers capable of considering the time dependency and comparing their performances with that of traditional Machine Learning classifiers (PUBLICATION VI).

3.2 Adopted Data Sources

In this thesis, we mainly focused on the analysis of commit data, and the use of the information retrieved via static analysis tools regarding the single commits and the files within it. More specifically, for the majority of the works that support this thesis, we used the Technical Debt Dataset [137]. This dataset contains the data from 33 open-source projects sourced from the Apache Software Foundation. All the projects' commits were analyzed using SonarQube (better described in Section 2.1.3, which allowed us to have the indication of a set of rules and metrics which characterize each commit. More specifically, it provides more than 500 rules for Java based software, and it gives the indication of whether these rules have been violated or not in the code. Regarding the metrics, the ones provided are linked to the characteristics of the software. A full list of the metrics provided can be found in Table 2.1.

Besides the metrics provided by SonarQube, we also calculated additional metrics based on the implementation by Pascarella et al. [138] for the metrics proposed by Rahman et al. [90] and Kamei et al. [26]. The details on these metrics can be found in Table 2.2.

Since we also used data from the National Vulnerability Database (NVD). This data comprises 9 Java projects. For these projects, we used the commit data alongside multiple metrics. More specifically we used 24 process metrics, 9 product metrics, and textual features collected as the frequency of appearance of each word of the commit message (bag-of-word). A total of 1,446 individual words token were used. The information on the process and product metrics can be found in Table 2.3.

3.3 Research Techniques

In this section, we discuss the multiple analysis techniques used throughout the work. Given the multitude of analyses performed for this thesis, and given the differ-

ent types of data used, we used different analysis techniques for the different steps of the research work. The common denominator is the use of data analysis techniques and models from the spectrum of machine learning.

3.3.1 Machine Learning Models

More specifically, we explored particularly machine learning classifiers for the purpose of categorizing commits data as faulty or vulnerable. Among the specific techniques used, we focused our attention on classical models like Support Vector Machine (SVM), KNearest Neighbors (KNN), and tree based classifiers like Decision Tree, Random Forest, Extremely Randomized Trees. We also explored more advance models built using boosting techniques like AdaBoost, Gradient Boost and XGBoost. All these techniques are thoroughly described in Section 2.2.1.

3.3.2 Deep Learning Models

Most of these models and techniques were used throughout the whole study and kept as reference and baseline for comparing other more advanced models. For example, we compared the performances of the machine learning classifiers based on boosting techniques with more advanced deep learning models aimed at classifying commits considering also the historical data and the time dependency. Among the variety of deep learning classifiers, we selected the Residual Network (ResNet) and a Fully Convolutional Neural Network (FCN). These are based on 1-dimentional convolutional neural networks (1D-CNN), an architecture that proved to be among the best performing in time series classification tasks also in our previous study [104]. These models have been described in Section 2.2.2.

3.3.3 Anomaly Detection Models

Besides normal classifiers, we also explored the use of techniques builds specifically for detecting rarity in data, or in other words for the detection of anomalies. The specific techniques used are based on classical machine learning models like the SVM, the KNN, and the Random Forest, but focus their scope is to isolate and detect abnormalities. The models used for this purposed are the OneClassSVM

(OCSVM), Local Outlier Factor (LOF), and Isolation Forest (IF). More details on these can be found in Section 2.2.4.

3.3.4 Data Analysis Techniques

In order to use the models and techniques described, we also had to preprocess the data and make it suitable for the analysis performed. For this purpose, we used mainly two techniques. The first is meant to remove variables that don't add additional information via checking the multicollinearity of the variables. For this we exploited the Variable Inflation Factor (VIF) [139]: it measures how much the variance of the model increases for each variable. Each variable that has the value of VIF coefficient > 5 is removed. The second preprocessing technique was used for solving the problem of unbalanced data. Throughout the thesis work, we noticed that most of the data used had some serious unbalancing problems, with the minority class accounting for a very small percentage of the total number of samples (i.e., in ?? < 5% of faulty commits). One approach used to solve this issue was the use of oversampling techniques, which work by generating artificial data belonging to the minority class, based on the characteristics of the existing samples of the minority class. Among the multiple oversampling techniques available, we used the Synthetic Minority Oversampling Technique (SMOTE), as it was shown to produce the best results in similar contexts [140].

Apart from these techniques, which have been used to preprocess the data, in this thesis work we also approached other statistical techniques that allowed us to further inspect the time dependency of the data. For this purpose, we used autocorrelation and partial autocorrelations. The *autocorrelation*, calculates the correlation between a variable at time t, i.e. $Y(t)$, and its previous (lagged) value $Y(t-k)$. As with normal correlation, the value of autocorrelation varies from -1 to 1. Similarly, the *partial autocorrelation* also measures the correlation between $Y(t)$ and $Y(t-k)$ but the effect of the values between $Y(t)$ and $Y(t-k)$ are removed.

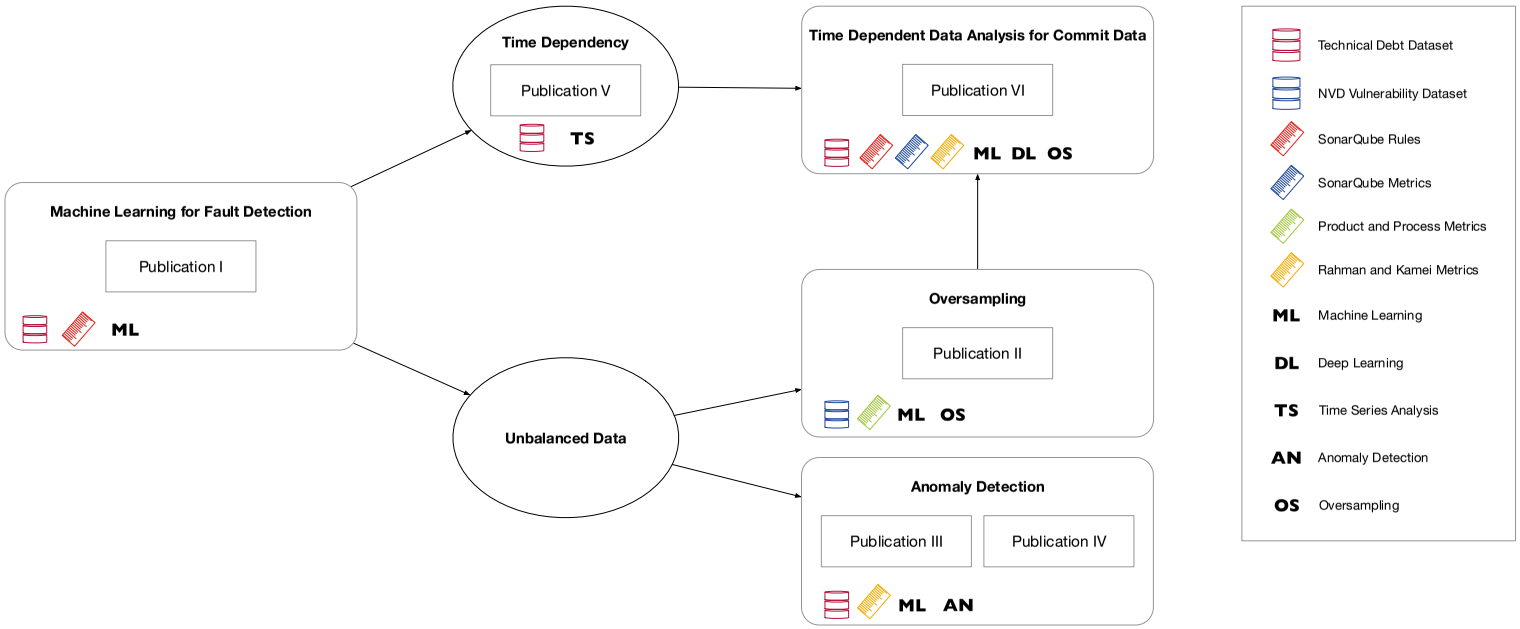


Figure 3.2 Methodology followed in the thesis, including the details of which data sources and techniques have been used.

4 RESULTS

This chapter presents the results obtained throughout the thesis. More specifically, Section 4.1 shows the results obtained in PUBLICATION I when classifying the commits as fault-inducing using machine learning, including the issues that arose. Section 4.2 presents the implementation of the oversampling technique (in PUBLICATION II) and how this helped in overcoming some of the issues found working with unbalanced data. Section 4.3 provides insight on the use of anomaly detection problem and how it can be applied for solving the problem linked to the rarity of faults in the commit data (PUBLICATIONS III and IV). Section 4.4 demonstrates how the time dependency issue can be solved by treating commit data as time series (PUBLICATIONS V). Finally, Section 4.5 shows the results obtained by applying various of the techniques tested throughout the work, including the use of oversampling techniques, taking into account the time dependency of the data, and using deep learning models (PUBLICATIONS VI).

4.1 Machine Learning for Fault Detection

RQ1 - PUBLICATION I

Given the popularity of tools for assessing the quality of software, and in particular, Static Analysis Tools (SAT), we started approaching our problem by investigating the accuracy and fault-proneness of SonarQube. SonarQube works by analyzing the code compliance against a set of predefined rules. When one of the rules is violated, this gets reported. Among the multiple rules provided by the SAT, a specific set is identified as "bugs", meaning that they represent something that is not working as it should in the code and that will soon trigger a fault. From practitioners' point of view, in contrast, there are opinions that led us to believe that what SonarQube classifies as a bug doesn't always result in an actual fault. This causes their interpretation to be subjective. For this reason, we performed an analysis using machine learning

classifiers in order to understand whether the information on the SonarQube rule violated (SQ-violations) can be used to successfully indicate the faultiness of a piece of software. More in detail, what we aimed at accomplishing during this first study was to understand:

- Whether the SQ-violations classified as bugs are more fault-prone compared to other types of rules;
- Which are the most fault-prone SQ-violations;
- What is the prediction accuracy of the SonarQube quality model based on the violation classified as bugs.

To understand whether SQ-violations can indicate the faultiness of a piece of code, we decided to perform a historical analysis of the commits of 21 open source projects from the Apache software foundation from the Technical Debt Dataset [137].

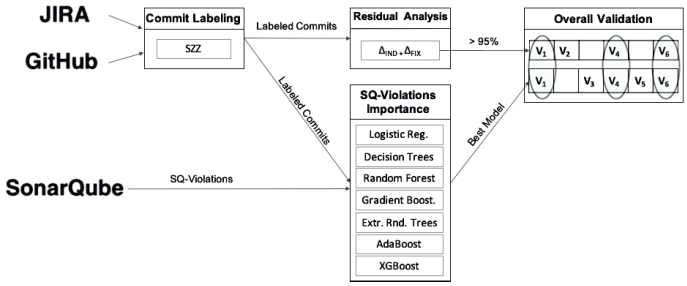


Figure 4.1 Data analysis pipeline (PUBLICATION I)

Firstly we extracted the commit data of the projects from the Github repositories and the corresponding information on issues from Jira. These were then classified using the SZZ algorithm [141] as either fault-inducing or not fault-inducing.

As it can be seen in Figure 4.1, we simultaneously run the machine learning analysis using the SQ-violation as predictors and the variable "fault-inducing/not fault-inducing" as target for the classification. The machine learning models (classifiers) used are described in Section 2.2.1.

The results of the machine learning analysis were in line with our expectations. It was in fact clear that it was possible to use machine learning models to classify commits as fault-inducing or not, using the SQ-violations as predictor. The accuracy metric used for understanding how the model worked and which was the best per-

forming machine learning model was the ROC-AUC. According to the ROC-AUC, the best performing model was the XGBoost, with and ROC-AUC of 83.2%.

4.1.1 Issues with Detection of Faults with Machine Learning

RQ2

Although the apparently positive results, we noticed that the models actually performed quite poorly when considering their ability to discriminate between the positive (fault-inducing) and negative (not fault-inducing) samples in the dataset. While the ROC-AUC was generally high, metrics like precision, recall, and as a consequence f-measure were very poor. The XGBoost, for example, despite the ROC-AUC of 83.2%, only had a precision of 60.8%, recall of 18.2%, and f-measure of 31.8%. Further analysis of the results showed that the problem was a quite heavy misclassification of the positive samples. This means that the machine learning models were able to correctly identify the not fault-inducing commits (true negative rate of 99.7%), while wrongly classifying the fault-inducing as not fault-inducing (false negative rate of 81.8%).

These results lead us to investigate further the motivation for this heavy misclassification. Analyzing the dataset used, we noticed that roughly 95% of the commits were actually not fault-inducing, while only 5% of the total actually induced a fault. This made our data extremely unbalanced. This unbalancing was actually confusing the machine learning models as they didn't get enough samples of both classes (the falt-inducing and the not fault-inducing commits). For this reason, all the classifiers learned to always predict the commit as not fault-inducing.

Unfortunately, this heavy unbalancing problem is quite common in software fault/vulnerability datasets, since faults are (or should be) a rarity. It is therefore clear that classical approaches will not work when analyzing and classifying this kind of data, but other approaches must be taken.

Investigating further on how to solve this issue, we decided to continue following two approaches:

- Artificially generate more samples of the minority class (non fault-inducing commits) via oversampling techniques.
- Treat the faults as anomalies, and therefore use anomaly detection techniques to find fault-inducing instances of the code

Besides the unbalanced data, another problem that came out while performing this study was that the commit data is, of course, sequential, and therefore there exists a time dependency between commits that must be taken into account. In this first work, we did not consider it since we were simply interested in understanding if we could classify the single commit as fault-inducing or not, based on its SQ-violations, without caring for the commits that came before that. Of course, the results obtained led us to believe that another possible way to improve the classification performance of the machine learning models could be that of including the commit history in the data used by the classifiers to perform the analysis.

4.2 Oversampling as a Way to Solve the Unbalanced Data Problem

RQ3 - PUBLICATION II

The first way that could be used to deal with the unbalanced data is to use oversampling techniques to artificially generate data belonging to the minority class. To test this we used another dataset that included information on vulnerability issues rather than faults. Besides faults, vulnerabilities also present problems to the quality of the code as they might represent threats to the security of the software which could lead to undesired effects. Moreover, similar to faults, also vulnerabilities are (or should be) a rarity in a software. It seems therefore fitting to test our findings in the context of software faults, also in the case of vulnerabilities, and to include oversampling techniques to try to solve the misclassification of vulnerable commits due to highly unbalanced data.

Table 4.1 Description of the 9 Java projects (PUBLICATION II)

Project	#Commit	LOC	#Sample Commits	#VCCs
CONVERSATION	5,810	16,035	1,000	10
CANDLEPIN	8,646	30,875	300	3
HAWTIO	8,354	3,705	1,200	12
JBOSS-NEGOTIATION	299	505	191	2
JENKINS	25,867	29,080	4,400	44
JOLOKIA	1,573	3,685	1,100	11
JUNRAR	221	1,325	100	1
LITEMALL	990	3,500	100	1
STRUTS1-FOREVER	4,526	4,025	600	6
	56,286	**92,735**	**8,991**	**90**

In this case, we used 9 Java projects which have the data on public software vulnerability recorded on the National Vulnerability Database (NVD): this is a database that aims at collecting known software vulnerabilities. The details of these projects can be seen in Table 4.1. In total, the projects considered account for 56,286 commits, but for the purpose of our study and computational reasons, we had to cap the number of total commits to 8,991. A total of 90 commits presented vulnerabilities.

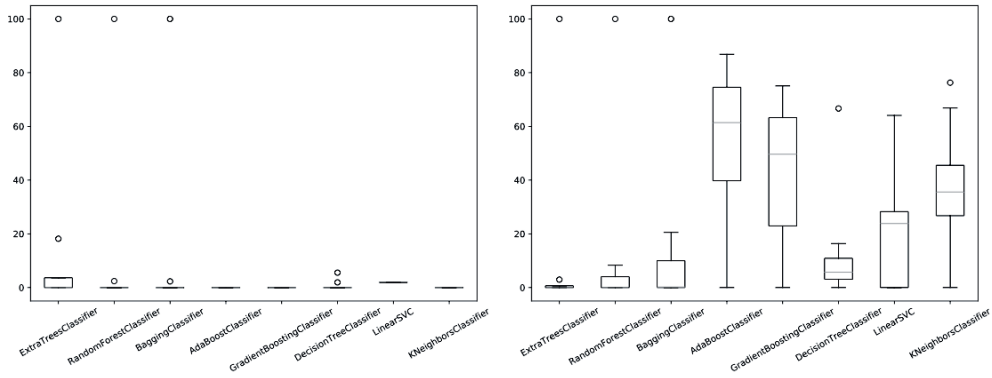
Also for this study, the data analysis pipeline included machine learning classifiers, more specifically we used classical machine learning models like Support Vector Machine (SVM), KNearest Neighbors (KNN), and Bagging, tree based classifiers like Decision Tree, Random Forest, Extremely Randomized Trees, and boosting techniques AdaBoost, and Gradient Boost. A description of how these models work can be found in Section 2.2.1.

Regarding the predictor variable used, we used three sets of metrics: process, product, and textual features. More specifically, regarding the process metrics, we considered metrics that allow us to quantify the changes in the projects, the contribution of committing authors, the number of files involved, etc. Regarding the product metrics, we considered metrics that allowed us to quantify the properties of the source code. The last set of metrics used are textual features: these have been extracted by collecting a bag-of-word (frequency of appearance of each individual word) of the commits data. In total, we used 24 process metrics, 9 product metrics, and the textual features had a total of 1,446 individual tokens (specific words). The full list of metrics used can be found in Table 2.3 (Section 3.2)

Before running the machine learning classifiers, we performed an oversampling of the data in order to balance the two classes: the commits presenting vulnerabilities versus the ones without vulnerability, since there was a proportion of 1:100. For this purpose, we used the Synthetic Minority Oversampling Technique (SMOTE) which generates samples of the minority class based on its preexisting samples.

After performing the classification, we found out that in general the results were much better than the ones obtained with the fault prediction. Of course, both the samples to classify are different and the metrics used as well, but we notice that in general that most of the machine learning models could correctly classify a good majority of the samples The boosting techniques particularly obtained fairly high results, with the highest F-measure obtained by the AdaBoost.

It can still be clearly seen the benefit of using oversampling in Figure 4.2. On



(a) F-measure without oversampling **(b)** F-measure with oversampling

Figure 4.2 F-measure comparison using all machine learning classifiers on the dataset without oversampling (left) and using oversampling (right)

the left, it is possible to see the F-measure obtained by all the models when using the normal dataset with no oversampling. We can see that it resembles the results obtained in our analysis performed for the faults. On the other hand, with the use of the SMOTE oversampling technique, it is clear from the Figure 4.2b on the right that most of the models improve their classification performance. This is particularly true for the AdaBoost and the Gradient Boost classifiers.

4.3 Software Faults as Anomalies

RQ3 - PUBLICATION III and IV

As it can be seen from the results shown in the previous section, it is clear that despite the classification improvement for some of the machine learning classifiers, the results obtained after balancing the dataset with oversampling techniques are still not satisfying. It is therefore hard to say if a system for detecting software vulnerabilities or faults based solely on these techniques could be used in real life situations. For this reason, we tried to investigate further the use of anomaly detection techniques for the detection (not anymore classification) of faulty commits.

4.3.1 A Preliminary Analysis

In order for us to gain more experience in the field of anomaly detection, we first investigated the anomaly detection problem in a traditional domain, focusing on the anomalies and their detection in cloud-native systems.

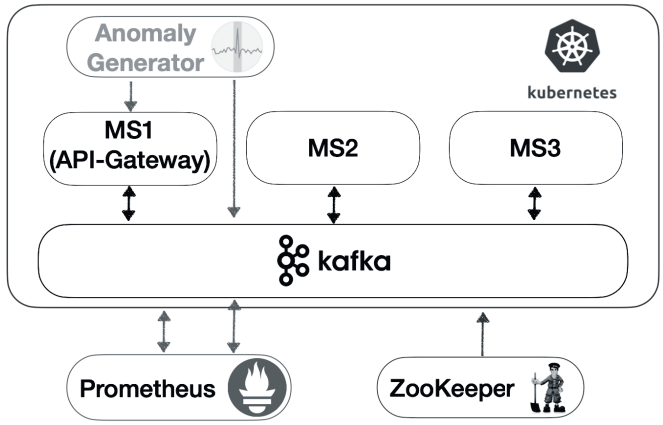


Figure 4.3 Cloud-native system set-up for anomaly generation (PUBLICATION III)

For this purpose, we set up a system based on Kubernetes to simulate a cloud-native system. The full structure of the system simulated can be seen in Figure 4.3. More specifically, the system was used to simulate a real cloud-native system consisting of 3 nodes, on top of which we deployed an anomaly generator that randomly injected memory anomalies. This was done by overloading the system memory until the request for memory exceeded the maximum allowed memory causing the restart of the system. By randomizing the injection of anomalies, we ensured that the data collected was similar to real data.

After the data collection, for the sake of understanding, if the dataset was actually viable, we analyzed the performance of a basic machine learning classifier, a RANDOM FOREST, when used for the prediction of a fault in a cloud-native system. Surprisingly enough, the RANDOM FOREST was able to correctly predict an anomaly with an F-measure of 92.4%. Of course, in this case, we trained and tested the model using single instances of the data, without really considering the history, but limiting the analysis to classifying each data point as either normal or anomalous.

The collected data was useful for us to be able to study and test approaches for the detection of anomalies. Since the data was collected at run time and simulated a

real life system, we could further study and understand how to approach this type of anomalies.

More specifically, as it can be seen from Figure 4.4, the fault most commonly happens after the injection of the anomaly has already stopped. This of course means that there is a brief time period between the injection (or occurrence) of the anomaly, and the actual fault that can be used to eventually correct possible mistakes, or at least limit the damages.

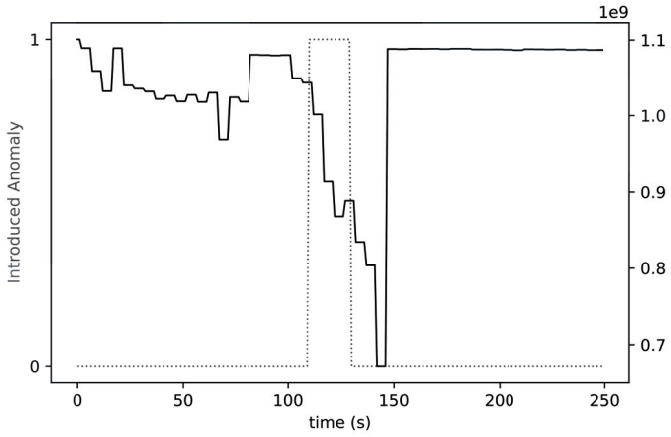


Figure 4.4 An example of anomaly: the dotted line represent the injection of the anomaly, while the full line indicates the available memory (in GB) (PUBLICATION III)

This might be very useful also when considering the analysis of software commits: if we manage to recognize an anomalous commit, where for example a fault or a vulnerability is injected, we have time to limit its possible damages and solve the issue that might arise.

4.3.2 Anomaly Detection for Software Fault Detection

To put in practice the lessons learned from building an anomaly dataset in the domain of cloud-native system to the software quality issue, we tried partially replicating our previous works on software fault detection using a newer version of the dataset Technical Debt Dataset [137]. This improved version allowed us to analyze 32 projects. In total, these account for 61,081 commits, and 193,800 files.

For this specific work, we focused our analysis at the file level rather than the commit level, using the same metrics and predictors used by Pascarella et al. in [138].

In order to verify if faults in software can be treated as anomalies, we compared the performances of normal machine learning classifiers with machine learning based anomaly detectors. More specifically, for this study, the data analysis pipeline included 3 machine learning classifiers, namely Support Vector Machine (SVM), Extremely Randomized Trees (ExtraTrees), and KNearest Neighbors (KNN), and 3 machine learning based anomaly detectors: OneClassSVM (OCSVM), Isolation Forest (IF), and Local Outlier Factor (LOF). More detail on these can be found in Section 2.2.1.

As for the metrics used as predictors, these were extracted using the implementation in [138] of the metrics proposed by Rahman [90] and Kamei [26]. The full list of metrics used can be found in Table 2.2 (Section 3.2)

Analyzing the data, we noticed that the percentage of faulty files was 34%. This indeed is much higher than the one found in our previous studies on fault prediction and vulnerability prediction. For this reason, we compared the machine learning models on a different portion of the dataset:

- Full dataset (~ 34% of defects)
- Three projects with the highest number of defective files (~ 50%)
- Three projects with the smallest number of defective files (< 20%)

As it can be seen in Figure 4.5, we noticed that while the anomaly detection models perform better than their classifiers counterparts in the case with the least defective files, their performance in terms of F-measure cannot be considered good enough. As a matter of fact, the anomaly detectors, reach an F-measure of ~ 30%, while their classifier counterparts have an F-measure that goes from ~ 5% to ~ 30%. In the cases in which we considered the whole dataset or only the projects with the most defective files, we can see that the performances of the anomaly detectors are similar to those of the machine learning classifiers.

From these results it is clear that there is no real advantage of using anomaly detectors, as even when the percentage of faulty instances is low, they don't bring many benefits to the detection.

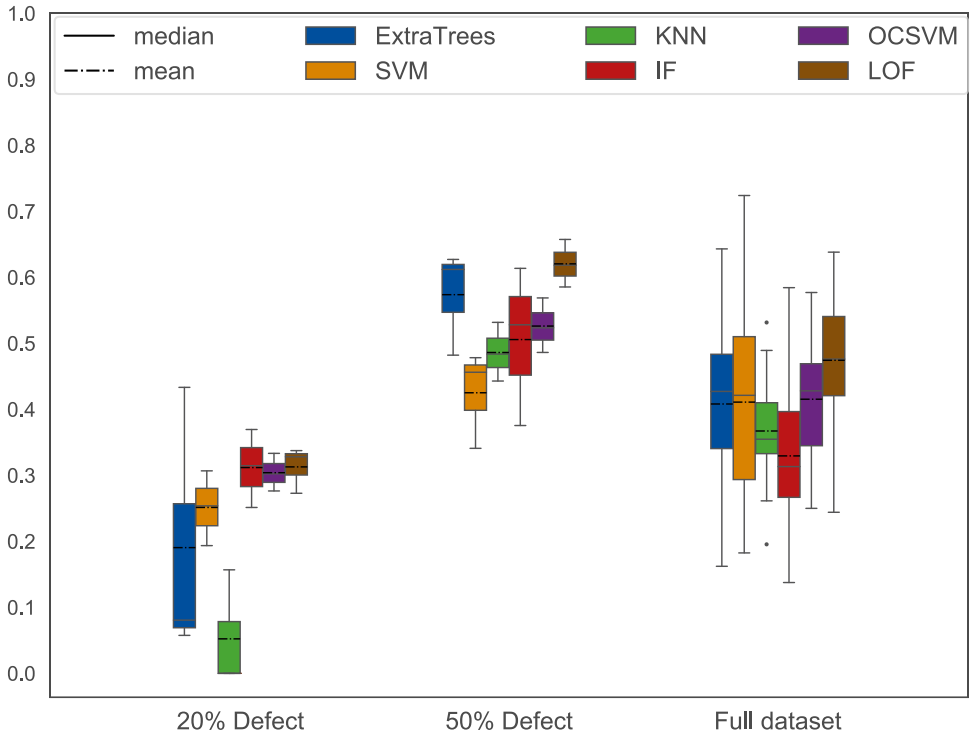


Figure 4.5 F-measure of the multiple models on the different portions of the dataset (Adapted from PUBLICATION IV)

4.4 Is the Time Dependency Important?

RQ3 - PUBLICATION V

Given the results obtained so far in solving the issues found with the software fault prediction, we tested additional methodologies trying to see if it is worth exploring the time dependency of the commit data (Section 2.2.3. We expect in fact that by taking into consideration the history of the commits, we can get more information that can help in understanding when a fault is happening.

For this reason, we considered the projects available in the Technical Debt Dataset [137] and calculated the autocorrelation and the partial autocorrelations of the commits over a span of 30 sequential commits.

These two measures allow us to understand what is the relation between consecutive data points. More specifically, the autocorrelation calculates the correlation between a variable $Y(t)$ with its previous values $Y(t-k)$. k is the lag, meaning the

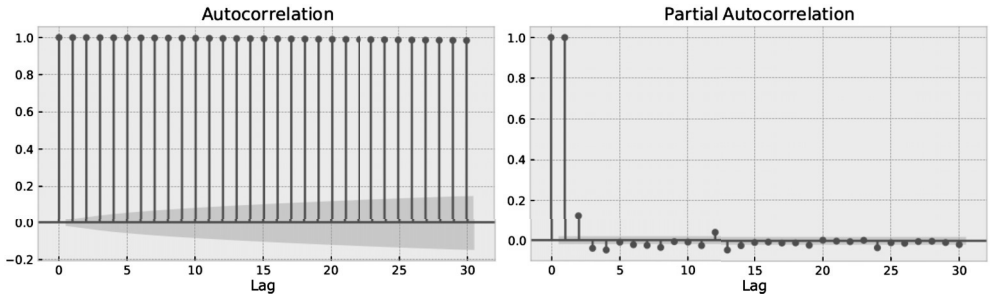


Figure 4.6 Autocorrelation and partial autocorrelation considering 30 consecutive commits (PUBLICATION V)

number of previous values considered. In other words, this measure gives us the value of the dependency of a variable on its historical values. Similarly, the partial autocorrelation measures the correlation between the same two values as before but excluding the effect of all the values in between.

In Figure 4.6 we can see for example a very strong autocorrelation for a commit with its history (almost alwasy 1). Similarly, the partial autocorrelation suggests a significant amount of information is lost if information between commits is not considered.

These results suggest that it might be worth investigating the time dependency of the commit data further, and possibly include it as additional information for the detection of faults.

4.5 Oversampling, Time Dependency and Deep Learning for Software Fault Detection

RQ3 - PUBLICATION VI

Given the results obtained so far, we decided to perform again an analysis of the software faults, using the lessons learned from the previous experiments. More specifically we used the same dataset used for the anomaly detection, the Technical Debt Dataset, from which we analyzed 28 projects.

For this analysis, we used multiple predictors. In fact, besides the SonarQube rules violation used in our previous study, we also used the SonarQube metrics (SQ-metrics). Besides these, we also used the same metrics used for the comparison between anomaly detectors and machine learning classifiers. Also in this case we used

the implementation given in [138] of the metrics proposed by Rahman [90] and Kamei [26]. The full list of metrics used can be found in Table 2.2 (Section 3.2)

Regarding the models used for the prediction of the fault inducing commits, we split the analysis into two parts, the first using classical techniques, and classifiers, while the second takes into account the time dependency of the data. More precisely, for the "classical" analysis we used the best performing machine learning classifiers from previous analyses for the classification of the single commits as fault inducing or not, namely RANDOM FOREST, GRADIENT BOOST and XGBOOST. Regarding the time dependency of the commits, in order to take it into account, we used deep learning models used for the classification of time series, which were already successfully used in other works [104]. The advantage of this approach is that it allows taking into account not only the information related to the single commit but that of the previous commits. The model used for this case are a RESIDUAL NETWORK (ResNet) and a FULLY CONVOLUTIONAL NEURAL NETWORK (FCN). More information on these can be found in Section 2.2.2.

