Empirical Software Engineering (2022) 27:189
https://doi.org/10.1007/s10664-022-10164-z

# A machine and deep learning analysis among SonarQube rules, product, and process metrics for fault prediction

**Francesco Lomio[1] · Sergio Moreschini[1] · Valentina Lenarduzzi[2]**

Accepted: 30 March 2022 / Published online: 1 October 2022


## Abstract

**Background** Developers spend more time fixing bugs refactoring the code to increase the maintainability than developing new features. Researchers investigated the code quality impact on fault-proneness, focusing on code smells and code metrics.

**Objective** We aim at advancing fault-inducing commit prediction using different variables, such as SonarQube rules, product, process metrics, and adopting different techniques.

**Method** We designed and conducted an empirical study among 29 Java projects analyzed with SonarQube and SZZ algorithm to identify fault-inducing and fault-fixing commits, computing different product and process metrics. Moreover, we investigated fault-proneness using different Machine and Deep Learning models.

**Results** We analyzed 58,125 commits containing 33,865 faults and infected by more than 174 SonarQube rules violated 1.8M times, on which 48 software product and process metrics were calculated. Results clearly identified a set of features that provided a highly accurate fault prediction (more than 95% AUC). Regarding the performance of the classifiers, Deep Learning provided a higher accuracy compared with Machine Learning models.

**Conclusion** Future works might investigate whether other static analysis tools, such as FindBugs or Checkstyle, can provide similar or different results. Moreover, researchers might consider the adoption of time series analysis and anomaly detection techniques.

Communicated by: Foutse Khomh, Gemma Catolino and Pasquale Salza

This article belongs to the Topical Collection: *Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*

✉ Francesco Lomio
francesco.lomio@tuni.fi

Sergio Moreschini
sergio.moreschini@tuni.fi

Valentina Lenarduzzi
valentina.lenarduzzi@oulu.fi

[1] Tampere University, Tampere, Finland

[2] University of Oulu, Oulu, Finland

**Keywords** SonarQube · Software metrics · Fault prediction · Machine learning · Deep learning

## 1 Introduction

Software teams spend a significant amount of time trying to locate defects and fix bugs (Zeller 2009). Fixing a bug involves isolating the part of the code that causes unexpected behavior of the program and changing it to correct the error (Beller et al. 2018). Bug fixing is a challenging task, and developers often spend more time fixing bugs and making the code more maintainable than developing new features (Murphy-Hill et al. 2015; Pan et al. 2009).

Different works addressed this problem (D'Ambros et al. 2010; Osman et al. 2017), relying on different information, such as process metrics (Nagappan and Ball 2005; Moser et al. 2008; Hassan 2009a) (number of changes, recent activity), code metrics (Subramanyam and Krishnan 2003; Gyimothy et al. 2005; Nagappan et al. 2006) (lines of code, complexity) or previous faults (Ostrand et al. 2005; Hassan and Holt 2005; Kim et al. 2007). The research community also considered the impact of different code quality issues on fault-proneness, with a special focus on Fowler's code smells (Palomba et al. 2018; Gatrell and Counsell 2015; D'Ambros et al. 2010; Saboury et al. 2017; Lenarduzzi et al. 2020b).

In our previous works, we investigated the fault-proneness of SonarQube rules, first with machine learning techniques (Lenarduzzi et al. 2020e), and second with classical statistic techniques (Lenarduzzi et al. 2020b). Also, the approaches adopted in our previous work did not allow us to identify the correlation of each individual SonarQube rule with fault-proneness. As a result, developers commonly struggle to understand which metric or SonarQube rules they should consider to decrease the fault-proneness of their code (Vassallo et al. 2018), mainly because the ruleset includes more than 500 rules per development language.

In this paper, we aim at advancing the state of the art on fault-inducing commit prediction based on an in depth investigation among several features, a large number of projects and commits, and multiple Machine learning and Deep Learning classifiers.

Starting from the results obtained in our previous work (Lenarduzzi et al. 2020b), we designed and conducted an empirical study among *29 of the 33 Java projects* of the Technical Debt dataset (Lenarduzzi et al. 2019b) analyzed with SonarQube version 7.5 that violated more than *1.8M of SonarQube rules*, and where the faults were determined applying the SZZ algorithm (Śliwerski et al. 2005). We compared the fault prediction power of different features (SonarQube rule and product and process metrics) using the three most accurate Machine Learning models identified in our previous work (Lenarduzzi et al. 2020b) and two Deep Learning models. Moreover, to increase the validity of our results, we better preprocessed the data to avoid multicollinearity and to account for the unbalanced dataset; we also adopted a more accurate data validation strategy.

The results of our study reveal a number of significant findings. Considering the features selection, SonarQube rules can be used as fault predictors only under specific conditions such as the classifier and the variables preprocessing. Using historical data (Deep Learning) allows for better results (AUC 90% in average) than adopting Machine Learning models. Grouping the SonarQube rules by types positively improves the accuracy only when using Machine Learning models. Also, the rule types grouping reduces the features (predictors) number allowing to manage the time and simplify the process.

However, even if the results regarding SonarQube rules and Machine Learning are contrasting with those obtained in the previous work (Lenarduzzi et al. 2019b), they are more reliable and realistic because of the new preprocessing approach and the more accurate validation strategy.

Looking at the selected product and process metrics, the results clearly identified a set of the metrics which provided a higher accurate fault prediction. Specifically, Rahman and Devanbu (2013) (92.45% in average) and Kamei et al. (2012) (96.53% in average) provide good results both adopting Machine or Deep Learning models. Considering the metrics calculated by SonarQube, only Deep Learning models provide a good level of accuracy (79% in average). Moreover, including the SonarQube rules in all the metrics combinations, the results are always impressive. We reach the best performance (AUC more than 97% on average) when Deep Learning is adopted as model category.

Regarding the best model selection, our results highlighted the higher accuracy performance achieved by Deep Learning models. Compared with Machine Learning models, Deep Learning increases the AUC, enables the correct fault identification, and decreases the probability of incorrect identification.

The contribution of this paper is three-fold:

- A comparison of the prediction power of the fault-proneness of SonarQube rules and product and process metrics
- A comparison of the effectiveness and accuracy of Machine Learning and Deep Learning models for the identification of fault-inducing SonarQube rules and product and process metrics
- A set of important features (SonarQube rules, product and process metrics) and models to achieve an accurate fault prediction.

The remainder of this paper is structured as follows. In Section 2 we introduce the background in this work, introducing the original study, SonarQube violations and the different machine and deep learning models. Section 3, describes the case study design, while Section 4 presents the obtained results. Section 5 discusses the results, and Section 6 identifies threats to validity. Section 7 describes the related works, while Section 8 draws the conclusion highlighting the future works.

## 2 Background

In this Section, we illustrate the background of this work, introducing our previous study (called "previous"), SonarQube static analysis tool, and the Machine and Deep Learning models adopted in this study.

### 2.1 The Previous Study

In this Section, we illustrate the previous study (Lenarduzzi et al. 2020e) and the obtained results. Moreover, we explain the reasons why we conducted this study, and we compare it with the previous one. We followed the guidelines proposed by Carver for reporting replications (Carver 2010).

We decided to consider for this study, only the paper (Lenarduzzi et al. 2020e) since – as far as we know – this is the only one that provide a ranking of importance of SonarQube issues that could induce bugs in the source code. Moreover, two of the authors of this paper are also authors of the previous study.

The previous study investigated the fault-proneness of SonarQube rules in order to understand if rules classified as "Bug" are more fault-prone than security and maintainability rules ("vulnerability" and "code smell"). Moreover, the previous study evaluated the accuracy of the SonarQube quality model for the bugs prediction. As context, the previous study analyzed 21 randomly selected mature Java projects from the Apache Software Foundation. All the commits of the projects were analyzed with SonarQube (version 6.4), and the commits that induced a fault were determined applying the SZZ algorithm (Śliwerski et al. 2005). The SonarQube rules fault proneness were investigated with seven Machine Learning algorithms (Decision Trees (Breiman et al. 1984), Random Forest (Breiman 2001), Bagging (Breiman 1996), Extra Trees (Geurts et al. 2006), Ada Boost (Freund and Schapire 1997), Gradient Boost (Friedman 2001), XG Boost Chen and Guestrin 2016). Results show that only a limited number of SonarQube rules can really be considered fault-prone.

Differently from the previous study (Table 1), we considered the *29 Java projects* of the Technical Debt dataset (Lenarduzzi et al. 2019b), analyzed with SonarQube version 7.5, that include more than *1.8M SonarQube rules* violated, and on which there was calculated *24 software metrics*, and where the faults are determined applying the SZZ algorithm (Śliwerski et al. 2005). Moreover, we considered process and product metrics proposed by Rahman and Devanbu (2013) and Kamei et al. (2012) to corroborate the software metrics included in the SonarQube suite. We adopted *Deep Learning models*, and we made a comparison between the detection accuracy of Deep Learning and Machine Learning models in order to identify which ones better predict a fault. We adopted the three Machine Learning models that exhibit the best accuracy performance (AUC = 80%) in the previous study.

In order to improve the previous results, we adopted a data pre-processing step to check for multicollinearity between the variables. This was done using the *Variable Inflation Factor* (VIF) (O'Brien 2007). Moreover, the authors reported that the commits labelled as fault inducing account for less than 5% of the total number of commits considered. This causes a highly unbalanced dataset, where the positive class (fault-inducing commit) accounts for less than 5% of the total number of samples. This type of data negatively impacts the performance of normal classifiers (both Machine Learning and Deep Learning). For this reason, we adopted an oversampling technique to rebalance the dataset. For this step we used a *Synthetic Minority Oversampling Technique* (SMOTE) (Chawla et al. 2002).

**Table 1** Study design comparison

| | Previous study (Lenarduzzi et al. 2020e) | New study |
|---|---|---|
| #Projects | 21 | 32 |
| #Commits | 39,518 | 77,932 |
| SonarQube tool version | 6.4 | 7.5 |
| SonarQube rules | 231,453 | 1,941,508 |
| Faults | 4,505 | 40,890 |
| Product and process metrics | 0 | 48 |
| Machine Learning models | 8 | 3[a] |
| Deep Learning models | 0 | 2 |

[a]The best ones among the 8 adopted in Lenarduzzi et al. (2020e)

### 2.2 SonarQube

SonarQube is one of the most common open-source static code analysis tools adopted both in academia (Lenarduzzi et al. 2017, 2020c) and in industry (Vassallo et al. 2019a). 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 rule 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 "represent something wrong in the code" and that will soon be reflected in a bug. "Code 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 and Beck (1999) but to a different set of rules. Fowler and Beck (1999) 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 by 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:[1] Blocker, Critical, Major, Minor, and Info.

In this work, we focus on the SonarQube violations, which are reliability rules classified as "bugs" by SonarQube, as we are interested in understanding whether they are related to faults. Moreover, we consider the 32 software metrics calculated by SonarQube. In the replication package (Section 3.5) 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 SonarQube violations with their ID (squid). The complete list of violations can be found in the file "SonarQube-rules.xsls" in the online raw data.

### 2.3 Machine Learning models

We selected the three machine learning models that turned out to be the most accurate in the faults prediction in our previous study (Lenarduzzi et al. 2020e): Random Forest (Breiman 2001), Gradient Boost (Friedman 2001), and XGboost (Chen and Guestrin 2016). As for Lenarduzzi et al. (2020e), Gradient Boosting and Random Forest are implemented using the library *Scikit-Learn*[2] with their default parameters. XGBoost model is implemented using the XGBoost library.[3] All the classifiers are fitted using 100 estimators.

[1] SonarQube Issues and Rules Severity:' https://docs.sonarqube.org/display/SONAR/Issues
[2] https://scikit-learn.org
[3] https://xgboost.readthedocs.io

**Random Forest** Random Forest (Breiman 2001) is an ensemble technique based on decision trees. The term ensemble indicates it uses a set of "weak" classifiers that help solve the assigned task. In this specific case, the week classifiers are multiple decision trees.

Using a randomly chosen subset of the original dataset, an arbitrary amount of decision trees is generated (Breiman 1996). In the case of random forest, the subset is created with replacement, meaning that a sample can appear multiple times. Moreover, it is also chosen a subset of the features of the original dataset, without replacement (appear only once). This helps reducing the correlation between the individual decision trees. With this setup, each tree is trained on a specific subset of the data, and it can make prediction on unseen data. The final classification given by the Random Forest is decided based on the majority vote of the individual decision trees.