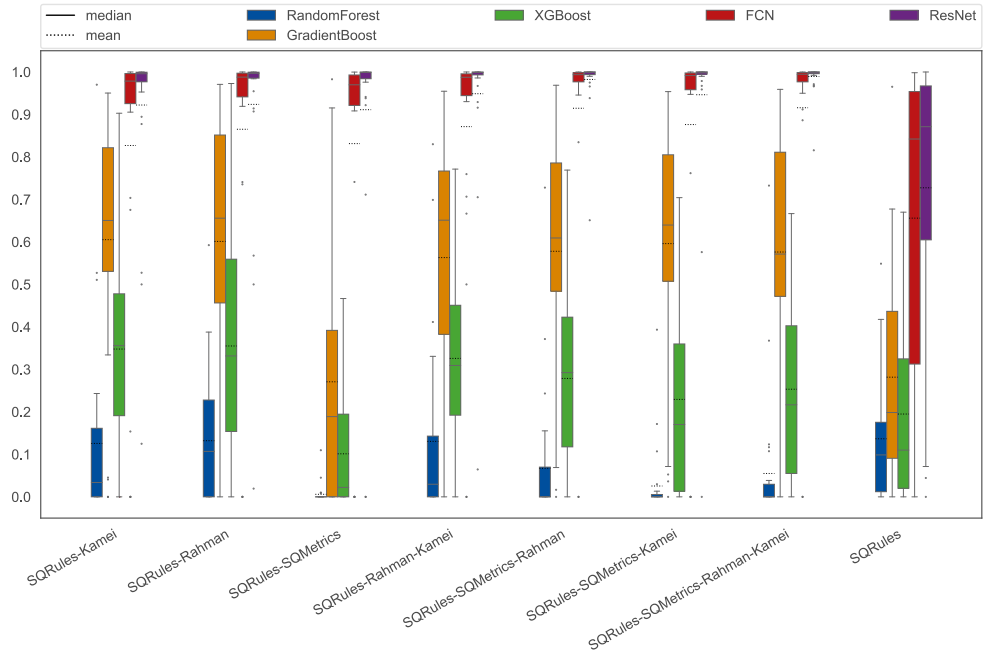


Figure 4.7 F-measure comparison among Machine Learning and Deep Learning models for SQ rules compared to software metrics (PUBLICATION VI)

As an additional step before performing the actual analysis, we also rebalanced

the data as it was shown previously, by using the SMOTE oversampling technique. It is also important to highlight that besides the classification task, we performed the analysis using multiple combinations of metrics: we were therefore able also to discriminate which type of metrics are better suited for the detection of fault-inducing commits.

Table 4.2 Accuracy metrics comparison for Deep Learning models

	Deep Learning			
	FCN		ResNet	
	All	Type	All	Type
AUC	94,8 %	96,0 %	99,8 %	99,8 %
F-Measure	91,6 %	91,4 %	98,9 %	95,3 %
Precision	92,1 %	92,0 %	99,5 %	96,0 %
Recall	91,2 %	91,0 %	98,5 %	94,9 %
MCC	90,2 %	89,9 %	98,2 %	94,4 %
FNR	8,8 %	9,0 %	1,5 %	5,1 %
TNR	99,0 %	98,9 %	99,6 %	99,4 %
FPR	1,0 %	1,1 %	0,4 %	0,6 %

After these steps, we trained and validate the models and obtained the results shown in Figure 4.7 It is clear from the figure that the deep learning models are substantially better suited for the fault-detection problem compared to the machine learning classifiers. More specifically it can be seen that both the ResNet and the FCN perform consistently regardless of the combination of metrics used, except for the case in which only SonarQube rule violations were used. From the boxplots, it appears evidently also that the machine learning classifiers have inconsistent results throughout the validation process, with the F-measure varying in some cases from almost 0% to > 90%. Looking at the metric combination, we can see that in general the ResNet and the FCN perform best when all the metrics are used, that means when SonarQube rule violations, SQ-metrics, Kamei, and Rahman metrics all are used as predictors. In Table 4.2 it can be seen the value of the other accuracy metrics calculated for both the ResNet and the FCN when all the metric sets are used.

5 DISCUSSION

In this Chapter, we discuss the results obtained in this thesis, providing therefore an answer to the research questions outlined in Section 1.1.

RQ1 - What machine learning techniques can be adopted for software fault detection?

In order to answer the RQ1, we trained and test multiple machine learning models from different underlying techniques, on data containing the information of commits from multiple projects of the Apache Software Foundation, and analyze with SonarQube. For this analysis, the machine learning models had to classify the individual commit as either fault-inducing or non fault-inducing. The results were that some of the machine learning models were accurate enough to be used for detecting software faults. Particularly, the models based on boosting techniques, AdaBoost, Gradient Boosting, and XGBoost, were undoubtedly the most accurate (in terms of ROC-AUC). Among these, the best performing technique was the XGBoost (Publication I). The reason for these techniques to be more accurate than the others used (like the tree based models), is probably due to the way they are built. As explained in Section 2.2.1, these models build many "shallow" classifiers that allow the model to be less prone to overfitting. This allows them to generalize better. Moreover, these multiple "shallow" classifiers are created iteratively during the training process in order to maximize the weights for the commits that were wrongly classified, pushing the model to give more importance to the mistakes.

Therefore, to summarize, the machine learning models best suited for software fault detection are in general the ones based on boosting techniques. More specifically we noticed that XGBoost is the best performing model in classifying the commits as fault-inducing or non fault-inducing. It is important to notice that this result is true in the case no other methods are used to solve the issues described in RQ2. Once those issues are addressed, it can be seen that while boosting methodology remains useful and accurate, they no longer represent the most accurate models.

RQ2 - What issues prevent the proper use of machine learning for software fault detection?

From the analysis of the performance of the machine learning classifiers in the task of software fault prediction, it was interesting to see that while the boosting techniques seen before were quite performing in terms of ROC-AUC (with the XGBoost having the best ROC-AUC of 83.2%), the other performance metrics were quite poor. More specifically, taking into consideration the XGBoost, this had an F-measure of only 31.8%, indicating difficulties in properly discriminating the fault-inducing commits from the non fault-inducing commits. This was even more evident when looking at the rate of commits wrongly classified as non fault-inducing (False Negative Rate (FNR)), and the rate of correctly identified non-fault-inducing commits (True Negative Rate (TNR)).

From these results, it is clear that there is a problem in the classification, and while the results in terms of ROC-AUC might seem high, the other measure shed some light on the real issue which is the difficulty of properly recognizing the data in the two distinct categories: fault-inducing and non fault-inducing commits. This is even more clear since, after analyzing the dataset, we noticed that the fault-inducing commits accounted for 95% of the total number of commits in the dataset, meaning that the non fault-inducing commits represented only the remaining 5% of the data. In other words, the data with what we have been dealing with was highly *unbalanced*. This is true not only for our specific dataset but for the software faults in general since these are usually an anomaly and therefore are rare.

Besides the issue linked to the unbalanced data, we also noticed that while the commits have been considered individually in this analysis, classifying each of them without taking into account the previous development of the software, it might be more useful to also consider the history of each commit. This could in fact bring more information that might be useful for the machine learning models to better detect the faults.

To summarize, two main issues were found when dealing with the detection of software faults. From one side we have in fact a highly *unbalanced data*, and from another side, we should deal with the data coming from commits considering its *time dependency*.

RQ3 - How can the issues linked to fault detection be solved?

After the definition of the issues found with the detection of faults, we approached them investigating different techniques. In light of the findings of RQ2, the answer to this question can be split into the following two parts.

- **Unbalanced data.** In order to tackle the issue of the unbalanced data, we proposed and tested two different approaches. The first was based on artificially generating multiple samples of the minority class, in order to re-balanced the two classes. This approach was tested on vulnerability data rather than software fault data since these share many of the same characteristics when it comes to the proportion between normal data and faulty/vulnerable data (PUBLICATION II). To re-balance the data we used the Synthetic Minority Oversampling Technique (SMOTE), which generates artificial samples of the minority class based on its preexisting samples. This allowed to partially ameliorate the results obtained in the detection of vulnerable commits. This was particularly true for the boosting techniques, namely ADABOOST and GRADIENT BOOSTING, which significantly improved their performances compared to the classification based on the raw dataset. The second approach for dealing with the unbalanced data was to consider the samples of the minority class as anomalies, and therefore to apply anomaly detection techniques for detecting the fault-inducing commits (PUBLICATION III AND IV). We noticed that, while the anomaly detection models do work better when dealing with highly unbalanced data (as they should), they still don't have positive enough performance to be considered a good substitute to the machine learning classifiers. Finally, while oversampling techniques indeed improved the classification performance of unbalanced data, treating them as anomalies did not lead to the expected results. In any case, we did not feel like discarding completely the anomaly detection approach, since we noticed that the data used for software fault detection share indeed many similarities with anomaly detection specific datasets. It is also worth noting that not all software systems are characterized by a small percentage of faults, and therefore the use of anomaly detection needs to be evaluated on a case-by-case basis.
- **Taking into account the time dependency.** Regarding the time dependency of the data, we analyzed the commit data using statistical techniques from the domain of time series analysis, namely autocorrelation and partial autocorre-

lation (PUBLICATION V). From this preliminary, it was clear that the commits are highly correlated with the previous commits, and therefore considering a commit's own history we could indeed include much more information that could be of use to the machine learning classifiers. Also, from the results obtained from the partial autocorrelation, we found that much of the information is lost if information between commits is not included.

To give a definitive answer to the RQ3, we included the best performing techniques found and inspected throughout the thesis work to analyze the same dataset used for answering the RQ1. More specifically, we compared the performance of the machine learning techniques based on boosting which best performed in the previous works, with additional models based on deep learning. These models were chosen in order to include the time dependency of the commit data in the analysis. Also, the whole dataset had been preprocessed using the SMOTE oversampling technique. From this work we found that using the oversampling, together with the inclusion of the information on the time dependency of the data, actually improved significantly the detection of software faults, substantially outperforming every other analysis done previously (PUBLICATION VI).

To summarize, the results obtained in this thesis and in the publications from which it derives have shown a set of methodologies that allow for a more accurate detection of software faults. These results can be also particularly useful for the practitioners, since they could know which combination of tools and techniques to use, alongside which type of metrics, to more accurately detect software faults. This would allow them to more easily find and react to faults, resulting in an overall improvement. Similarly, researchers could benefit from these results as they show how a combination of different techniques can improve the detection of software faults; more specifically this show how adapting methodology coming from a different domain (see anomaly detection or time series analysis) can indeed improve the detection of software faults.

6 CONCLUSION

The application of machine learning and deep learning techniques for the detection of faults in software, is a topic towards which the research community has increased its interests. While many works have been done on the use of multiple techniques, there is no real answer yet on which are the technique that are proven to correctly and confidently predict faults in software. Neither there is a real answer on why some of the techniques don't work or have sub-optimal performance when compared to other fields.

In this thesis we aimed to shed some lights on the use of machine learning models for software faults detection and how to improve it by proposing some solution to the issues linked to the use of machine learning. More specifically we investigated multiple machine learning classifiers and noted which of these were the ones to perform best in the fault detection problems. After finding the best performing model, we investigate some of the pitfalls that it had. This were mainly due to the type of data used, since it had two main problems: the data related to faults in software is heavily unbalanced, and each sample depends on the previous. For this reason we formalized the two issues of unbalancing and time dependency of the data. We therefore proposed multiple approach to address both the issues, and find some solution that help in improving the performance of machine learning techniques in the detection of software faults. More specifically, we found that both balancing the data and taking into account its time dependency could considerably improve the fault detection capabilities of the machine and deep learning models used.

By addressing these issues, we have contributed to the body of research on the machine learning tools for software quality and solved some of the issues that are associated with the detection of software quality issues, being them faults or vulnerabilities. We are also confident that the results presented in this thesis, and in the associated published work, will be useful to further improve the detection of software quality issues, and also improve the models and techniques used.

6.1 Future Works

In light of the findings of this thesis, it is clear that much work is still needed before coming up with a unifying solution for the detection of faults in software. More specifically, we can clearly see that there are still many approaches that can be further improved and investigated.

Throughout the thesis, we have used machine and deep learning models and techniques for the detection of faults. These models were selected since they were the ones most used and already investigate in other fields of software quality. With the extremely fast improvements that characterize the research around machine and deep learning, it would be interesting to study and analyze the performances of more advanced and updated machine learning models. For example, in the last part of this work we used deep learning models adapted from the generic task of time series classification to classify commits taking into account their history and therefore accounting for the time dependency. The models selected were two of the best-performing models already analyzed in other research works. Nevertheless, there exist many more models that could be more accurate and could bring interesting insights when considering the time dependency. LONG SHORT TERM MEMORY (LSTM) networks, for example, have been known for a few years now to be particularly suited for dealing with time series, since they can take into account a whole series of data by automatically defining what it is important to take into account and what's not. Similarly, newer TRANSFORMERS deep learning models could bring an added value, by using attention mechanisms to discriminate on what to consider important.

Regarding considering software faults as anomalies, since our findings clearly indicate that the faults in a software resemble anomalies found in other similar problems, it could be interesting to further investigate the use of anomaly detection techniques for software faults detection. It could be interesting, for example, to apply more advanced techniques commonly used for the detection of anomalies, such as the AUTOENCODERS. These are a type of deep neural network that learns to replicate the input, and can therefore be trained to recognize the NORMAL samples. In this way, when the input is an ANOMALOUS sample it raises an error since the network has never seen it before.

In terms of metrics used as features to detect the faults, these could be investigated further as well. We have seen in our multiple works that the metric-set used as input

features affect the result of the detection. For example, one of the side results of our work was that using the so-called *product* and *process* metrics could bring additional information overall ameliorating the performance of the machine learning classifiers. It would therefore be interesting to see which other metrics could be calculated and how these affect the results. We have already seen in our other studies that some newer metrics can be of help in the daily development process of services, according to the practitioners [142]. Such metrics could be useful also for the detection of software faults.

Finally, but equally important, it would be interesting to see how the methods and techniques investigated in this thesis work could be of use in other fields of software engineering and software quality specifically. They share many similarities with software faults, including their rarity and the fact that should be found and addressed as quickly as possible. Moreover, most of the research done on software vulnerabilities has focused on the metrics to use for the detection. Since we have seen already that oversampling techniques can help in the detection of software vulnerabilities, it could be interesting to see how the other methodologies used in this work can improve their detection and be further generalized.

REFERENCES

[1] V. Lenarduzzi, A. Sillitti, and D. Taibi, "Analyzing forty years of software maintenance models", in *39th International Conference on Software Engineering Companion*, ser. ICSE-C '17, 2017, pp. 146–148.

[2] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection", *Empirical Softw. Engg.*, vol. 21, no. 3, pp. 1143–1191, Jun. 2016.

[3] D. Di Nucci, F. Palomba, D. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: Are we there yet?", Mar. 2018.

[4] F. Pecorelli, F. Palomba, F. Khomh, and A. De Lucia, "Developer-driven code smell prioritization", in *International Conference on Mining Software Repositories*, 2020.

[5] S. Lujan, F. Pecorelli, F. Palomba, A. De Lucia, and V. Lenarduzzi, "A preliminary study on the adequacy of static analysis warnings with respect to code smell prediction", in *4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation*. 2020, pp. 1–6.

[6] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities", *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, 2011.

[7] H. Perl, S. Dechand, M. Smith, *et al.*, "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits", in *ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 426–437.

[8] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with applying vulnerability prediction models", in *Symposium and Bootcamp on the Science of Security*, 2015.

[9] Y. Zhang, D. Lo, X. Xia, B. Xu, J. Sun, and S. Li, "Combining software metrics and text features for vulnerable file prediction", in *International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2015, pp. 40–49.

[10] M. Jimenez, R. Rwemalika, M. Papadakis, F. Sarro, Y. Le Traon, and M. Harman, "The importance of accounting for real-world labelling when predicting software vulnerabilities", in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 695–705.

[11] 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.

[12] 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*, vol. 44, no. 5, pp. 412–428, 2018.

[13] Y. Kamei, E. Shihab, B. Adams, *et al.*, "A large-scale empirical study of just-in-time quality assurance", *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2013.

[14] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data", in *IEEE International Conference on Software Engineering*, vol. 2, 2015, pp. 99–108.

[15] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction", in *IEEE International Conference on Software Quality, Reliability and Security*, 2015, pp. 17–26.

[16] I. O. for Standardization (ISO), *Iso/iec 25000:2005, software engineering - software product quality requirements and evaluation (square)*, 2005.

[17] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: Current results, limitations, new approaches", *Automated Software Engg.*, pp. 375–407, Dec. 2010.

[18] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction", in *International Conference on Software Engineering*, ser. ICSE '11, 2011, pp. 481–490.

[19] N. Bettenburg, M. Nagappan, and A. E. Hassan, "Think locally, act globally: Improving defect and effort prediction models", in *Working Conference on Mining Software Repositories (MSR)*, 2012, pp. 60–69.

[20] L. Prechelt and A. Pepper, "Why software repositories are not used for defect-insertion circumstance analysis more often: A case study", *Information and Software Technology*, vol. 56, no. 10, 2014.

[21] M. Kondo, D. Germán, O. Mizuno, and E.-H. Choi, "The impact of context metrics on just-in-time defect prediction", *Empirical Software Engineering*, vol. 25, pp. 890–939, 2019.

[22] T. Hoang, H. Khanh Dam, Y. Kamei, D. Lo, and N. Ubayashi, "Deepjit: An end-to-end deep learning framework for just-in-time defect prediction", in *16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 34–45.

[23] V. Lenarduzzi, F. Lomio, H. Huttunen, and D. Taibi, "Are sonarqube rules inducing bugs?", in *27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 501–511.

[24] L. Pascarella, D. Spadini, F. Palomba, M. Bruntink, and A. Bacchelli, "Information needs in contemporary code review", *ACM on Human-Computer Interaction*, vol. 2, no. CSCW, pp. 1–27, 2018.

[25] Y. Yang, Y. Zhou, J. Liu, *et al.*, "Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models", in *International symposium on foundations of software engineering*, 2016, pp. 157–168.

[26] Y. Kamei, E. Shihab, B. Adams, *et al.*, *A large-scale empirical study of just-in-time quality assurance*, 2012.

[27] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models", *Empirical Software Engineering*, vol. 21, 2016.

[28] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators", *IEEE Transactions on software engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[29] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design", *IEEE Transactions on software engineering*, vol. 20, 1994.

[30] 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, pp. 531–577, 2012.

[31] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history", *IEEE Transactions on software engineering*, vol. 26, no. 7, pp. 653–661, 2000.

[32] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code! examining the effects of ownership on software quality", in *13th European conference on Foundations of software engineering*, 2011, pp. 4–14.

[33] L. Pascarella, F. Palomba, and A. Bacchelli, "On the performance of method-level bug prediction: A negative result", *Journal of Systems and Software*, vol. 161, 2020.

[34] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, "Toward a smell-aware bug prediction model", *IEEE Transactions on Software Engineering*, vol. 45, no. 2, pp. 194–218, 2017.

[35] L.-P. Querel and P. C. Rigby, "Warningsguru: Integrating statistical bug models with static analysis to provide timely and specific bug warnings", in *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018, 2018, pp. 892–895.

[36] A. Trautsch, S. Herbold, and J. Grabowski, "Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction", in *International Conference on Software Maintenance and Evolution (ICSME 2020)*, 2020.

[37] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history", *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653–661, 2000.

[38] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering", *IEEE Transactions on Software Engineering*, vol. 38, 2012.

[39] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, and J. Liu, "Dictionary learning based software defect prediction", in *International Conference on Software Engineering*, ser. ICSE 2014, 2014, pp. 414–423.

[40] W. Fu and T. Menzies, "Revisiting unsupervised learning for defect prediction", in *11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, 2017, pp. 72–83.

[41] W. Li, W. Zhang, X. Jia, and Z. Huang, "Effort-aware semi-supervised just-in-time defect prediction", *Information and Software Technology*, vol. 126, p. 106 364, 2020.

[42] A. E. Hassan, "Predicting faults using the complexity of code changes", in *International conference on software engineering*, IEEE, 2009, pp. 78–88.

[43] L. Pascarella, F. Palomba, and A. Bacchelli, "Re-evaluating method-level bug prediction", in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 592–601.

[44] F. Khomh, "Squad: Software quality understanding through the analysis of design", ser. WCRE '09, Washington, DC, USA: IEEE Computer Society, 2009, pp. 303–306.

[45] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "Bdtex: A gqm-based bayesian approach for the detection of antipatterns", *Journal of Systems and Software*, vol. 84, no. 4, pp. 559–572, 2011.

[46] D. Falessi, B. Russo, and K. Mullen, "What if i had no smells?", *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 78–84, 2017.

[47] F. A. F. I. Tollin, M. Zanoni, and R. Roveda, "Change prediction through coding rules violations", *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, EASE'17, pp. 61–64, 2017.

[48] V. Lenarduzzi, V. Nikkola, N. Saarimäki, and D. Taibi, "Does code quality affect pull request acceptance? an empirical study", *Journal of Systems and Software*, vol. 171, 2021.

[49] X. Yang, D. Lo, X. Xia, and J. Sun, "Tlel: A two-layer ensemble learning approach for just-in-time defect prediction", *Information and Software Technology*, vol. 87, pp. 206–220, 2017.

[50] F. Arcelli Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques", *Know.-Based Syst.*, vol. 128, no. C, pp. 43–58, Jul. 2017.

[51] N. Maneerat and P. Muenchaisri, "Bad-smell prediction from software design model using machine learning techniques", in *8th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, 2011, pp. 331–336.

[52] F. Pecorelli, D. Di Nucci, C. De Roover, and A. De Lucia, "A large empirical assessment of the role of data balancing in machine-learning-based code smell detection", *Journal of Systems and Software*, p. 110 693, 2020.

[53] V. Lenarduzzi, A. Martini, D. Taibi, and D. A. Tamburri, "Towards surgically-precise technical debt estimation: Early results and research roadmap", in *International Workshop on Machine Learning Techniques for Software Quality Evaluation*, 2019, pp. 37–42.

[54] N. Saarimaki, M. Baldassarre, V. Lenarduzzi, and S. Romano, "On the accuracy of sonarqube technical debt remediation time", *SEAA Euromicro 2019*, 2019.

[55] M. T. Baldassarre, V. Lenarduzzi, S. Romano, and N. Saarimaki, "On the diffuseness of technical debt items and accuracy of remediation time when using sonarqube", in *Information Software System*, 2020.

[56] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks", *Science*, vol. 313, no. 5786, pp. 504–507, 2006.

[57] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", *Commun. ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[58] R. Sarikaya, G. E. Hinton, and A. Deoras, "Application of deep belief networks for natural language understanding", *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 22, no. 4, pp. 778–784, 2014.

[59] M. White, C. Vendome, M. Linares-Vasquez, and D. Poshyvanyk, "Toward deep learning software repositories", in *12th Working Conference on Mining Software Repositories*, 2015, pp. 334–345.

[60] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Combining deep learning with information retrieval to localize buggy files for bug reports (n)", in *International Conference on Automated Software Engineering (ASE)*, 2015, pp. 476–481.

[61] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning", in *International Symposium on Foundations of Software Engineering*, ser. FSE 2016, 2016, pp. 631–642.

[62] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically enhanced software traceability using deep learning techniques", in *International Conference on Software Engineering (ICSE)*, 2017, pp. 3–14.

[63] X. Gu, H. Zhang, and S. Kim, "Deep code search", in *International Conference on Software Engineering (ICSE)*, 2018, pp. 933–944.

[64] S. M. Abozeed, M. Y. ElNainay, S. A. Fouad, and M. S. Abougabal, "Software bug prediction employing feature selection and deep learning", in *International Conference on Advances in the Emerging Computing Technologies (AECT)*, 2020, pp. 1–6.

[65] R. Ferenc, D. Bán, T. Grósz, and T. Gyimóthy, "Deep learning in static, metric-based bug prediction", *Array*, vol. 6, p. 100 021, 2020.

[66] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction", *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1267–1293, 2020.

[67] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for predicting vulnerable software components", *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 67–85, 2021.

[68] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network", in *International Conference on Software Quality, Reliability and Security (QRS)*, 2017, pp. 318–328.

[69] T. J. McCabe, "A complexity measure", *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[70] Y. Shin and L. Williams, "An empirical model to predict security vulnerabilities using code complexity metrics", in *International Symposium on Empirical Software Engineering and Measurement*, 2008, pp. 315–317.

[71] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?", *Empirical Software Engineering*, vol. 18, pp. 25–59, 2011.

[72] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities", *Journal of Systems Architecture*, vol. 57, no. 3, pp. 294–313, 2011.

[73] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista", in *International Conference on Software Testing, Verification and Validation*, 2010, pp. 421–428.

[74] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics vs text mining", in *International Symposium on Software Reliability Engineering*, 2014, pp. 23–33.

[75] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components", in *Conference on Computer and Communications Security*, 2007, pp. 529–540.

[76] V. H. Nguyen and L. M. S. Tran, "Predicting vulnerable software components with dependency graphs", in *International Workshop on Security Measurements and Metrics*, 2010.

[77] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining", *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, 2014.

[78] Y. Zhang, R. Jin, and Z.-H. Zhou, "Understanding bag-of-words model: A statistical framework", *International Journal of Machine Learning and Cybernetics*, vol. 1, no. 1-4, pp. 43–52, 2010.

[79] Z. S. Harris, "Distributional structure", *Word*, vol. 10, no. 2-3, pp. 146–162, 1954.

[80] B. Smith and L. Williams, "Using sql hotspots in a prioritization heuristic for detecting all types of web application vulnerabilities", in *International Conference on Software Testing, Verification and Validation*, 2011, pp. 220–229.

[81] C. Theisen and L. Williams, "Better together: Comparing vulnerability prediction models", *Information and Software Technology*, vol. 119, p. 106 204, 2020.

[82] C. Theisen, K. Herzig, P. Morrison, B. Murphy, and L. Williams, "Approximating attack surfaces with stack traces", in *International Conference on Software Engineering*, vol. 2, 2015, pp. 199–208.

[83] N. Bhatt, A. Anand, and V. Yadavalli, "Exploitability prediction of software vulnerabilities", *Quality and Reliability Engineering*, vol. 37, Sep. 2020.

[84] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison", *Empirical Softw. Engg.*, vol. 17, no. 4–5, pp. 531–577, 2012.

[85] K. Z. Sultana, V. Anu, and T.-Y. Chong, "Using software metrics for predicting vulnerable classes and methods in java projects: A machine learning approach", *Journal of Software: Evolution and Process*, vol. 33, 2020.

[86] V. Lenarduzzi, A. Sillitti, and D. Taibi, "A survey on code analysis tools for software maintenance prediction", in *6th International Conference in Software Engineering for Defence Applications*, Springer International Publishing, 2020, pp. 165–175.

[87] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, "How developers engage with static analysis tools in different contexts", in *Empirical Software Engineering*, 2019.

[88] M. Fowler and K. Beck, "Refactoring: Improving the design of existing code", *Addison-Wesley Longman Publishing Co., Inc.*, 1999.

[89] I. Sommerville, "Software engineering 10th edition", *ISBN-10*, vol. 137035152, p. 18, 2015.

[90] F. Rahman and P. Devanbu, "How, and why, process metrics are better", in *International Conference on Software Engineering*, IEEE Press, 2013, pp. 432–441.

[91] D. R. Cox, "The regression analysis of binary sequences", *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 20, no. 2, pp. 215–242, 1958, ISSN: 00359246. [Online]. Available: http://www.jstor.org/stable/2983890.

[92] L. Breiman, J. Friedman, C. J. Stone, and R. Olshen, *Classification and regression trees Regression trees*. Chapman and Hall, 1984.

[93] L. Breiman, "Bagging predictors", *Machine Learning*, vol. 24, no. 2, pp. 123–140, Aug. 1996.

[94] L. Breiman, "Random forests", *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[95] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees", *Machine Learning*, vol. 63, no. 1, pp. 3–42, Apr. 2006.

[96] Y. Freund and R. E. Schapire, "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting", *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119–139, Aug. 1997.

[97] J. H. Friedman, "Greedy function approximation: A gradient boosting machine", *The Annals of Statistics*, vol. 29, pp. 1189–1232, 2001.

[98] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System", in *22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16*, 2016, pp. 785–794.

[99] C. Cortes and V. Vapnik, "Support-vector networks", *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[100] Z. Zhang, "Introduction to machine learning: K-nearest neighbors", *Annals of translational medicine*, vol. 4, no. 11, 2016.

[101] T. M. Mitchell, *Machine Learning*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997, ISBN: 0070428077, 9780070428072.

[102] Z. Wang, W. Yan, and T. Oates, "Time series classification from scratch with deep neural networks: A strong baseline", in *2017 International Joint Conference on Neural Networks (IJCNN)*, 2017, pp. 1578–1585.

[103] H. I. Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P. Muller, "Deep learning for time series classification: A review", *Data Mining and Knowledge Discovery*, vol. 33, no. 4, pp. 917–963, 2019.

[104] F. Lomio, E. Skenderi, D. Mohamadi, J. Collin, R. Ghabcheloo, and H. Huttunen, "Surface type classification for autonomous robot indoor navigation", *27th European Signal Processing Conference (EUSIPCO) - Workshop on Signal Processing, Computer Vision and Deep Learning for Autonomous Systems*, 2019.

[105] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", in *IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[106] M. Lin, Q. Chen, and S. Yan, "Network in network", *arXiv preprint arXiv:1312.4400*, 2013.

[107] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3431–3440.

[108] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", *arXiv preprint arXiv:1502.03167*, 2015.

[109] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines", in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[110] G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[111] R. S. Tsay, *Analysis of financial time series*. John wiley & sons, 2005.

[112] J. Arroyo, R. Espınola, and C. Maté, "Different approaches to forecast interval time series: A comparison in finance", *Computational Economics*, vol. 37, no. 2, pp. 169–191, 2011.

[113] S. D. Campbell and F. X. Diebold, "Weather forecasting for weather derivatives", *Journal of the American Statistical Association*, vol. 100, no. 469, pp. 6–16, 2005.

[114] C. Chatfield, *Time-series forecasting*. CRC press, 2000.

[115] A. Sutcliffe, "Time-series forecasting using fractional differencing", *Journal of Forecasting*, vol. 13, no. 4, pp. 383–393, 1994.

[116] D. A. Dickey and W. A. Fuller, "Distribution of the estimators for autoregressive time series with a unit root", *Journal of the American statistical association*, vol. 74, no. 366a, pp. 427–431, 1979.

[117] Y.-W. Cheung and K. S. Lai, "Lag order and critical values of the augmented dickey–fuller test", *Journal of Business & Economic Statistics*, vol. 13, no. 3, pp. 277–280, 1995.

[118] P. C. Phillips and P. Perron, "Testing for a unit root in time series regression", *Biometrika*, vol. 75, no. 2, pp. 335–346, 1988.

[119] D. M. Hawkins, *Identification of outliers*. Springer, 1980, vol. 11.

[120] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey", *ACM computing surveys (CSUR)*, vol. 41, no. 3, pp. 1–58, 2009.

[121] E. Aleskerov, B. Freisleben, and B. Rao, "Cardwatch: A neural network based database mining system for credit card fraud detection", in *Proceedings of the IEEE/IAFE 1997 computational intelligence for financial engineering (CIFEr)*, IEEE, 1997, pp. 220–226.

[122] V. Kumar, "Parallel and distributed computing for cybersecurity", *IEEE Distributed Systems Online*, vol. 6, no. 10, 2005.

[123] C. Spence, L. Parra, and P. Sajda, "Detection, synthesis and compression in mammographic image analysis with a hierarchical image probability model", in *Proceedings IEEE workshop on mathematical methods in biomedical image analysis (MMBIA 2001)*, IEEE, 2001, pp. 3–10.

[124] R. Fujimaki, T. Yairi, and K. Machida, "An approach to spacecraft anomaly detection problem using kernel feature space", in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, 2005, pp. 401–410.

[125] X. Song, M. Wu, C. Jermaine, and S. Ranka, "Conditional anomaly detection", *IEEE Transactions on knowledge and Data Engineering*, vol. 19, no. 5, pp. 631–645, 2007.

[126] A. S. Weigend, M. Mangeas, and A. N. Srivastava, "Nonlinear gated experts for time series: Discovering regimes and avoiding overfitting", *International Journal of Neural Systems*, vol. 6, no. 04, pp. 373–399, 1995.

[127] S. Salvador, P. Chan, and J. Brodie, "Learning states and rules for time series anomaly detection.", in *FLAIRS conference*, 2004, pp. 306–311.

[128] Y. Kou, C.-T. Lu, and D. Chen, "Spatial weighted outlier detection", in *Proceedings of the 2006 SIAM international conference on data mining*, SIAM, 2006, pp. 614–618.

[129] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: Alternative data models", in *Proceedings of the 1999 IEEE symposium on security and privacy (Cat. No. 99CB36344)*, IEEE, 1999, pp. 133–145.

[130] P. Sun, S. Chawla, and B. Arunasalam, "Mining for outliers in sequential databases", in *Proceedings of the 2006 SIAM international conference on data mining*, SIAM, 2006, pp. 94–105.

[131] C. C. Noble and D. J. Cook, "Graph-based anomaly detection", in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2003, pp. 631–636.

[132] S. Shekhar, C.-T. Lu, and P. Zhang, "Detecting graph-based spatial outliers: Algorithms and applications (a summary of results)", in *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, 2001, pp. 371–376.

[133] N. V. Chawla, N. Japkowicz, and A. Kotcz, "Special issue on learning from imbalanced data sets", *ACM SIGKDD explorations newsletter*, vol. 6, no. 1, pp. 1–6, 2004.

[134] B. Schölkopf, J. C. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson, "Estimating the support of a high-dimensional distribution", *Neural computation*, vol. 13, no. 7, pp. 1443–1471, 2001.

[135] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest", in *Int.Conference on Data Mining*, 2008, pp. 413–422.

[136] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "Lof: Identifying density-based local outliers", in *Int.conference on Management of data*, 2000, pp. 93–104.

[137] V. Lenarduzzi, N. Saarimäki, and D. Taibi, "The technical debt dataset", in *Conference on Predictive Models and Data Analytics in Software Engineering*, 2019.

[138] 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.

[139] R. M. O'brien, "A caution regarding rules of thumb for variance inflation factors", *Quality & quantity*, vol. 41, no. 5, pp. 673–690, 2007.

[140] F. Pecorelli, D. Di Nucci, C. De Roover, and A. De Lucia, "On the role of data balancing for machine learning-based code smell detection", in *Proceedings of the 3rd ACM SIGSOFT international workshop on machine learning techniques for software quality evaluation*, 2019, pp. 19–24.

[141] a. A. Z. J. Śliwerski T. Zimmermann, "When do changes induce fixes?", in *International Workshop on Mining Software Repositories*, ser. MSR '05, St. Louis, Missouri: ACM, 2005, pp. 1–5.

[142] F. Lomio, Z. Codabux, D. Birtch, D. Hopkins, and D. Taibi, "On the benefits of the accelerate metrics: An industrial survey at vendasta", in *29th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022.

APPENDIX A PUBLICATION SUMMARY

Publication I - In this work we analysed the performance of multiple machine learning models for the purpose of classifying commit data as either fault-inducing or not. For this work we used a dataset containing the commits from 21 open source projects. For the classification of each commit we used the information on the SonarQube rules violated. As a result we selected the best performed model as the XGBoost classifier. Once the best performing model was known, we used it as a proxy to find which of the SonarQube rules violation is more informative for the classification. This allowed us to select a very small number of SonarQube rules that can be considered as highly informative for the purpose of the detection of faulty commits. We also raised some issue linked to the detection of faulty commits using machine learning.

Publication II - This second work we used the machine learning models already seen in the previous work to detect commits with vulnerability issues. While performing this analysis we analyzed multiple aspects of the detection. Firstly, we used oversampling techniques to overcome the problem linked to the unbalanced data. For this reason we used the SMOTE oversampling technique to artificially balance the two classes (vulnerable commits and not). Thanks to this we noticed an overall increase in classification performances of the machine learning classifiers. We also used this analysis to find out which of the metrics used as predictors was more informative. More specifically, for this work we used three type of metrics: product, process and textual metrics. We trained and tested the model using all possible combinations of the three metric-sets and we selected which combination of metrics yield the best classification performance.

Publication III - This work proposes RARE, a dataset containing data with anomalies. More specifically, we created a replica system of a cloud-native system, and designed an anomaly generator service that would randomly inject a byte stream in the system in order to overload the memory. We therefore collected the data

including a set of metrics monitoring the system, which would describe both the normal and the anomalous status. We also collected information on the injection of the anomalies. In this publication we also proposed a simple machine learning based method to classify the data collected from the system.

Publication IV - In this work we used what we learned in Publicatoin III on anomalies to try another approach to solve the issue related to unbalanced data. In fact, we performed an analysis on commit data with a file level granularity, to see if using anomaly detection techniques could improve the classification performance of the machine learners. For this purpose we compared three machine learning classifiers and three machine learning based anomaly detector to detect faulty files. The results of this analysis showed that there is indeed an improvements of the performance of the anomaly detectors when the data is heavily unbalanced, but it doesn't justify their use over the classical machine learning models. These in fact, perform overall better in detecting faulty files in commits.

Publication V - Given the nature of the commit data, in this publication we focused on demonstrating using statistical tools that commit data is indeed time dependent. To do so, we analyzed the commit data with the value of the Squale Index, a metric that indicates the "effort to fix all Code Smells in minutes". We analyzed the commit data using statistical tools use to evaluate time series data. For this reason we firstly calculated the autocorrelation and partial autocorrelation of the commits, finding that each commit is indeed dependent on its history. After this confirmation, we used an autoregressive integrated moving average (ARIMA), a model typically used for forecasting time series data, to forecast the future value of the Squale Index. With this we demonstrated that we could actually consider commit data as a time series, and therefore use time series specific analysis tools to analyze the commit data.

Publication VI - The last publication aimed at summarizing and putting into practice the lesson learnt in the precedent publications. In this we tackled the software fault prediction using the techniques that were found to work best in our previous works. More precisely, we compared the performance of the best machine learning models found in Publication I and II, and compared them with two deep learning classifiers specifically designed for time series classification. In this way we could incorporate in the analysis the time dependency of the commit data. Beside this, we also used the SMOTE oversampling technique in order to balance the dataset, and have roughly the same number of fault-inducing commits and non fault-inducing

commits. In order to be as generic as possible, we run this classification analysis using multiple set of metrics as predictor. More precisely we used the SonarQube rules, SonarQube metrics, and the metrics proposed by Rahman et al. [90] and Kamei et al. [26]. This analysis allowed us to confirm that the time dependency of the data does give valuable information, since the two models using the history of the commits performed best. We were also able to define the best combination of metrics, since using both the metrics defined in SonarQube and by Rahman and Kamei sensibly increased the classification performance of all the classifiers in general.

PUBLICATIONS

PUBLICATION I

Are sonarqube rules inducing bugs?

V. Lenarduzzi, F. Lomio, H. Huttunen, and D. Taibi

International Conference on Software Analysis, Evolution and Reengineering (SANER)
DOI: 10.1109/SANER48275.2020.9054821

Are SonarQube Rules Inducing Bugs?

Valentina Lenarduzzi
Lahti-Lappeenranta University
Lahti-Lappeenranta, Finland
valentina.lenarduzzi@lut.fi

Francesco Lomio
Tampere University
Tampere, Finland
francesco.lomio@tuni.fi

Heikki Huttunen
Tampere University
Tampere, Finland
heikki.huttunen@tuni.fi

Davide Taibi
Tampere University
Tampere, Finland
davide.taibi@tuni.fi

***Abstract*—The popularity of tools for analyzing Technical Debt, and particularly the popularity of SonarQube, is increasing rapidly. SonarQube proposes a set of coding rules, which represent something wrong in the code that will soon be reflected in a fault or will increase maintenance effort. However, our local companies were not confident in the usefulness of the rules proposed by SonarQube and contracted us to investigate the fault-proneness of these rules.**

In this work we aim at understanding which SonarQube rules are actually fault-prone and to understand which machine learning models can be adopted to accurately identify fault-prone rules.

We designed and conducted an empirical study on 21 well-known mature open-source projects. We applied the SZZ algorithm to label the fault-inducing commits. We analyzed the fault-proneness by comparing the classification power of seven machine learning models.

Among the 202 rules defined for Java by SonarQube, only 25 can be considered to have relatively low fault-proneness. Moreover, violations considered as "bugs" by SonarQube were generally not fault-prone and, consequently, the fault-prediction power of the model proposed by SonarQube is extremely low.

The rules applied by SonarQube for calculating technical debt should be thoroughly investigated and their harmfulness needs to be further confirmed. Therefore, companies should carefully consider which rules they really need to apply, especially if their goal is to reduce fault-proneness.

***Index Terms*—Technical Debt, SonarQube, coding style, code smells, architectural smells, static analysis, machine learning**

I. Introduction

The popularity of tools for analyzing technical debt, such as SonarQube, is increasing rapidly. In particular, SonarQube has been adopted by more than 85K organizations [1] including nearly 15K public open-source projects [2]. SonarQube analyzes code compliance against a set of rules. If the code violates a rule, SonarQube adds the time needed to refactor the violated rule as part of the technical debt. SonarQube also identifies a set of rules as "bugs", claiming that they "represent something wrong in the code and will soon be reflected in a fault"; moreover, they also claim that zero false positives are expected from "bugs" [3].

Four local companies have been using SonarQube for more than five years to detect possible issue in their code, reported that their developers do not believe that the rules classified as bugs can actually result in faults. Moreover, they also reported that the manual customization of the SonarQube out-of-the-box set of rules (named "the Sonar way"[4]) is very subjective and their developers did not manage to agree on a common set of rules that should be enforced. Therefore, the companies asked us to understand if it is possible to use machine learning to reduce the subjectivity of the customization of the SonarQube model, considering only rules that are actually fault-prone in their specific context.

SonarQube is not the most used static analysis tool on the market. Other tools such as Checkstyle, PMD and FindBugs are more used, especially in Open Source Projects [1] and in research [2]. However, the adoption of another tool in the DevOps pipeline requires extra effort for the companies, including the training and the maintenance of the tool itself. If the SonarQube rules actually resulted fault-prone, our companies would not need to invest extra effort to adopt and maintain other tools.

At the best of our knowledge, not studies have investigated the fault-proneness of SonarQube rules, and therefore, we accepted the challenge and we designed and conducted this study. At best, only a limited number of studies have considered SonarQube rule violations [3], [4], but they did not investigate the impact of the SonarQube violations considered as "bugs" on faults.

The goal of this work is twofold:

- Analyze the fault-proneness of SonarQube rule violations, and in particular, understand if rules classified as "bugs" are more fault-prone than security and maintainability rules.
- Analyze the accuracy of the quality model provided by SonarQube in order to understand the fault-prediction accuracy of the rules classified as "bugs".

SonarQube and issue tracking systems adopt similar terms for different concepts. Therefore, in order to clarify the terminology adopted in this work, we define *SQ-Violation* as a violated SonarQube rule that generated a SonarQube "issue" and *fault* as an incorrect step, process, data definition, or any unexpected behavior in a computer program inserted by a developer, and reported by Jira issue-tracker. We also use the term "fault-fixing" commit for commits where the developers have clearly reported bug fixing activity and "fault-inducing" commits for those commits that are responsible for the introduction of a fault.

[1]https://www.sonarqube.org
[2]https://sonarcloud.io/explore/projects
[3]SonarQube Rules: https://tinyurl.com/v7r8rqo
[4]SonarQube Quality Profiles: https://tinyurl.com/wkejmgr

The remainder of this paper is structured as follows. In Section II-A we introduce SonarQube and the SQ-Violations adopted in this work. In Section II we present the background of this work, introducing the SonarQube violations and the different machine learning algorithms applied in this work. In Section III, we describe the case study design. Section IV presents respectively the obtained results. Section V identifies threats to validity while Section VI describes related works. Finally, conclusions are drawn in Section VII.

II. Background

A. SonarQube

SonarQube is one of the most common Open Source static code analysis tools adopted both in academia [5],[2] and in industry [1]. SonarQube is provided as a service from the sonarcloud.io platform or it can be downloaded and executed on a private server.

SonarQube calculates several metrics such as the number of lines of code and the code complexity, and verifies the code's compliance against a specific set of "coding rules" defined for most common development languages. In case the analyzed source code violates a coding rule or if a metric is outside a predefined threshold, SonarQube generates an "issue". SonarQube includes Reliability, Maintainability and Security rules.

Reliability rules, also named "bugs" create issues (code violations) that "represents something wrong in the code" and that will soon be reflected in a bug. "Cod5e smells" are considered "maintainability-related issues" in the code that decreases code readability and code modifiability. It is important to note that the term "code smells" adopted in SonarQube does not refer to the commonly known code smells defined by Fowler et al. [6] but to a different set of rules. Fowler et al. [6] consider code smells as "surface indication that usually corresponds to a deeper problem in the system" but they can be indicators of different problems (e.g., bugs, maintenance effort, and code readability) while rules classified by SonarQube as "Code Smells" are only referred to maintenance issues. Moreover, only four of the 22 smells proposed my Fowler et al. are included in the rules classified as "Code Smells" by SonarQube (Duplicated Code, Long Method, Large Class, and Long Parameter List).

SonarQube also classifies the rules into five *severity* levels[5]: Blocker, Critical, Major, Minor, and Info.

In this work, we focus on the sq-violations, which are reliability rules classified as "bugs" by SonarQube, as we are interested in understanding whether they are related to faults.

SonarQube includes more than 200 rules for Java (Version 6.4). In the replication package (Section III-D) we report all the violations present in our dataset. In the remainder of this paper, column "*squid*" represents the original rule-id (SonarQube ID) defined by SonarQube. We did not rename it, to ease the replicability of this work. In the remainder of this work, we will refer to the different sq-violations with their id (squid). The complete list of violations can be found in the file "SonarQube-rules.xsls" in the online raw data.

B. Machine Learning Techniques

In this Section, we describe the machine learning techniques adopted in this work to predict the fault-proneness of sq-violations. Due to the nature of the task, all the models used for this work were used for classification. We compared eight machine learning models. Among these, we used a generalized linear model: Logistic Regression [7]; one tree based classifier: Decision Tree [8]; and 6 *ensemble classifiers*: Bagging [9], Random Forest [10], Extremely Randomized Trees [11], AdaBoost [12], Gradient Boosting [13], and XGBoost [14] which is an optimized implementation of Gradient Boosting. All the models, except the XGBoost, were implemented using the library *Scikit-Learn*[6], applying the default parameters for building the models. For the ensamble classifiers we alwasys used 100 estimators. The XGBoost classifier was implemented using the *XGBoost* library[7] also trained with 100 estimators.

1) Logistic Regression [7]: Contrary to the linear regression, which is used to predict a numerical value, Logistic Regression is used for predicting the category of a sample. Particularly, a binary Logistic Regression model is used to estimate the probability of a binary result (0 or 1) given a set of independent variables. Once the probabilities are known, these can be used to classify the inputs in one of the two classes, based on their probability to belong to either of the two.

Like all linear classifiers, Logistic Regression projects the P-dimensional input $\mathbf{x}$ into a scalar by a dot product of the learned weight vector $\mathbf{w}$ and the input sample: $\mathbf{w} \cdot \mathbf{x} + w_0$, where $w_0 \in \mathbb{R}$ the constant intercept. To have a result which can be interpreted as a class membership probability—a number between 0 and 1—Logistic Regression passes the projected scalar through the logistic function (sigmoid). This function, for any given input x, returns an output value between 0 and 1. The logistic function is defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

Where the class probability of a sample $\mathbf{x} \in \mathbb{R}^P$ is modeled as

$$Pr(c = 1 \mid \mathbf{x}) = \frac{1}{1 + e^{-(\mathbf{w} \cdot \mathbf{x} + w_0)}}.$$

Logistic Regression is trained through maximum likelihood: the model's parameters are estimated in a way to maximize the likelihood of observing the inputs with respect to the parameters $\mathbf{w}$ and w_0. We chose to use this model as baseline as it requires limited computational resources and it is easy to implement and fast to train.

[5]SonarQube Issues and Rules Severity:' https://docs.sonarqube.org/display/SONAR/Issues Last Access:May 2018

[6]https://scikit-learn.org

[7]https://xgboost.readthedocs.io

2) Decision Tree Classifier [8]: Utilizes a decision tree to return an output given a series of input variables. Its tree structure is characterized by a *root node* and multiple *internal nodes*, which are represented by the input variable, and *leaf*, corresponding to the output. The nodes are linked between one another through branches, representing a test. The output is given by the decision path taken. A decision tree is structured as a if-then-else diagram: in this structure, given the value of the variable in the root node, it can lead to subsequent nodes through branches following the result of a test. This process is iterated for all the input variables (one for each node) until it reaches the output, represented by the leaves of the tree.

In order to create the best structure, assigning each input variable to a different node, a series of metrics can be used. Amongst these we can find the *GINI impurity* and the *information gain*:

- Gini impurity measures how many times randomly chosen inputs would be wrongly classified if assigned to a randomly chosen class;
- Information gain measures how important is the information obtained at each node related to its outcome: the more important is the information obtained in one node, the purer will be the split.

In our models we used the Gini impurity measure to generate the tree as it is more computationally efficient. The reasons behind the choice of decision tree models and Logistic Regression, are their simplicity and easy implementation. Moreover, the data does not need to be normalized, and the structure of the tree can be easily visualized. However, this model is prone to overfitting, and therefore it cannot generalize the data. Furthermore, it does not perform well with imbalanced data, as it generates a biased structure.

3) Random Forest [10]: is an ensemble technique that helps to overcome overfitting issues of the decision tree. The term ensemble indicates that these models use a set of simpler models to solve the assigned task. In this case, Random Forest uses an ensemble of decision trees.

An arbitrary number of decision trees is generated considering a randomly chosen subset of the samples of the original dataset [9]. This subset is created with replacement, hence a sample can appear multiple times. Moreover, in order to reduce the correlation between the individual decision trees a random subset of the features of the original dataset. In this case, the subset is created without replacement. Each tree is therefore trained on its subset of the data, and it is able to give a prediction on new unseen data. The Random Forest classifier uses the results of all these trees and averages them to assign a label to the input. By randomly generating multiple decision trees, and averaging their results, the Random Forest classifier is able to better generalize the data. Moreover, using the random subspace method, the individual trees are not correlated between one another. This is particularly important when dealing with a dataset with many features, as the probability of them being correlated between each other increases.

4) Bagging [9]: Exactly like the Random Forest model, the Bagging classifier is applied to an arbitrary number of decision trees which are constructed choosing a subset of the samples of the original dataset. The difference with the Random Forest classifier is in the way in which the split point is decided: while in the Random Forest algorithm the splitting point is decided base on a random subset of the variables, the Bagging algorithm is allowed to look at the full set of variable to find the point minimizing the error. This translates in structural similarities between the trees which do not resolve the overfitting problem related to the single decision tree. This model was included as a mean of comparison with newer and better performing models.

5) Extremely Randomized Trees [11]: (ExtraTrees) [11], provides a further randomization degree to the Random Forest. For the Random Forest model, the individual trees are created by randomly choosing subsets of the dataset features. In the ExtraTrees model the way each node in the individual decision trees are split is also randomized. Instead of using the metrics seen before to find the optimal split for each node (Gini impurity and Information gain), the cut-off choice for each node is completely randomized, and the resulting splitting rule is decided based on the best random split. Due to its characteristics, especially related to the way the splits are made at the node level, the ExtraTrees model is less computationally expensive than the Random Forest model, while retaining a higher generalization capability compared to the single decision trees.

6) AdaBoost [12]: is another ensemble algorithm based on *boosting* [15] where the individual decision trees are grown sequentially. Moreover, a weight is assigned to each sample of the training set. Initially, all the samples are assigned the same weight. The model trains the first tree in order to minimize the classification error, and after the training is over, it increases the weights to those samples in the training set which were misclassified. Moreover, it grows another tree and the whole model is trained again with the new weights. This whole process continues until a predefined number of trees has been generated or the accuracy of the model cannot be improved anymore. Due to the many decision trees, as for the other ensemble algorithms, AdaBoost is less prone to overfitting and can, therefore, generalize better the data. Moreover, it automatically selects the most important features for the task it is trying to solve. However, it can be more susceptible to the presence of noise and outliers in the data.

7) Gradient Boosting [13]: also uses an ensemble of individual decision trees which are generated sequentially, like for the AdaBoost. The Gradient Boosting trains at first only one decision tree and, after each iteration, grows a new tree in order to minimize the loss function. Similarly to the AdaBoost, the process stops when the predefined number of trees has been created or when the loss function no longer improves.

8) XGBoost [14]: can be viewed as a better performing implementation of the Gradient Boosting algorithm, as it allows for faster computation and parallelization. For this reason it can yield better performance compared to the latter, and can be more easily scaled for the use with high dimensional data.

III. Case Study Design

We designed our empirical study as a case study based on the guidelines defined by Runeson and H´ost [16]. In this Section, we describe the empirical study including the goal and the research questions, the study context, the data collection and the data analysis.

A. Goal and Research Questions

As reported in Section 1, our goals are to analyze the fault-proneness of SonarQube rule violations (SQ-Violations) and the accuracy of the quality model provided by SonarQube. Based on the aforementioned goals, we derived the following three research questions (**RQs**).

RQ1 Which are the most fault-prone SQ-Violations?
In this RQ, we aim to understand whether the introduction of a set of SQ-Violations is correlated with the introduction of faults in the same commit and to prioritize the SQ-Violations based on their fault-proneness.
Our hypothesis is that a set of SQ-Violations should be responsible for the introduction of bugs.

RQ2 Are SQ-Violations classified as "bugs" by SonarQube more fault-prone than other rules?
Our hypothesis is that reliability rules ("bugs") should be more fault-prone that maintainability rules ("code smells") and security rules.

RQ3 What is the fault prediction accuracy of the SonarQube quality model based on violations classified as "bugs"?
SonarQube claims that whenever a violation is classified as a "bug", a fault will develop in the software. Therefore, we aim at analyzing the fault prediction accuracy of the rules that are classified as "bugs" by measuring their precision and recall.

B. Study Context

In agreement with the four companies, we considered open source projects available in the Technical Debt Dataset [17]. The reason for considering open source projects instead of their private projects is that not all the companies would have allowed us to perform an historical analysis of all their commits. Moreover, with closed source projects the whole process cannot be replicated and verified transparently.

For this purpose, the four companies selected together 21 out of 31 projects available, based on the ones that were more similar to their internal projects considering similar project age, size, usage of patterns used and other criteria that we cannot report for reason of NDA.

The dataset includes the analysis of each commit of the projects from their first commit until the end of 2015 with SonarQube, information on all the Jira issues, and a classification of the fault-inducing commits performed with the SZZ algorithm [18].

In Table I, we report the list of projects we considered together with the number of analyzed commits, the project size (LOC) of the last analyzed commits, the number of faults identified in the selected commits, and the total number of SQ-Violations.

TABLE I
The selected projects

Project Name	Analyzed commits	Last commit LOC	Faults	SonarQube Violations
Ambari	9727	396775	3005	42348
Bcel	1255	75155	41	8420
Beanutils	1155	72137	64	5156
Cli	861	12045	59	37336
Codec	1644	34716	57	2002
Collections	2847	119208	103	11120
Configuration	2822	124892	153	5598
Dbcp	1564	32649	100	3600
Dbutils	620	15114	21	642
Deamon	886	3302	4	393
Digester	2132	43177	23	4945
FileUpload	898	10577	30	767
Io	1978	56010	110	4097
Jelly	1914	63840	45	5057
Jexl	1499	36652	58	34802
Jxpath	596	40360	43	4951
Net	2078	60049	160	41340
Ognl	608	35085	15	4945
Sshd	1175	139502	222	8282
Validator	1325	33127	63	2048
Vfs	1939	59948	129	3604
Sum	**39.518**	**1,464,320**	**4,505**	**231,453**

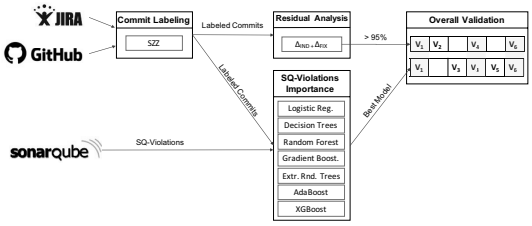


Fig. 1. The Data Analysis Process

C. Data Analysis

Before answering our RQs, we first executed the eight machine learning (ML) models, we compared their accuracy, and finally performed the residual analysis.

The next subsections describe the analysis process in details as depicted in Figure 1.

1) Machine Learning Execution: In this step we aim at comparing fault-proneness prediction power of SQ-Violations by applying the eight machine learning models described in Section II-B.

Therefore we aim at predicting the fault-proneness of a commit (labeled with the SZZ algorithm) by means of the SQ-Violations introduced in the same commit. We used the SQ-Violations introduced in each commits as independent variables (predictors) to determine if a commit is fault-inducing (dependent variable).

After training the eight models described in Section II-B, we performed a second analysis retraining the models using a *drop-column mechanism* [19]. This mechanism is a simplified variant of the exhaustive search [20], which iteratively tests every subset of features for their classification performance. The full exhaustive search is very time-consuming requiring 2^P train-evaluation steps for a P-dimensional feature space. Instead, we look only at dropping individual features one at a time, instead of all possible groups of features.

More specifically, a model is trained P times, where P is the number of features, iteratively removing one feature at a time, from the first to the last of the dataset. The difference in cross-validated test accuracy between the newly trained model and the baseline model (the one trained with the full set of features) defines the importance of that specific feature. The more the accuracy of the model drops, the more important for the classification is the specific feature.

The feature importance of the SQ-Violation has been calculated for all the machine learning models described, but we considered only the importance calculated by the most accurate model (cross-validated with all P features, as described in the next section), as the feature importances of a poor classifier are likely to be less reliable.

2) Accuracy Comparison: Apart from ranking the SQ-Violations by their importance, we first need to confirm the validity of the prediction model. If the predictions obtained from the ML techniques are not accurate, the feature ranking would also become questionable. To assess the prediction accuracy, we performed a 10-fold cross-validation, dividing the data in 10 parts, *i.e.,* we trained the models ten times always using 1/10 of the data as a testing fold. For each fold, we evaluated the classifiers by calculating a number of accuracy metrics (see below). The data related to each project have been split in 10 sequential parts, thus respecting the temporal order, and the proportion of data for each project. The models have been trained iteratively on group of data preceding the test set. The temporal order was also respected for the groups included in the training set: as an example, in fold 1 we used group 1 for training and group 2 for testing, in fold 2 groups 1 and 2 were used for training and group 3 for testing, and so on for the remaining folds.

As accuracy metrics, we first calculated precision and recall. However, as suggested by [21], these two measures present some biases as they are mainly focused on positive examples and predictions and they do not capture any information about the rates and kind of errors made.

The contingency matrix (also named confusion matrix), and the related f-measure help to overcome this issue. Moreover, as recommended by [21], the Matthews Correlation Coefficient (MCC) should be also considered to understand possible disagreement between actual values and predictions as it involves all the four quadrants of the contingency matrix.

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. The way these measures were calculated can be found in Table II.

TABLE II
ACCURACY METRICS FORMULAE

Accuracy Measure	Formula
Precision	$\frac{TP}{FP+TP}$
Recall	$\frac{TP}{FN+TP}$
MCC	$\frac{TP*TN-FP*FN}{\sqrt{(FP+TP)(FN+TP)(FP+TN)(FN+TN)}}$
f-measure	$2 * \frac{precision*recall}{precision+recall}$
TNR	$\frac{TN}{FP+TNe}$
FPR	$\frac{FP}{TN+FP}$
FNR	$\frac{FN}{FN+TP}$

TP: True Positive; TN: True Negative; FP: False Positive; FN: False Negative

Finally, to graphically compare the true positive and the false positive rates, we calculated the Receiver Operating Characteristics (ROC), and the related Area Under the Receiver Operating Characteristic Curve (AUC): the probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one.

In our dataset, the proportion of the two types of commits is not even: a large majority (approx. 90 %) of the commits were non-fault-inducing, and a plain accuracy score would reach high values simply by always predicting the majority class. On the other hand, the ROC curve (as well as the precision and recall scores) are informative even in seriously unbalanced situations.

3) SQ-Violations Residual Analysis: The results from the previous ML techniques show a set of SQ-Violations related with fault-inducing commits. However, the relations obtained in the previous analysis do not imply causation between faults and SQ-Violations.

In this step, we analyze which violations were introduced in the fault-inducing commits and then removed in the fault-fixing commits. We performed this comparison at the file level. Moreover, we did not consider cases where the same violation was introduced in the fault-inducing commit, removed, re-introduced in commits not related to the same fault, and finally removed again during the fault-fixing commit.

In order to understand which SQ-Violations were introduced in the fault-inducing commits (IND) and then removed in the fault-fixing commit (FIX), we analyzed the residuals of each SQ-Violation by calculating:

$$Residual = \Delta_{IND} + \Delta_{FIX}$$

where Δ_{IND} and Δ_{FIX} are calculated as:

Δ_{IND} = *#SQ-Violations introduced in the fault-inducing commit*

Δ_{FIX} =*#SQ-Violations removed in the fault-fixing commit*

Figure 2 schematizes the residual analysis.

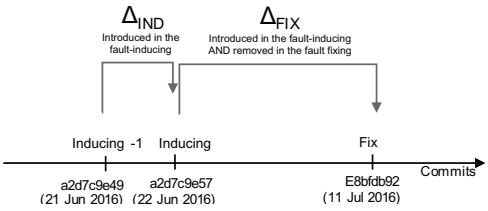


Fig. 2. Residuals Analysis

We calculated the residuals for each commit/fix pair, verifying the introduction of the SQ-Violation V_i in the fault-inducing commit (IND) and the removal of the violation in the fault-fixing commit (FIX). If Δ_{IND} was lower than zero, no SQ-Violations were introduced in the fault-inducing commit. Therefore, we tagged such a commit as not related to faults.