The process of averaging the prediction of multiple decision trees, allows the random forest classifier to better generalize the data and overcome the overfitting problem to which decision trees are prone. Also, using a randomly selected subset of the original dataset, the individual trees are not correlated with one another. This is particularly important in our case, as in this study we are using a high number of features, and therefore the probability of the features being correlated to one another, increases.

**Gradient Boosting** Gradient Boosting (Friedman 2001) is another ensemble model which, compared to the random forest, generates the individual weak classifiers sequentially during the training process. In this case, we are also using a series of decision trees as weak classifiers. The gradient boosting model creates and trains only one decision tree at first. After each iteration, another tree is grown to improve the accuracy of the model and minimize the loss function. This process continues until a predefined number of decision trees has been created, or the loss function no longer improves.

**XGBoost** The last classical model used, is the XGBoost (Chen and Guestrin 2016). This is nothing but a better-performing implementation of the Gradient Boosting algorithm. It allows for faster computation and parallelization compared to gradient boosting. It can therefore result in better computational and overall performance compared to the latter, and can be more easily scaled for the use with high dimensional data, as it is the one we are using.

### 2.4 Deep Learning Models

Deep learning is a subset of Machine Learning (ML) based on the use of artificial neural networks. The term deep indicates the use of multiple layers in the neural network architecture: the classical artificial neural network is the multilayer perceptron (MLP), which comprises an input layer, output, and a hidden layer in between. This structure limits the quantity of information that the network can learn and use for its task. Adding more layers allows the network to increase the amount of information that the network can extract from the raw input, improving its performance.

While machine learning models become progressively better at whatever their function is, they still need some guidance, especially in how the features are provided in input. In most cases, it is necessary to perform some basic to advance feature engineering before feeding them to the model for training. Deep learning models, on the other hand, thanks to their ability to progressively extract higher-level features from the input in the multiple

layers of their architecture, require little to no previous feature engineering. This is particularly helpful when dealing with high-dimensional data.

Also, as seen Section 2.3, most of the classical machine learning models suffer from performance degradation when dealing with large datasets and high dimensional data. Deep learning models, on the other hand, can be helpful as thanks to the different types of architectures, they can be more scalable and flexible.

In this Section, we briefly introduce the Deep Learning-based techniques we adopted in this work: *Fully Convolutional Network* (FCN) (Wang et al. 2017) and *Residual Network* (ResNet) (Wang et al. 2017).

These two approaches are adopted from Fawaz et al. (2019), where it was shown that their performance is superior to multiple other methods tested. In particular, *Fawaz et al.* showed in their work that the FCN and the ResNet were the best performing classifiers in the context of the multivariate time series classification. This conclusion was obtained by testing 9 different deep learning classifiers on 12 multivariate time series datasets.

**Residual Network** The first deep learning model used is a residual network (ResNet) (Wang et al. 2017). Among the many different types of ResNet developed, the one we used is composed of 11 layers, of which 9 are convolutional. Between the convolutional layers, it has some shortcut connection which allows the network to learn the residual (He et al. 2016). 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 used the ResNet shown in Fawaz et al. (2019). It consists of 3 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 (Lin et al. 2013) instead of the more traditional fully connected layer. The GAP layer allows the features maps of the convolutional layers to be recognised as a category confidence map. Moreover, it reduces the number of parameters to train 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) (Wang et al. 2017). 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 (Long et al. 2015). 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. (2017), 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 Fawaz et al. (2019). This implementation consists of 3 convolutional blocks, each composed of a 1-dimensional convolutional layer and by a batch normalization layer (Ioffe and Szegedy 2015). It uses a rectified linear unit (ReLU) (Nair and Hinton 2010) 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. tThis 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 (Wang et al. 2017). Moreover, it is smaller than the ResNet, which would make the FCN model more computationally efficient.

## 3 Empirical Study Design

We designed our empirical study based on the guidelines defined by Runeson and Höst (2009). 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.

### 3.1 Goal and Research Questions

The goal of this paper is to conduct an in depth investigation among several features, a large number of projects and commits, and multiple Machine learning and Deep Learning classifiers to predict the commits fault proneness. This study allows us to: 1) corroborate our assumption that SonarQube rules fault proneness was low, extending our previous works Lenarduzzi et al. 2020b, e, and 2) build models to predict whether a commit is fault-prone with the highest accuracy as possible. As features, we selected the SonarQube rules and different product and process metrics (Section 3.4).

The perspective is of both practitioners and researchers since they are interested in understanding which variables.

Based on the aforementioned goal, we derived the following Research Questions ($RQ_s$).

**$RQ_1$** What is the fault proneness of the SonarQube rules?
**$RQ_2$** What is the fault proneness of product and process metrics?
**$RQ_3$** To what extent can SonarQube rules impact the performance of fault prediction models that leverage process and product metrics?
**$RQ_4$** Which is the best combination of features and the best model for the fault prediction?

More specifically, in **$RQ_1$** we aim at investigating the impact of all the SonarQube rules on fault-proneness. The goal is to understand how accurate the prediction can be for fault-proneness if developers do not violate all the SonarQube rules. To provide a complete evaluation, we considered all the SonarQube rules first, and then grouped by type (Bug, Code Smell, and Vulnerability). We selected SonarQube, since it is by far one of the most popular tools and its popularity is increased in the last years, considering discussion in platforms such as Stack Overflow, LinkedIn, and Google groups (Vassallo et al. 2018; Lenarduzzi et al. 2021a, d; Avgeriou et al. 2020). However, as reported by Vassallo et al. (2018), developers commonly get confused by the large number of rules, especially because their severity assigned by SonarQube is not actually correlated with the fault proneness (Lenarduzzi et al. Lenarduzzi et al. 2020b, e).

Software metrics have been considered good predictors for fault-proneness for several decades (D'Ambros et al. 2010; Pascarella et al. 2019). Therefore, in **$RQ_2$** we are interested in investigating the fault proneness of different software metrics combined, including the ones proposed by Rahman and Devanbu (2013), Kamei et al. (2012), and SonarQube suites. In order to have a baseline for the next RQ, in this RQ we aim at investigating the impact of each product and process metrics set on fault proneness.

In **$RQ_3$**, we assess the actual prediction capability using the relevant features coming from the previous research questions ($RQ_1$ and $RQ_2$) when predicting the presence of a fault in the source code.

Finally, in **$RQ_4$**, thanks to the achieved results for each feature and model, we identify their best combination of predictors and models that allows developers to reach the highest accuracy when predicting a fault in the source code.

### 3.2 Study Context

As context, we considered the projects included in the Technical Debt Dataset (Lenarduzzi et al. 2019b). The data set contains 33 Java projects from the Apache Software Foundation (ASF) repository.[4] The projects in the data set were selected based on "criterion sampling" (Patton 2002), that fulfill all of the following criteria: developed in Java, older than three years, more than 500 commits and 100 classes, and usage of an issue tracking system with at least 100 issues reported. The projects were selected also maximizing their diversity and representation by considering a comparable number of projects with respect to project age, size, and domain. Moreover, the 33 projects can be considered mature, due to the strict review and inclusion process required by the ASF. Moreover, the included projects regularly review their code and follow a strict quality process.[5] More details on the data set can be found in Lenarduzzi et al. (2019b).

For each project, Table 2 reports the number of commits analyzed, the number of faults detected, and the number of occurrences of SonarQube rules violated.

### 3.3 Data Collection

The Technical Debt Dataset (Lenarduzzi et al. 2019b) contains the information of the analysis of the commits of the 33 Open Source Java projects. In this work, we considered the following information, as depicted in Fig. 1:

- *SonarQube Rules Violations.* We considered the data from the Table "SONAR_ISSUES" that includes data on each rule violated in the analyzed commits. The complete list of rules is available online[6] but can also be found in the file "sonar_rules.csv" of the Technical Debt Dataset while the diffuseness of each rule is reported in Saarimäki et al. (2019). As reported in Table 2, the analyzed projects violated 174 SonarQube rules for 1,914,508 times. Since in our previous work (Lenarduzzi et al. 2020b) we found incongruities in the rules type and severity assigned by SonarQube, we decided to consider all the detected rules. Table 3 shows the SonarQube ruled violated grouped by type and severity.
- *Product and Process Metrics.* We considered the *24 software metrics* measured by SonarQube (Table "SONAR_MEASURES" of the Technical Debt data set) as listed in Table 4, related to
  - Size (11 types)
  - Complexity (5 types)
  - Test coverage (4 types)
  - Duplication (4 types)
- *Fault-inducing and Fault-fixing commits identification.* In the dataset, the fault-inducing and fault-fixing commits are determined using the SZZ algorithm (Śliwerski et al. 2005; Lenarduzzi et al. 2020a) and reported in the Table "SZZ_FAULT_INDUCING_COMMITS". The SZZ algorithm identifies the fault-introducing commits from a set of fault-fixing commits. The fault-introducing commits are extracted from a bug tracking system such as Jira or looking at commits that state

[4] http://apache.org
[5] https://incubator.apache.org/policy/process.html
[6] https://rules.sonarsource.com/java

**Table 2** The selected projects

| Project | #Commits | #Faults | # SQ rules | |
|---|---|---|---|---|
| | | | Violated | Occurrences |
| Accumulo | 2,641 | 2,250 | 118 | 1,429,757 |
| Ambari | 13,397 | 17,722 | 110 | 41,612 |
| Atlas | 2,336 | 1,990 | 111 | 35,776 |
| Aurora | 4,012 | 628 | 90 | 7526 |
| Batik | 2,097 | 1,160 | 114 | 31,691 |
| Beam | 2,865 | 1,723 | 109 | 8,449 |
| Bcel | 10,210 | 3,218 | 98 | 85,018 |
| Beanutils | 1,324 | 242 | 81 | 5,182 |
| Cli | 1,192 | 346 | 81 | 37,408 |
| Codec | 896 | 182 | 65 | 58,073 |
| Cocoon | 1,726 | 327 | 131 | 2,041 |
| Collections | 2,982 | 135 | 103 | 11,118 |
| Configuration | 2,895 | 73 | 96 | 5,612 |
| Deamon | 980 | 190 | 30 | 393 |
| Dbcp | 1,861 | 284 | 79 | 3,696 |
| Dbutils | 645 | 159 | 40 | 644 |
| Digester | 2,145 | 149 | 72 | 4,947 |
| Exec | 617 | 444 | 57 | 762 |
| Felix | 596 | 147 | 104 | 11,340 |
| FileUpload | 922 | 282 | 52 | 769 |
| Httpcomponents Client | 2,867 | 463 | 97 | 10,803 |
| HttpComponents Core | 1,941 | 188 | 84 | 9,531 |
| Io | 2,118 | 368 | 85 | 5849 |
| Jelly | 1,939 | 56 | 77 | 5,060 |
| Jexl | 1,551 | 119 | 101 | 34,994 |
| Jxpath | 597 | 265 | 71 | 4,951 |
| MINA Sshd | 1,370 | 1,588 | 97 | 9,031 |
| Net | 2,088 | 438 | 86 | 41,340 |
| Ognl | 608 | 3,415 | 90 | 4,945 |
| Santuario | 2,697 | 1,302 | 107 | 22,398 |
| Validator | 1,339 | 397 | 61 | 2,050 |
| Vfs | 2,067 | 84 | 97 | 3,719 |
| Zookeeper | 411 | 1,859 | 70 | 5,023 |
| **Sum** | **77,932** | **40,470** | **2,864** | **1,941,508** |

that they are fixing an issue. A complete description of the steps adopted in the SZZ algorithm is available in Śliwerski et al. (2005).