For each violation, the analysis of the residuals led us to two groups of commits:

- *Residual* > 0: The SQ-Violations introduced in the fault-inducing commits were not removed during the fault-fixing.
- *Residual* ≤ 0: All the SQ-Violations introduced in the fault-inducing commits were removed during the fault-fixing. If Residual < 0, other SQ-Violations of the same type already present in the code before the bug-inducing commit were also removed.

For each SQ-Violations, we calculated descriptive statistics so as to understand the distribution of residuals.

Then, we calculated the residual sum of squares (RSS) as:

$$RSS = \sum (Residual)^2$$

We calculated the percentage of residuals equal to zero as:

$$\frac{\#zero\_residuals}{\#residuals} * 100\%$$

Based on the residual analysis, we can consider violations where the percentage of zero residuals was higher than 95% as a valid result.

4) RQ1: Which are the most fault-prone SQ-Violations?: In order to analyze RQ1, we combined the results obtained from the best ML technique and from the residual analysis. Therefore, if a violation has a high correlation with faults but the percentage of the residual is very low, we can discard it from our model, since it will be valuable only in a limited number of cases. As we cannot claim a cause-effect relationship without a controlled experiment, the results of the residual analysis are a step towards the identification of this relationship and the reduction of spurious correlations.

5) RQ2: Are SQ-Violations classified as "bugs" by SonarQube more fault-prone than other rules?: The comparison of rules classified as "bugs" with other rules has been performed considering the results of the best ML techniques and the residual analysis, comparing the number of violations classified as "bug" that resulted to be fault-prone from RQ1. We expect bugs to be in the most faults-prone rules.

6) RQ3: What is the fault prediction accuracy of the SonarQube quality model based on violations classified as "bugs": Since SonarQube considers every SQ-Violation tagged as a "bug" as "something wrong in the code that will soon be reflected in a bug", we also analyzed the accuracy of the model provided by SonarQube.

In order to answer our RQ3, we calculated the percentage of SQ-Violations classified as "bugs" that resulted in being highly fault-prone according to the previous analysis. Moreover, we also analyzed the accuracy of the model calculating all the accuracy measures reported in Section III-C2.

D. Replicability

In order to allow the replication of our study, we published the raw data in the replication package [8].

IV. Results

In this work, we considered more than 37 billion effective lines of code and retrieved a total of 1,464,320 violations from 39,518 commits scanned with SonarQube. Table 1 reports the list of projects together with the number of analyzed commits and the size (in Lines of Code) of the latest analyzed commit. We retrieved a total of 4,505 faults reported in the issue trackers.

All the 202 rules available in SonarQube for Java were found in the analyzed projects. For reasons of space limitations, we will refer to the SQ-Violations only with their SonarQube id number (SQUID). The complete list of rules, together with their description is reported in the online replication package (file SonarQube-rules.xlsx). Note that in column "Type" MA means Major, Mi means Minor, CR means Critical, and BL means Blocker.

A. RQ1: Which are the most fault-prone SQ-Violations?

In order to answer this RQ, we first analyzed the importance of the SQ-Violations by means of the most accurate ML technique and then we performed the residual analysis.

1) SQ-Violations Importance Analysis: As shown in Figure 3, XGBoost resulted in the most accurate model among the eight machine learning techniques applied to the dataset. The 10-fold cross-validation reported an average AUC of 0.83. Table III (column RQ1) reports average reliability measures for the eight models.

Despite the different measures have different strengths and weaknesses (see Section III-C2), all the measures are consistently showing that XGBoost is the most accurate technique.

The ROC curves of all models are depicted in Table III while the reliability results of all the 10-folds models are available in the online replication package.

Therefore, we selected XGBoost as classification model for the next steps, and utilized the feature importance calculated applying the drop-column method to this classifier. The XGBoost classifier was retrained removing one feature at a time sequentially.

[8]Replication Package: https://figshare.com/s/fe5d04e39cb74d6f20dd

TABLE III
MODEL RELIABILITY

	RQ1 (Average between 10-fold validation models)							**RQ2**	**RQ3**
Measure	**Logistic Regr.**	**Decision Tree**	**Bagging**	**Random Forest**	**Extra Trees**	**AdaBoost**	**Gradient Boosting**	**XGBoost**	**SQ "bugs"**
Precision	0.417	0.311	0.404	0.532	0.427	0.481	0.516	0.608	0.086
Recall	0.076	0.245	0.220	0.156	0.113	0.232	0.192	0.182	0.028
MCC	0.162	0.253	0.279	0.266	0.203	0.319	0.300	0.318	0.032
f-measure	0.123	0.266	0.277	0.228	0.172	0.301	0.275	0.275	0.042
TNR	0.996	0.983	0.990	0.995	0.995	0.993	0.995	0.997	0.991
FPR	0.004	0.002	0.010	0.004	0.005	0.007	0.005	0.003	0.009
FNR	0.924	0.755	0.779	0.844	0.887	0.768	0.808	0.818	0.972
AUC	0.670	0.501	0.779	0.802	0.775	0.791	0.825	0.832	0.509

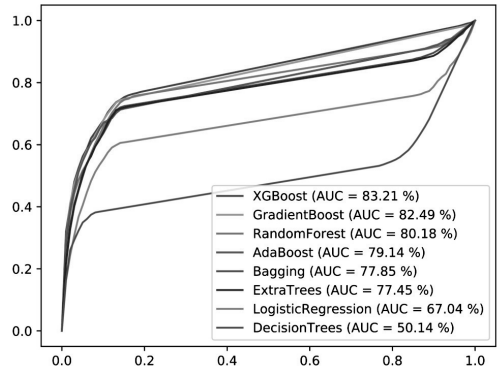


Fig. 3. ROC Curve (Average between 10-fold validation models)

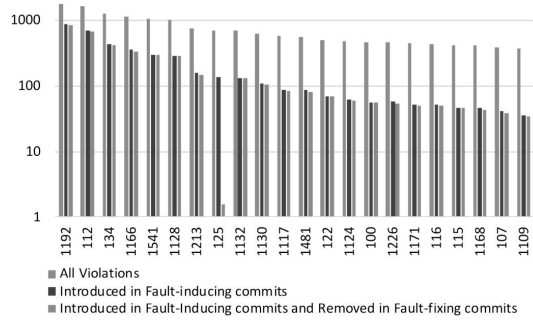


Fig. 4. Comparison of Violations introduced in fault-inducing commits and removed in fault-fixing commits

23 SQ-Violations have been ranked with an importance higher than zero by the XGBoost. In Table V, we report the SQ-Violations with an importance higher or equal than 0.01 % (coloumn "Intr. & Rem. (%)" reports the number of violations introduced in the fault-inducing commits AND removed in the fault-fixing commits). The remaining SQ-Violations are reported in the raw data for reasons of space. coloumn "Intr. & Rem. (%)" means

The combination of the 23 violations guarantees a good classification power, as reported by the AUC of 0.83. However, the drop column algorithm demonstrates that SQ-Violations have a very low individual importance. The most important SQ-Violation has an importance of 0.62%. This means that the removal of this variable from the model would decrease the accuracy (AUC) only by 0.62%. Other three violations have a similar importance (higher than 0.5%) while others are slightly lower.

2) Model Accuracy Validation: The analysis of residuals shows that several SQ-Violations are introduced in fault-inducing commits in more than 50% of cases. 32 SQ-Violations out of 202 had been introduced in the fault-inducing commits and then removed in the fault-fixing commit in more than 95% of the faults. The application of the XGBoost, also confirmed an importance higher than zero in 26 of these SQ-Violations. This confirms that developers, even if not using SonarQube, pay attention to these 32 rules, especially in case of refactoring or bug-fixing.

Table V reports the descriptive statistics of residuals, together with the percentage residuals = 0 (number of SQ-Violations introduced during fault-inducing commits and removed during fault-fixing commits).

Column "Res >95%", shows a checkmark (✓) when the percentage of residuals=0 was higher than 95%.

Figure 4 compares the number of violations introduced in fault-inducing commits, and the number of violations removed in the fault-fixing commits.

B. Manual Validation of the Results

In order to understand the possible causes and to validate the results, we manually analyzed 10 randomly selected instances for the first 20 SQ-Violations ranked as more important by the XGBoost algorithm.

The first immediate result is that, in 167 of the 200 manually inspected violations, the bug induced in the fault-inducing commit was not fixed by the same developer that induced it.

We also noticed that violations related to duplicated code and empty statements (eg. "method should not be empty") always generated a fault (in the randomly selected cases). When committing an empty method (often containing only a "TODO" note), developers often forgot to implement it and then used it without realizing that the method did not return the expected value. An extensive application of unit testing could definitely reduce this issue. However, we are aware that is is a very common practice in several projects. Moreover, SQ-Violations such as 1481 (unused private variable should

TABLE IV
SONARQUBE CONTINGENCY MATRIX (PREDICTION MODEL BASED ON SQ-VIOLATIONS CONSIDERED AS "BUG" BY SONARQUBE)

Predicted	Actual	
	IND	NOT IND
IND	32	342
NOT IND	1,124	38,020

be removed) and 1144 (unused private methods should be removed) unexpectedly resulted to be an issue. In several cases, we discovered methods not used, but expected to be used in other methods, resulted in a fault. As example, if a method A calls another method B to compose a result message, not calling the method B results in the loss of the information provided by B.

C. RQ2: Are SQ-Violations classified as "bugs" by SonarQube more fault-prone than other rules?

Out of the 57 violations classified as "bugs" by SonarQube, only three (squid 1143, 1147, 1764) were considered fault-prone with a very low importance from the XGBoost and with residuals higher than 95%. However, rules classified as "code smells" were frequently violated in fault-inducing commits. However, considering all the SQ-Violations, out of 40 the SQ-Violations that we identified as fault-prone, 37 are classified as "code smells" and one as security "vulnerability".

When comparing severity with fault proneness of the SQ-Violations, only three SQ-Violations (squid 1147, 2068, 2178) were associated with the highest severity level (blocker). However, the fault-proneness of this rule is extremely low (importance $<= 0.14\%$). Looking at the remaining violations, we can see that the severity level is not related to the importance reported by the XGBoost algorithm since the rules of different level of severity are distributed homogeneously across all importance levels.

D. RQ3: Fault prediction accuracy of the SonarQube model

"Bug" violations were introduced in 374 commits out of 39,518 analyzed commits. Therefore, we analyzed which of these commits were actually fault-inducing commits. Based on SonarQube's statement, all these commits should have generated a fault.

All the accuracy measures (Table III, column "RQ2") confirm the very low prediction power of "bug" violations. The vast majority of "bug" violations never become a fault. Results are also confirmed by the extremely low AUC (50.95%) and by the contingency matrix (Table IV). The results of the SonarQube model also confirm the results obtained in RQ2. Violations classified as "bugs" should be classified differently since they are hardly ever injected in fault-inducing commits.

V. THREATS TO VALIDITY

In this Section, we discuss the threats to validity, including internal, external, construct validity, and reliability. We also explain the different adopted tactics [22].

Construct Validity. As for construct validity, the results might be biased regarding the mapping between faults and commits. We relied on the ASF practice of tagging commits with the issue ID. However, in some cases, developers could have tagged a commit differently. Moreover, the results could also be biased due to detection errors of SonarQube. We are aware that static analysis tools suffer from false positives. In this work we aimed at understanding the fault proneness of the rules adopted by the tools without modifying them, so as to reflect the real impact that developers would have while using the tools. In future works, we are planning to replicate this work manually validating a statistically significant sample of violations, to assess the impact of false positives on the achieved findings. As for the analysis timeframe, we analyzed commits until the end of 2015, considering all the faults raised until the end of March 2018. We expect that the vast majority of the faults should have been fixed. However, it could be possible that some of these faults were still not identified and fixed.

Internal Validity. Threats can be related to the causation between SQ-Violations and fault-fixing activities. As for the identification of the fault-inducing commits, we relied on the SZZ algorithm [18]. We are aware that in some cases, the SZZ algorithm might not have identified fault-inducing commits correctly because of the limitations of the line-based diff provided by git, and also because in some cases bugs can be fixed modifying code in other location than in the lines that induced them. Moreover, we are aware that the imbalanced data could have influenced the results (approximately 90% of the commits were non-fault-inducing). However, the application of solid machine learning techniques, commonly applied with imbalanced data could help to reduce this threat.

External Validity. We selected 21 projects from the ASF, which incubates only certain systems that follow specific and strict quality rules. Our case study was not based only on one application domain. This was avoided since we aimed to find general mathematical models for the prediction of the number of bugs in a system. Choosing only one or a very small number of application domains could have been an indication of the non-generality of our study, as only prediction models from the selected application domain would have been chosen. The selected projects stem from a very large set of application domains, ranging from external libraries, frameworks, and web utilities to large computational infrastructures. The dataset only included Java projects. We are aware that different programming languages, and projects different maturity levels could provide different results.

Reliability Validity. We do not exclude the possibility that other statistical or machine learning approaches such as Deep Learning, or others might have yielded similar or even better accuracy than our modeling approach.

VI. RELATED WORK

In this Section, we introduced the related works analyzing literature on SQ-Violations and faults predictions.

TABLE V
SUMMARY OF THE MOST IMPORTANT SONARQUBE VIOLATIONS RELATED TO FAULTS (XGBOOST IMPORTANCE $> 0.2\%$)

SonarQube				**SZZ**		**Residuals**					**XG Boost**	**Res. >95%**
SQUID	Severity	Type	# Occ.	Intr. & Rem.(%)	Intr. in fault-ind	Mean	Max	Min	Stdev	RSS	Imp.	
S1192	CRITICAL	CS	1815	50,87	95,10	245,60	-861	2139	344,42	1726	0,66	✓
S1444	MINOR	CS	96	2,69	97,92	4,59	-7	73	10,34	94	0,62	✓
Useless Import Check	MAJOR	CS	1026	28,76	97,27	33,37	-170	351	61,58	998	0,41	✓
S00105	MINOR	CS	263	7,37	97,72	1,96	-13	32	10,22	257	0,41	✓
S1481	MINOR	CS	568	15,92	95,25	10,41	-6	83	14,60	541	0,39	✓
S1181	MAJOR	CS	200	5,61	97,00	8,87	0	88	13,43	194	0,31	✓
S00112	MAJOR	CS	1644	46,08	94,77	188,26	-279	1529	270,34	1558	0,29	
S1132	MINOR	CS	704	19,73	93,75	121,75	-170	694	134,91	660	0,24	
Hidden Field	MAJOR	CS	584	16,37	92,98	26,96	-12	143	29,42	543	0,23	
S134	CRITICAL	CS	1272	35,65	94,65	70,66	-66	567	88,07	1204	0,20	

Falessi et al. [3] studied the distribution of 16 metrics and 106 SQ-Violations in an industrial project. They applied a *What-if* approach with the goal of investigating what could happen if a specific SQ-Violation would not have been introduced in the code and if the number of faulty classes decrease in case the violation is not introduced. They compared four ML techniques applying the same techniques on a modified version of the code where they manually removed SQ-Violations. Results showed that 20% of faults were avoidable if the code smells would have been removed.

Tollin et al. [4] investigated if SQ-Violations introduced would led to an increase in the number of changes (code churns) in the next commits. The study was applied on two different industrial projects, written in C# and JavaScript. They reported that classes affected by more SQ-Violations have a higher change proneness. However they did not prioritize or classified the most change prone SQ-Violations.

Digkas et al. [23] studied weekly snapshots of 57 Java projects of the ASF investigating the amount of technical debt paid back over the course of the projects and what kind of issues were fixed. They considered SQ-Violations with severity marked as *Blocker, Critical, and Major*. The results showed that only a small subset of all issue types was responsible for the largest percentage of technical debt repayment. Their results thus confirm our initial assumption that there is no need to fix all issues. Rather, by targeting particular violations, the development team can achieve higher benefits. However, their work does not consider how the issues actually related to faults.

Falessi and Reichel [24] developed an open-source tool to analyze the technical debt interest occurring due to violations of quality rules. Interest is measured by means of various metrics related to fault-proneness. They use SonarQube rules and uses linear regression to estimate the defect-proneness of classes. The aim of MIND is to answer developers' questions like: is it worth to re-factor this piece of code? Differently than in our work, the actual type of issue causing the defect was not considered.

Codabux and Williams [25] propose a predictive model to prioritize technical debt. They extracted class-level metrics for defect- and change-prone classes using *Scitool Understanding* and *Jira Extracting Tool* from Apache Hive and determined significant independent variables for defect- and change-prone classes, respectively. Then they used a Bayesian approach to build a prediction model to determine the "technical debt proneness" of each class. Their model requires the identification of "technical debt items", which requires manual input. These items are ultimately ranked and given a risk probability by the predictive framework.

Saarimäki investigated the diffuseness of SQ-violations in the same dataset we adopted [26] and the accuracy of the SonarQube remediation time [27].

Regarding other code quality rules detection, 7 different machine learning approaches (Random Forest, Naive Bayes, Logistic regression, IBl, IBk, VFI, and J48) [28] were successfully applied on 6 code smells (Lazy Class, Feature Envy, Middle Man Message Chains, Long Method, Long Parameter Lists, and Switch Statement) and 27 software metrics (including Basic, Class Employment, Complexity, Diagrams, Inheritance, and MOOD) as independent variables.

Code smells detection was also investigated from the point of view of how the severity of code smells can be classified through machined learning models [29] such as J48, JRip, Random Forest, Naive Bayes, SMO, and LibSVM with best agreement to detection 3 code smells (God Class, Large Class, and Long Parameter List).

VII. DISCUSSION AND CONCLUSION

SonarQube classifies 57 rules as "bugs", claiming that they will sooner or later they generate faults. Four local companies contacted us to investigate the fault prediction power of the SonarQube rules, possibly using machine learning, so as to understand if they can rely on the SonarQube default rule-set or if they can use machine learning to customize the model more accurately.

We conducted this work analyzing a set of 21 well-known open source project selected by the companies, analyzing the presence of all 202 SonarQube detected violations in the complete project history. The study considered 39,518 commits, including more than 38 billion lines of code, 1.4 million violations, and 4,505 faults mapped to the commits.

To understand which sq-violations have the highest fault-proneness, we first applied eight machine learning approaches

to identify the sq-violations that are common in commits labeled as fault-inducing. As for the application of the different machine learning approaches, we can see an important difference in their accuracy, with a difference of more than 53% from the worst model (Decision Trees AUC=47.3%±3%) and the best model (XGBoost AUC=83.32%±10%). This confirms also what we reported in Section II-B: ensemble models, like the XGBoost, can generalize better the data compared to Decision Trees, hence it results to be more scalable. The use of many *weak* classifiers, yields an overall better accuracy, as it can be seen by the fact that the *boosting* algorithms (AdaBoost, GradientBoost, and XGBoost) are the best performers for this classification task, followed shortly by the Random Forest classifier and the ExtraTrees.

As next step, we checked the percentage of commits where a specific violation was introduced in the fault-inducing commit and then removed in the fault-fixing commit, accepting only those violations where the percentage of cases where the same violations were added in the fault-inducing commit and removed in the fault-fixing commit was higher than 95%.

Our results show that 26 violations can be considered fault-prone from the XGBoost model. However, the analysis of the residuals showed that 32 sq-violations were commonly introduced in a fault-inducing commit and then removed in the fault-fixing commit but only two of them are considered fault-prone from the machine learning algorithms. It is important to notice that all the sq-violations that are removed in more than 95% of cases during fault-fixing commits are also selected by the XGBoost, also confirming the importance of them.

When we looked at which of the sq-violations were considered as fault-prone in the previous step, only four of them are also classified as (”bugs”) by SonarQube. The remaining fault-prone sq-violations are mainly classified as ”code smells” (SonarQube claims that ”code smells” increase maintenance effort but do not create faults). The analysis of the accuracy of the fault prediction power of the SonarQube model based on ”bugs” showed an extremely low fitness, with an AUC of 50.94%, confirming that violations classified as ”bugs” almost never resulted in a fault.

An important outcome is related to the application of the machine learning techniques. Not all the techniques performed equally and XGBoost was the most more accurate and fastest technique in all the projects. Therefore, the application XGBoost to historical data is a good alternative to the manual tuning of the model, where developers should select which rules they believe are important based on their experience.

The result confirmed the impression of the developers of our companies. Their developers still consider it very useful to help to develop clean code that adhere to company standards, and that help new developers to write code that can be easily understood by other developers. Before the execution of this study the companies were trying to avoid to violate the rules classifies as bugs, hoping to reduce fault proneness. However, after the execution of this study, the companies individually customized the set of rules considering only coding standards aspects and rules classified as ”security vulnerabilities”. The main result for the companies is that they will need to invest in the adoption of other tools to reduce the fault proneness and therefore, we will need to replicate this work considering other tools such as FindBugs, PMD but also commercial tools such as Coverity Scan, Cast Software and others.

Based on the overall results, we can summarize the following lessons learned:

Lesson 1: SonarQube violations are not good predictors of fault-proneness if considered individually, but can be good predictors if considered together. Machine learning techniques, such as XGBoost can be used to effectively train a customized model for each company.

Lesson 2: SonarQube violations classified as ”bugs” do not seem to be the cause of faults.

Lesson 3: SonarQube violation severity is not related to the fault-proneness and therefore, developers should carefully consider the severity as decision factor for refactoring a violation.

Lesson 4: Technical debt should be calculated differently, and the non-fault prone rules should not be accounted as ”fault-prone” (or ”buggy”) components of the technical debt while several ”code smells” rules should be carefully considered as potentially fault-prone.

The lessons learned confirm our initial hypothesis about the fault-proneness of the SonarQube violations. However, we are not claiming that SonarQube violations are not harmful in general. We are aware that some violations could be more prone to changes [3], decrease code readability, or increase the maintenance effort.

Our recommendation to companies using SonarQube is to customize the rule-set, taking into account which violations to consider, since the refactoring of several sq-violations might not lead to a reduction in the number of faults. Furthermore, since the rules in SonarQube constantly evolve, companies should continuously re-consider the adopted rules.

Research on technical debt should focus more on validating which rules are actually harmful from different points of view and which will account for a higher technical debt if not refactored immediately.

Future works include the replication of this work considering the severity levels of SonarQube rules and their importance. We are working on the definition of a more accurate model for predicting TD [30] Moreover, we are planning to investigate whether classes that SonarQube identify as problematic are more fault-prone than those not affected by any problem. Since this work did not confirmed the fault proneness of SonarQube rules, the companies are interested in finding other static analysis tool for this purpose. Therefore, we are planning to replicate this study using other tools such as FindBugs, Checkstyle, PMD and others. Moreover, we will focus on the definition of recommender systems integrated in the IDEs [31][32], to alert developers about the presence of potential problematic classes based on their (evolution of) change- and fault-proneness and rank them based on the potential benefits provided by their removal.

REFERENCES

[1] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C. Gall, and Andy Zaidman. How Developers Engage with Static Analysis Tools in Different Contexts. In *Empirical Software Engineering*, 2019.

[2] Valentina Lenarduzzi, Alberto Sillitti, and Davide Taibi. A survey on code analysis tools for software maintenance prediction. In *6th International Conference in Software Engineering for Defence Applications*, pages 165–175. Springer International Publishing, 2020.

[3] D. Falessi, B. Russo, and K. Mullen. What if i had no smells? *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 78–84, Nov 2017.

[4] F. Arcelli Fontana I. Tollin, M. Zanoni, and R. Roveda. Change prediction through coding rules violations. *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pages 61–64, 2017.

[5] Valentina Lenarduzzi, Alberto Sillitti, and Davide Taibi. Analyzing forty years of software maintenance models. In *39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 146–148, Piscataway, NJ, USA, 2017. IEEE Press.

[6] M. Fowler and K. Beck. Refactoring: Improving the design of existing code. *Addison-Wesley Longman Publishing Co., Inc.*, 1999.

[7] D. R. Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society. Series B (Methodological)*, 20(2):215–242, 1958.

[8] Leo Breiman, Jerome Friedman, Charles J. Stone, and R.A. Olshen. *Classification and regression trees Regression trees*. Chapman and Hall, 1984.

[9] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 8 1996.

[10] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[11] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine Learning*, 63(1):3–42, 4 2006.

[12] Yoav Freund and Robert E Schapire. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 8 1997.

[13] Jerome H. Friedman. Greedy Function Approximation: A Gradient Boosting Machine.

[14] Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In *22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16*, pages 785–794, New York, New York, USA, 2016. ACM Press.

[15] Robert E. Schapire. The Strength of Weak Learnability. *Machine Learning*, 5(2):197–227, 1990.

[16] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg.*, 14(2):131–164, 2009.

[17] Valentina Lenarduzzi, Nyyti Saarimäki, and Davide Taibi. The technical debt dataset. In *15th conference on PREdictive Models and data analycs In Software Engineering*, PROMISE '19, 2019.

[24] D. Falessi and A. Reichel. Towards an open-source tool for measuring and visualizing the interest of technical debt. In *7th International Workshop on Managing Technical Debt (MTD)*, pages 1–8, 2015.

[18] and A. Zeller J. Śliwerski, T. Zimmermann. When do changes induce fixes? In *International Workshop on Mining Software Repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.

[19] Parr Terence, Turgutlu Kerem, Csiszar Christopher, and Howard Jeremy. Beware default random forest importances. http://explained.ai/rf-importance/index.html. Accessed: 2018-07-20.

[20] Hyunjin Yoon, Kiyoung Yang, and Cyrus Shahabi. Feature subset selection and feature ranking for multivariate time series. *IEEE transactions on knowledge and data engineering*, 17(9):1186–1198, 2005.

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

[22] R.K. Yin. *Case Study Research: Design and Methods, 4th Edition (Applied Social Research Methods, Vol. 5)*. SAGE Publications, Inc, 4th edition, 2009.

[23] G. Digkas, M. Lungu, P. Avgeriou, A. Chatzigeorgiou, and A. Ampatzoglou. How do developers fix issues and pay back technical debt in the apache ecosystem? In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, volume 00, pages 153–163, March 2018.

[25] B.J. Williams Z. Codabux. Technical debt prioritization using predictive analytics. In *38th International Conference on Software Engineering Companion*, ICSE '16, pages 704–706, New York, NY, USA, 2016. ACM.

[26] Nyyti Saarimäki, Valentina Lenarduzzi, and Davide Taibi. On the diffuseness of code technical debt in open source projects of the apache ecosystem. *International Conference on Technical Debt (TechDebt 2019)*, 2019.

[27] N. Saarimaki, M.T. Baldassarre, V. Lenarduzzi, and S. Romano. On the accuracy of sonarqube technical debt remediation time. *SEAA Euromicro 2019*, 2019.

[28] N. Maneerat and P. Muenchaisri. Bad-smell prediction from software design model using machine learning techniques. In *8th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 331–336, May 2011.

[29] Francesca Arcelli Fontana and Marco Zanoni. Code smell severity classification using machine learning techniques. *Know.-Based Syst.*, 128(C):43–58, July 2017.

[30] Valentina Lenarduzzi, Antonio Martini, Davide Taibi, and Damian Andrew Tamburri. Towards surgically-precise technical debt estimation: Early results and research roadmap. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, MaLTeSQuE 2019, pages 37–42, 2019.

[31] Andrea Janes, Valentina Lenarduzzi, and Alexandru Cristian Stan. A continuous software quality monitoring approach for small and medium enterprises. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, pages 97–100, 2017.

[32] Valentina Lenarduzzi, Christian Stan, Davide Taibi, Davide Tosi, and Gustavs Venters. A dynamical quality model to continuously monitor software maintenance. In *11th European Conference on Information Systems Management*, 2017.

PUBLICATION II

Just-in-time software vulnerability detection: Are we there yet?

F. Lomio, E. Iannone, A. De Lucia, F. Palomba, and V. Lenarduzzi

Journal of Systems and Software
DOI: 10.1016/j.jss.2022.111283

The Journal of Systems & Software 188 (2022) 111283

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Just-in-time software vulnerability detection: Are we there yet?☆

Francesco Lomio [a], Emanuele Iannone [b], Andrea De Lucia [b], Fabio Palomba [b], Valentina Lenarduzzi [c,*]

[a] Tampere University, Finland
[b] SeSa Lab — Department of Computer Science, University of Salerno, Italy
[c] University of Oulu, Finland

ARTICLE INFO

Article history:
Received 6 July 2021
Received in revised form 23 December 2021
Accepted 21 February 2022
Available online 25 February 2022

Keywords:
Software vulnerabilities
Machine learning
Empirical SE

ABSTRACT

Background: Software vulnerabilities are weaknesses in source code that might be exploited to cause harm or loss. Previous work has proposed a number of automated machine learning approaches to detect them. Most of these techniques work at release-level, meaning that they aim at predicting the files that will potentially be vulnerable in a future release. Yet, researchers have shown that a commit-level identification of source code issues might better fit the developer's needs, speeding up their resolution.
Objective: To investigate how currently available machine learning-based vulnerability detection mechanisms can support developers in the detection of vulnerabilities at commit-level.
Method: We perform an empirical study where we consider nine projects accounting for 8991 commits and experiment with eight machine learners built using process, product, and textual metrics.
Results: We point out three main findings: (1) basic machine learners rarely perform well; (2) the use of ensemble machine learning algorithms based on boosting can substantially improve the performance; and (3) the combination of more metrics does not necessarily improve the classification capabilities.
Conclusion: Further research should focus on just-in-time vulnerability detection, especially with respect to the introduction of smart approaches for feature selection and training strategies.

1. Introduction

Software security plays a crucial role in modern software development (Dowd et al., 2006). In software engineering terms, this has to do with the implementation of programs that can continue working under malicious circumstances (McGraw, 2004). Specifically, the source code should be designed to be resilient to external attacks: unfortunately, software vulnerabilities represent threats to security that may potentially be exploited by externals to cause loss of data, privilege escalation, race conditions, and other undesired effects that may affect the source code (Decan et al., 2018; Plate et al., 2015).

The research community has been addressing the problem of vulnerabilities under different perspectives, by proposing empirical studies aiming at characterizing them and their impact on source code (Finifter et al., 2013; Kim and Lee, 2018; Gonzalez et al., 2019), but more importantly by devising automated techniques that could support their identification (Hydara et al., 2015; McKinnel et al., 2019; Svacina et al., 2020).

Most of the approaches defined so far are based on source code and/or dynamic analysis (Li et al., 2016; Junjin, 2009; Sotirov, 2005; Trinh et al., 2014), symbolic execution (Li et al., 2013; Wang et al., 2009; Saxena et al., 2009), and fuzz-testing (Jiang et al., 2018; Wang et al., 2010). Some of them are also implemented within automated tools, e.g., SonarQube[1] and Eclipse Steady[2] that are widely adopted in practice (Vassallo et al., 2019).

Despite the research and industrial effort spent so far for building techniques and tools able to identify software vulnerabilities, the current solutions are still rarely effective in practice as they suffer from high false positive rates and/or scalability issues (Antunes and Vieira, 2010; Do et al., 2020; Johnson et al., 2013).

For these reasons, the research around vulnerability detection is still highly active. The last years have seen a growing interest in the application of artificial intelligence algorithms to software

☆ Editor: Earl Barr.

* Corresponding author.
E-mail addresses: francesco.lomio@tuni.fi (F. Lomio), eiannone@unisa.it (E. Iannone), adelucia@unisa.it (A. De Lucia), fpalomba@unisa.it (F. Palomba), valentina.lenarduzzi@oulu.fi (V. Lenarduzzi).

[1] SonarQube: https://www.sonarqube.org.
[2] Eclipse Steady: https://projects.eclipse.org/proposals/eclipse-steady.

https://doi.org/10.1016/j.jss.2022.111283

security (Chernis and Verma, 2018; Harer et al., 2018). Techniques based on machine learning, in particular, have reached promising results: starting from a set of vulnerability data collected through the change history analysis of files over previous releases of an application, these techniques train machine learning algorithms (e.g., Decision Tree) in order to predict the likelihood of new, unseen source code files to be affected by vulnerabilities in future releases (Ghaffarian and Shahriari, 2017).

While the performance reported in previous studies (Scandariato et al., 2014; Shin et al., 2011; Zimmermann et al., 2010; Theisen et al., 2015; Theisen and Williams, 2020) highlighted the suitability of machine learning approaches to predict vulnerabilities on future releases, it is still unclear how these approaches support developers in finding the exact location of the vulnerable code. As a matter of fact, traditional vulnerability predictions (Zimmermann et al., 2010; Theisen et al., 2015; Scandariato et al., 2014; Jimenez et al., 2019) would produce a large set of potentially vulnerable files or binaries, that should be manually inspected to establish the actual presence of the flaw, requiring a non-negligible amount of extra work. Moreover, such a task requires the selection of the most appropriate group of developers that can comprehend the rationale behind the last changes applied to the files. These limitations calls for novel solutions that better suits real case scenarios. Specifically, contemporary pull-based development practices (Gousios et al., 2014) make long-term recommendations, like those given by release-based predictions, not really suitable (Singer et al., 2010). Shorter-term recommendations, also known as *just-in-time* or *commit-level* predictions, should be preferred instead as they allow developers to receive an immediate feedback on the newly committed work and improve code quality while having the context of the modification still fresh in mind (Kamei et al., 2012; Kang et al., 2021). In addition, techniques able to work at this granularity become not only suitable at commit-time, but also while developers perform code review (Pascarella et al., 2018). As a consequence of these recent advances, vulnerability detection mechanisms should be re-assessed at a lower granularity.

Hence, this paper proposes an empirical investigation into the performance of just-in-time software vulnerability detection techniques. We mine nine Java projects available in the *National Vulnerability Database* (NVD)[3] in order to collect known vulnerabilities that affected them during their change history. Afterwards, we experiment with eight machine learning algorithms that we train using three different sets of features based on code, change, and textual metrics—both algorithms and features were previously employed in the context of vulnerability detection research. In addition, we employ a set of machine learning engineering steps (Tantithamthavorn and Hassan, 2018) aiming at improving the performance of the experimented models, such as dropping correlated features (O'brien, 2007), balancing the dataset (Pecorelli et al., 2020), and tuning hyperparameters (Bergstra and Bengio, 2012).

The results of our study reveal a number of findings. In the first place, we observe that basic machine learning algorithms, e.g., Support Vector Machine, have low performance when applied for the task of detecting vulnerabilities at commit-level, in contrast with previous work on vulnerability prediction. Moreover, the use of ensemble techniques do not necessarily provide benefits, even tough approaches based on boosting, like AdaBoost, seems promising and might be further investigated. Finally, we point out the limitations of existing metrics: for instance, we observe that previously devised textual metrics based on a raw *bag-of-words* source code representation lead the machine learners to have high variability and low prediction accuracy.

To sum up, we provide the following contributions:

1. Empirical evidence on the limited capabilities of commit-level vulnerability prediction models built using traditional techniques without proper setup;
2. A set of insights into the likely causes of failure of the current solutions, which forms the future research direction on the matter;
3. An online appendix.[4] providing all data and scripts used to conduct our study and that can be used by the research community to replicate and build upon our empirical study.

Structure of the paper. Section 2 discusses the related literature and motivates our work. In Section 3 we report the methodology employed to address our goals, while Section 4 analyzes the achieved results. The key implications of the study are presented in Section 5. The discussion of the threats to validity and how we mitigated them is reported in Section 6. Finally, Section 7 concludes the paper and outlines our future research agenda on the matter.

2. Related work

Research on software vulnerability prediction models (VPMs) mainly focused on identifying the best set of predictors correlated with the presence of vulnerabilities. Almost all works involved software product metrics directly computed on the source or binary files, such as size (e.g., Lines of Code) or structural metrics (Chidamber and Kemerer, 1994). Among these, complexity metrics (e.g., McCabe's Cyclomatic Complexity (McCabe, 1976)) are the ones that have received more attention. Shin and Williams (2008), Shin et al. (2011) and Shin and Williams (2011), in the context of Mozilla Firefox, found a strong positive correlation between the number of decisions in the code and the vulnerability-proneness of a file. Specifically, the VPMs – built using complexity metrics as predictors – achieve higher precision scores if the predictions are restricted to the top vulnerable files only, hinting that the files that were subject to many vulnerabilities in the past have high complexity values. This finding is further confirmed in other studies (Chowdhury and Zulkernine, 2011; Zimmermann et al., 2010; Walden et al., 2014; Jimenez et al., 2019). Similarly, coupling and cohesion metrics have been shown to be, respectively, positively and negatively correlated with vulnerabilities, corroborating the common wisdom that poor quality code raised the risk of introducing flaws (Chowdhury and Zulkernine, 2011). Moreover, Nguyen and Tran (2010) exploited a set of metrics extracted from the Component Dependency Graphs (CDG) to predict vulnerable C++ files in JS Engine of Firefox, observing an improvement in both accuracy and recall with respect to models built considering complexity metrics only. Neuhaus et al. (2007) found a correlation between the number of imports and functions with vulnerabilities in C functions, hinting their usefulness in a VPM. In particular, they devised a Support Vector Machine (SVM) relying on the number of past vulnerabilities on the imported C files in the context of Mozilla Firefox achieving a high precision of 70%, at the cost of a lower recall of 45%. Furthermore, Scandariato et al. (2014) were the first to investigate on the predictive power of text mining techniques. Namely, they used the *bag-of-words* method (Zhang et al., 2010; Harris, 1954) to extract the most frequent terms (i.e., words) from source code Java files to predict the presence of vulnerabilities on 20 Android apps. They managed to score a high performance in within-project predictions (i.e., making prediction on files belonging to the same project in which the model was trained), but failing in

[3] The National Vulnerability Database: https://nvd.nist.gov/.

[4] Our online appendix: https://figshare.com/s/0ef0f484a058e2297df4.

Table 1
Comparison with previous works concerning vulnerability prediction models. The focus is on the granularity level (i.e., the components that is subject to the predictions), the set of metrics used as predictors, the involved systems and the mined vulnerability data sources.

Study	Granularity	Predictors/Features	Context	Data sources
Neuhaus et al. (2007)	Function	Past vulnerable imports	Firefox	MFSA
Sultana et al. (2020)	Class/Method	Product metrics	4 Java systems	Vendor Advisories
Zimmermann et al. (2010)	Binary	Process and product metrics	Windows Vista	NVD
Theisen et al. (2015)	Binary/File	Past crashes, Process and product metrics	Windows 8	Microsoft Error Reporting
Theisen and Williams (2020)	Binary/File	Past crashes, Process and product metrics, Bag-of-words	Firefox	MFSA
Morrison et al. (2015)	Binary/File	Product metrics	Windows 7 and 8	NVD
Nguyen and Tran (2010)	File	Product metrics	Firefox JS Engine	MFSA, Bugzilla, NVD
Chowdhury and Zulkernine (2011)	File	Product metrics	Firefox	MFSA, Bugzilla
Smith and Williams (2011)	File	SQL Hotspots, LOC	WordPress, WakkaWiki	Bugzilla
Shin et al. (2011)	File	Process and product metrics	Firefox, RHEL	MFSA, RHSR, Bugzilla
Shin and Williams (2011)	File	Past faults, Process and Product metrics	Firefox	Bugzilla
Scandariato et al. (2014)	File	Bag-of-words	20 Android apps	Fortify SCA
Walden et al. (2014)	File	Product metrics, Bag-of-words	3 PHP systems	Vendor Security Advisories, NVD
Zhang et al. (2015)	File	Product metrics, Bag-of-words	3 PHP systems	Dataset from Walden et al. (2014).
Jimenez et al. (2019)	File	Bag-of-words, Process and product metrics	Linux, OpenSSL and Wireshark	NVD
Perl et al. (2015)	Commit	Process and product metrics	66 C/C++ systems	CVE DB
Yang et al. (2017)	Commit	Process and product metrics	Firefox	MFSA
Our study	Commit	Process and product metrics, bag-of-words	9 Java systems	NVD

cross-project scenario (i.e., making prediction on files not belonging to the projects in which the model was trained), as further confirmed by Walden et al. (2014). Later, Zhang et al. (2015) combined the above bag-of-words method with traditional product metrics, achieving higher F-measure value with respect to the VPM in Walden et al. (2014). On the other hand, Zimmermann et al. (2010) analyzed the impact of organizational (e.g., the number of developers) and code churn (i.e., the rate of changes applied to binaries) metrics to vulnerabilities in Windows Vista, achieving high precision but low recall, in line with the findings of later studies (Shin et al., 2011; Jimenez et al., 2019; Perl et al., 2015; Yang et al., 2015). Smith and Williams (2011) tested the usage of warnings of possible SQL Injections as predictors in two VPMs for WordPress and WakkaWiki, finding a positive correlation with many vulnerability types—other than SQL Injection. All the above findings are mixed together in the study of Theisen and Williams (2020), in which the authors claimed that the best prediction models are the one encompassing many different set of metrics (namely, product, process, text metrics and past faults).

Regarding the model selection, vulnerability prediction have been based on different supervised machine learning models, such as Decision Trees (Shin et al., 2011; Scandariato et al., 2014; Zhang et al., 2015; Jimenez et al., 2019; Theisen and Williams, 2020), Support Vector Machines (SVM) (Neuhaus et al., 2007; Nguyen and Tran, 2010; Zimmermann et al., 2010; Scandariato et al., 2014; Theisen et al., 2015; Perl et al., 2015; Morrison et al., 2015), Naïve Bayes (Nguyen and Tran, 2010; Shin et al., 2011; Scandariato et al., 2014; Morrison et al., 2015; Zhang et al., 2015; Theisen and Williams, 2020) and Random Forests (Shin et al., 2011; Scandariato et al., 2014; Walden et al., 2014; Morrison et al., 2015; Zhang et al., 2015; Jimenez et al., 2019; Theisen and Williams, 2020). Among these, Naïve Bayes resulted in higher recall values (which means lower false negative rate), while Random Forests regularly score high precision (meaning low false positive rate) in different contexts. Such models are also popular in similar tasks, e.g., defect (Shin and Williams, 2011; D'Ambros et al., 2012) and exploitability prediction (Bhatt et al., 2020).

Most studies have been conducted on predicting vulnerabilities at source code file level (Nguyen and Tran, 2010; Chowdhury and Zulkernine, 2011; Smith and Williams, 2011; Shin and Williams, 2008; Shin et al., 2011; Scandariato et al., 2014; Walden et al., 2014; Zhang et al., 2015; Jimenez et al., 2019), which means that the VPM tells whether a given file is or is not affected by a vulnerability. In such a scenario, the developers can invest their effort on inspecting and testing the problematic files with dedicated attention. The same concept is applied for VPMs working on binary files (Zimmermann et al., 2010; Theisen et al., 2015; Theisen and Williams, 2020; Morrison et al., 2015), which contain machine code produced by a compiler. Neuhaus et al. (2007) designed a tool, Vulture, that predicts the vulnerabilities in C/C++ functions, whereas Sultana et al. (2020) do this on Java methods, instead. To the best of our knowledge, only few works have considered the predictions at commit-level. Perl et al. (2015) devised a method for obtaining the vulnerability-contributing commit on 66 C/C++ open-source projects. They essentially relied on the `git blame` command that reaches the commits that changed last the deleted lines of a public fixing commit of known vulnerabilities reported in NVD. Then, they labeled the most blamed commit as a vulnerability-contributing commit. Finally, they trained a Support Vector Machine on this dataset, outperforming the detection capabilities of equivalent static analysis tools. The entire pipeline was replicated some years later by Riom et al. (2021), in which the authors, among other things, delve into the possibility to improve VPM provided by Perl et al. (2015) by experimenting on a different feature set containing metrics capturing more security-related aspects—e.g., the number of `sizeof` operators, which are known to be closely linked to improper sizing of dynamically-allocated buffers (CWE, 2006). However, they could not fully replicate the experiment (Perl et al., 2015) as the datasets and scripts were not available anymore, and the original paper did not provide sufficient detail on how to re-implement the features extraction step. For these reasons, Riom et al. could not provide a faithful comparison. Yang et al. (2017) considered the case of web vulnerabilities arising in Mozilla Firefox, and, using a large set of process and product metrics drawn by Kamei et al. (2012), they provided a VPM that achieved high precision (over 90%), at the cost of having a very low recall score (below 15%) at the best possible configuration.

Our work and contribution. Table 1 summarizes and compares the works in the vulnerability prediction field, other than highlighting the main differences of our contribution. Our research aims at shedding lights on the capabilities of a large variety of machine learning models for just-in-time vulnerability detection. Hence, with respect to most of the papers discussed, our study has a different level of granularity and aims at assessing whether and how the promising research on machine learning for vulnerability detection can be applied at commit-level.

In particular, our study can be considered complementary with respect to previous works by Perl et al. (2015) and Yang et al. (2017) that targeted a commit-level granularity. First, we exploited multiple machine learning algorithms with the aim of providing a broader overview of how effective these techniques are for the just-in-time vulnerability detection, instead of employing only a single learner (e.g., SVM or Random Forest). Then, we employed a set of techniques to improve the model performance, such as removing features exhibiting multicollinearity (O'brien, 2007), balancing the dataset (Pecorelli et al., 2020), and fine-tuning the model hyper-parameters (Bergstra and Bengio, 2012). Such techniques were not always considered in the past when building VPMs, as also pointed by Jimenez et al. (2019). We also considered the role of textual metrics, which have been shown as highly relevant by Scandariato et al. (2014). In particular, we wanted to assess whether the raw use of the textual metrics actually provides an improvement in terms of predictive performance when considered with other features, as show by Theisen and Williams (2020). Finally, we targeted a different programming language, like Java, which has its own peculiarities and, more importantly, vulnerabilities. Indeed, a large part of the current body of knowledge covered types of weaknesses strictly tied to the programming language, e.g., the Buffer Overflow (CWE, 2006) vulnerability predominantly affecting C/C++ code.

On the basis of these considerations, the main contributions of our study pose an additional ground for software engineering researchers working on the identification of vulnerabilities, who can exploit our results to understand and build upon the current limitations and challenges connected to the application of machine learning-based vulnerability detectors at commit-level.

3. Research methodology

In this section we provide a formulation of the design of our study according to the Goal-Question-Metric (GQM) paradigm (Basili et al., 1994). In Section 3.1 we define the goal of our study and the consequent research question. Then, we describe the context of our empirical study, i.e., the projects we selected (Section 3.2), the procedures behind the automated extraction of vulnerability-contributing commits (Section 3.3), and the computation of software metrics (Section 3.4). All these data are required to build the dataset exploited by our machine learning pipeline, for which we provide a detailed description (Section 3.5). We conclude the section by presenting the evaluation methods we employed to answer our research question (Section 3.6).

3.1. Goal and research question

The *goal* of this empirical study was to investigate the performance of machine learning methods when employed for the task of just-in-time vulnerability detection, with the *purpose* of assessing their suitability in a pull-based development scenario. The *perspective* is both of practitioners and researchers: the former are interested in understanding whether and to what extent machine learning-based vulnerability detectors can be used during their daily activities; the latter are interested in evaluating strengths, weaknesses, and challenges for the use of machine learning for just-in-time vulnerability detection and that can be investigated further in future research.

We analyzed how well different machine learners can identify commits contributing to vulnerabilities. In this respect, we were inspired by previous research on vulnerability prediction (Theisen and Williams, 2020) and assessed the impact of three families of software metrics on the performance of different machine learning algorithms. We asked:

Table 2
Summary of projects considered in this study. These statistics are related to the period before 8th Match 2021.

Project	#Commit	LOC	#Sample commits	#VCCs
Conversation	5,810	16,035	1000	10
Candlepin	8,646	30,875	300	3
Hawtio	8,354	3,705	1200	12
Jboss-Negotiation	299	505	191	2
Jenkins	25,867	29,080	4400	44
Jolokia	1 573	3,685	1100	11
Junrar	221	1,325	100	1
Litemall	990	3,500	100	1
Struts1-Forever	4,526	4,025	600	6
	56,286	**92,735**	**8991**	**90**

RQ. *How well do machine learning algorithms perform when employed in the context of just-in-time vulnerability detection?*

We set up a machine learning pipeline that implements well-established guidelines for to the creation of unbiased supervised learning techniques (Song et al., 2010; Tantithamthavorn and Hassan, 2018). As further explained in the next sections, we considered and mitigated common pitfalls related to feature selection, hyper-parameter configuration, data balancing, selection of performance metrics, and statistical tests. When designing and reporting our study, we adopted the guidelines by Wohlin et al. (2000) and followed the *ACM/SIGSOFT Empirical Standards* (Ralph et al., 2021).[5]

3.2. Context of the study

The *context* of the empirical study was composed of nine Java projects, whose main characteristics are reported in Table 2. These projects account for a total of 56,286 commits but, due to computational reasons, we randomly sampled 8991 of them (16% of the total commits). When selecting the commits to analyze, we made sure not to discard commits containing vulnerabilities, whose collection is explained later in Section 3.3.

More in general, we considered all the Java projects having public software vulnerability data stored on the *National Vulnerability Database* (NVD). This database was originally created by the U.S. NIST Computer Security Division (NIST, 2021) with the aim of collecting and disclosing known vulnerabilities affecting software systems and their causes. It includes a comprehensive set of publicly known vulnerabilities: each of them is described through CVE (Common Vulnerabilities and Exposure (MITRE, 2021)) records and is enriched with additional pieces of information such as external references, severity (computed using the Common Vulnerability Scoring System — CVSS), the related weakness type (Common Weakness Enumeration — CWE), and the known affected software configurations (Common Platform Enumerations — CPEs). NVD aggregates information from multiple data sources and is widely considered a reliable data source (Alhazmi et al., 2007; Huang et al., 2010; Zhang et al., 2011). As a matter of fact, vulnerability reports must fulfill a well-defined set of requirements [6] before being added into NVD. As an example, vendors requesting for the creation of a CVE record have to provide a prose description of the issue, containing enough information for readers to understand which are the

[5] Given the nature of our study and the currently available empirical standards, we followed the *"General Standard"* and *"Data Science"* definitions and guidelines.

[6] https://www.cve.org/ResourcesSupport/AllResources/CNARules#section_8-1_cve_record_information_requirements.

known products affected (e.g., application, operating system, or hardware). Such a description has to be supported by at least one accessible reference, e.g., a public mailing list. Moreover, a CVE describes one and only one independently fixable vulnerability, meaning that each record describes a single instance of an issue concerning a violation to the security policy of a product. This makes us confident enough about the validity and quality of the information contained in NVD.

Our focus on Java was motivated by the fact that previous research on software vulnerabilities did not extensively targeted this programming language (see Table 1): as such, our study can be considered as the first investigation of the capabilities of just-in-time detection approaches for the identification of known Java vulnerabilities. In addition, our choice was based on the availability of metrics that could characterize different aspects of Java source code, as well as the tools that could automate the data collection procedures.

3.3. Collecting vulnerability-contributing commits

When collecting software vulnerabilities, we mined data exploiting CVE-Search,[7] an open-source tool that imports the entire set of CVE records from NVD into a MongoDB database for easier search and processing. We performed some additional filtering steps with the aim of removing incomplete/incorrect data that might have biased our conclusions: (1) we discarded CVEs that reported commits pointing to more than one GitHub repository, since we could not establish which project was involved in the first place; (2) we filtered out vulnerabilities whose fixes were marked as `merge` commits, as these do not apply any modification in the project history but simply incorporate the changes from a branch into another, i.e., we could not consider them as actual patches since we were interested in getting precise information about the time when fixes were added into the history rather than the time when they were sent into the main branch. After these filtering, we ended up with a total of 27 vulnerabilities (CVEs) of 12 different types (CWEs).

Afterwards, we implemented a mining procedure based leveraging the well-known SZZ algorithm (Sliwerski et al., 2005) to fetch the vulnerability-contributing commits (VCCs) (Meneely et al., 2013), i.e., the commits that are likely to have contributed to the introduction of a vulnerability. To this purpose, we started from vulnerability-fixing commits that we mined from NVD. Specifically, for each file f_i touched by the fixing commit c_{fix}, our algorithm runs the `git-diff` command to extract the list of modified lines in f_i with respect to the previous commit c_{fix-1}; then, it runs the `git-blame` command on the deleted lines in order to retrieve the commits where these were changed last. We consider these commits as VCCs of the vulnerability fixed in c_{fix}. As a result, for each vulnerability we obtain a set of VCCs as more than one commit might contribute to its introduction.

To improve the precision of this procedure, we applied some additional adjustments to reduce the risk of catching false positive VCCs. We excluded files from c_{fix} that were (1) non-source Java files, (2) test classes, (3) build files, (4) documentation and blob resources (the entire list of blacklisted files is available in our online appendix[4]). We also filtered out the VCCs that appeared as `merge` commits, as they do not report any *actual* modifications to the project's history—indeed, we were interested in the moments in which the patches were added in the history for the first time, not when they were merged into the main branch (Git, 2021). Finally, we managed the cases where the fixing commit c_{fix} consisted only of added lines. In these situations, there are no lines to blame and we assumed that the files involved in the commit were born vulnerable: as such, we marked the commits that introduced the files as vulnerable. Overall, we managed to obtain a total of 90 distinct VCCs among the nine projects—a detailed list reporting these commits is available in our online appendix.[4] Whether or not a commit contributes to a vulnerability represents the *dependent variable* of the models built, i.e., the information that we aimed at predicting using machine learning techniques.

[7] https://github.com/cve-search/cve-search.

3.4. Collecting software metrics

Once we had collected vulnerability data, we focused on the *independent variables*. In this respect, we exploited three families of metrics that were investigated in previous studies on software vulnerability detection: process, product, and textual features. The detail of each of these metrics, along with the description and the rational behind their usage, is described in Table 3.

With respect to process metrics, we considered different aspects previously treated in vulnerability research (Shin and Williams, 2008; Zimmermann et al., 2010; Shin et al., 2011; Perl et al., 2015; Yang et al., 2015) and able to characterize the change history of the projects, like the churn metrics (concerning added and deleted lines, methods, conditions, method calls, and assignments), the extent of contribution made by the committing author (i.e., the developer implementing the change), the number of files involved in the commit, the scattering of the changes, the number of previous changes and author of the files, etc. To compute these metrics, we developed our own tool, available in our online appendix.[4] It is worth point out that most of these metrics concerns metadata directly extracted from the commit metadata – e.g., the number of days between the commit date and the project creation date – while two of them, namely *Mean Days Since Creation* and *Mean of Past Changes*, were obtained by analyzing the `git` metadata related to each file involved in the commit. For these metrics, we aggregated the values obtained from each valid file (with the same filters used in Section 3.3) using the mean operator to bring them at commit-level, enabling their use as predictors for the machine learning models.

As for product metrics, we took into account the Chidamber and Kemerer (1994), a set of well-known Object-Oriented metrics able to quantify different structural properties of the source code, such as cohesion and coupling. Similarly, to process metrics, we exploited an ad-hoc tool, available in our online appendix,[4] able to extract structural metrics from a given parsable Java files. To reach our goals, we run it against all the Java files involved in the commits to extract the traditional set of CK metrics (listed in Table 3); afterwards, we computed the mean of the metric values to bring them at commit level, similarly to what was done by Yang et al. (2017) for the SLOC metric.

Finally, we extracted the textual features experimented by Scandariato et al. (2014). For each commit, we selected the valid files (which underwent to the usual filters described in Section 3.3) so that we could make our document corpus. Then, we extracted its *bag-of-words* (Zhang et al., 2010; Harris, 1954), which is a compact representation of the documents in the corpus through the number of occurrences of the words (a.k.a. terms) appearing in the entire corpus (which constitute the *vocabulary*). Namely, a file is represented as a vector of M integers, each representing the counting of the M words appearing in the vocabulary. At this point, the bags-of-words of the files involved in a commit were summed together, so that any commit could have its own bag-of-words made of the total number of times each words appeared in the modified valid files only. We treated each term as an independent variable for our models. To remove any noise that could damage the models performance (De Lucia et al., 2013; Silva and Ribeiro, 2003), we filtered out the high-frequency words—removing the ones appearing in more than 80%

documents, as they add poor information to the text; in addition, we also dropped low-frequency words, appearing in less than 5% documents, to reduce the dimensionality of the feature space, which was shown to improve the training process (Martins, 2003; Joachims, 1998). All in all, we ended up with 1318 distinct tokens, each encoded as a numeric feature. In addition, following the approach adopted by Perl et al. (2015), we extracted the bag-of-words of the sole commits' patches to count the terms involved in the actual change, without considering the unaffected code areas. Specifically, for each commit we obtained the bag-of-words of the added lines only, and the bag-of-words of the deleted lines sharing the same vocabulary. Then, we computed the absolute difference between the two vectors, so that we could obtain the number of times each term was involved, either in an addition or deletion, in the actual patch. Also in this case we filtered out high- and low-frequency words using the same filters used for the entire files. In this case, we ended up with 128 distinct tokens, encoded as 128 integer features. It is worth remarking that we did not compute the overlap between these 128 terms and the 1318 extracted in the previous step, as they originate from two different corpuses (i.e., files and patches). Thus, we considered a total of 1446 tokens. These two approaches were implemented in our own scripts, which relied on Scikit-Learn's `CountVectorizer` class,[8] and made it publicly available into our appendix.[4]

3.5. Setting the machine learning methods

After having collected dependent and independent variables to be used, we configured the machine learning models to detect vulnerable commits. The design of our machine learning pipeline is described hereafter.

3.5.1. Design of the models

As we had collected different families of metrics, we could experiment with various models. We first devised three supervised techniques that relied, individually, on *product*, *process*, and *textual* metrics to predict the proneness of the commits to be vulnerable: in this way, we could assess the contribution given by each metrics set. Afterwards, we started combining them by adopting a stepwise method: we created models based on *product+process*, *product+textual*, and *process+textual* features. Finally, we also considered the model using all the features together. As a consequence, we designed and experimented with seven different combinations of features.

3.5.2. Selection of the classifier

We treated the problem as a binary classification task: determining whether a commit contributed to a vulnerability or not. As discussed in Section 2, the related literature did not pose conclusive results on the machine learning algorithms that are more suitable for the classification of software vulnerabilities. For this reason, we experimented with the following eight learning algorithms:

Support Vector Machine (SVM) (Cortes and Vapnik, 1995). This is a statistical model that constructs the best hyper-plane out of the infinite possibilities in a N-dimensional space—with N being the number of features. The best hyper-plane is capable of distinctly separate the data points, having the maximum margin (namely the largest distance to the nearest training data points of any class).

KNearestNeighbors (KNN) (Zhang, 2016). This a non-parametric technique that classifies the samples using the dataset alone (i.e., without building a model). The classification is made as a majority vote, i.e., based on the class of the majority of its k nearest neighbors data points.

Decision Tree (Breiman et al., 1984). This is a classifier with a tree-like structure, characterized by multiple *nodes* and *leaf*. The nodes are linked through branches, representing a test. The output is given by the decision path taken. The decision tree is structured as an if-then-else diagram: given an input variable (root node), it leads to multiple sub nodes through branches. The process is iterated until the output (leaves) is reached.

Random Forest (Breiman, 2001). This is an ensemble technique that helps to overcome the overfitting issues of the decision tree. Ensemble means that this model uses a set of *weak* classifiers (decision trees in this case) to solve the assigned problem. Each individual tree is generated using a random subset of samples in the dataset. To reduce the correlation between the individual trees, the splitting point is chosen using a random subset of the dataset, *without replacement*. Using this method, a Random Forest is able to better generalize the data and reduce the overfitting problem faced by other classifiers.

Extremely Randomized Trees (Geurts et al., 2006). Extra-Trees adds a further randomization to the Random Forest, as each node of the weak classifiers is split randomly. This means that instead of relying to specific metrics for choosing the optimal splitting point, this model randomly generates a series of splits and choose the one which gives the best result. This characteristic allows the model to be less computationally expensive compared to the others, while maintaining high generalization capabilities.

AdaBoost (Freund and Schapire, 1997). This is an ensemble model based on *boosting* (Schapire, 1990), in which each individual tree is trained in a *sequential* fashion. Initially, a single decision tree is created and the same weight is assigned to all samples in the training set. Progressively, the weights are increased for the misclassified samples and another tree is generated. The whole process continues until a predefined number of trees has been generated or the accuracy of the model cannot be improved anymore. With respect to the other ensemble models, AdaBoost is less prone to overfitting.

Gradient Boosting (Friedman, 2001). As AdaBoost, it uses an ensemble of individual trees which are generated sequentially. A tree is generated after each iteration to minimize a differential loss function. The process stops when the predefined number of trees has been created or when the loss function no longer improves.

XGBoost (Chen and Guestrin, 2016). An improved implementation of Gradient Boosting algorithm, allowing faster computation and parallelization.

The choice of focusing on these classifiers was driven by our willingness to investigate the classification performance of a large variety of algorithms, including ensemble methods. It is worth remarking that in our research we were interested in benchmarking *narrow* artificial intelligence techniques (Coppin, 2004): the evaluation of other approaches belonging to the category of *strong* artificial intelligence, e.g., deep learning, is part of our future research agenda.

3.5.3. Preprocessing steps

As recommended in literature (Tantithamthavorn and Hassan, 2018), we performed a number of steps aimed at building a machine learning pipeline that could avoid bias in the interpretation of the results. In the first place, we applied a *feature selection* in order to avoid multi-collinearity (O'brien, 2007). This step was required to remove correlated metrics that provide the machine learners with the same (or similar) information and that might cause them to not being able to derive the correct explanatory meaning of the features. In this respect, we exploited

[8] https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html.

Table 3
List of metrics extracted from each commit in the dataset, used as independent variables (features) for the machine learners. The table reports a description, the rationale behind their selection, and related works in which they have been used for VPMs.