Moreover, to enrich the data regarding the *product and process metrics* contained in the dataset, we considered the product and process metrics proposed by Rahman and Devanbu (2013) and Kamei et al. (2012), implemented by Pascarella et al. (2019). Moreover,

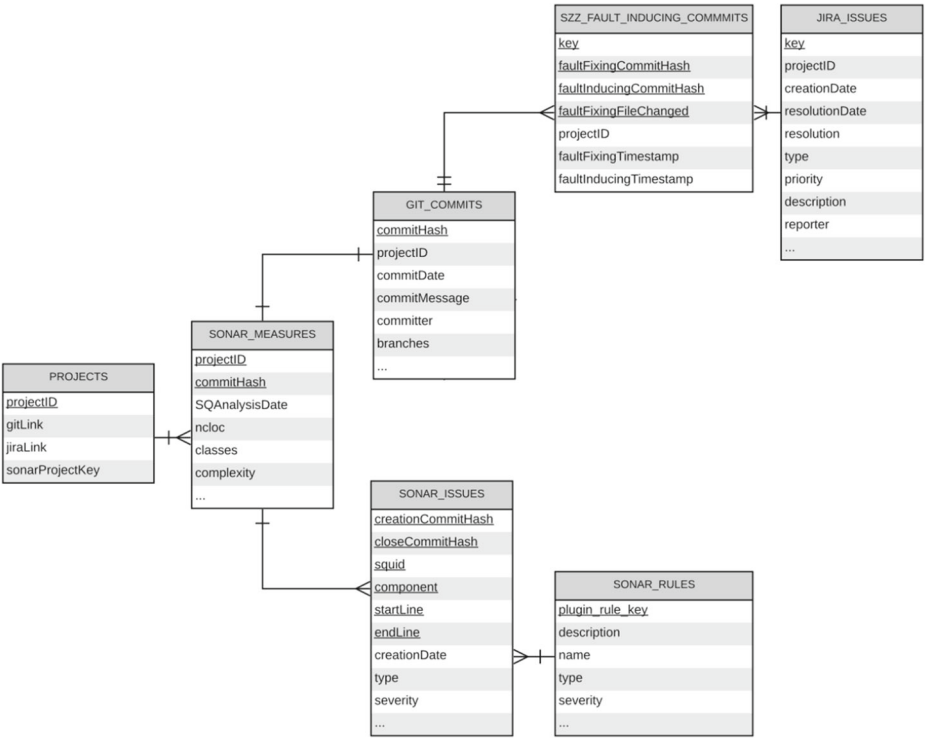


**Fig. 1** Technical Debt Dataset Tables (Lenarduzzi et al. 2019b) considered in this study

these metrics were previously validated in the context of fine-grained just-in-time defect prediction. These metrics cover various aspects of the development process (Table 5):

- Developers' expertise (e.g., the contribution frequency of a developer Kamei et al. 2012)
- The structure of changes (e.g., the number of changed lines in a commit Rahman and Devanbu 2013)
- The evolution of the changes (e.g., the frequency of changes Rahman and Devanbu 2013)
- The dimensional footprint of a committed change (e.g., the relation between uncorrelated changes in a commit Tan et al. 2015).

### 3.4 Data Analysis

In this Section, we report the data analysis protocol adopted in this study including data preprocessing, data analysis, and accuracy comparison metrics.

#### 3.4.1 Data Preprocessing

In order to investigate our RQs we need to preprocess the data available in the Technical Debt Dataset. Moreover, since we are planning to adopt machine learning and Deep

**Table 3** Type and severity of SonarQube rules violated in our projects

| SonarQube rules | | # | Occurrences |
|---|---|---|---|
| Type | Bugs | 37 | 22,620 |
| | Code Smells | 130 | 1,861,999 |
| | Vulnerability | 7 | 57,489 |
| Severity | Blocker | 8 | 18,083 |
| | Critical | 42 | 143,293 |
| | Major | 90 | 983,647 |
| | Minor | 32 | 727,155 |
| | Info | 2 | 69,330 |

Learning techniques, we need to preprocess the data accordingly to the models we aim to adopt.

The preprocessing was composed of three steps:

- Data extraction from the Technical Debt Dataset
- Data pre-processing
- Data preparation for the Machine Learning Analysis
- Data preparation for the Deep Learning Analysis

**Data extraction from the Technical Debt Dataset** The data in the tables SZZ_FAULT_INDUCING_COMMITS, and SONAR_MEASURES of the Technical Debt Dataset already list the information per commit. However, the table SONAR_ISSUES contains one row for each file where a rule has been violated. Therefore, we extracted a new table by means of an SQL query (see the replication package for details Lomio et al. 2022). The result is the new table SONAR_ISSUE_PER_COMMIT. Then, we joined the newly created table SONAR_ISSUE_PER_COMMIT with the tables SZZ_FAULT_INDUCING_COMMITS and SONAR_MEASURES using the commit hash as key. This last step resulted in the final dataset that we used for our analysis (Table FullTable.csv in the replication package Lomio et al. 2022), which contains the following information: the commit hash, the project to which the commit refers to, the boolean label *Inducing*, which indicates if the commit is fault inducing or not, and the set of sonar measures and sonar issues introduced in the commit.

Moreover, we calculated the software metrics proposed by Rahman and Devanbu (2013) and Kamei et al. (2012) according to Pascarella et al. (2019) procedure. Pascarella et al. (2019) provided a publicly accessible replication package with all the scripts used to compute the metrics. The tool collects the new metrics as soon as a new file $F_i$ was added to a repository, (2) updated the metrics of $F_i$ whenever a commit modified it, (3) kept track of possible file renaming by relying on the GIT internal rename heuristic and subsequently updating the name of $F_i$, and (4) removed $F_i$ in the case it was permanently deleted.

Due to the characteristics of the projects, we were able to calculate the metrics proposed by Rahman and Kamei only on 29 of the 33 projects, leaving out the following projects: Batik, Beam, Cocoon, and Santuario. In order to be able to compare the results obtained using the different metrics as features, we excluded these projects also for the analysis with the SonarQube rules.

We combined the metrics by a step-wise method: we grouped the metrics based on Rahman and Devanbu (2013) + Kamei et al. (2012), Rahman and Devanbu (2013) + SonarQube metrics, and Kamei et al. (2012) + SonarQube metrics. Finally, we also considered all the

**Table 4** Product and process metrics detected by SonarQube

| 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 5** Product and process metrics proposed by Rahman and Devanbu (2013) and Kamei et al. (2012) (from Pascarella et al. 2019)

| | Metric | Description |
|---|---|---|
| Rahman and Devanbu (2013) | COMM | The cumulative number of changes in a given file up to the considered commit. |
| | ADEV | The cumulative number of active developers who modified a given file Rahman and Devanbu (2013)up to the considered commit. |
| | DDEV | The cumulative number of distinct developers contributed to a given file up to the considered commit. |
| | ADD | The normalized number of lines added to a given file in the considered commit. |
| | DEL | The normalized number of lines removed from a given file in the considered commit. |
| | OWN | The value indicating whether the owner of the file does the commit. |
| | MINOR | The number of contributors who contributed less than 5% of a given file up to the considered commit. |
| | SCTR | The number of packages modified by the committer in the considered commit. |
| | 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 has been involved. |
| | NSCTR | The number of different packages touched by the developer in commits where the file has been modified. |
| | 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. (2012) | ND | The number of directories involved in a commit. |
| | ENTROPY | The distribution of the modified code across each given file in the considered commit. |
| | LA | Ten number of lines added to the given file in the considered commit (absolute number of the ADD metric). |
| | LD | The number of lines removed from the given file in the considered commit (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. |

metrics together. Based on this grouping, we designed seven different metrics combinations. We also extended this grouping in order to combine each of the metrics also with SonarQube rules and SonarQube rules type, hence resulting in 14 additional combinations. The full list of combinations can be seen in Table 6.

The complete process is depicted in Fig. 2.

**Data Pre-processing** As recommended in literature, we applied a set of pre-processing steps to avoid bias in the interpretation of the results (Tantithamthavorn and Hassan 2018).

Firstly, each SonarQube violation has been normalized for each project, so that the impact of the specific violation becomes more evident.

We applied a *feature selection* method to remove correlated variables that provide the classifiers with the same (or similar) information, and that might cause them not to be able to derive the correct explanatory meaning of the features. This step allows avoiding multi-collinearity (O'Brien 2007). We exploited the Variable Inflation Factor (VIF) method (O'Brien 2007): for each independent variable, the `VIF` function measures how much the variance of the model increases because of collinearity. The features having a `VIF`

**Table 6** The selected features

| Subset | Features selected | # Samples |
|---|---|---|
| SonarQube Rules | 90 | 59,912 |
| SQ Rules Type | 3 | 59,912 |
| SQ Metrics | 9 | 59,912 |
| Kamei et al. (2012) | 8 | 59,912 |
| Rahman and Devanbu (2013) | 9 | 59,912 |
| Kamei et al. (2012) + Rahman and Devanbu (2013) | 15 | 59,912 |
| SQ Metrics + Kamei et al. (2012) | 17 | 59,912 |
| SQ Metrics + Rahman and Devanbu (2013) | 18 | 59,912 |
| SQ Metrics + Kamei et al. (2012) + Rahman and Devanbu (2013) | 24 | 59,912 |
| SQ Rules Type + SQ Metrics | 12 | 59,912 |
| SQ Rules Type + Kamei et al. (2012) | 11 | 59,912 |
| SQ Rules Type + Rahman and Devanbu (2013) | 12 | 59,912 |
| SQ Rules Type + Kamei et al. (2012) + Rahman and Devanbu (2013) | 18 | 59,912 |
| SQ Rules Type + SQ Metrics + Kamei et al. (2012) | 20 | 59,912 |
| SQ Rules Type + SQ Metrics + Rahman and Devanbu (2013) | 21 | 59,912 |
| SQ Rules Type + SQ Metrics + Kamei et al. (2012) + Rahman and Devanbu (2013) | 27 | 59,912 |
| SQ Rules + SQ Metrics | 99 | 59,912 |
| SQ Rules + Kamei et al. (2012) | 98 | 59,912 |
| SQ Rules + Rahman and Devanbu (2013) | 98 | 59,912 |
| SQ Rules + Kamei et al. (2012) + Rahman and Devanbu (2013) | 103 | 59,912 |
| SQ Rules + SQ Metrics + Kamei et al. (2012) | 107 | 59,912 |
| SQ Rules + SQ Metrics + Rahman and Devanbu (2013) | 107 | 59,912 |
| SQ Rules + SQ Metrics + Kamei et al. (2012) + Rahman and Devanbu (2013) | 112 | 59,912 |

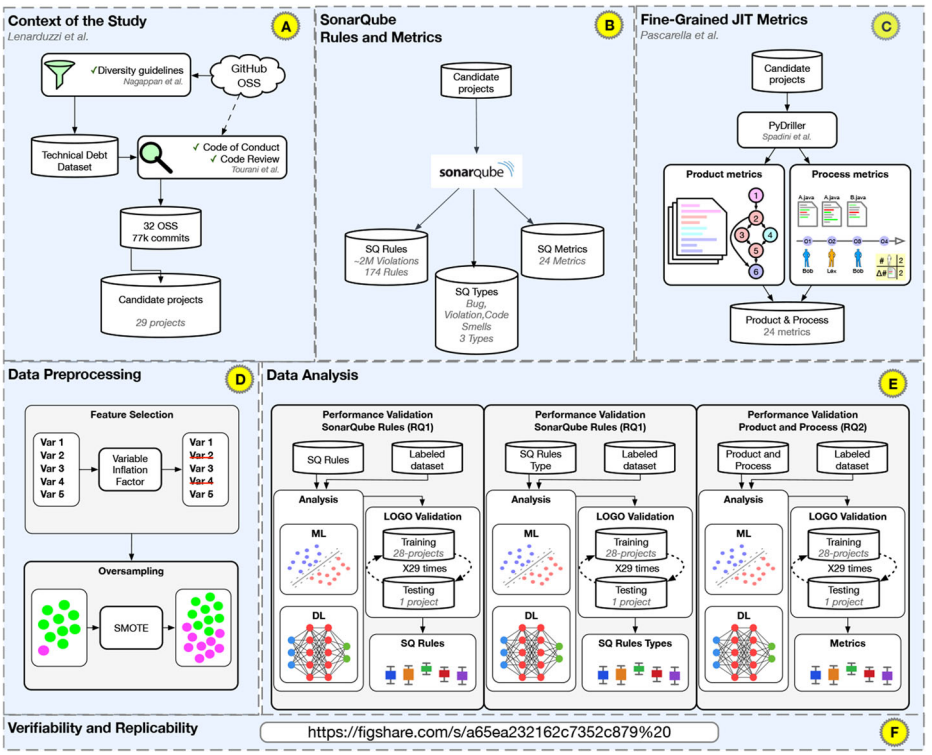


**Fig. 2** The data preprocessing process

coefficient higher than 5 were removed; the process was repeated, iteratively, until the point where all the remaining features had a VIF coefficient lower than the defined threshold.