Name	Description	Rationale	VPMs
Process metrics			
Added lines	The number of lines added in the commit.	A high amount of added lines indicates a large commit, which has a higher risk of introducing defects (Meneely and Williams, 2012; Nagappan and Ball, 2005; Graves et al., 2000) or vulnerabilities (Zimmermann et al., 2010; Meneely et al., 2013; Shin et al., 2011).	Zimmermann et al. (2010), Shin and Williams (2011), Perl et al. (2015) and Yang et al. (2017)
Deleted lines	The number of lines removed in the commit.	Same as *Added Lines*.	Zimmermann et al. (2010), Shin and Williams (2011), Perl et al. (2015) and Yang et al. (2017)
Added methods	The number of new functions/methods added in the commit.	New functions or methods may add new security check or increase the attack surface (Piancó et al., 2016; Yang et al., 2017).	Perl et al. (2015) and Yang et al. (2017)
Deleted methods	The number of removed functions/methods in the commit.	Deleting security-critical functions or methods may remove security checks or reduce the attack surface (Piancó et al., 2016; Yang et al., 2017).	Perl et al. (2015) and Yang et al. (2017)
Modified methods	The number of changed functions/methods in the commit.	The removal of security-critical functions or methods may modify the security profile (Piancó et al., 2016; Yang et al., 2017).	Perl et al. (2015) and Yang et al. (2017)
Added conditions	The number of added conditional expressions in the commit.	Same as *Added Methods*.	Yang et al. (2017) and Riom et al. (2021)
Removed conditions	The number of removed conditional expressions in the commit.	Same as *Removed Methods*.	Yang et al. (2017) and Riom et al. (2021)
Added method calls	The number of added function or method call in the commit.	Same as *Added Methods*.	Yang et al. (2017) and Riom et al. (2021)
Removed method calls	The number of removed function or method call in the commit.	Same as *Removed Methods*.	Yang et al. (2017) and Riom et al. (2021)
Added assignments	The number of assignments added in the commit.	Adding new assignments may improve or drop security constraints (Piancó et al., 2016; Yang et al., 2017).	Yang et al. (2017) and Riom et al. (2021)
Removed assignments	The number of assignments removed in the commit.	Same as *Added Assignments*.	Yang et al. (2017) and Riom et al. (2021)
Mean days since creation	The mean number of days elapsed from the creation dates of each modified file to the commit date.	The "age" of each file could be correlated with the presence (or absence Graves et al., 2000) of vulnerabilities.	N/A
Mean of past changes	The mean number of previous changes (i.e., commits) of each touched file.	A file that was changed many times is more prone to defects (Nagappan and Ball, 2005) and vulnerabilities (Zimmermann et al., 2010; Shin et al., 2011; Perl et al., 2015).	Zimmermann et al. (2010), Shin et al. (2011), Perl et al. (2015) and Yang et al. (2017)
Past different authors	The size of the set of distinct authors that touched the files modified in the commit.	A file touched by many different developers is more prone to defects (Nagappan and Ball, 2005) and vulnerabilities (Zimmermann et al., 2010; Shin et al., 2011; Perl et al., 2015; Meneely and Williams, 2012; Meneely et al., 2013).	Zimmermann et al. (2010), Shin et al. (2011), Perl et al. (2015) and Yang et al. (2017)
Author past contributions	The number of commits done by the author before the commit.	Inexpert developers may involuntarily contribute to vulnerabilities (Meneely et al., 2013).	Yang et al. (2017)
Author past contributions ratio	*Author Past Contributions* divided by the total number of commits made to the entire project.	Same as *Author Past Contributions*.	Perl et al. (2015)
Author 30-days past contributions	The number of commits done by the author within 30 days before the commit date.	Same as *Author Past Contributions*.	N/A
Author 30-days past contributions ratio	*Author 30-days Past Contributions* divided by the total number of commits made 30 days before the commit date.	Same as *Author Past Contributions*.	N/A
Author workload	The amount of work that an author has invested within a 30-days time window. Given a commit x performed on date d, we considered the distribution of commits done by the developers involves within 30 days before d, scaled to [0..1] range, and assigned its percentile value (Tufano et al., 2017b).	A developer with high workload could implement poor quality code (Tufano et al., 2017b), defects (Nagappan et al., 2008) or vulnerabilities (Zimmermann et al., 2010).	N/A
Days after creation	The number of days elapsed from the project's repository creation (i.e., the first commit) date to the commit date.	The "age" of the repository has an impact on the general code quality (Tufano et al., 2017b) and the introduction of errors (Parnas, 1994).	N/A

(*continued on next page*)

Table 3 (*continued*).

Name	Description	Rationale	VPMs
Fix	Whether or not the commit had the goal to fix an issue or a defect. This is done by looking at specific keywords in the commit message (reported in our online appendix).	Fix commits may cause collateral damages of introducing new bugs (Kamei et al., 2012) or vulnerabilities (Bandara et al., 2020).	Yang et al. (2017)
Touched files	The number of files modified in the commit, excluding the irrelevant ones (test, documentation, build, and blob files).	A commit touching many files lacks of cohesion and may have a higher risk of introducing defects (Perl et al., 2015; Herzig et al., 2016; Yang et al., 2017).	Yang et al. (2017) and Riom et al. (2021)
Entropy of changes	Distribution of changes across each modified file, measured using the Normalized Static Entropy, as used by Kamei et al. (Kamei et al., 2012).	A high entropy indicates a highly-fragmented commit, i.e., scattered changes touching many files, which indicates a highly-complex commit (Hassan, 2009; Kamei et al., 2012; Tufano et al., 2017a).	Yang et al. (2017)
Number of Hunks	The number of continuous blocks of changes in the commit diff.	Similar to *Entropy of Changes*.	Perl et al. (2015)
PRODUCT METRICS			
LOC	Lines of Code, counting both source and comment lines.	Large files tend to have higher risk of becoming vulnerable (Koru et al., 2009; Zimmermann et al., 2010; Meneely and Williams, 2012; Perl et al., 2015; Theisen et al., 2015).	Zimmermann et al. (2010), Chowdhury and Zulkernine (2011), Shin and Williams (2011) and Zhang et al. (2015)
SLOC	Source Lines of Code, i.e., LOC without comment and documentation lines.	Same as *LOC*.	Chowdhury and Zulkernine (2011) and Yang et al. (2017)
WMC	Weighted Methods per Class, i.e., the sum of the complexities (i.e., McCabe's Cyclomatic Complexity) of all the methods in a class (Chidamber and Kemerer, 1994).	Complex code is difficult to maintain and test (Shin et al., 2011; Chowdhury and Zulkernine, 2011; McCabe, 1976) and thus has higher chance of having vulnerabilities (Shin et al., 2011; Chowdhury and Zulkernine, 2011; Zimmermann et al., 2010).	Zimmermann et al. (2010), Chowdhury and Zulkernine (2011) and Zhang et al. (2015)
CBO	Coupling Between Object, i.e., the number of dependencies a class has with other classes (Chidamber and Kemerer, 1994).	Highly coupled code makes input from external sources harder to trace (Shin et al., 2011), and has positive correlation with vulnerabilities (Chowdhury and Zulkernine, 2011).	Zimmermann et al. (2010) and Chowdhury and Zulkernine (2011)
RFC	Response For a Class, i.e., the number of methods (including inherited) that can potentially be called by other classes (Chidamber and Kemerer, 1994).	Same as *CBO*.	Chowdhury and Zulkernine (2011), Shin et al. (2011) and Zhang et al. (2015)
DIT	Depth of Inheritance, i.e., the depth of the class within its inheritance tree (Chidamber and Kemerer, 1994).	A deep class is likely to have a larger number of inherited methods, making it more complex to predict its behavior as it is affected by many ancestor classes (Chowdhury and Zulkernine, 2011).	Zimmermann et al. (2010) and Chowdhury and Zulkernine (2011)
NOC	Number of Children, i.e., the number of direct sub-classes (Chidamber and Kemerer, 1994).	Changing a class with many incoming dependencies may introduce defects (Chowdhury and Zulkernine, 2011).	Zimmermann et al. (2010) and Chowdhury and Zulkernine (2011)
LCOM1	Lack of Cohesion of Methods version 1, i.e., the number of pairs of methods not sharing all the fields they access to Chidamber and Kemerer (1994).	Poor cohesive code has been shown to be positively correlated with vulnerabilities (Chowdhury and Zulkernine, 2011).	Chowdhury and Zulkernine (2011)
LCOM2	Lack of Cohesion of Methods version 2, i.e., the percentage of methods not accessing a specific attribute averaged over all attributes in the class (Henderson-Sellers et al., 1996).	Same as *LCOM1*.	Chowdhury and Zulkernine (2011)
TEXT METRICS			
Files Term(s) Frequency	The counting of each word that appears in the full text of the modified JAVA files.	Term frequency has been shown to improve the prediction power if considered with other metrics (Zhang et al., 2015; Theisen and Williams, 2020).	Scandariato et al. (2014), Walden et al. (2014) and Zhang et al. (2015)
Patches Term(s) Frequency	The number of times in which the words appearing in the patches involving JAVA files were changed (added or removed).	Same as *Files Term(s) Frequency*.	Perl et al. (2015)

the Variable Inflation Factor (VIF) method (O'brien, 2007): for each independent variable and for each experimented model, the `vif` function measures how much the variance of the model increases because of collinearity. The features having a `vif` coefficient higher than 5 were removed; the process was repeated until the point where all the features had coefficients lower than the threshold. Afterwards, we considered the problem of hyper-parameter configuration. In particular, we run the RANDOM SEARCH algorithm (Bergstra and Bengio, 2012), which performs a randomized search of the hyper-parameter space with the aim of identifying the optimal hyper-parameter values to use for the classification task. Bergstra and Bengio (2012) proved

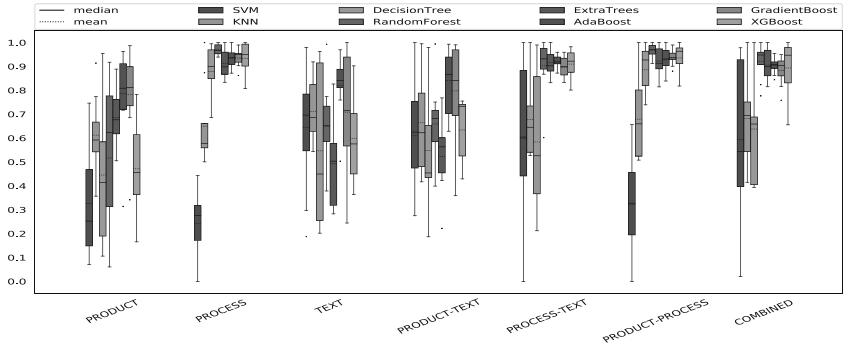


Fig. 1. The AUC–ROC scores obtained during the LOGO validation of the 56 models, grouped by the seven features combinations.

that this search algorithm is able to reach, using less computational resources, the same – or even better – hyper-parameter configuration as an exhaustive search, e.g., GRID SEARCH.

3.6. Evaluating the machine learning methods

Our empirical investigation led to the training and validation of a total of 56 different models, coming from the combination of the eight machine learning algorithms (Section 3.5.2) and the seven features combinations (Section 3.5.1). The results of the comparison of these models are reported in Section 4. After setting the machine learners, we defined the data analysis procedure to address our research question.

3.6.1. Training and validation strategy

To assess the capabilities of the considered models, we had to define a training and validation strategy. We took into account the imbalance of the dataset: as previously shown (see Table 2), each project has around 1% of vulnerable commits. As such, we applied the *Synthetic Minority Oversampling Technique* (SMOTE) (Chawla et al., 2002): for each project, this technique generates artificial samples of the minority class (i.e., vulnerable commits in our case) in order to rebalance the classes. Unfortunately, we found that the technique could not be applied on all the considered projects. In particular, SMOTE requires the presence of at least two samples of the minority class; otherwise, it does not have enough data to oversample the dataset. In two projects, i.e., JUNRAR and LITEMALL, only one commit was labeled vulnerable and it was not possible to apply the balancing approach. This problem influenced our training procedures, as we could not effectively train machine learners using a *within-project* strategy.

Hence, we went for a *cross-project* training. This means that we aggregated data coming from *n-1* projects, balance the training set, and then verify the performance of the models on the remaining project. More specifically, we adopted a *Leave One Group Out* (LOGO) validation strategy, which divides the entire dataset into folds, each containing all the commits of a single project for a total of 9 folds. The validation consisted of 9 iterations, each using 8 folds to build the training set, and the remaining one for the test set. As a consequence, each project was used in *n–1* training sessions, and only once for the testing.

3.6.2. Detection performance measures

For each fold experimented during the validation, we assessed the machine learning models capabilities using a number of performance measures. First, we computed *precision* and *recall*. However, as suggested by Powers (2011), these two measures present some biases as they are mainly focused on positive examples (i.e., vulnerable commits in our context) and predictions, so they 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) (Matthews, 1975) 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 Characteristic* (ROC) curve, and the related *Area Under the Curve* (AUC–ROC). This measure gave us the probability of ranking a randomly chosen positive instance higher than a randomly chosen negative one.

3.6.3. Statistical analysis

The final step of our methodology consisted of the application of statistical tests to verify whether the differences in the performance achieved by the various experimented models were statistically significant. Such an analysis was useful to assess the existence of metrics and/or classifiers that were more suitable for the problem of just-in-time vulnerability detection. Since the data are not normally distributed, we exploited the Friedman Test with the Nemenyi post-hoc test (Nemenyi, 1962) on all the machine learning models. This is a post-hoc test that identifies the groups of data that differ after a statistical test of multiple comparisons has rejected the null hypothesis (the groups are similar), making a pair-wise performance. We selected this test because it is robust to multiple comparisons – which is our case since we had to compare multiple models on multiple features – and does not require the underlying distribution to be normally distributed. To conduct the statistical analysis, we used the `Nemenyi` package for PHYTON.[9]

4. Empirical study results

Figs. 1 and 2 depict the box plots reporting the distribution of AUC–ROC and F-measure values obtained during the LOGO validation of the 56 machine learning models on the considered

[9] https://scikit-posthocs.readthedocs.io/en/latest/.

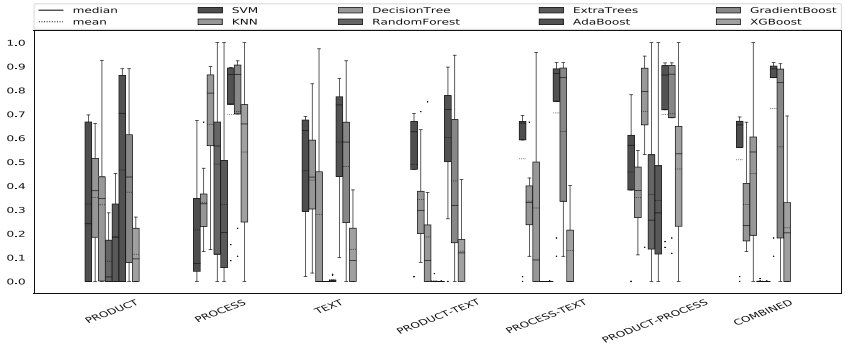


Fig. 2. The F-measure scores obtained during the LOGO validation of the 56 models, grouped by the seven features combinations.

dataset. In both figures, each color indicates the model produced by the selected learning algorithms (Section 3.5); the box plots were also grouped by the seven different combinations of features. For the sake of readability and comprehensibility, we only report in detail the results of two of the seven performance metrics described in Section 3.6; however, the complete results are included in the online appendix.[4]

Considering the AUC–ROC distributions (Fig. 1) the ensemble methods (Random Forest, Extra Trees, AdaBoost, Gradient Boosting, and XGBoost) generally performed better than the three basic classifiers (SVM, KNN and Decision Tree) over all the seven combinations of features. Among all the feature sets, the *product* group alone (label *"PRODUCT"*) caused the models to obtain the worst AUC–ROC scores. Something similar, though with lesser extent, happened for the *textual* metrics (label *"PRODUCT"*). The combination of these two groups (label *"PRODUCT-TEXT"*) did not yield any relevant positive effect, i.e., the addition of *product* metrics does not provide substantial changes in terms of AUC–ROC. Moreover, the sole presence of *product* and/or *textual* metrics did not highlight any relevant difference between basic classifiers and ensemble models, i.e., they are comparable in terms of AUC–ROC. Although the ensemble models still show better performance, the SVM and KNN models are the only ones that greatly benefit from the presence of *textual* metrics. This phenomenon becomes even more evident when moving from the *product+process* group to the *combined* one. The most interesting results occurred when *process* metrics were involved (all the groups having the *"PROCESS"* label in Fig. 1). On the one hand, these metrics further increased the differences among the AUC–ROC distributions—e.g., the large gap between the box plots of AdaBoost and SVM. On the other hand, almost all the models – with the notable exception of the SVMs – received a general improvement. What is more, the ensemble models achieved the best AUC–ROC scores in *product+process* feature combination (label *"PRODUCT-PROCESS"*), hinting that the addition of *textual* metrics causes negative, though marginal, effects. Once again, SVM and KNN were not subject to these phenomena: their models did not receive any positive effect from the presence of *process* metrics. Indeed, similarly to the *product* metrics, they seem to be quite "insensitive" from the presence or absence of *process* metric when the *textual* metrics are already involved. This can be seen by comparing the set having the *textual* metrics alone (label *"TEXT"*) with the ones including them (*"PRODUCT-TEXT"*, *"PROCESS-TEXT"*, and *"COMBINED"*).

The F-measure trends (Fig. 2) are largely different from the ones seen with the AUC–ROC. In the first place, not all the ensemble methods benefit from the presence of *process* metrics. Random Forest, Extra Trees and XGBoost classifiers scored even lower F-measures than the basic classifiers; this difference becomes even larger when *textual* metrics are added: their F-measures collapsed around 0. Differently, AdaBoost Gradient Boosting preserve the general behavior seen with the AUC–ROC: adding *process* metrics is always beneficial, i.e., the inter-quartile ranges shrunk, while the mean and median values increased. To a far lesser extent, these two models suffer from the presence of *textual* metrics, as they may slightly worsen their performance. As an example, XGBoost dropped for about 0.1 point in F-measure when *textual* metrics were added to *product* and *process* models (i.e., from *"PRODUCT-PROCESS"* to *"COMBINED"*). In any case, the sole presence of *process* metrics is enough to achieve acceptable performance.

Finding #1

The majority of the models benefit – in terms of both AUC–ROC and F-measure scores – from the presence of *process* metrics in the set of predictors. SVM- and KNN-trained classifiers give the best of themselves when *textual* metrics are involved, while the opposite, to varying degrees, occurs for the other models, especially in terms of F-measure—which is much more susceptible than AUC–ROC.

From the point of view of the machine learning algorithms, the Decision Tree provides the most unstable models, highly influenced by the set of predictors used, i.e., they received the largest drop in terms of both AUC–ROC and F-measure when *textual* metrics were added. This could be explained by the fact that decision trees are particularly sensible to noise in the training data, and cannot properly generalize. This effect is more obvious in the case of a high dimensional feature space—i.e., the one created when all the tokens from the two *bag-of-words* built are added – or highly imbalanced data – which is true in this context since the number of vulnerable instances is far lower than the number of "safe" instances. Such a limitation is partially solved by using ensemble methods. Conversely, the classifiers trained using SVM and KNN are the only models positively influenced by the presence of *textual* metrics. In particular, KNNs resulted to produce the most stable models, being the least influenced by other predictors that are not part of the *textual* group, and achieving very similar scores in most combinations of predictors. Between the two, KNN outperformed SVM in terms of AUC–ROC, but it scored very low performance in terms F-measure. Nevertheless, both algorithms did not manage to train models with high scores, making them unsuitable in the context we considered.

Random Forest and Extra Trees, despite having a similar learning mechanisms, obtained quite different distributions: they

both scored the worst possible F-measures, being around 0 in most cases, even lower than a traditional DECISION TREE. They draw benefit from the inclusion of *process* metrics, but they are too much negatively influenced by the tokens of the *bag-of-words*. Curiously enough, they still managed to reach very high AUC–ROC scores, sometimes even outperforming all the other learners. Such contrasting AUC–ROC and F-measure values implies that there is a possibility to improve the predictive capabilities of these models by tuning the decision threshold: instead of keeping it to the default value 0.5, change it accordingly to the specific needs, finding the best trade-off between the recall and the false positive rate, which also have an impact to the F-measure.

The scenario is thoroughly different for boosting-based models: ADABOOST, followed by GRADIENT BOOSTING, outperformed the other learners on all fronts. This result was somehow expected since both these models build sequential *shallow* weak classifier, usually a single split decision tree, which are less prone to overfit compared to the *deep* weak classifiers used by other ensemble models like RANDOM FOREST. Moreover, the aggregation of the prediction is weighted in the case of ADABOOST and GRADIENT BOOSTING, hence the individual weak classifiers who performed better have a higher weight compared to those who performed poorly. In the other ensemble models, the prediction of each weak classifier carries the same weight. Oddly enough, XGBOOST, despite being a boosting-based model, was very far from the performance of ADABOOST and GRADIENT BOOSTING, and more similar to RANDOM FOREST and EXTRA TREES.

Finding #2

Boosting-based classifiers, especially ADABOOST and GRADIENT BOOSTING, achieved the best overall results. The other ensemble models scored far worse F-measures, even lower than basic classifier. RANDOMFOREST and EXTRATREES, however, obtained very high AUC–ROC scores, hinting the possibility to improve their predictive capabilities by properly tuning the decision threshold. SVMs and KNNs are the only models to benefit from *textual* metrics, and generally ignore the effect of other features.

To assess whether the distributions of the performance metrics were statistically different when considering different combinations of predictors, we run the post hoc Nemenyi rank test (Nemenyi, 1962) on all the machine learning models. For the sake of readability, in this paper we only report and describe the results for ADABOOST, i.e., the algorithm that provided the best results over all the seven combinations of features. For consistency, we show the *p*-values of the Nemenyi rank test computed on the distribution of AUC–ROC and F-measure values by the means of heatmaps (Figs. 3a and 3b). In addition, we report the statistical results (in terms of AUC–ROC and F-measure) of the eight experimented machine learners trained using the *product+process* features set, i.e., the best combination according to our results (Figs. 4a and 4b). The complete results are reported in our online appendix.[4]

Fig. 3a shows statistically significant differences (depicted in dark violet) in AUC–ROC values between the models built using the *product* metrics alone (label *"PRODUCT"* label) and both (1) those built using *process* metrics (label *"PRODUCT"*), and (2) the ones trained using only the *textual* metrics (label *"TEXT"*). This confirms the large positive effect that *process* metrics have on the AUC–ROC measure on ADABOOST-trained models. Between the *product* and *textual* groups there are no statistically significant differences, implying that there is no sufficient evidence to establish which provides higher predictive capabilities. On a similar note, Fig. 3b shows the presence of statistically significance differences between the *combined* group (label *"COMBINED"* and the groups involving either *product* or *textual* metrics (labels *"PRODUCT"*, *"TEXT"*, and *"PRODUCT-TEXT"*) in terms of F-measure. This is a further evidence on the contribution provided by the *process* metrics.

Focusing on the *process+product* combination, which provided the best models overall, Fig. 4a better highlights the comparable performance obtained by the ensemble methods which significantly differs from the ones obtained by SVM and KNN—which did not benefit from the *process* metrics, but, rather, from *textual* ones. Fig. 4b provides a different view of what could be seen from the box plots (Fig. 2): ADABOOST and GRADIENT BOOSTING far greatly surpassed the F-measures scored by RANDOMFOREST, EXTRATREES, and DECISIONTREE models. Surprisingly, the DECISIONTREES were able to significantly surpass the performance of RANDOMFOREST and EXTRATREES. This does not immediately implies that decision trees are better than the related ensemble methods. As a matter of fact, RANDOMFOREST and EXTRATREES still scored higher AUC–ROC, suggesting the need to fine tune the decision threshold to achieve better predictive capabilities, instead of relaying on the default one (which could also be the best choice in certain cases). This aspect, however, deserves further investigation.

Finding #3

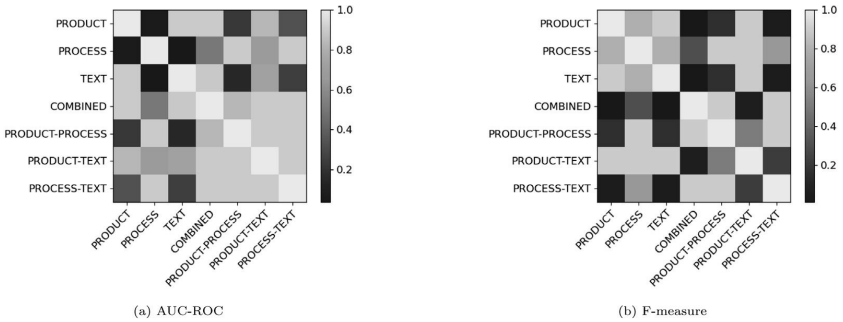
Significance tests confirm the findings discovered during the qualitative analysis of the distributions by the means of box plots: the adoption of *process* metrics to ADABOOST models provides improvements in terms of both AUC–ROC and F-measure. More in general, the boosting-based algorithms are better than other classifiers, while non-boosting ensemble methods still need further investigation on how to improve their capabilities by tuning their decision thresholds.

5. Discussion and implications

The results achieved in our empirical study revealed a number of insights that may lead to concrete implications for the software engineering research community, and that we further discuss hereafter.

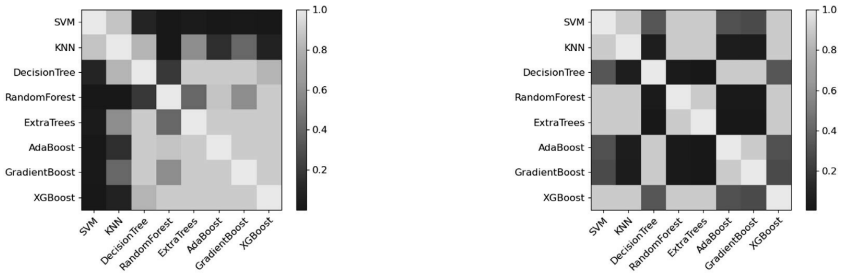
Comparison with other just-in-time VPMs. Our analyses revealed a number of insights that could be related to the ones discovered by Perl et al. (2015) and Yang et al. (2017), i.e., the closest studies to our work and that represent the current state-of-the-art in just-in-time vulnerability prediction modeling. Similarly to what Riom et al. experienced (Riom et al., 2021), we could not provide a precise comparison with the VPMs described in Perl et al. (2015) and Yang et al. (2017), as the original papers point to appendices that no longer exist, preventing us to access to the raw results they achieved. Moreover, the description of the metrics extraction provided in those papers do not report implementation details, making the reproduction even harder. For all these reasons, we only compared our findings with the ones reported in Perl et al. (2015) and Yang et al. (2017), leaving out any detailed comments on the actual performance scores achieved by the models. In this respect, our goal was to find any possible point of agreement and/or disagreement between our contribution and the current state-of-the-art VPMs.

The performance of the two VPMs (Perl et al., 2015; Yang et al., 2017) were reported in terms of precision, using the rationale that, in the context of predicting software vulnerabilities, a higher precision is preferable as the minimization of the false positive rates is instrumental for preventing developers to pointlessly inspect a large number of commits that will not



(a) AUC-ROC (b) F-measure

Fig. 3. Nemenyi test *p*-values obtained for comparing the eight AdaBoost models trained on the seven features combinations.



(a) AUC-ROC (b) F-measure

Fig. 4. Nemenyi test *p*-values obtained for comparing the models trained on *product+process* feature combination using the eight machine learning algorithms.

contribute to the insertion of vulnerable code. Because of this, the authors compared their models with a baseline static analysis tools, i.e., FlawFinder (FlawFinder, 2021), to find whether they could outperform its detection capabilities at the same recall level (done by varying the decision threshold). In both studies, the models were able to achieve much higher precision, i.e., they largely reduced the amount of false positives discovered by FlawFinder. Yet, such a comparison is still limited, as it does assess the actual effectiveness of machine-learning models. As a matter of fact, under these configurations – i.e., when setting the decision threshold to have the same recall level as FlawFinder – the SVM built in Perl et al. (2015) obtained an F-measure of 0.343, while the RandomForest used in Yang et al. (2017) scored 0.198. Both these results indicate limited effectiveness. It is worth remarking that we could not compare these scores with ours as all the studies considered different contexts and feature sets, making any comparison unfair and leading to wrong conclusions. In any case, the F-measure, together with precision and recall, cannot be the sole measure to be taken into account, especially when working with imbalanced datasets. Indeed, other measures, such as AUC–ROC and MCC, are recommended to provide a better overview on the predictive capabilities of the models (Shepperd et al., 2014; Pecorelli et al., 2020). To the best of our knowledge, our study is one of the first in JIT vulnerability prediction that does not consider the precision alone, and whose primary goal is not overcoming static analysis tools, but rather comparing many learning algorithms to find which provides the best models, as well as adopting critical pre-processing steps aimed at improving the training session.

Both Perl et al. (2015) and Yang et al. (2017) considered the use of *textual* metrics under the "code metrics" feature group. Specifically, Perl et al. (2015), not only they run the bag-of-words on the patch content, but they also added the counting of the C/C++ language keywords (e.g., `if`, `goto`, etc.) in the same group; similarly, Yang et al. (2017) counted the C/C++ keywords appearing in the modified files of the commit. The behavior of our SVM models seems to be in line with the one by Perl et al. (2015): the *textual* metrics seems to be beneficial, as opposed to the *process* metrics (which they called GitHub *meta-data*). This may be explained by the fact that SVMs are able to perform well even with large and sparse feature space, i.e., when considering words counting (Joachims, 1998). On the other hand, the RandomForest in Yang et al. (2017) obtained the worst performance when involving the *textual* metrics, the same encountered with our RandomForests. In both studies the authors managed to obtained the best performance when considering all metrics together, confirming the results observed in Theisen and Williams (2020) at the file granularity level. Our SVMs did not experience this effect, as the best model is obtained when only *textual* features were considered, while our RandomForests confirmed this effect only for the AUC–ROC scores, as the addition of *textual* metrics dropped the F-measure close to 0. This hints the need of better data pre-processing activities tailored on the requirements of each learning algorithm.

JIT vulnerability detection: Are we there yet? In the title of the paper, we pose this question. According to our results, the answer is: *"No"*. The accuracy of the existing vulnerability prediction models is not enough to make developers aware of possible vulnerabilities when committing new changes onto a repository. Our study identifies a number of open issues and challenges that the research community should further consider and on which we elaborate more in the remainder of the section. From the predictive power of the features to the machine learning pipelines configured for the prediction exercise, the currently available solutions cannot provide a *just-in-time* feedback to developers.

From a practical perspective, our results indicate the lack of techniques that can analyze the changes done within a commit and detect possible inconsistencies inducing vulnerabilities. As such, developers must still rely on longer-term predictions that analyze entire releases to identify vulnerable files. This represents a threat to the usability and usefulness of the available approaches, as indicated by previous work (Kamei et al., 2012). Hence, our work points out the need for further research on the matter and that should be devoted to all components of the machine learning pipelines.