Since we have an imbalanced dataset, with the commits labelled as fault inducing accounting for less than 5% of the total number of commits considered, we included an oversampling step to improve the performance of the classifiers used. 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., faulty 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. Particularly, SMOTE requires the presence of at least two samples of the minority class to be able to replicate them and properly oversample the dataset. The total number of samples considered for the analysis after the SMOTE was applied, along with the number of features selected through the VIF method for each subset considered, can be found in Table 6.

Moreover, since our commit data is dependent on the time, we also included Deep Learning models, in order to include the effect of past commits in determining the faultiness of the current ones. Compared to Machine Learning models, it is, in fact, possible to include also past data as input, instead of only the current data point.

**Data Preparation for the Machine Learning Analysis** In order to predict if a commit is fault-inducing or not, based on the violation of a SonarQube rule or to the change of a metric, we identified the fault inducing (Boolean) variable as the target (dependent) variable.

The machine learning models described in Section 2.3, allow only to have a two-dimensional input (N,M), where N is the number of samples and M is the number of features. This means that we can classify a commit as fault inducing or not, only based on the information related to that commit itself: we cannot include the commit's history. For this reason, to prepare the data for answering RQs, for each commit, we selected the target variable, which is the boolean label *Inducing*. As input features, we prepared multiple sets, including SonarQube rules and SonarQube rules type ($RQ_1$), product and process metrics ($RQ_2$), and their combinations ($RQ_3$).

It is important to notice that at this point, we are interested in classifying a snapshot of the commit as fault inducing or not; therefore, the time dependency information is not taken into account.

**Data Preparation for the Deep Learning Analysis** The deep learning models described in Section 2.4, allow the use of three-dimensional input (N,h,M), where N and M are the numbers of samples and features, as for the machine learning models, while h indicates the number of commits in each sample. This means that we are able to include the features related to the past commits in the classification of another commit (Fig. 3): we can include the history of the commit and are not limited to using only its current status.

For this reason, we had to reshape the data in order to include the past status of the commits. We used the previous 10 commits as input variables for our models and the label of the following commit as the target variable. Going more in detail, as we have multiple projects in our dataset, we first divided the data into subsets, including only one project. This helps us include only commits from the same project in each sample. After doing this, we reshaped the data using a rolling window of length 10 and step 1, selecting 10 commits and storing the following commit label as target variable. We did this iteratively for all the commits for each project. Similarly to what was done for the machine learning case, we prepared multiple sets of inputs, including SonarQube rules and SonarQube rules type ($RQ_1$), product and process metrics ($RQ_2$), and their combinations ($RQ_3$).

Once the new samples are obtained, they are shuffled and divided into train and test sets. Contrary to the machine learning case, here we take into account the time dependency

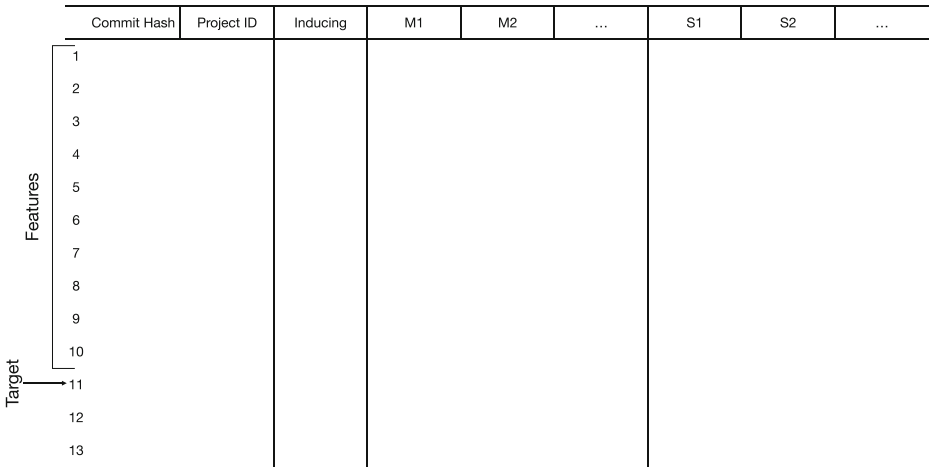


**Fig. 3** The Deep Learning preprocessing

between commits. Still, it is indeed important to notice that this is done in each individual sample. Therefore it is not necessary to consider any temporal order in the train-test split.

### 3.4.2 Data Analysis

We first analyzed the fault-proneness of SonarQube rule ($RQ_1$) and of software metrics ($RQ_{2.1}$) with the three Machine Learning models that better performed on this task in our previous work (Lenarduzzi et al. 2020e). Then, we applied Deep Learning models on the same data to get better insights of the data with more advanced analysis techniques. Finally, we compared the results obtained and applied statistical tests to assess the results.

**Machine Learning Analysis** The three machine learning models presented in Section 2.3, were all implemented using *Scikit-learn* library, except for the XGBoost model, implemented using its own library. All the classifiers were trained using 100 decision trees. The models were trained using a LOGO (Leave One Group Out) validation strategy. All three ML models were run on an Intel Xeon W-2145 with 16 cores and 64GB of RAM.

**Deep Learning Analysis** The deep learning models described in Section 2.4, were implemented in TensorFlow (Abadi et al. 2015) and Chollet et al. (2015), using a similar approach as Fawaz et al. (2019). Both models were trained for 50 epochs, with a mini-batch size of 64 and using as optimizer the Adadelta algorithm (Zeiler 2012), which allow the model to adapt the learning rate. In order to better compare the results with the ones obtained using classical machine learning methods, also the deep learning models were trained using a LOGO validation strategy. Both models have been trained on a computational cluster with a total of 32 NVIDIA Tesla P100 and 160 CPU cores specific for training deep learning models. Each of our model had available 1 NVIDIA Tesla P100 with 16GB of VRAM, 1 CPU core, and 40GB of RAM.

**Accuracy Comparison** As validation technique we adopted the Leave One Group Out (LOGO) validation. This technique divides the dataset into train and test sets using a 'group as discriminant (in our case the *project* is used). All the groups but one are used to train the model, and the remaining is used for testing. This is done for each group in the dataset. This means that $n$ models are trained, with $n$ the number of projects in our data. For each fold, $n - 1$ groups are used for training, and 1 for testing. This means that for our analysis, the training set was composed 28 projects. The remaining 1 project was used to validate the model. This process was repeated 29 times, so that all the projects in the dataset were in the test set exactly once. It is important to highlight that the commit of a project cannot be split between train and test set. This constraint avoids the possible bias due to the time-sensitive nature of code commits: in other words, we never allow a commit belonging to a project to be seen by the model before the train.

The selection of the LOGO validation technique was based on the need to have a validation strategy which would minimize the possible bias given the nature of the data that we had for our analysis. More specifically, a normal *k-fold cross-validation* would not be suitable as it would include commits from projects in the test set, already in the train set, resulting in a bias classification. Also, a *time based* validation would not work with our data as there would be many folds in which there would not be any fault-inducing commit (as they represent less than 5% of the data), hence the classifiers used would not work. This problem would arise also considering a *within project* validation, especially for those projects that had very few fault-inducing commits. tThis validation could be used without

any probelm with larger projects (i.e., Ambari, Bcel), but it would leave out many of the smaller projects which are necessary to strengthen and better generalize our results. It is obvious that also using a time based validation, mixing all commits from all projects would create a bias as for the k-fold cross validation. Also, it is important to notice that for both the machine learning and for the deep learning classifiers, we intrinsically take into consideration the time nature of the data: we are using models which consider the samples statically, without having memory of any time-based dependency between samples. For this reason, we could avoid using a strict time based validation.

The alternative we were left with, was therefore to use a validation strategy that would eliminate as many biases as possible while ensuring to have enough samples of both classes in all the folds of the validation strategy.

As for accuracy metrics, we first calculated precision and recall. However, as suggested by Powers (2011), 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 Powers (2011), the Matthews Correlation Coefficient (MCC) should also be considered to understand the 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 7.

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). This gives us 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. 99%) of the commits were non-fault-inducing, and a plain accuracy 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.

**Statistical Analysis** To assess our results, we also compared the distributions of the software metrics groups and SonarQube rules using statistical tests. We needed to compare

**Table 7** 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 (True Negative Rate) | $\frac{TN}{FP+TNe}$ |
| FPR (False Positive Rate) | $\frac{FP}{TN+FP}$ |
| FNR (False Negative Rate) | $\frac{FN}{FN+TP}$ |

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

**Table 8** SonarQube rules violated in the fault-inducing commits

| SonarQube rules in the fault-inducing commits | | # | Occurrences |
|---|---|---|---|
| Type | Bugs | 26 | 4,491 |
| | Code Smells | 116 | 374,106 |
| | Vulnerability | 7 | 18,998 |
| Severity | Blocker | 6 | 7,959 |
| | Critical | 31 | 28,647 |
| | Major | 81 | 216,655 |
| | Minor | 29 | 125,993 |
| | Info | 2 | 18,341 |

more than 2 groups with not normally distributed data (we tested the normality applying Wilkinson test[7]), and dependent samples (two (or more) samples are called dependent if the members chosen for one sample automatically determine which members are to be included in the second sample). To identify a set of important features and models for fault prediction, we need to verify whether the differences in the performance achieved by the various experimented models were statistically significant. We had two possible options adopting ScottKnott test (Tantithamthavorn et al. 2017, 2018) or Nemenyi test[8] post-hoc test (Nemenyi 1962). The selection depends on the data distribution: if the normality is proved we will opt for ScottKnott, otherwise we will select Nemenyi. Based on the result achieved from the test, we identified the best models and built them using only the most important features and compared with the ones using all the features. For each RQ, we identified which data groups differ after a statistical test of multiple comparisons (null hypothesis is that the groups are similar), making a pair-wise comparison.

### 3.5 Replicability

In order to allow the replication of our study, we published the complete raw data, including all the scripts adopted to perform the analysis and all the results in the replication package (Lomio et al. 2022).

## 4 Results

In this Section, we first report a summary of the data analyzed, and then we answer our RQs.

### 4.1 $RQ_1$. What is the Fault Proneness of the SonarQube Rules?

We considered 59,912 commits in 29 Java projects that violated 174 different rules a total of 1,823,118 times. Out of 174 rules detected in our projects, only 161 are categorized with a SonarQube ID, and these are the ones that we used as input for our analysis, as described in Section 3.4. The 455 commits labelled by SZZ as fault-inducing, violated 149 Sonarqube rules 397,595 times, as reported in Table 8.

[7] https://www.spss-tutorials.com/spss-shapiro-wilk-test-for-normality/
[8] `Nemenyi` package for PHYTON.[9]

**Table 9** The top-10 violated SonarQube rules

| SonarQube rules | Occurrences | Type | Severity |
|---|---|---|---|
| S134 | 23,192 | Code Smells | Major |
| S00112 | 22,185 | Code Smells | Major |
| RTDC | 17,324 | Code Smells | Minor |
| S1166 | 16,164 | Code Smells | Critical |
| S1192 | 15,827 | Code Smells | Minor |
| S1213 | 15,615 | Code Smells | Minor |
| S1133 | 15,236 | Code Smells | Info |
| S106 | 14,196 | Code Smells | Major |
| S1132 | 13,815 | Code Smells | Major |
| MCC | 13,447 | Code Smells | Major |

RTDC means RedundantThrows-DeclarationCheck

MCC means "MethodCyclomaticComplexity"

In the remainder of this Section, we refer to the SonarQube Violations only with their SonarQube ID number (e.g. S108). The complete list of rules, together with their description is reported in the online replication package (Lomio et al. 2022).

It is important to remember that, according to the SonarQube model, a Bug "represents 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.[10] Therefore, we should expect that Bugs represented the vast majority of the rules detected in the fault-inducing commits. However, all the three types present a similar distribution: 19.85% of Bug, 20.09% of Code Smells, and 33.04% of Security Vulnerabilities. In Table 9 we report the occurrences of the top-10 violated SonarQube rules in the fault-inducing commits. Considering the average of each rule per commit, the distribution shows that the top-10 recurrent SonarQube rules are detected in almost all the cases in the fault inducing commits. Only a little portion (less than 3%) is also detected in the not-inducing commits (Fig. 4).

We analyzed our projects with the three selected Machine Learning models (Gradient Boost, Random Forest, and XG Boost) and with two Deep Learning models (FCN and ResNet) to predict a fault based on SonarQube rules.

We considered both the rules individually and grouped by types (Bug, Code Smell, and Vulnerability). We aimed to understand if the presence of a Sonar issue of different types has a higher probability of introducing a fault in the source code.

Figures 5 and 6 depict the box plots reporting the distribution of AUC and F-measure values obtained during the LOGO validation of the three Machine Learning and the two Deep Learning models on the considered dataset. Instead, Figs. 7 and 8 refer to FNR and FPR values. In both figures, each color indicates the model produced considering the rules individually (Blue) and grouped by types (Orange).

Considering the SonarQube rules individually, the three Machine Learning validation results reported an average AUC of 50% (as also shown in Table 10 and Fig. 5). In our previous work (Lenarduzzi et al. 2020e), the AUC obtained an average value of 80%. We believe that avoiding multi-collinearity (O'Brien 2007) (VIP) and adopting a more accurate and realistic validation approach (LOGO) provided a more reliable prediction accuracy.

Deep Learning models, instead, enabled to better predict a fault (Table 14 and Fig. 5). We can see that in terms of AUC both Deep Learning models over-performed the machine

[10] SonarQube Rules: https://tinyurl.com/v7r8rqo

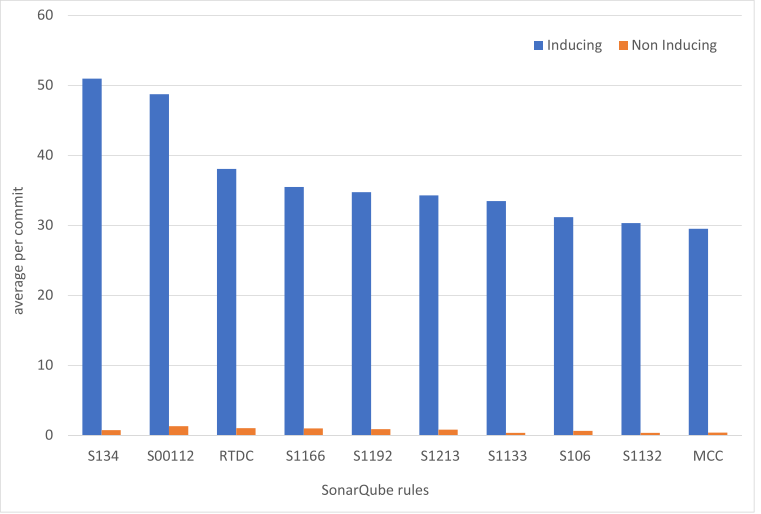


**Fig. 4** Average distribution of inducing and non-inducing commits for the top-10 SonarQube violation types

learning models, with an average AUC of 90%. For the other accuracy metrics, we have good results (better than with the machine learning models).

Moreover, the FNR is higher in the case of the Machine Learning models, as they incorrectly identified normal commits as faulty (Fig. 7). It must be said that even if Deep Learning models look better for FNR, they incorrectly identify faulty classes (FPR - Fig. 8).

Grouping the SonarQube rules by types increases the prediction accuracy (Table 10) in terms of AUC (Fig. 5) and F-measure (Fig. 6) when we applied the Machine Learning models. Instead, Deep Learning models seem to not be affected by the grouping. The same trend can be observed looking at FNR (Fig. 7) and FPR (Fig. 8).

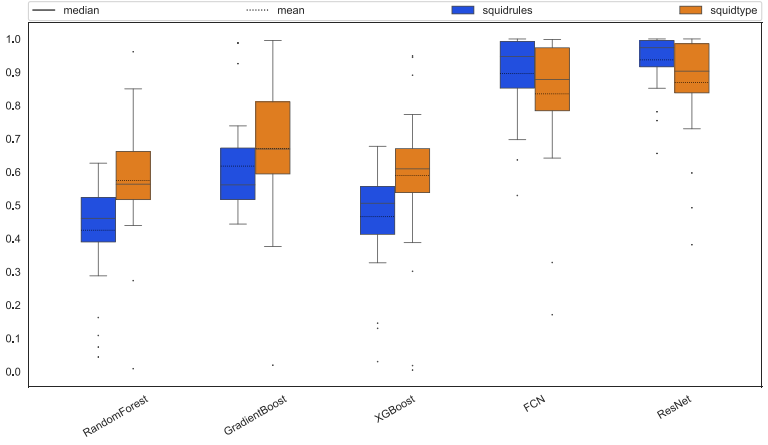


**Fig. 5** AUC comparison among machine learning and deep learning models for SonarQube rules and for SonarQube rules grouped by type ($RQ_1$)

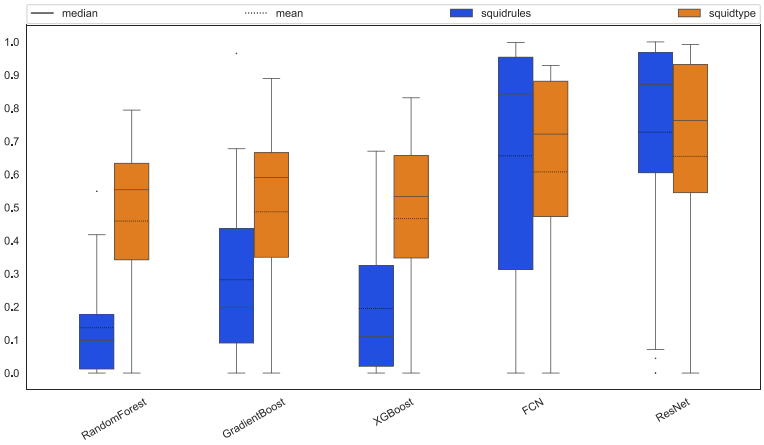


**Fig. 6** F-measure comparison among Machine Learning and Deep Learning models for SonarQube rules and for SonarQube rules grouped by type ($RQ_1$)

These differences in results and performance improvement can be explained with the *curse of dimensionality*. The data we are using can be considered as *high dimensional* data, when considering all the SQ rules individually. This type of data has been shown to limit machine learning models' performance, while affecting less (in this case, for instance) the performance of deep learning models. Machine learning models slightly improve their overall performances when dealing with fewer features instead (i.e. SQ rule types).

Based on the overall results, Deep Learning models are good fault predictors considering all the accuracy metrics.

Moreover, adopting LOGO validation strategy, increases the overall performance of both Deep and Machine Learning models, as we can see in Table 15 and Figs. 23 and 24 reported in the Appendix.

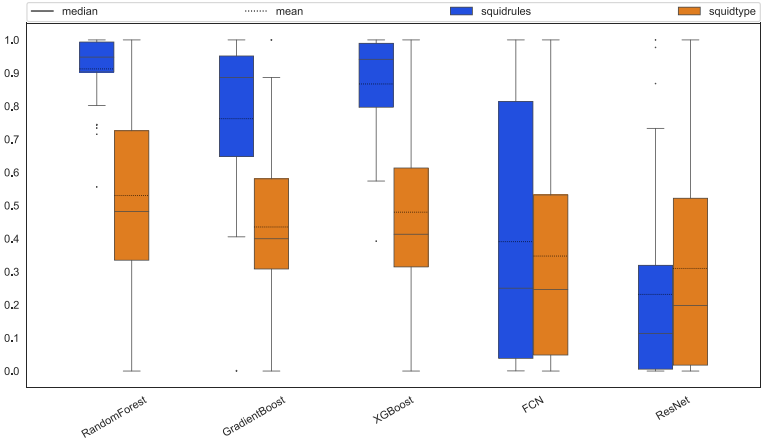


**Fig. 7** FNR comparison among Machine Learning and Deep Learning models for SonarQube rules and for SonarQube rules grouped by type ($RQ_1$)

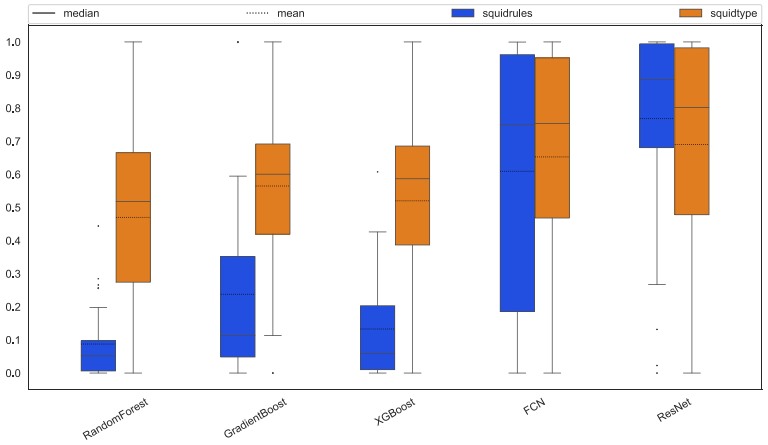


**Fig. 8** TPR comparison among Machine Learning and Deep Learning models for SonarQube rules and for SonarQube rules grouped by type ($RQ_1$)

## 4.2 $RQ_2$. What is the Fault Proneness of Software Metrics?

In this Section, we investigated the fault proneness of product and process metrics considering the ones proposed by Rahman and Devanbu (2013), Kamei et al. (2012), and SonarQube suites (Table 11).

As for $RQ_1$, Figs. 9 and 10 depict the box plots reporting the distribution of AUC and F-measure values obtained during the LOGO validation of the three Machine Learning and the two Deep Learning models on the considered dataset. Instead, Figs. 13 and 14 refer to FNR and TNR values. In both figures, each color indicates the model produced considering different features.

Similarly to $RQ_1$, we used the three selected Machine Learning models (Gradient Boost, Random Forest, and XG Boost) and with the two Deep learning models (FCNN and ResNet)

**Table 10** Accuracy metrics (%) comparison for SonarQube rules with Machine Learning ($RQ_1$)

| SQ rules | Machine learning | | | | | |
|---|---|---|---|---|---|---|
| | Gradient boost | | Random forest | | XG boost | |
| | All | Type | All | Type | All | Type |
| AUC | 61,7 | 67 | 42,5 | 57,4 | 46,5 | 58,9 |
| F-Measure | 28,1 | 48,6 | 13,6 | 45,9 | 19,4 | 46,6 |
| Precision | 61,5 | 53,4 | 52,6 | 50,8 | 61,5 | 50,8 |
| Recall | 23,8 | 56,4 | 8,7 | 47 | 13,3 | 52 |
| MCC | 2 | 22,1 | 7,9 | 17,3 | 14 | 17,9 |
| FNR | 76,1 | 43,5 | 91,2 | 52,9 | 86,6 | 47,9 |
| TNR | 94,3 | 715 | 96,4 | 70,3 | 96,2 | 68,8 |
| FPR | 5,6 | 28,4 | 3,5 | 29,6 | 3,7 | 31,1 |

**Table 11** Accuracy metrics (%) comparison for SonarQube rules with Deep Learning ($RQ_1$)

| SQ rules | Deep learning | | | |
|---|---|---|---|---|
| | FCNN | | ResNet | |
| | All | Type | All | Type |
| AUC | 89,5 | 83,4 | 93,7 | 86,9 |
| F-Measure | 65,5 | 6 | 72,7 | 65,4 |
| Precision | 78 | 67,7 | 83,8 | 72 |
| Recall | 60,9 | 65,2 | 76,8 | 68,9 |
| MCC | 59,7 | 462, | 68,5 | 53,9 |
| FNR | 39 | 34,7 | 23,1 | 31 |
| TNR | 96,2 | 83,9 | 96,5 | 88,7 |
| FPR | 3,7 | 16 | 3,4 | 11,2 |

to predict a fault based on software metrics. Table 12 reports all the accuracy metrics for the machine learning and the deep learning models.

Considering the results obtained with Machine Learning model, *Kamei* (2012) metrics and *Rahman* (2013) metrics work better individually (91% and 90% in average respectively), while SonarQube metrics presents the lowest accuracy (60% in average). Combining together different metrics provide a benefit only for sonarqube metrics (Table 12, Figs. 9 and 10).

On the contrary, Machine Learning models correctly identified the non-faulty classes (TNR - Fig. 14), while for Deep Learning models it depends on which software metrics are used as predictors.

As happened for $RQ_1$, adopting LOGO validation strategy increases the overall performance of both Deep and Machine Learning models (Table 16 and Figs. 25 and 26 reported in the Appendix Section).