The existing metrics are not enough. One of the key outcomes of our research is the inability of current metrics to characterize vulnerable commits in an effective and consistent manner. Indeed, despite having considered most of the metrics exploited in previous work on VPMs, in many cases the performance achieved in terms of F-measure are low. This is particularly evident when considering the textual metrics: the bag-of-words source code representation which was found successful by Scandariato et al. (2014) was instead poorly accurate in our case. This is true for both models exploiting this representation individually and those where textual features are combined with other metrics. This might be due to the fact that, when run on the set of modified files, the representation takes into account too many irrelevant tokens possibly creating noise, hindering the predictive capabilities of the bag-of-words to indicate the whether a commit contributes to a vulnerability. For this reason we also (1) employed the use of thresholds to discard both high- and low-frequency words, and (2) added the tokens extracted from the commit patch only, with the aim of reducing the noise and using more relevant features. Yet, these actions did not improve the overall quality of the textual metric set, highlighting the need for additional specific pre-processing activities aiming at further reducing noise. On a similar note, general-purpose code metrics alone often lead to poor results. For instance, the product metrics exploited in our study – and in vulnerability research in general – refer to the quantification of code quality aspects like cohesion, coupling, and complexity: while these have been successfully employed in other branches, e.g., code smell or defect prediction (Azeem et al., 2019; Hall et al., 2011), we observed that their contribution for just-in-time vulnerability detection is limited. Therefore, our results represent a call for new software metrics that can better characterize additional aspects of the source code, e.g., capturing security-related aspects (Alshammari et al., 2011; Chowdhury et al., 2008; Younis et al., 2016), and evolutionary properties correlated to the presence of vulnerabilities.

Better together? On the combination of feature sets. As a follow-up discussion, it is worth analyzing the results achieved while combining multiple metrics. As recently reported by Theisen and Williams (2020), vulnerability prediction models relying on a mixture of code, process, and textual metrics perform better than models based on individual features. When lowering the granularity of the prediction to commit-level, we found that this is not always the case, hence partially contrasting their results. As a matter of fact, there are some specific learning algorithms that appear to perform well under certain performance measures but fail when evaluated with different measures. For instance, despite showing very high AUC–ROC scores, the Random Forest models resulted to be one of the worst models in terms of F-measure when all the features were involved, apparently owing to the addition of textual metrics. At the same time, Theisen and Williams (2020) show that the combination of textual and software metrics lead to a considerable drop in precision, hence affecting F-measure as well, in line with the results we observed in Fig. 2. This suggests the need for automated mechanisms that can exploit contextual information to recommend which features would best fit the needs of the system where vulnerabilities must be diagnosed. A partial exception to this general finding was represented by process metrics: as shown in our study, these are the features that allow machine learners to significantly improve their detection capabilities. Our results seem to be in line with previous research showcasing the positive impact that change history information has on predictive modeling approaches (Pascarella et al., 2019; Rahman and Devanbu, 2013). As such, it seems reasonable to argue that further research on the processes around the introduction of vulnerabilities should be performed to better characterize and improve their detection.

Ensemble learning for vulnerability prediction. In our investigation, we observed that the choice of the classifier has an impact on the resulting capabilities of just-in-time vulnerability detection models. While base learning algorithms typically have low performance, we noticed that the use of ensemble methods improves the classification capabilities. On the one hand, this result does not come as a surprise, as ensemble learning has been introduced with the aim of overcoming the performance of base classifiers. On the other hand, however, it is also worth pointing out that previous investigations in the field of software engineering have revealed that the improvements given by ensemble methods might be limited when other aspects (e.g., availability of a balanced training set) come into play (Laradji et al., 2015; Pecorelli and Di Nucci, 2021). Our findings specifically highlight that boosting methods might be promising for vulnerability detection and, indeed, the AdaBoost learner is the one obtaining the best performance. As observed in Section 4, its characteristics allow it to iteratively train a weak classifier on subsequent training data, assigning a weight to each instance of the training set, and leading to boost the learning capabilities. These results might drive practitioners in the selection of the technique to use when predicting vulnerabilities at commit-level, but also researchers to build upon these characteristics to engineer *ad-hoc* methodologies to further improve the boosting performance.

6. Threats to validity

This section discusses the possible biases to our results and reports the employed mitigation strategies.

Threats to construct validity. A first threat in this category relates to the dataset exploited. We mined the *National Vulnerability Database* with the aim of collecting real, verified data on the vulnerabilities that affected software projects in the past. The nature of the information contained in NVD allowed us to be confident about the reliability of the dataset. Nonetheless, we cannot exclude imprecision: for instance, a patch reported in the database might have not removed a vulnerability as intended.

We relied on a technique based on SZZ to fetch the vulnerability-contributing commits that are likely to have caused the patch applied in the vulnerability-fixing commits mined from NVD. Previous studies have shown that this algorithm may frequently produce false positives (Rodríguez-Pérez et al., 2018); to mitigate this risk we adopted some precautions. We exploited the implementation of SZZ provided by PyDriller (Spadini et al., 2018), which follows the standard version of the algorithm (Sliwerski et al., 2005) on which some adjustments have been included, i.e., discarding the candidate commits where only comments, cosmetic changes, or empty lines were blamed. This implementation achieved the highest recall with respect to the other variants (Rosa et al., 2021), and so we opted for it to reduce the risk of missing relevant VCCs.

Over the initial population of 56,286 commits considered in our context, we sampled 8991 commits due to computational constraints. We are aware that this sampling could have affected the performance of the machine learning models during the training and testing phases; however, our sampling criterion was

carefully made random with the aim of mitigating the selection bias.

Another potential threat may be related to the selection of the independent variables used to build the experimented models. In this respect, we have carefully considered the related literature and the features previously used by researchers who targeted the problem of file-level vulnerability detection. Perhaps more importantly, our analyses targeted three different families of metrics, hence allowing us to experiment with features capturing different aspects of source code. Nevertheless, we cannot rule out that other metrics, not considered in the study, could provide additional contribution to the performance of just-in-time vulnerability detection methods. We plan to investigate this aspect further in the future.

Finally, when using the bag-of-words method we discarded the words appearing in over 80% of the documents (i.e., files or patches) or less than 5%. While this step could have removed some relevant features, and so possibly hindered the performance of the models, it is a recommended pre-processing step to remove noisy data and reduce the dimensionality of the dataset, which has been seen to have positive effects on the training process (Martins, 2003; Joachims, 1998). The choice of these thresholds was directed by the need to have a reasonable number of features to train the models in an acceptable time without removing important tokens.

Threats to internal validity. In the context of our work, we selected and experimented eight machine learning models to better understand their strengths and weaknesses. Of course, the setting up of these approaches might have biased our results. However, we followed well-established guidelines (Song et al., 2010; Tantithamthavorn and Hassan, 2018) through which we addressed possible issues due to multi-collinearity, missing hyper-parameter configuration, and data balancing issues. When focusing on these issues, we used methods and techniques that have been widely employed in the past (e.g., the `vif` function to deal with correlated variables) and that are recognized as effective.

Threats to external validity. Our study involved nine systems written in Java. On the one hand, we recognize that larger-scale studies would be desirable to further understand the capabilities of machine learning models for vulnerability detection. On the other hand, we are aware that different results might be obtained when addressing our research question on projects written in different programming languages or developed in different contexts (e.g., industrial systems). To enable replicability, we made all data and scripts available in our online appendix.[4] In any case, our future research agenda includes a large-scale replication of the study.

Threats to conclusion validity. To derive conclusive results on the performance of just-in-time vulnerability detectors, we first computed a number of evaluation metrics in an effort of capturing various angles of their capabilities. All of them uniformly indicated the poor performance of the experimented models, hence confirming our conclusions. In addition, we also applied statistical tests to verify the significance of the differences observed: we run the Nemenyi rank test (Nemenyi, 1962) to deal with the problem of multiple comparisons. This test is particularly useful in our context as it is suitable for non-normal distributions like the ones we experienced.

7. Conclusion

This paper proposed an empirical investigation into the capabilities of machine learning models for just-in-time vulnerability prediction. We took into account a set of eight machine learners and three families of features to provide a broad overview of how software vulnerabilities can be identified at commit-level.

Our key results indicated that the problem should be further investigated, as elaborated in Section 5. First, the currently available metrics seem to be not enough and, perhaps more importantly, their combination does not necessarily improve the detection capabilities. The research community should invest effort in defining empirical investigations into the features connected to the introduction of vulnerabilities at commit-level, other than the features that developers consider more relevant. For instance, we can envision the definition of longitudinal studies where developers are monitored for a given time period so that their activities might be closely analyzed in order to identify the key inducers of vulnerabilities. Similarly, we can envision studies aiming at elaborating catalogs of micro-antipatterns that developers frequently apply when contributing to vulnerabilities. An improved understanding of the features that more characterize the problem of software vulnerabilities would definitively improve the accuracy of just-in-time prediction models. On the basis of these empirical investigations, the definition of novel instruments able to compute those metrics and, perhaps more importantly, novel comprehensive datasets would be key to enable more and more research on the matter.

Second, our results indicate that the choice of the classifier impacts the performance: while most of the algorithms experimented achieve low F-measure scores, we observed that an ensemble method like AdaBoost seems to provide promising results that should be further analyzed and possibly improved by the research community. In other terms, our findings stimulate research targeting the engineering of software vulnerability prediction models. For instance, we could envision empirical studies and/or novel software engineering for artificial intelligence methods that could mix together the capabilities of individual classifiers or even dynamically adapt the classifier to use based on the peculiar characteristics of code commits and of the developers applying changes.

Last but not least, a collateral finding of our study concerns with the lack of public data and scripts that can be used to replicate/reproduce previous studies. This is pretty worrisome and allows us to recommend further research effort on the definition of standards and guidelines to make research reproducible, especially to enable researchers to compare the previous findings with new ones, hence leading to advance the state of the art in a safe and sustainable manner.

Our future research agenda includes a larger-scale replications of our study, other than the definition of novel techniques for (1) selecting features to use when identifying vulnerabilities at commit-level and (2) improving the training capabilities of ensemble approaches.

CRediT authorship contribution statement

Francesco Lomio: Data collection, Data analysis, Writing – original draft. **Emanuele Iannone:** Data collection, Data analysis, Writing – original draft. **Andrea De Lucia:** Supervision, Writing – original draft. **Fabio Palomba:** Conceptualization, Methodology, Writing – original draft. **Valentina Lenarduzzi:** Conceptualization, Methodology, Writing – original draft.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Alhazmi, O., Malaiya, Y., Ray, I., 2007. Measuring, analyzing and predicting security vulnerabilities in software systems. Comput. Secur. 26 (3), 219–228.

Alshammari, B., Fidge, C., Corney, D., 2011. A hierarchical security assessment model for object-oriented programs. In: 2011 11th International Conference on Quality Software. pp. 218–227.

Antunes, N., Vieira, M., 2010. Benchmarking vulnerability detection tools for web services. In: 2010 IEEE International Conference on Web Services. IEEE, pp. 203–210.

Azeem, M.I., Palomba, F., Shi, L., Wang, Q., 2019. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. Inf. Softw. Technol. 108, 115–138.

Bandara, V., Rathnayake, T., Weerasekara, N., Elvitigala, C., Thilakarathna, K., Wijesekera, P., Keppitiyagama, C., 2020. Fix that Fix Commit: A real-world remediation analysis of JavaScript projects. In: 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 198–202.

Basili, V.R., Caldiera, G., Rombach, H.D., 1994. The goal question metric approach. In: Encyclopedia of Software Engineering. Wiley.

Bergstra, J., Bengio, Y., 2012. Random search for hyper-parameter optimization. J. Mach. Learn. Res. 13 (2).

Bhatt, N., Anand, A., Yadavalli, V., 2020. Exploitability prediction of software vulnerabilities. Qual. Reliab. Eng. 37.

Breiman, L., 2001. Random forests. Mach. Learn. 45 (1), 5–32.

Breiman, L., Friedman, J., Stone, C.J., Olshen, R., 1984. Classification and Regression Trees Regression Trees. Chapman and Hall.

Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P., 2002. SMOTE: Synthetic minority over-sampling technique. J. Artif. Int. Res. 16 (1), 321–357.

Chen, T., Guestrin, C., 2016. XGBoost: A scalable tree boosting system. In: 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16. pp. 785–794.

Chernis, B., Verma, R., 2018. Machine learning methods for software vulnerability detection. In: International Workshop on Security and Privacy Analytics. pp. 31–39.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20 (6), 476–493.

Chowdhury, I., Chan, B., Zulkernine, M., 2008. Security metrics for source code structures. In: Proceedings of the Fourth International Workshop on Software Engineering for Secure Systems. In: SESS '08, Association for Computing Machinery, New York, NY, USA, pp. 57–64.

Chowdhury, I., Zulkernine, M., 2011. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. J. Syst. Archit. 57 (3), 294–313.

Coppin, B., 2004. Artificial Intelligence Illuminated. Jones & Bartlett Learning.

Cortes, C., Vapnik, V., 1995. Support-vector networks. Mach. Learn. 20 (3), 273–297.

2006. CWE-119: Improper restriction of operations within the bounds of a memory buffer. https://cwe.mitre.org/data/definitions/119.html. Online; accessed 02 December 2021.

D'Ambros, M., Lanza, M., Robbes, R., 2012. Evaluating defect prediction approaches: A benchmark and an extensive comparison. Empir. Softw. Eng. 17 (4–5), 531–577.

De Lucia, A., Di Penta, M., Oliveto, R., Panichella, A., Panichella, S., 2013. Applying a smoothing filter to improve IR-based traceability recovery processes: An empirical investigation. Inf. Softw. Technol. 55 (4), 741–754.

Decan, A., Mens, T., Constantinou, E., 2018. On the impact of security vulnerabilities in the npm package dependency network. In: International Conference on Mining Software Repositories. pp. 181–191.

Do, L.N.Q., Wright, J., Ali, K., 2020. Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. IEEE Trans. Softw. Eng..

Dowd, M., McDonald, J., Schuh, J., 2006. The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities. Pearson Education.

Finifter, M., Akhawe, D., Wagner, D., 2013. An empirical study of vulnerability rewards programs. In: 22nd {USENIX} Security Symposium ({USENIX} Security 13). pp. 273–288.

2021. Flawfinder. https://dwheeler.com/flawfinder. Online; accessed 24 November 2021.

Freund, Y., Schapire, R.E., 1997. A decision-theoretic generalization of on-line learning and an application to boosting. J. Comput. System Sci. 55 (1), 119–139.

Friedman, J.H., 2001. Greedy function approximation: A gradient boosting machine. Ann. Statist. 29 (5), 1189–1232.

Geurts, P., Ernst, D., Wehenkel, L., 2006. Extremely randomized trees. Mach. Learn. 63 (1), 3–42.

Ghaffarian, S.M., Shahriari, H.R., 2017. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. ACM Comput. Surv. 50 (4), 1–36.

2021. Git - git-merge documentation. https://git-scm.com/docs/git-merge. Online; accessed 19 November 2021.

Gonzalez, D., Alhenaki, F., Mirakhorli, M., 2019. Architectural security weaknesses in industrial control systems (ICS) an empirical study based on disclosed software vulnerabilities. In: International Conference on Software Architecture (ICSA). pp. 31–40.

Gousios, G., Pinzger, M., Deursen, A.v., 2014. An exploratory study of the pull-based software development model. In: International Conference on Software Engineering. pp. 345–355.

Graves, T., Karr, A., Marron, J., Siy, H., 2000. Predicting fault incidence using software change history. IEEE Trans. Softw. Eng. 26 (7), 653–661.

Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S., 2011. A systematic literature review on fault prediction performance in software engineering. IEEE Trans. Softw. Eng. 38 (6), 1276–1304.

Harer, J.A., Kim, L.Y., Russell, R.L., Ozdemir, O., Kosta, L.R., Rangamani, A., Hamilton, L.H., Centeno, G.I., Key, J.R., Ellingwood, P.M., et al., 2018. Automated software vulnerability detection with machine learning. arXiv preprint arXiv: 1803.04497.

Harris, Z.S., 1954. Distributional structure. Word 10 (2–3), 146–162.

Hassan, A.E., 2009. Predicting faults using the complexity of code changes. In: 2009 IEEE 31st International Conference on Software Engineering. pp. 78–88.

Henderson-Sellers, B., Constantine, L., Graham, I., 1996. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). Object Oriented Syst. 3, 143–158.

Herzig, K., Just, S., Zeller, A., 2016. The impact of tangled code changes on defect prediction models. Empir. Softw. Eng. 21 (2), 303–336.

Huang, S., Tang, H., Zhang, M., Tian, J., 2010. Text clustering on national vulnerability database. In: International Conference on Computer Engineering and Applications, Vol. 2. pp. 295–299.

Hydara, I., Sultan, A.B.M., Zulzalil, H., Admodisastro, N., 2015. Current state of research on cross-site scripting (XSS)–A systematic literature review. Inf. Softw. Technol. 58, 170–186.

Jiang, B., Liu, Y., Chan, W., 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 259–269.

Jimenez, M., Rwemalika, R., Papadakis, M., Sarro, F., Le Traon, Y., Harman, M., 2019. The importance of accounting for real-world labelling when predicting software vulnerabilities. In: European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 695–705.

Joachims, T., 1998. Text categorization with support vector machines: Learning with many relevant features. In: Nédellec, C., Rouveirol, C. (Eds.), Machine Learning: ECML-98. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 137–142.

Johnson, B., Song, Y., Murphy-Hill, E., Bowdidge, R., 2013. Why don't software developers use static analysis tools to find bugs? In: International Conference on Software Engineering (ICSE). pp. 672–681.

Junjin, M., 2009. An approach for SQL injection vulnerability detection. In: International Conference on Information Technology: New Generations. IEEE, pp. 1411–1414.

Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N., 2012. A large-scale empirical study of just-in-time quality assurance. IEEE Trans. Softw. Eng. 39 (6), 757–773.

Kang, J., Ryu, D., Baik, J., 2021. Predicting just-in-time software defects to reduce post-release quality costs in the maritime industry. Softw. - Pract. Exp. 51 (4), 748–771.

Kim, S., Lee, H., 2018. Software systems at risk: An empirical study of cloned vulnerabilities in practice. Comput. Secur. 77, 720–736.

Koru, A.G., Zhang, D., El Emam, K., Liu, H., 2009. An investigation into the functional form of the size-defect relationship for software modules. IEEE Trans. Softw. Eng. 35 (2), 293–304.

Laradji, I.H., Alshayeb, M., Ghouti, L., 2015. Software defect prediction using ensemble learning on selected features. Inf. Softw. Technol. 58, 388–402.

Li, H., Kim, T., Bat-Erdene, M., Lee, H., 2013. Software vulnerability detection using backward trace analysis and symbolic execution. In: 2013 International Conference on Availability, Reliability and Security. IEEE, pp. 446–454.

Li, Z., Zou, D., Xu, S., Jin, H., Qi, H., Hu, J., 2016. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In: Annual Conference on Computer Security Applications. pp. 201–213.

Martins, C.A., 2003. Reducing the dimensionality of bag-of-words text representation used by learning algorithms.

Matthews, B., 1975. Comparison of the predicted and observed secondary structure of T4 phage lysozyme. Biochim. Biophys. Acta (BBA) - Protein Struct. 405 (2), 442–451.

McCabe, T.J., 1976. A complexity measure. IEEE Trans. Softw. Eng. (4), 308–320.

McGraw, G., 2004. Software security. IEEE Secur. Priv. 2 (2), 80–83.

McKinnel, D.R., Dargahi, T., Dehghantanha, A., Choo, K.-K.R., 2019. A systematic literature review and meta-analysis on artificial intelligence in penetration testing and vulnerability assessment. Comput. Electr. Eng. 75, 175–188.

Meneely, A., Srinivasan, H., Musa, A., Tejeda, A.R., Mokary, M., Spates, B., 2013. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In: 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 65–74.

Meneely, A., Williams, O., 2012. Interactive churn metrics: Socio-technical variants of code churn. SIGSOFT Softw. Eng. Notes 37 (6), 1–6.
MITRE, 2021. Common vulnerabilities and exposures. https://cve.mitre.org/. Online; accessed 06 July 2021.
Morrison, P., Herzig, K., Murphy, B., Williams, L., 2015. Challenges with applying vulnerability prediction models. In: Symposium and Bootcamp on the Science of Security.
Nagappan, N., Ball, T., 2005. Use of relative code churn measures to predict system defect density. In: International Conference on Software Engineering, 2005. ICSE 2005. pp. 284–292.
Nagappan, N., Murphy, B., Basili, V., 2008. The influence of organizational structure on software quality: An empirical case study. In: International Conference on Software Engineering. pp. 521–530.
Nemenyi, P., 1962. Distribution-free multiple comparisons. In: Biometrics, Vol. 18. International Biometric Soc, 1441 I ST, NW, SUITE 700, WASHINGTON, DC 20005-2210, p. 263.
Neuhaus, S., Zimmermann, T., Holler, C., Zeller, A., 2007. Predicting vulnerable software components. In: Conference on Computer and Communications Security. pp. 529–540.
Nguyen, V.H., Tran, L.M.S., 2010. Predicting vulnerable software components with dependency graphs. In: International Workshop on Security Measurements and Metrics.
NIST, 2021. U.S. NIST computer security division. https://www.nist.gov. Online; accessed 06 July 2021.
O'brien, R.M., 2007. A caution regarding rules of thumb for variance inflation factors. Qual. Quant. 41 (5), 673–690.
Parnas, D.L., 1994. Software aging. In: International Conference on Software Engineering. In: ICSE '94, IEEE Computer Society Press, Washington, DC, USA, pp. 279–287.
Pascarella, L., Palomba, F., Bacchelli, A., 2019. Fine-grained just-in-time defect prediction. J. Syst. Softw. 150, 22–36.
Pascarella, L., Spadini, D., Palomba, F., Bruntink, M., Bacchelli, A., 2018. Information needs in contemporary code review. ACM Hum.-Comput. Interact. 2 (CSCW), 1–27.
Pecorelli, F., Di Nucci, D., 2021. Adaptive selection of classifiers for bug prediction: A large-scale empirical analysis of its performances and a benchmark study. Sci. Comput. Program. 205, 102611.
Pecorelli, F., Di Nucci, D., De Roover, C., De Lucia, A., 2020. A large empirical assessment of the role of data balancing in machine-learning-based code smell detection. J. Syst. Softw. 110693.
Perl, H., Dechand, S., Smith, M., Arp, D., Yamaguchi, F., Rieck, K., Fahl, S., Acar, Y., 2015. VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits. In: ACM SIGSAC Conference on Computer and Communications Security. pp. 426–437.
Piancó, M., Fonseca, B., Antunes, N., 2016. Code change history and software vulnerabilities. In: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W). pp. 6–9.
Plate, H., Ponta, S.E., Sabetta, A., 2015. Impact assessment for vulnerabilities in open-source software libraries. In: International Conference on Software Maintenance and Evolution (ICSME). pp. 411–420.
Powers, D.M.W., 2011. Evaluation: From precision, recall and f-measure to roc., informedness, markedness & correlation. J. Mach. Learn. Technol. 2 (1), 37–63.
Rahman, F., Devanbu, P., 2013. How, and why, process metrics are better. In: 2013 35th International Conference on Software Engineering (ICSE). IEEE, pp. 432–441.
Ralph, P., et al., 2021. Empirical standards for software engineering research.
Riom, T., Sawadogo, A., Allix, K., Bissyandé, T.F., Moha, N., Klein, J., 2021. Revisiting the VCCFinder approach for the identification of vulnerability-contributing commits. Empir. Softw. Eng. 26 (3), 46.
Rodríguez-Pérez, G., Robles, G., González-Barahona, J.M., 2018. Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the SZZ algorithm. Inf. Softw. Technol. 99, 164–176.
Rosa, G., Pascarella, L., Scalabrino, S., Tufano, R., Bavota, G., Lanza, M., Oliveto, R., 2021. Evaluating SZZ implementations through a developer-informed oracle. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 436–447.
Saxena, P., Poosankam, P., McCamant, S., Song, D., 2009. Loop-extended symbolic execution on binary programs. In: International Symposium on Software Testing and Analysis. pp. 225–236.
Scandariato, R., Walden, J., Hovsepyan, A., Joosen, W., 2014. Predicting vulnerable software components via text mining. IEEE Trans. Softw. Eng. 40 (10), 993–1006.
Schapire, R.E., 1990. The strength of weak learnability. Mach. Learn. 5 (2), 197–227.
Shepperd, M., Bowes, D., Hall, T., 2014. Researcher bias: The use of machine learning in software defect prediction. IEEE Trans. Softw. Eng. 40 (6), 603–616.
Shin, Y., Meneely, A., Williams, L., Osborne, J.A., 2011. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Trans. Softw. Eng. 37 (6), 772–787.
Shin, Y., Williams, L., 2008. An empirical model to predict security vulnerabilities using code complexity metrics. In: International Symposium on Empirical Software Engineering and Measurement. pp. 315–317.
Shin, Y., Williams, L., 2011. Can traditional fault prediction models be used for vulnerability prediction? Empir. Softw. Eng. 18, 25–59.
Silva, C., Ribeiro, B., 2003. The importance of stop word removal on recall values in text categorization. In: Proceedings of the International Joint Conference on Neural Networks, 2003., Vol. 3. pp. 1661–1666.
Singer, J., Lethbridge, T., Vinson, N., Anquetil, N., 2010. An examination of software engineering work practices. In: CASCON First Decade High Impact Papers. pp. 174–188.
Sliwerski, J., T., Z., Zeller, A., 2005. When do changes induce fixes? In: International Workshop on Mining Software Repositories. In: MSR '05, pp. 1–5.
Smith, B., Williams, L., 2011. Using SQL hotspots in a prioritization heuristic for detecting all types of web application vulnerabilities. In: International Conference on Software Testing, Verification and Validation. pp. 220–229.
Song, Q., Jia, Z., Shepperd, M., Ying, S., Liu, J., 2010. A general software defect-proneness prediction framework. IEEE Trans. Softw. Eng. 37 (3), 356–370.
Sotirov, A.I., 2005. Automatic Vulnerability Detection using Static Source Code Analysis (Ph.D. thesis). Citeseer.
Spadini, D., Aniche, M., Bacchelli, A., 2018. PyDriller: Python framework for mining software repositories. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018. ACM Press, New York, New York, USA, pp. 908–911.
Sultana, K.Z., Anu, V., Chong, T.-Y., 2020. Using software metrics for predicting vulnerable classes and methods in Java projects: A machine learning approach. J. Softw.: Evol. Process 33.
Svacina, J., Raffety, J., Woodahl, C., Stone, B., Cerny, T., Bures, M., Shin, D., Frajtak, K., Tisnovsky, P., 2020. On vulnerability and security log analysis: A systematic literature review on recent trends. In: International Conference on Research in Adaptive and Convergent Systems. pp. 175–180.
Tantithamthavorn, C., Hassan, A.E., 2018. An experience report on defect modelling in practice: Pitfalls and challenges. In: International Conference on Software Engineering: Software Engineering in Practice. pp. 286–295.
Theisen, C., Herzig, K., Morrison, P., Murphy, B., Williams, L., 2015. Approximating attack surfaces with stack traces. In: International Conference on Software Engineering, Vol. 2. pp. 199–208.
Theisen, C., Williams, L., 2020. Better together: Comparing vulnerability prediction models. Inf. Softw. Technol. 119, 106204.
Trinh, M.-T., Chu, D.-H., Jaffar, J., 2014. S3: A symbolic string solver for vulnerability detection in web applications. In: Conference on Computer and Communications Security. pp. 1232–1243.
Tufano, M., Bavota, G., Poshyvanyk, D., Penta, M.D., Oliveto, R., Lucia, A.D., 2017a. An empirical study on developer-related factors characterizing fix-inducing commits. J. Softw.: Evol. Process 29.
Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., Poshyvanyk, D., 2017b. When and why your code starts to smell bad (and whether the smells go away). IEEE Trans. Softw. Eng. 43 (11), 1063–1088.
Vassallo, C., Panichella, S., Palomba, F., Proksc, S., Gall, H., Zaidman, A., 2019. How developers engage with static analysis tools in different contexts. Empir. Softw. Eng..
Walden, J., Stuckman, J., Scandariato, R., 2014. Predicting vulnerable components: Software metrics vs text mining. In: International Symposium on Software Reliability Engineering. pp. 23–33.
Wang, T., Wei, T., Gu, G., Zou, W., 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: 2010 IEEE Symposium on Security and Privacy. IEEE, pp. 497–512.
Wang, T., Wei, T., Lin, Z., Zou, W., 2009. Intscope: Automatically detecting integer overflow vulnerability in X86 binary using symbolic execution. In: NDSS. Citeseer.
Wohlin, C., Runeson, P., Host, M., Ohlsson, M., Regnell, B., Wesslen, A., 2000. Experimentation in Software Engineering: An Introduction.
Yang, J., Hotta, K., Higo, Y., Igaki, H., Kusumoto, S., 2015. Classification model for code clones based on machine learning. Empir. Softw. Eng. 20 (4), 1095–1125.
Yang, L., Li, X., Yu, Y., 2017. VulDigger: A just-in-time and cost-aware tool for digging vulnerability-contributing changes. In: IEEE Global Communications Conference. pp. 1–7.
Younis, A., Malaiya, Y., Anderson, C., Ray, I., 2016. To fear or not to fear that is the question: Code characteristics of a vulnerable function with an existing exploit. In: Conference on Data and Application Security and Privacy. pp. 97–104.
Zhang, Z., 2016. Introduction to machine learning: k-nearest neighbors. Ann. Transl. Med. 4 (11).

Zhang, S., Caragea, D., Ou, X., 2011. An empirical study on using the national vulnerability database to predict software vulnerabilities. In: International Conference on Database and Expert Systems Applications. pp. 217–231.
Zhang, Y., Jin, R., Zhou, Z.-H., 2010. Understanding bag-of-words model: a statistical framework. Int. J. Mach. Learn. Cybern. 1 (1–4), 43–52.
Zhang, Y., Lo, D., Xia, X., Xu, B., Sun, J., Li, S., 2015. Combining software metrics and text features for vulnerable file prediction. In: International Conference on Engineering of Complex Computer Systems (ICECCS). pp. 40–49.
Zimmermann, T., Nagappan, N., Williams, L., 2010. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In: International Conference on Software Testing, Verification and Validation. pp. 421–428.

Francesco Lomio is a Doctoral Researcher at the Cloud, Software Engineering, Evolution, and Assessment with AI (CloudSEA.AI) group at Tampere University, Finland. His main research focuses on machine and deep learning application for software quality, and fault detection. His research interests include machine and deep learning architectures, cloud-native system monitoring and anomaly detection, and software quality for artificial intelligence based systems. He is also interested in the broader field of machine learning, included but not limited to computer vision and time series analysis. Among his activities, he served on the program committee for international conferences and reviewer for journals in the fields of software engineering.

Emanuele Iannone is a Ph.D. student at University of Salerno, Italy. His main research focuses on the analysis of software vulnerabilities, particularly in the broader context of software security testing. His research interests also include the development of novel tools and techniques of Mining Software Repositories and Search-based Software Testing. He also served as a reviewer for the main international conferences and journals of the software engineering field.