To assess whether the accuracy metric distributions were statistically different when considering different metrics combinations, we first determine the normality of the data and since it was not satisfied, we run the post-hoc Nemenyi rank test (Nemenyi 1962) on all the Machine and Deep Learning models. For the sake of space limitations, we only report the results for the more accurate Machine and Deep Learning models for all the considered software metrics: XGBoost and ResNet. We report the statistical results achieved when considering the AUC and F-Measure of he models trained using the Rahman and Devanbu (2013), Kamei et al. (2012), metrics suite (Figs. 11a and 12a), and F-measure (Figs. 11b, 12b). Statistically significant differences are depicted in dark violet. The complete results are reported in our online appendix (Lomio et al. 2022).

Considering XGBoost, AUC values (Fig. 11a) obtained between the models built with SonarQube metrics (SQ) are statistically significant differences and the Rahman and Devanbu (2013), Kamei et al. (2012) metrics. Moreover, there is a statistically significant difference considering the other metrics combined together. The trend is observable for the values of F-measure (Fig. 11b). Looking at ResNet model, AUC (Fig. 12a) statistically significant differences results are observed between Kamei et al. (2012) and SonarQube metrics, while there is no substantial difference between Rahman and Devanbu (2013), Kamei et al. (2012) ones (Table 13).

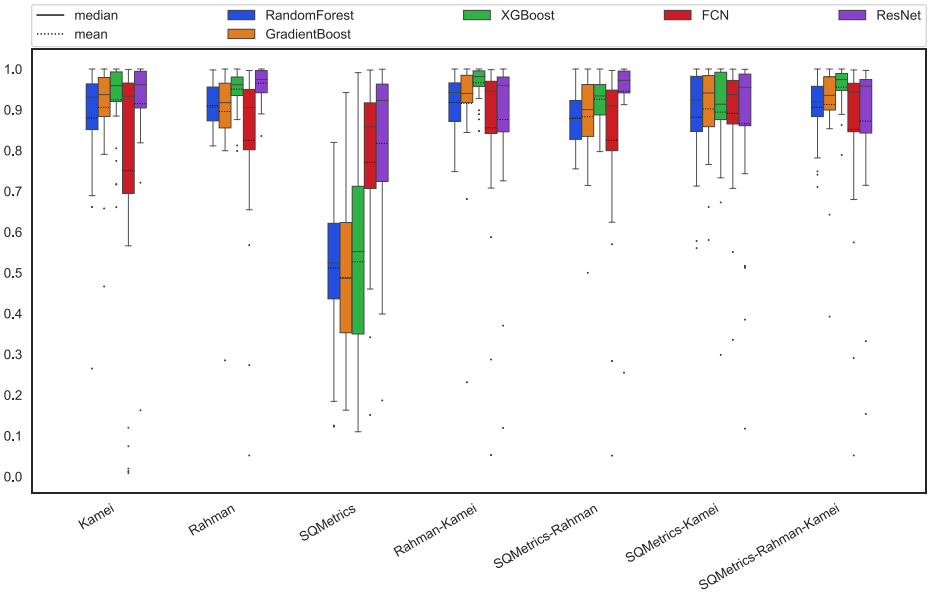


**Fig. 9** AUC comparison among Machine Learning and Deep Learning models for software metrics ($RQ_2$)

***Finding 1***. *Rahman and Devanbu (2013)' and `Kamei et al. (2012) metrics are confirmed as good fault predictors, while SonarQube metrics provided the lowest accuracy. Combining them did not show significant improvement.*

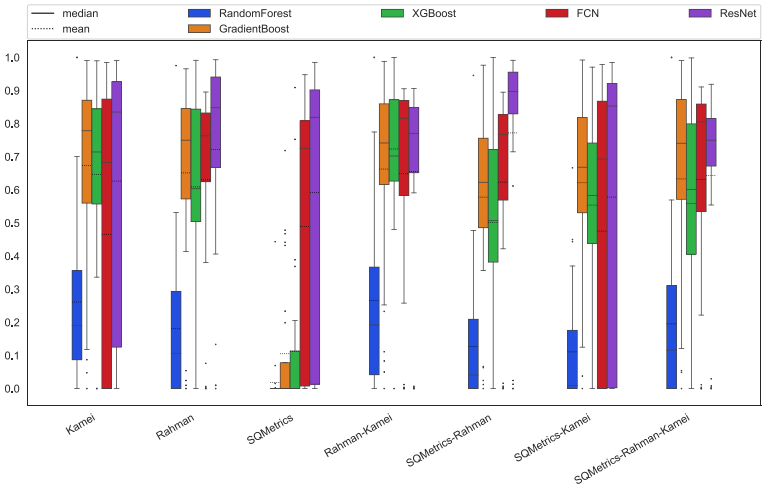


**Fig. 10** F-measure comparison among Machine Learning and Deep Learning models for software metrics ($RQ_2$)

**Table 12** Accuracy metrics (%) comparison for software metrics ($RQ_2$)

| Metrics | Machine learning | | | Deep learning | |
|---|---|---|---|---|---|
| | Gradient boost | Random forest | XG boost | FCNN | ResNet |
| SonarQube (SQ) metrics | | | | | |
| AUC | 48.83 | 51.18 | 52.68 | 77.13 | 81.76 |
| F-measure | 10.53 | 1.83 | 11.31 | 48.97 | 59.24 |
| Precision | 12.46 | 6.66 | 30.20 | 53.09 | 64.36 |
| Recall | 10.29 | 1.26 | 9.33 | 57.45 | 58.70 |
| MCC | -8.95 | -0.03 | 5.00 | 32.88 | 47.32 |
| FPR | 17.22 | 0.89 | 5.73 | 23.94 | 13.64 |
| TNR | 82.78 | 99.11 | 94.27 | 76.06 | 86.36 |
| FNR | 89.71 | 98.74 | 90.67 | 42.55 | 41.30 |
| Kamei et al. (2012) metrics | | | | | |
| AUC | 89.59 | 91.00 | 95.06 | 82.51 | 96.53 |
| F-measure | 65.13 | 18.06 | 60.91 | 63.06 | 72.17 |
| Precision | 76.07 | 63.24 | 84.63 | 58.42 | 78.46 |
| Recall | 69.20 | 12.18 | 56.39 | 78.19 | 80.80 |
| MCC | 54.32 | 18.06 | 55.38 | 40.33 | 65.21 |
| FPR | 8.90 | 0.54 | 1.36 | 39.24 | 10.21 |
| TNR | 91.10 | 99.46 | 98.64 | 60.76 | 89.79 |
| FNR | 30.80 | 87.82 | 43.61 | 21.81 | 19.20 |
| Rahman and Devanbu (2013) metrics | | | | | |
| AUC | 90.59 | 87.99 | 92.45 | 75.07 | 91.49 |
| F-measure | 67.34 | 26.14 | 64.70 | 46.58 | 62.65 |
| Precision | 80.35 | 76.98 | 87.00 | 53.82 | 68.45 |
| Recall | 70.12 | 18.99 | 57.67 | 48.94 | 68.54 |
| MCC | 59.68 | 26.38 | 58.54 | 40.31 | 55.37 |
| FPR | 5.41 | 0.38 | 0.65 | 6.78 | 9.18 |
| TNR | 94.59 | 99.62 | 99.35 | 93.22 | 90.82 |
| FNR | 29.88 | 81.01 | 42.33 | 51.06 | 31.46 |
| Rahman and Devanbu (2013) + Kamei et al. (2012) metrics | | | | | |
| AUC | 82.33 | 76.02 | 80.48 | 70.41 | 71.68 |
| F-measure | 5.48 | 6.68 | 8.82 | 5.44 | 4.29 |
| Precision | 13.60 | 16.67 | 25.29 | 3.13 | 2.44 |
| Recall | 5.81 | 5.32 | 6.75 | 42.21 | 66.52 |
| MCC | 6.81 | 8.29 | 11.16 | 6.70 | 5.17 |
| FPR | 0.13 | 0.01 | 0.05 | 14.97 | 41.38 |
| TNR | 99.87 | 99.99 | 99.95 | 85.03 | 58.62 |
| FNR | 94.19 | 94.68 | 93.25 | 57.79 | 33.48 |
| SonarQube (SQ) + Kamei et al. (2012) metrics | | | | | |
| AUC | 88.30 | 87.96 | 92.60 | 82.53 | 94.54 |
| F-measure | 57.80 | 12.71 | 50.17 | 62.34 | 77.21 |
| Precision | 74.07 | 52.31 | 82.89 | 58.11 | 78.47 |
| Recall | 60.15 | 8.64 | 42.72 | 76.80 | 84.11 |

**Table 12** (continued)

| Metrics | Machine learning | | | Deep learning | |
|---|---|---|---|---|---|
| | Gradient boost | Random forest | XG boost | FCNN | ResNet |
| MCC | 46.55 | 13.35 | 45.04 | 39.87 | 70.19 |
| FPR | 8.40 | 0.24 | 1.62 | 37.95 | 9.42 |
| TNR | 91.60 | 99.76 | 98.38 | 62.05 | 90.58 |
| FNR | 39.85 | 91.36 | 57.28 | 23.20 | 15.89 |
| SonarQube (SQ) + Rahman and Devanbu (2013) metrics | | | | | |
| AUC | 90.22 | 88.20 | 89.43 | 89.12 | 86.59 |
| F-measure | 62.17 | 11.08 | 55.41 | 47.55 | 57.83 |
| Precision | 78.90 | 47.90 | 83.29 | 57.86 | 57.96 |
| Recall | 63.10 | 8.72 | 48.23 | 49.80 | 65.87 |
| MCC | 52.89 | 12.03 | 50.06 | 41.05 | 49.54 |
| FPR | 6.31 | 0.21 | 0.74 | 7.17 | 12.15 |
| TNR | 93.69 | 99.79 | 99.26 | 92.83 | 87.85 |
| FNR | 36.90 | 91.28 | 51.77 | 50.20 | 34.13 |
| SonarQube (SQ) + Rahman and Devanbu (2013) + Kamei et al. (2012) metrics | | | | | |
| AUC | 91.31 | 90.61 | 95.55 | 85.20 | 87.23 |
| F-measure | 63.32 | 19.54 | 55.86 | 63.12 | 64.33 |
| Precision | 79.00 | 61.10 | 83.47 | 63.12 | 58.45 |
| Recall | 66.04 | 14.67 | 46.77 | 73.89 | 84.25 |
| MCC | 55.68 | 20.35 | 50.79 | 46.79 | 43.51 |
| FPR | 5.57 | 0.19 | 0.69 | 25.41 | 38.48 |
| TNR | 94.43 | 99.81 | 99.31 | 74.59 | 61.52 |
| FNR | 33.96 | 85.33 | 53.23 | 26.11 | 15.75 |

***Finding 2.*** *Using historical data (Deep Learning) did not significantly improve the accuracy reached with a single snapshot (Machine Learning), except for SonarQube Metrics. Both Deep and Machine Learning models suffer from a higher True Negative Rate (TNR), while only Deep learning models detect more False Positive (FPR).*

### 4.3 $RQ_3$.To What Extent Can SonarQube Rules Impact the Performance of Fault Prediction Models that Leverage Process and Product Metrics

In this Section, we considered in the metrics combination used in $RQ_2$, including also the SonarQube rules. Table 10 depicts the accuracy metrics results for the SonarQube individually and with the Sonarqube rules types using the Machine learning, while Table 14 presents the results adopting Deep Learning models (Figs. 13 and 14). Figures 15 and 16 depict the box plots reporting the distribution of AUC and F-measure values obtained during the LOGO validation of the three Machine Learning and the two Deep Learning models on the considered dataset considering the SonarQube individually. Instead, Figs. 17 and 18

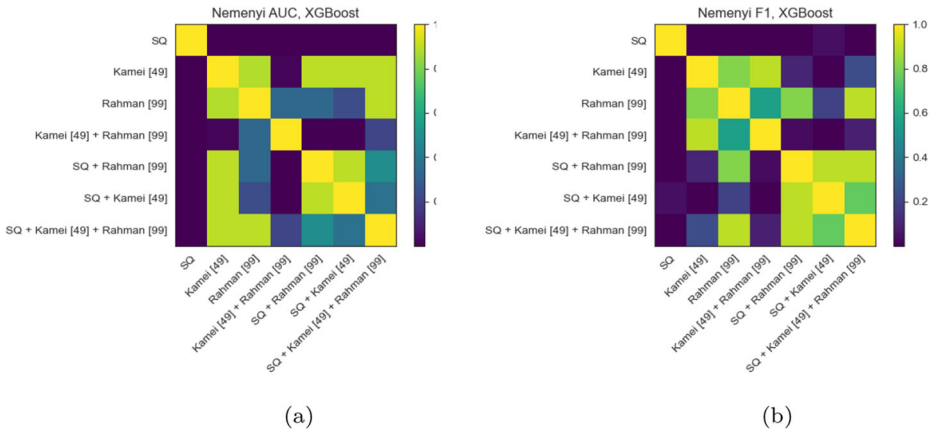


**Fig. 11** Nemenyi test for comparing the different product and process metrics group within XGBoost ($RQ_2$)

refer the Sonarqube rules grouped by types. In both figures, each color indicates the model produced considering different models.

As for the other RQs, to assess whether the accuracy metric distributions were statistically different when considering in the first case SonarQube rules and in the second case the rule types, we run the post-hoc Nemenyi rank test (Nemenyi 1962). We considered all the metric combinations and all the models (Figs. 27a, 28, 29, 30, 31, 32, and 33b in Appendix).

**SonarQube Rules** Evaluating the effect obtained including SonarQube rules with each metric combination, the observed change in terms of AUC and F-measure is not substantial (Table 10) adopting Machine Learning models. Instead, and unsuspected, the change is negative in all the combinations except for the pair *SQ rules* + *SQ metrics* with *Gradient Boost* as model and for *SQ rules* + Rahman and Devanbu (2013) + Kamei et al. (2012) *metrics* with *Random Forest* as model, where the change is significant. Instead, the results obtained with Deep Learning models turned out the best in terms of AUC. All the combinations significantly benefit from the inclusion of SonarQube rules. Considering the other accuracy

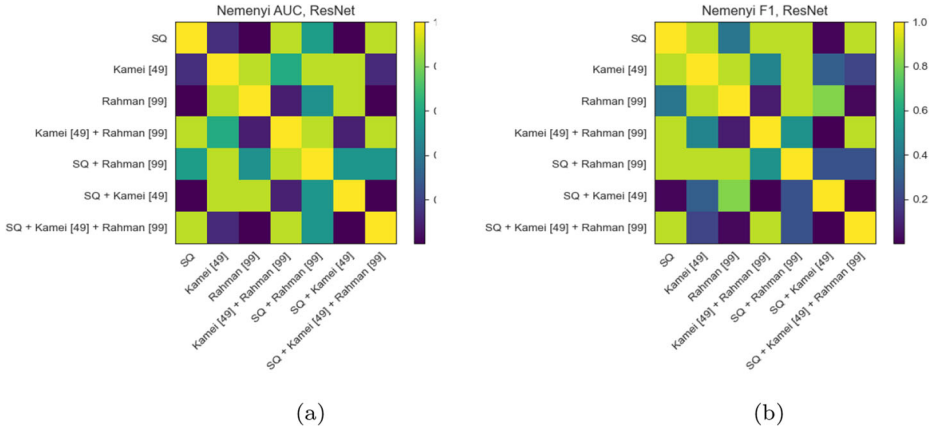


**Fig. 12** Nemenyi test for comparing the different product and process metrics within ResNet ($RQ_2$)

**Table 13** Accuracy metrics (%) comparison for SonarQube rules with Machine Learning ($RQ_3$)

| | Machine learning | | | | | |
|---|---|---|---|---|---|---|
| | Gradient boost | | Random forest | | XG boost | |
| | All | Type | All | Type | All | Type |
| SQ Rules + SQ metrics | | | | | | |
| AUC | 62.30 | 72.14 | 55.85 | 62.61 | 49.16 ↓ | 61.48 |
| F-Measure | 27.08 | 42.68 | 0.60 ↓ | 1.51 ↓ | 10.13 ↓ | 15.33 |
| Precision | 52.19 | 53.62 | 9.18 | 21.54 | 44.96 | 43.36 |
| Recall | 21.39 | 48.50 | 0.32 ↓ | 0.81 ↓ | 6.35 ↓ | 10.26 |
| MCC | 17.84 ↓ | 24.01 ↓ | -0.36 | 0.84 ↓ | 3.87 ↓ | 6.64 |
| FNR | 78.61 | 51.51 | 99.68 | 99.19 | 93.65 | 89.74 |
| TNR | 93.77 | 81.61 ↓ | 99.68 | 99.64 | 96.06 | 94.86 |
| FPR | 6.23 ↓ | 18.39 ↓ | 0.32 ↓ | 0.36 ↓ | 3.94 ↓ | 5.14 ↓ |
| SQ Rules + Kamei et al. (2012) metrics | | | | | | |
| AUC | 83.76 ↓ | 86.01 ↓ | 82.15 ↓ | 82.63 ↓ | 75.48 ↓ | 80.20 ↓ |
| F-Measure | 60.54 ↓ | 60.48 ↓ | 12.57 ↓ | 22.80 | 34.79 ↓ | 39.05 ↓ |
| Precision | 69.61 ↓ | 71.15 ↓ | 55.55 ↓ | 62.01 ↓ | 68.96 ↓ | 71.69 ↓ |
| Recall | 63.15 ↓ | 65.50 ↓ | 8.81 ↓ | 19.88 | 25.25 ↓ | 33.23 ↓ |
| MCC | 45.67 ↓ | 46.80 ↓ | 12.51 ↓ | 22.10 | 26.04 ↓ | 32.06 ↓ |
| FNR | 36.85 | 34.50 | 91.19 | 80.12 | 74.75 | 66.77 |
| TNR | 85.09 ↓ | 86.75 ↓ | 99.54 | 99.35 ↓ | 95.34 ↓ | 96.11 ↓ |
| FPR | 14.91 ↓ | 13.25 ↓ | 0.46 ↓ | 0.65 ↓ | 4.66 ↓ | 3.89 ↓ |
| SQ Rules + Rahman and Devanbu (2013) metrics | | | | | | |
| AUC | 83.41 ↓ | 84.22 ↓ | 82.70 ↓ | 82.28 ↓ | 80.04 ↓ | 80.31 ↓ |
| F-Measure | 60.12 ↓ | 60.24 ↓ | 13.22 ↓ | 23.93 ↓ | 35.51 ↓ | 37.10 ↓ |
| Precision | 70.38 ↓ | 69.90 ↓ | 60.24 ↓ | 67.67 ↓ | 67.42 ↓ | 70.02 ↓ |
| Recall | 65.48 ↓ | 66.12 ↓ | 7.87 ↓ | 19.03 | 32.58 ↓ | 30.44 ↓ |
| MCC | 46.33 ↓ | 46.32 ↓ | 13.49 ↓ | 22.90 ↓ | 27.89 ↓ | 29.75 ↓ |
| FNR | 34.52 | 33.88 | 92.13 | 80.97 | 67.42 | 69.56 |
| TNR | 86.02 ↓ | 85.84 ↓ | 99.53 ↓ | 99.27 ↓ | 95.96 ↓ | 96.29 ↓ |
| FPR | 13.98 ↓ | 14.16 ↓ | 0.48 ↓ | 0.74 ↓ | 4.04 ↓ | 3.71 ↓ |
| SQ Rules + Rahman and Devanbu (2013)+ Kamei et al. (2012) metrics | | | | | | |
| AUC | 84.30 | 84.33 | 84.47 | 83.05 | 81.19 | 80.76 |
| F-Measure | 56.31 | 57.15 | 13.06 | 22.92 | 32.59 | 34.37 |
| Precision | 68.93 | 69.26 | 58.46 | 63.35 | 74.15 | 73.14 |
| Recall | 60.07 | 61.82 | 8.74 | 17.11 | 28.52 | 27.09 |
| MCC | 41.95 | 44.09 | 13.47 | 21.78 | 28.07 | 30.00 |
| FNR | 39.94 | 38.18 | 91.26 | 82.89 | 71.48 | 72.91 |
| TNR | 87.44 ↓ | 88.04 ↓ | 99.74 ↓ | 99.54 ↓ | 97.65 ↓ | 97.95 ↓ |
| FPR | 12.56 ↓ | 11.96 ↓ | 0.26 ↓ | 0.46 ↓ | 2.35 ↓ | 2.05 ↓ |
| SQ Rules + SQ+ Kamei et al. (2012) metrics | | | | | | |
| AUC | 83.08 ↓ | 85.51 ↓ | 83.19 ↓ | 83.72 ↓ | 77.80 ↓ | 81.23 ↓ |
| F-Measure | 59.61 | 64.39 | 2.56 ↓ | 13.51 | 22.94 ↓ | 35.13 ↓ |
| Precision | 71.78 ↓ | 74.90 | 32.42 ↓ | 63.16 | 65.95 ↓ | 81.09 ↓ |

**Table 13** (continued)

| | Machine learning | | | | | |
|---|---|---|---|---|---|---|
| | Gradient boost | | Random forest | | XG boost | |
| | All | Type | All | Type | All | Type |
| Recall | 64.23 | 69.94 | 1.49 ↓ | 9.62 | 15.08 ↓ | 28.16 ↓ |
| MCC | 46.83 | 53.86 | 3.16 ↓ | 15.19 | 19.18 ↓ | 32.05 ↓ |
| FNR | 35.77 | 30.06 | 98.52 | 90.38 | 84.92 | 71.84 |
| TNR | 87.48 ↓ | 89.36 ↓ | 99.95 | 99.79 | 98.08 ↓ | 98.34 ↓ |
| FPR | 12.52 ↓ | 10.64 ↓ | 0.05 ↓ | 0.21 ↓ | 1.92 ↓ | 1.66 ↓ |
| SQ Rules + SQ+ Rahman and Devanbu (2013) metrics | | | | | | |
| AUC | 82.51 ↓ | 85.58 ↓ | 83.50 ↓ | 83.35 ↓ | 79.10 ↓ | 83.79 ↓ |
| F-Measure | 57.80 ↓ | 60.46 ↓ | 6.71 ↓ | 17.06 | 27.87 ↓ | 28.46 ↓ |
| Precision | 71.84 ↓ | 71.44 ↓ | 37.09 ↓ | 59.62 | 69.98 ↓ | 73.21 ↓ |
| Recall | 61.38 ↓ | 66.83 | 4.34 ↓ | 14.37 | 23.94 ↓ | 24.27 ↓ |
| MCC | 44.65 ↓ | 49.27 ↓ | 7.49 ↓ | 18.01 | 24.86 ↓ | 26.44 ↓ |
| FNR | 38.62 | 33.17 | 95.66 | 85.64 | 76.06 | 75.73 |
| TNR | 87.84 ↓ | 88.08 ↓ | 99.90 | 99.74 ↓ | 98.18 ↓ | 98.71 ↓ |
| FPR | 12.16 ↓ | 11.92 ↓ | 0.10 ↓ | 0.26 ↓ | 1.82 ↓ | 1.29 ↓ |
| SQ Rules + SQ+Rahman and Devanbu (2013)+Kamei et al. (2012) metrics | | | | | | |
| AUC | 83.92 ↓ | 85.22 ↓ | 84.36 ↓ | 84.63 ↓ | 80.18 ↓ | 83.06 ↓ |
| F-Measure | 57.63 ↓ | 60.51 ↓ | 5.50 ↓ | 14.78 ↓ | 25.33 ↓ | 28.86 ↓ |
| Precision | 71.57 ↓ | 73.11 ↓ | 37.60 ↓ | 65.24 | 69.58 ↓ | 77.28 ↓ |
| Recall | 61.03 ↓ | 65.67 ↓ | 3.67 ↓ | 10.44 ↓ | 21.41 ↓ | 20.51 ↓ |
| MCC | 44.59 ↓ | 49.65 ↓ | 6.03 ↓ | 16.20 ↓ | 23.22 ↓ | 26.71 ↓ |
| FNR | 38.97 | 34.33 | 96.33 | 89.56 | 78.59 | 79.49 |
| TNR | 88.32 ↓ | 89.12 ↓ | 99.89 | 99.79 ↓ | 98.65 ↓ | 98.82 ↓ |
| FPR | 11.68 ↓ | 10.88 ↓ | 0.11 ↓ | 0.21 ↓ | 1.35 ↓ | 1.18 ↓ |

The red arrows (↓) indicate the values that decreased compared to the results of the analysis shown in Table 12. All other values increased

metrics, we can observe the same trend as for AUC and F-measure. FNR rate is consistently below 20%, TNR up to 97%, and FRP below 3%. These results confirmed the better accuracy of Deep Learning compared with Machine Learning models. Deep Learning models are able to correctly identity a faulty commit, with a low probability of incorrect identification.