Andrea De Lucia is a full professor of Software Engineering at the University of Salerno, Italy, Head of the Software Engineering Lab, and Director of the Ph.D. program in Computer Science. He received Ph.D. degree in electronic engineering and computer science from the University of Naples "Federico II", Italy, in 1996. His research interests include software maintenance and testing, reverse engineering and reengineering, source code analysis, code smell detection and refactoring, defect prediction, traceability management, visual modelling languages, and collaborative software development. He has published more than 250 papers on these topics in international journals, books, and conference proceedings and has edited books and journal special issues. Prof. De Lucia is co-editor in chief of Science of Computer Programming (Elsevier) and serves on the editorial board of Empirical Software Engineering (Springer) and Journal of Software Evolution and Process (Wiley). He also serves on the organizing and program committees of several international conferences in the field of software engineering. Prof. De Lucia is a senior member of the IEEE and was member-at-large of the executive committee of the IEEE Technical Council on Software Engineering. Salerno.

Fabio Palomba is an assistant professor at the Software Engineering (SeSa) Lab of the University of Salerno. He received the European Ph.D. degree in Management & Information Technology in 2017. His Ph.D. Thesis was the recipient of the 2017 IEEE Computer Society Best Ph.D. Thesis Award. His research interests include software maintenance and evolution, empirical software engineering, source code quality, and mining software repositories. He was the recipient of two ACM/SIGSOFT and one IEEE/TCSE Distinguished Paper Awards at the IEEE/ACM International Conference on Automated Software Engineering (ASE'13), the International Conference on Software Engineering (ICSE'15), and the IEEE International Conference on Software Maintenance and Evolution (ICSME'17), respectively, and Best Paper Awards at the ACM Computer Supported Cooperative Work (CSCW'18) and the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'18). In 2019 he was the recipient of an SNSF Ambizione grant, one of the most prestigious individual research grants in Europe. He is a member of the Steering Committee of ICPC (elected in 2021). He has been program co-chair of ICPC 2021, industrial track co-chair of SANER 2022, NIER/ERA track co-chair of ASE 2022, SCAM 2022, and MobileSoft 2022, FOSS Award co-chair of MSR 2022, other than program co-chair of MaLTeSQuE 2018 and 2019. In addition, he has been a member of the organizing committee of ICPC 2015 and SANER 2018. Since 2021, he is Editorial Board Member of the Springer's Empirical Software Engineering Journal (EMSE), where he already was Review Board Member since 2016, and the e-Informatica Software Engineering Journal (EISEJ). Since 2020 he is Review Board Member of the IEEE Transactions on Software Engineering. Since 2019, he is Editorial Board Member of ACM Transactions on Software Engineering and Methodology (TOSEM), Elsevier's Journal of Systems and Software (JSS), and Elsevier's Science of Computer Programming (SCICO). For his reviewing activities, he was the recipient of 12 Distinguished/Outstanding Reviewer Awards.

Valentina Lenarduzzi is an assistant professor (tenure track) at University of Oulu (Finland). Her research activities are related to modern software development practices and methodologies, including data analysis in software engineering, software quality, software maintenance and evolution, focusing on Technical Debt as well as code and architectural smells. She got the Ph.D. in Computer Science in 2015 and was a postdoctoral researcher at the Free University of Bozen-Bolzano, (Italy), at the Tampere University (Finland), and at LUT University (Finland). Moreover, she was visiting researcher at the University of Kaiserslautern (TUK) and the Fraunhofer Institute for Experimental Software Engineering IESE (Germany). She served as a program committee member of various international conferences (e.g., ICPC, ICSME, ESEM), and for various international journals (e.g., TSE, EMSE, JSS, IST) in the field of software engineering. She has been program co-chair of OSS 2021 and TechDebt 2022. She was also one of the organizers of the last edition of MaLTeSQuE workshop (2022) collocated with ESEC/FSE. Dr. Lenarduzzi is recognized by the Journal of Systems and Software (JSS) as one of the most active SE researchers in top-quality journals in the period 2013 to 2020.

PUBLICATION III

Rare: A labeled dataset for cloud-native memory anomalies

F. Lomio, D. M. Baselga, S. Moreschini, H. Huttunen, and D. Taibi

International Workshop on Machine-Learning Techniques for Software-Quality Evaluation
DOI: 10.1145/3416505.3423560

RARE: A Labeled Dataset for Cloud-Native Memory Anomalies

Francesco Lomio
Tampere University
Tampere, Finland
francesco.lomio@tuni.fi

Diego Martínez Baselga
Tampere University
Tampere, Finland
diego.martinezbaselga@tuni.fi

Sergio Moreschini
Tampere University
Tampere, Finland
sergio.moreschini@tuni.fi

Heikki Huttunen
Tampere University
Tampere, Finland
heikki.huttunen@tuni.fi

Davide Taibi
Tampere University
Tampere, Finland
davide.taibi@tuni.fi

ABSTRACT

Anomaly detection has been attracting interest from both the industry and the research community for many years, as the number of published papers and services adopted grew exponentially over the last decade. One of the reasons behind this is the wide adoption of cloud systems from the majority of players in multiple industries, such as online shopping, advertisement or remote computing. In this work we propose a Dataset foR cloud-nAtive memoRy anomaliEs: RARE. It includes labelled anomaly time-series data, comprising of over 900 unique metrics. This dataset has been generated using a microservice for injecting artificial byte stream in order to overload the nodes, provoking memory anomalies, which in some cases resulted in a crash. The system was built using a Kafka server deployed on a Kubernetes system. Moreover, in order to get access and download the metrics related to the server, we utilised Prometheus. In this paper we present a dataset that can be used coupled with machine learning algorithms for detecting anomalies in a cloud based system. The dataset will be available in the form of CSV file through an online repository. Moreover, we also included an example of application using a Random Forest algorithm for classifying the data as anomalous or not. The goal of the RARE dataset is to help in the development of more accurate and reliable machine learning methods for anomaly detection in cloud based systems.

CCS CONCEPTS

• **Software and its engineering → Software organization and properties**; *Correctness*; *Designing software*.

KEYWORDS

Dataset, anomaly detection, kubernetes, self healing, machine learning

ACM Reference Format:
Francesco Lomio, Diego Martínez Baselga, Sergio Moreschini, Heikki Huttunen, and Davide Taibi. 2022. RARE: A Labeled Dataset for Cloud-Native Memory Anomalies. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Cloud-native systems have many more moving parts than traditional monolithic ones. These systems are composed of several connected services, usually deployed on different machines. The different services communicate together through the network and there are orchestrators, load balancers, message buses, and many other components not needed in monolithic systems that can go wrong. In such a complex system, runtime failures are unavoidable and must be kept under control [4, 12].

Therefore, the introduction of anomaly detection systems is of paramount importance, to maintain the quality of service and to avoid to run to unexpected costs, due to anomalous usage of cloud resources [12, 14].

An anomaly can be defined as a rare event where the system behaviour deviates from what is standard, normal, or expected. This event usually differs significantly from the rest of the data. For example, a service could use an anomalous amount of memory or processors that will not enable other services to run properly. As another example, some service consumer might stop sending fetch requests to the broker, meaning that the service could be stalled or dead.

Different researchers proposed anomaly detection mechanisms. However, the vast majority are based on proprietary datasets or on very generic datasets [7, 12, 14] Moreover, memory-related issues are some of the most common anomalies in cloud-native systems [12] and the lack of labeled dataset clearly reporting where memory-related anomalies occur and in particular, when the event that generate the anomaly is introduced, do not allow researchers to define accurate prediction models to prevent these type of anomalies.

The goal of this work is to provide a dataset generated from a cloud-native system, with a series of labelled memory-related anomalies, to enable researchers to apply machine learning models enabling them to compare the performances of their models.

For this purpose, in this paper we present the "*dataset foR cloud-nAtive memoRy anomaliEs*" (**RARE**). RARE is a labeled dataset specifically created for cloud-native memory-related anomalies. It

includes 7,062 metrics collected on 10K data points, and 600 memory anomalies randomly injected.

The RARE dataset will enable the benchmark of machine learning models for memory-related anomaly detection. The usage of a common dataset will also allow researchers to replicate existing works, and to improve the performances of the machine learning algorithms adopted for the anomaly detection phase.

At the best of our knowledge, two other datasets have been presented for the analysis of anomalies in cloud-native systems: the Yahoo Webscope S5 [1] and the Numenta Anomaly Benchmark [9]. However, none of them includes memory-related anomalies.

(1) **Yahoo Webscope S5** was created following some recent events in Yahoo's website who almost caused a crash in their website. They provided a dataset that includes a set of anomalies (crashes) of the system. The dataset is composed of 367 time series, each one with length equal to 1,500. To increase the simplicity of the dataset it has been divided into 4 classes with count: 67/100/100/100. The class $A1$ is composed of real data from computational services, while the following three (namely $A2$, $A3$, $A4$) are composed of synthetic anomaly data with increasing complexity. For all time series, the Ground Truth (GT) is available.
(2) **Numenta Anomaly Benchmark (NAB)** is a rigorous benchmark for evaluating real-time anomaly detection algorithms. We followed three requirements to generate the dataset: include all possible types of streaming anomaly, include multiple data metrics and represent common challenges including noise. It is composed of 58 data files each with 1,000-22,000 data instances. The whole dataset has been labeled manually. The main characteristic of the dataset however, is the very rigorous scoring function which weights different kinds of errors differently. The main three points at the foundation of scoring in NAB are: the anomaly window, its scoring function and the application profile. The anomaly window is the range of point which are centered around a GT anomaly, the scoring function use such windows to identify true positives, false positives and false negatives. The grading of the application profile uses a sigmoidal function, which gives higher points to the early detection of anomalies and negative points to the detection outside such windows.

The remainder of this paper is structured as follows. Section 2 presents the RARE dataset while Section 3 describes the technologies adopted to generate it. Section 4 presents the method we adopted to collect the data. In Section 5 we propose a case study to validate the applicability of machine learning on the dataset. Section 5 presents possible applications, and in particular possible machine learning avenues. Section 7, finally draws conclusions and discusses future works.

2 RARE: THE MEMORY ANOMALY DATASET

RARE aims at proposing a dataset generated from a cloud-native system affected by lack of memory in its containers. We define memory anomalies, and in the following only anomalies, increased amount of usage of memory in the containers running the cloud-native system, that might decrease the system performance.

Table 1: List of metrics in the dataset grouped by tool.

TOOL	No. Metrics	Example of Metric
Java_lang	206	java_lang_runtime_starttime
jmx	3	jmx_config_reload_failure_total
kafka	187	kafka_exporter_build_info
kube	62	kube_node_created
kubernetes	1	kubernetes_build_info
node	155	node_arp_entries
prometheus	130	prometheus_engine_queries
workqueue	11	workqueue_adds_total
Other	180	get_token_count

In particular, we considered an anomaly the availability of less than 85% of memory in the containers. In the remainder of this Section, we present the RARE dataset, together with its structure. A detailed description of the system adopted to generate the dataset is available in Section 4.1 while the description of the technologies adopted is available in Section 3.

2.1 The Dataset

In the current version of the RARE dataset, we provide time-series data where each row contains a time stamp plus a single scalar value for a set of 7,062 metrics. The goal of the RARE dataset is to (i) include all types of metrics collected from the industry leading tools such as Prometheus, Kafka, and Kubernetes, (ii) a labelled time-series that indicates where we started to inject the anomalies and when the anomaly ended. In Figure 1, it is possible to see an example of an anomaly present in the dataset.

Targeting a specific range of applications is fundamental in a time-series based dataset. When it comes to training a model to prevent and overcome a specific event in time, it is important to have a dataset which is capable of faithfully describing such an event. It is thus important for RARE to provide a sequence of data which represents the evolution of the state of the nodes before, during and after a data overloading. Therefore, we opted for artificially-generated data to reduce confounding variables, and to ensure that the anomalies are injected because of a specific (artificially generated) event. A real dataset would have not allowed us to identify the exact reason of the anomalies, and even more, the exact event that triggered the anomaly.

The dataset contains a total of 942 unique metrics, a summary of which is described in Table 1. Most metrics have values related to different instances, therefore the full dataset has a total of 7,062 metrics.

2.2 Dataset Structure

The dataset presented in this work consists of two tables, each of them stored in a different CSV file: List_of_anomalies and RARE.

List_of_anomalies includes an explanation of the anomalies injected. As the dataset is artificially-generated, the overloading has been injected through a script, this provides us very accurate labels of the timestamps related to the beginning and the end of any event in the system. An example of 5 entrances of anomalies in the dataset are presented in Table 2. Specifically, we can see the following fields:

Table 2: Example of 5 entrances of the List_of_anomalies.

Timestamp_from	Timestamp_to	Before_anomaly
1579033135	1579033155	11
1579033582	1579033602	10
1579033997	1579034017	8
1579034411	1579034431	7
1579034846	1579034866	13

Table 3: Example of 5 entrances of RARE dataset.

Time	Anomaly	Instance	Node_load1_0	Kafka_server_brokertopicmetrics_bytesinpersec_count_2	machine_memory_bytes_0
1579033025	0	Waiting for Kafka stream to start	0.21	207	2089807872
1579033132	0	Kafka running	1.2	276	2089807872
1579033142	1	Generating anomaly	1.47	414	2089807872
1579033167	0	Waiting for the script to start	0.97	552	2089807872
1579034005	1	Generating anomaly	1.36	828	2089807872

- *Timestamp_from*: when the anomaly injection starts (expressed in Unix/Epoch time [5])
- *Timestamp_to*: when the anomaly injection finishes (also expressed in Unix time [5]).
- *Before_anomaly*: The number of seconds that Kafka has been sending messages before the beginning of the anomaly.

RARE is the file that contains the actual dataset. It includes 10K entries, each of which is characterized by the following variables:

- *Time*: the timestamp for each datapoint, recorded every one second, expressed in Epochs.
- *Anomaly*: a label (boolean) indicating whether the data point is anomalous or not.
- *Instance* which can have four different values: (i) *Waiting for the script to start* which indicates that the Pod is being created after having been destroyed. This action usually takes 300 to 350 seconds, (ii) *Waiting for the Kafka stream to start* which indicates the time between the container starts and the Kafka producer starts to send a bytes stream, (iii) *Kafka running*, meaning that the Kafka producer sends bytes but the anomaly is not being generated, and (iv) *Generating anomaly* indicating that the anomaly is being generated.
- the list of metrics, with one variable for each metric presented. There are a total of 942 unique metrics (summarized in Table 1), with each present for each pod and instance created, for a total of 7,062 metrics.

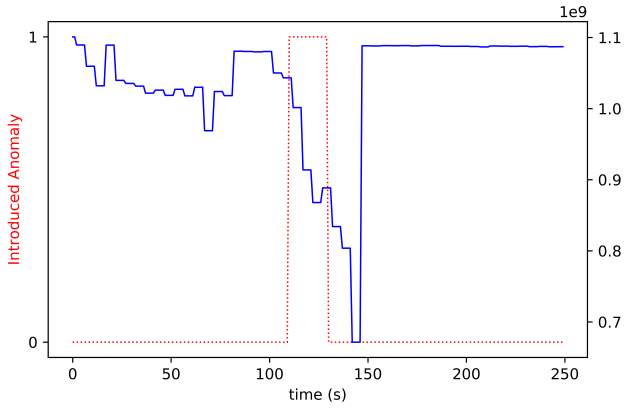


Figure 1: An example of anomaly in the dataset: the red dotted line indicates the presence of the anomaly, while the blue line indicates the memory available (in GB).

3 THE TECHNOLOGIES ADOPTED

In order to generate the dataset we built a microservice-based system using Kubernetes, Docker, Kafka, and Prometheus. In this Section, we briefly introduce these technologies.

- *Docker* [1] allows packaging and deploying of applications in containers. Each container wraps up all it needs to be an independent running system and, even if it is isolated from other containers, it can communicate with the others when requested. For its characteristics a container is usually mistaken for a virtual machine however, compared to the latter, it is lighter as a group of containers are run by a single operative single kernel. For this reason they do not need to start and deploy a whole operative system to work. Moreover, they only use resources which are needed and share such resources without being aware of it, which makes it necessary to use an external entity for such management.
- *Kubernetes*[2] is an open source platform used for running and managing containerized applications across clusters of machines. For each application, Kubernetes is capable of managing the entire life cycle, transforming them into a complete distributed system. Such systems are complete, reliable and scalable. The platform is capable of ensuring such qualities, since it is composed of multiple layers. At the base of such system there are Pods. Pods are containers that are guaranteed to be on the host machine and can share resources without conflict. Their management can be delegated to a controller. A set of Pods working together are defined as a Service while a pod contains a set of containers like Docker.
- *Prometheus*[3] is the most used monitoring system in Kubernetes. It is an open source platform capable of generating humanly readable event logging and metrics. The Prometheus server is periodically scraping the targets to generate such

[1] https://www.docker.com
[2] https://kubernetes.io
[3] https://prometheus.io

data moreover, it has also an auto-discover feature to increase targets. The data model generated is based on key-value pairs, in which no accident is the same as in Prometheus.

- *Kafka*[4] is a scalable messaging system based on publish-subscribe architecture. This means that a publisher sends streams of records so that customers may receive them. In this system, which has been adopted exponentially from some of the most famous web-based companies, messages are organized in topics therefore, the use of keywords help the grouping of messages belonging to the same class. For our purposes the most important characteristics are that it allows fault tolerance and produces real time streaming messages [8].

4 DATA COLLECTION AND PREPARATION

In this Section, we first introduce the system we monitored and how we injected memory anomalies. Then, we describe how we collected the data from the same system.

4.1 The System Monitored

The previously described tools have been assembled to create and deploy the microservice-based system, the anomaly generator systems and the complete infrastructure including Kubernetes, Kafka, and Prometheus as shown in Figure 2.

The first step in developing the system was related to the deployment of Kubernetes on 4 different Virtual Machines (VMs) where one was acting as the master and the other three as workers. We selected VMs with 2 CPUs, 2 GB RAM and Ubuntu Operating Systems. All machine were configured on the same network.

As second step, we set up Kafka through Zookeeper, an open source software for distributed systems coordination [17]. Following, Prometheus and its Alertmanager were installed and then configured using a custom configuration file. Once the latter has been run, it is possible to check the status of Prometheus through its GUI, which is accessible from a web browser. The implementation of Kafka resulted in a "StatefulSet" of 3 Kafka brokers pods and a "StatefulSet" of 3 Zookeeper Pods. In order to allow other applications from the cluster and outside the cluster to access Kafka brokers, a service to expose Kafka endpoints to a static port was created using NodePort. This means that Kafka producers and consumers may be easily created and connected to the brokers. The chart also provides options to configure metrics that Prometheus collects, in this particular case we exported JMX metrics and others provided by Kafka. Prometheus was deployed using and editing some manifest from [16]. It was defined as a Deployment of 1 replica and exposed a NodePort similarly as Kafka. By doing so, Prometheus GUI could be accessed by any browser using the master's IP and the external port provided by NodePort. Prometheus alert manager was set up using the same approach of a Deployment and Service. Additionally two more services were launched to generate more metrics: Kube-state-metrics [6] and Node Exporter [2], the first was responsible for the update of the cluster state, while the second was a hardware information collector. The main elements presented on the cluster were 4 deployments (Kafka brokers, Zookeeper, Prometheus and Kube-state-metrics) and one DaemonSet (NodeExporter).

[4] https://kafka.apache.org

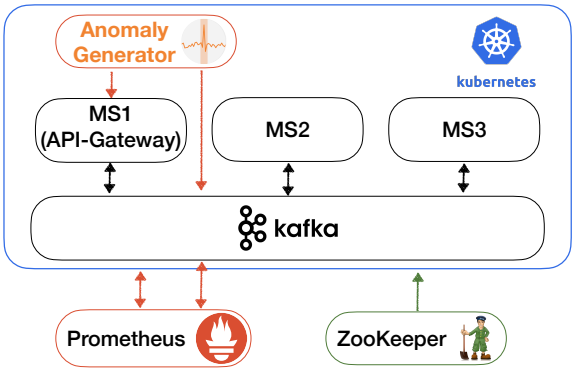


Figure 2: Kubernetes cluster.

The third step was to deploy the system to be monitored (MS1, MS2, and MS3 in Figure 2. Then, once the system was correctly up and running, we needed to generate anomalies (i.e. strange or unexpected conditions that provoke unusual situations).

To generate anomalies, we implemented an "anomaly generator" microservice that provokes a memory anomaly. The anomaly generator microservice starts in a separate Kubernetes Pod inside the cluster, then it connects to a Kafka broker and sends a byte-stream for a random number of seconds until such a request exceeds the maximum allowed and provokes the destruction of the container. The randomization is intended to make the dataset closer to a real life situation, while retaining the advantages of artificially created anomalies. The anomaly generator is based on a script that includes an infinite loop which runs in the background once the connection with the broker is established.

More details on the system implemented and on the configuration adopted can be found in [13].

4.2 Data Collection and Preparation

In this section we provide a small summary of the key points described in the previous section

- Setting-up the system: We first installed the system on a cluster of 4 nodes where one was working as a master and the other 3 as workers. We also installed the monitoring system (i.e. Prometheus) and Kafka.
- Anomalies generation: We created a script to injected anomalies into the system. The scripts starts a Kafka stream, it randomly waits a time frame between 5 and 15 seconds, and after produces the anomaly.
- Anomalies duration: The anomalies have a random duration between 4 and 20 seconds.
- System duration: We executed the system for three hours setting the monitoring systems to collect data every second.
- Data Exporting: The data has been exported in a CSV format.
- Dataset upload: Finally, we uploaded the generated dataset to Figshare [11].

5 A MACHINE LEARNING BASED APPROACH FOR DATASET VALIDATION

In this section we conducted an example case study to understand if it is possible to apply machine learning algorithms to the dataset.

The goal of the case study is to understand whether the anomaly depends on the metrics available in the dataset.

Therefore, we formulated our Research Question (RQ) as

RQ1: Is it possible to predict the memory-anomaly based on the metrics available on the dataset?

5.1 Data Analysis

In order to answer our RQ, we adopted a Random Forest [3] binary classifier, using the anomaly as dependent variable (boolean variable), and the remaining metrics as independent variables.

We fitted the classifier using 1,000 trees as base classifiers. We decided to adopt a Random Forest classifier as it is less prone to over fitting compared to a simpler model, and at the same time has a good level of randomization when sub-sampling the data that it uses for building the model [3]. Also, we didn't choose more advanced tools like deep neural networks, because our scope for this work was to give an example application of machine learning to the RARE dataset.

To assess the detection accuracy of the Random Forest algorithm, we performed a 10-fold cross-validation, dividing the data into ten parts; *i.e.,* we trained the models ten times, always using 1/10 of the data as a testing fold. The data related to each project was split into ten sequential parts, thus respecting the temporal order and the proportion of data for each project. The models were trained iteratively on groups of data preceding the test set. The temporal order was also respected for the groups included in the training set: For example, in fold 1, we used group 1 for training and group 2 for testing; in fold 2, groups 1 and 2 were used for training and group 3 for testing, and so on for the remaining folds.

As accuracy metrics, we first calculated precision and recall. However, as suggested by [15], 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 kinds of errors made.

The contingency matrix (also called confusion matrix) and the related f-measure help to overcome this issue. Moreover, as recommended by Powers [15], the Matthews Correlation Coefficient (MCC) should also be considered to understand any potential disagreement between the actual values and the predictions, as it involves all four quadrants of the contingency matrix.

From the contingency matrix, we retrieved the *true negative rate* (TNR) measure, which measures the percentage of negative samples correctly categorized as negative; the *false positive rate* (FPR), which measures the percentage of negative samples misclassified as positive; and the *false negative rate* (FNR), which measures the percentage of positive samples misclassified as negative. The *true positive rate* measure was left out as it is equivalent to the recall. The way these measures were calculated can be found in Table 4.

5.2 Results

The application of the Random Forest algorithm [3] to the data showed interesting dependencies between the existing metrics.

Table 4: Accuracy Metrics Formulas

Accuracy Measure	Formula
Precision	$\frac{TP}{FP+TP}$
Recall	$\frac{TP}{FN+TP}$
MCC	$\frac{TP*TN-FP*FN}{\sqrt{(FP+TP)(FN+TP)(FP+TN)(FN+TN)}}$
f-measure	$2 * \frac{precision*recall}{precision+recall}$
TNR	$\frac{TN}{FP+TN}$
FPR	$\frac{FP}{TN+FP}$
FNR	$\frac{FN}{FN+TP}$

TP: True Positive; TN: True Negative; FP: False Positive; FN: False Negative

In particular, we can see that the model trained using all the variables, performed remarkably well, confirming the dependency between the metrics and the anomaly injected. The contingency matrix (Table 5) confirm the accuracy of the model, as well as the other accuracy measures collected (Table 6).

Table 5: Contingency matrix

		Predicted	
Actual		*True Positive*	*False Negative*
		122	14
		False Positive	*True Negative*
		6	2361

Table 6: Accuracy Metrics Results

Accuracy Measure	Formula
Precision	95.31%
Recall	89.71%
MCC	92.05%
f-measure	92.42%
TNR	99.75%
FPR	0.25%
FNR	10.29%

We are aware that different metrics with different roles (e.g., metrics of communication between services) can be more frequent than others and might have a different impact on the results. Moreover, we do not exclude the possibility that other statistical or machine learning approaches, such as Deep Learning, might have yielded similar or even better accuracy than our modeling approach.

This validation was mainly aimed at verifying dependencies between the metrics, without considering the evolution of the values. The identification of dependencies in each datapoint, opens for further investigations, and confirm the possibility of conducting time-series analysis.

6 POSSIBLE APPLICATIONS

This dataset will open different avenues for researchers in Machine Learning and in Software Engineering. It will be possible to investigate different machine-learning aspects such as:

- Identify Real Time applications for preventing anomalies. As the RARE is a time based artificial dataset, it relies on specific

time windows which can help the trained machine learning method to learn to forecast the anomaly and therefore prevent the congestion before happening.
- Define Machine learning based methods. In fact, the dataset might be used for building tailored made machine learning methods to tackle the specific case of memory congestion anomaly.
- Propose self-healing mechanism to proactively prevent the anomalies. As an example, when the real time anomaly detection algorithm will detect a possible anomaly, it will be possible to activate the Kubernetes self healing mechanism to prevent the out of memory error.

We are planning to apply different machine learning algorithms to the RARE dataset. Algorithms include decision tree-based, ensemble techniques, neural networks, and others. The comparison will be performed in terms of prediction accuracy, training and test time, but also in terms of resources required. As an example, an algorithm might be much more accurate than another one, but requiring a too long training or testing time, or too much resources.

Moreover, once a suitable machine learning method has been defined, this could be used to test how much time before it is possible to forecast the anomaly, together with what are the essential metrics needed for the forecasting. This can serve as an indicator for practitioners to collect and focus only on a specific set of metrics.

7 CONCLUSIONS

In this paper we present RARE, a dataset containing 10k data points and 7,062 metrics describing artificially generated memory anomaly. The dataset has been meticulously prepared in order to provide a faithful time-based anomaly test bed. The different stages regarding the development of the dataset have been described in detail in the paper. Moreover, to complete such description, a test example has been provided, showing a simple random forest algorithm used for classifying a data point as anomalous or not.

The RARE dataset will allow researchers to benchmark their machine learning algorithms for memory-related anomaly detection. Specifically, some possible applications have been introduced with the hope of stimulating new research questions related to the time based anomaly detection.

Although we believe that the RARE dataset will be used for the development and advancement in the area of machine learning and software engineering, we recognize that the dataset can be further extended and improved. For this reason, future work includes the extension of the dataset, executing different type of systems, and injecting different types of anomalies, including cpu-related anomalies, but also other kinds such as denial of services attacks. Last but not least, we are planning to conduct different experiments, to test the most suitable AI method [10].

REFERENCES

[1] 2015. Yahoo anomaly detection dataset S5. Available: https://webscope.sandbox.yahoo.com/catalog.php?datatype=s&did=70. [Accessed September 2020].

[2] 2020. Prometheus Node Exporter. Available: https://github.com/prometheus/node_exporter. [Accessed September 2020].

[3] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[4] Xin Chen, Charng-Da Lu, and Karthik Pattabiraman. 2014. Failure Analysis of Jobs in Compute Clouds: A Google Cluster Case Study. In *International Symposium on Software Reliability Engineering*. 167–177.

[5] The Open Group. 2019. The Open Group Base Specifications Issue 7, Rationale: Base Definitions, section A.4 General Concepts. (2019).

[6] Elana Hashman. 2019. Operating Within Normal Parameters: Monitoring Kubernetes. , 17 pages.

[7] Shi Jin, Zhaobo Zhang, Krishnendu Chakrabarty, and Xinli Gu. 2017. Changepoint-based anomaly detection in a core router system. In *International Test Conference (ITC)*. 1–10.

[8] Apache Kafka. 2017. Apache Kafka. Available: https://kafka.apache.org/. [Accessed September 2020].

[9] Alexander Lavin and Subutai Ahmad. 2015. Evaluating Real-Time Anomaly Detection Algorithms–The Numenta Anomaly Benchmark. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 38–44.

[10] Valentina Lenarduzzi, Francesco Lomio, Sergio Moreschini, Davide Taibi, and Damian Andrew Tamburri. 2021. Software Quality for AI: Where we are now?. In *International Conference on Software Quality (SWQD 2020)*.

[11] Francesco Lomio, Diego Martínez Baselga, Sergio Moreschini, Heikki Huttunen, and Davide Taibi. 2020. RARE: A Labeled Dataset for Cloud-Native Memory Anomalies. https://zenodo.org/record/3943826#.Xw17DBFRXmE.

[12] Francesco Lomio, Pino Surace, and Davide Taibi. 2020. Anomaly Detection in Cloud-Native Systems: A Case Study in a Large Telecommunication Industry. *Under review* (2020).

[13] Diego Martínez Baselga. 2020. Anomaly detection with Prometheus. http://urn.fi/URN:NBN:fi:tuni-202004294645. *Master Thesis, Tampere University* (2020).

[14] Cristina Monni, Mauro Pezzè, and Gaetano Prisco. 2019. An RBM Anomaly Detector for the Cloud. *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 148–159.

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

[16] Vaibhav Thakur. 2020. K8s monitoring. Available: https://github.com/Thakurvaibhav/k8s/tree/master/monitoring. [Accessed September 2020].

[17] Apache ZooKeeper. 2010. Apache Zookeeper. Available: https://zookeeper.apache.org. [Accessed September 2020].

PUBLICATION IV

Regularity or anomaly? on the use of anomaly detection for fine-grained just-in-time defect prediction

F. Lomio, L. Pascarella, F. Palomba, and V. Lenarduzzi

PUBLICATION V

Towards a robust approach to analyze time-dependent data in software engineering

N. Saarimäki, S. Moreschini, F. Lomio, R. Penaloza, and V. Lenarduzzi

International Conference on Software Analysis, Evolution and Reengineering (SANER), In press.

PUBLICATION VI

A machine and deep learning analysis among sonarqube rules, product, and process metrics for faults prediction

F. Lomio, S. Moreschini, and V. Lenarduzzi

Empirical Software Engineering (EMSE), In press.