**SonarQube rule types** The scenario is thoroughly different including SonarQube rule types, since we obtained different results from the ones seen with Machine Learning models. For all the combination of SonarQube rules and metrics, we observed a significant discrepancy of results for AUC and F-measure in both models. *SQ metrics* and Rahman and Devanbu (2013) + Kamei et al. (2012) *metrics* benefit from the inclusion of the *SonarQube rules*, while Kamei et al. (2012) *metrics* and Rahman and Devanbu (2013) are not affected. The other combinations see a decreased in the AUC. Instead, the effect observed with Deep Learning model is negligible. Considering the other accuracy metrics, we can observe the same trend as for the results obtained with the individual rules.

**Table 14** Accuracy metrics (%) comparison for SonarQube rules with Deep Learning ($RQ_3$)

| SQ rules | Deep learning | | | |
|---|---|---|---|---|
| | FCNN | | ResNet | |
| | All | Type | All | Type |
| SQ Rules+SQ metrics | | | | |
| AUC | 91.67 | 93.69 | 99.10 | 98.12 |
| F-Measure | 83.14 | 80.07 | 91.13 | 90.20 |
| Precision | 84.32 | 83.31 | 91.87 | 91.74 |
| Recall | 82.22 | 78.59 | 90.64 | 89.25 |
| MCC | 80.53 | 76.39 | 89.65 | 88.29 |
| FNR | 17.78 ↓ | 21.41 ↓ | 9.36 ↓ | 10.75 ↓ |
| TNR | 98.13 | 97.11 | 98.85 | 98.71 |
| FPR | 1.87 ↓ | 2.89 ↓ | 1.15 ↓ | 1.29 ↓ |
| SQ Rules+Kamei et al. (2012) metrics | | | | |
| AUC | 95.66 | 96.81 | 99.36 | 99.54 |
| F-Measure | 82.70 | 90.27 | 92.22 | 94.88 |
| Precision | 84.85 | 91.50 | 93.57 | 95.59 |
| Recall | 84.05 | 92.09 | 95.80 | 97.14 |
| MCC | 80.46 | 87.89 | 91.38 | 93.55 |
| FNR | 15.95 ↓ | 7.91 ↓ | 4.20 ↓ | 2.86 ↓ |
| TNR | 98.34 | 98.07 | 99.20 | 98.79 |
| FPR | 1.66 ↓ | 1.93 ↓ | 0.81 ↓ | 1.21 ↓ |
| SQ Rules+Rahman and Devanbu (2013) metrics | | | | |
| AUC | 98.73 | 96.14 | 99.47 | 99.59 |
| F-Measure | 86.52 | 83.45 | 92.37 | 93.23 |
| Precision | 88.65 | 85.75 | 93.60 | 94.24 |
| Recall | 85.04 | 83.73 | 96.48 | 96.49 |
| MCC | 84.42 | 81.14 | 91.65 | 92.10 |
| FNR | 14.96 | 16.27 | 3.53 ↓ | 3.52 ↓ |
| TNR | 98.96 | 98.57 | 99.23 | 99.32 |
| FPR | 1.04 ↓ | 1.43 ↓ | 0.78 ↓ | 0.68 ↓ |
| SQ Rules + Rahman and Devanbu (2013)+Kamei et al. (2012) metrics | | | | |
| AUC | 98.88 | 98.07 | 99.63 | 99.59 |
| F-Measure | 87.14 | 84.91 | 94.89 | 91.31 |
| Precision | 88.08 | 86.60 | 96.09 | 92.81 |
| Recall | 88.62 | 84.62 | 97.21 | 92.90 |
| MCC | 85.57 | 82.69 | 94.08 | 90.30 |
| FNR | 11.38 ↓ | 15.38 | 2.79 ↓ | 7.10 ↓ |
| TNR | 99.04 | 98.70 | 99.44 | 99.45 |
| FPR | 0.96 ↓ | 1.30 ↓ | 0.56 ↓ | 0.55 ↓ |
| SQ Rules+SQ metrics+Kamei et al. (2012) metrics | | | | |
| AUC | 96.01 | 95.38 | 99.58 | 99.74 |
| F-Measure | 87.63 | 91.49 | 94.65 | 91.87 |
| Precision | 88.48 | 91.90 | 96.09 | 92.60 |

**Table 14** (continued)

| SQ rules | Deep learning | | | |
|---|---|---|---|---|
| | FCNN | | ResNet | |
| | All | Type | All | Type |
| Recall | 87.06 | 91.15 | 94.02 | 91.37 |
| MCC | 86.04 | 89.96 | 93.93 | 91.01 |
| FNR | 12.94 ↓ | 8.85 ↓ | 5.98 ↓ | 8.63 ↓ |
| TNR | 98.80 | 98.77 | 99.57 | 99.50 |
| FPR | 1.20 ↓ | 1.23 ↓ | 0.43 ↓ | 0.50 ↓ |
| SQ Rules+SQ metrics+Rahman and Devanbu (2013) metrics | | | | |
| AUC | 98.73 | 96.66 | 99.75 | 99.83 |
| F-Measure | 91.46 | 88.14 | 98.24 | 98.85 |
| Precision | 92.21 | 88.57 | 99.61 | 99.64 |
| Recall | 90.88 | 87.76 | 97.49 | 98.24 |
| MCC | 90.06 | 86.68 | 97.40 | 98.02 |
| FNR | 9.12 | 12.24 | 2.51 ↓ | 1.76 ↓ |
| TNR | 99.08 | 98.87 | 99.63 | 99.66 |
| FPR | 0.92 ↓ | 1.13 ↓ | 0.37 ↓ | 0.34 ↓ |
| SQ Rules+SQ metrics+Rahman and Devanbu (2013)+Kamei et al. (2012) metrics | | | | |
| AUC | 94.75 | 96.00 | 99.77 | 99.80 |
| F-Measure | 91.59 | 91.44 | 98.94 | 95.32 |
| Precision | 92.12 | 91.99 | 99.55 | 95.97 |
| Recall | 91.16 | 90.99 | 98.50 | 94.86 |
| MCC | 90.22 | 89.92 | 98.17 | 94.42 |
| FNR | 8.84 ↓ | 9.01 ↓ | 1.50 ↓ | 5.14 ↓ |
| TNR | 98.99 | 98.86 | 99.56 | 99.44 |
| FPR | 1.01 ↓ | 1.14 ↓ | 0.44 ↓ | 0.56 ↓ |

The red arrows (↓) indicate the values that decreased compared to the results of the analysis shown in Table 12. All other values increased

***Finding 3***. *Including SonarQube rules and adopting historical data (Deep Learning) largely increase (more than 90%) the accuracy of all the metrics combinations. SonarQube rules do not provide substantial changes using Machine Learning models.*

***Finding 4***. *Considering the rule types provides a negative effect using Machine Learning models, while the effect is negligible using historical data (Deep Learning).*

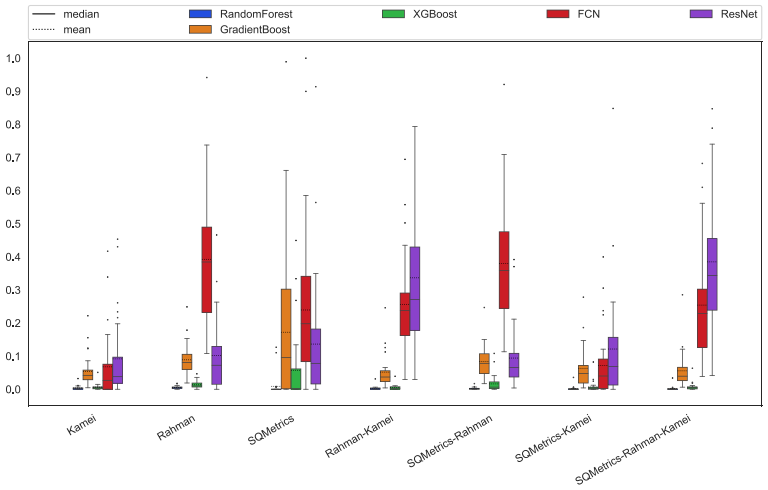


**Fig. 13** FPR comparison among Machine Learning and Deep Learning models for software metrics ($RQ_2$)

### 4.4 $RQ_4$.Which is the Best Combination of Metrics and the Best Model for the Fault Prediction?

As for the previous RQ, to assess whether the performance distributions of the different software metrics and SonarQube rules were statistically different when considering different combinations of Machine Learning and Deep Learning models, we run the post hoc Nemenyi rank test (1962). For the sake of space limitations, we only report the results for the more accurate combinations features (SonarQube rules, product and process metrics) and for more accurate models for all the considered features. For consistency, we show the p-values of the Nemenyi rank test computed on the distribution of AUC and F-measure values by the means of heatmaps (Figs. 19a, b, 21a and b) where statistically significant differences

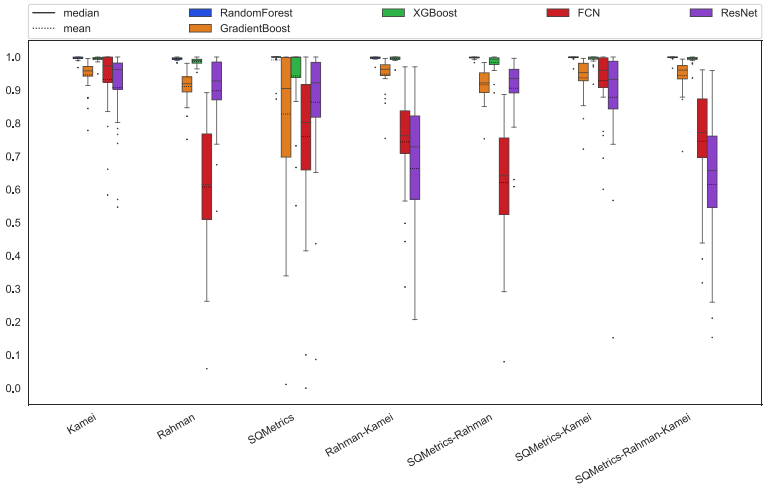


**Fig. 14** TNR comparison among Machine Learning and Deep Learning models for software metrics ($RQ_2$)

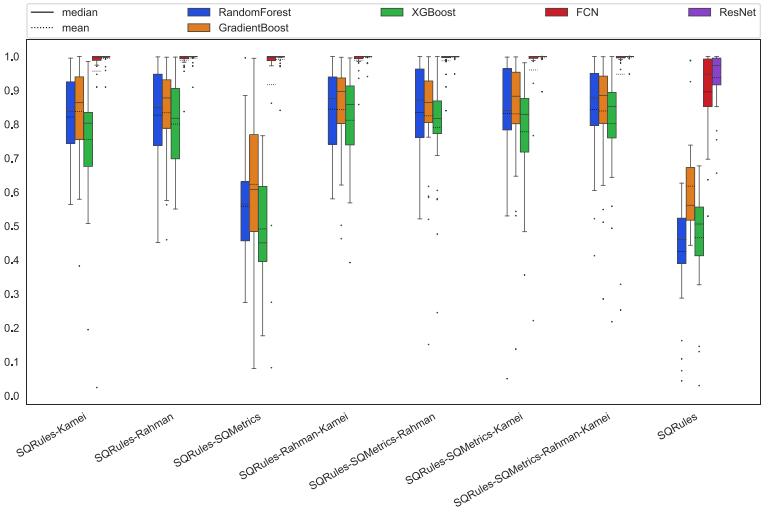


**Fig. 15** AUC comparison among Machine Learning and Deep Learning models for SQ rules compared to software metrics ($RQ_3$)

are depicted in dark violet. The complete results are reported in our online appendix (Lomio et al. 2022).

Looking at the results obtained in the previous RQs, and considering the values of the accuracy metrics obtained, we identified the Deep Learning models as more accurate than the Machine learning ones. Notably, the *ResNet* was shown to outperform all the other models, including the FCN.

The two feature sets in which the ResNet achieves the best results are:

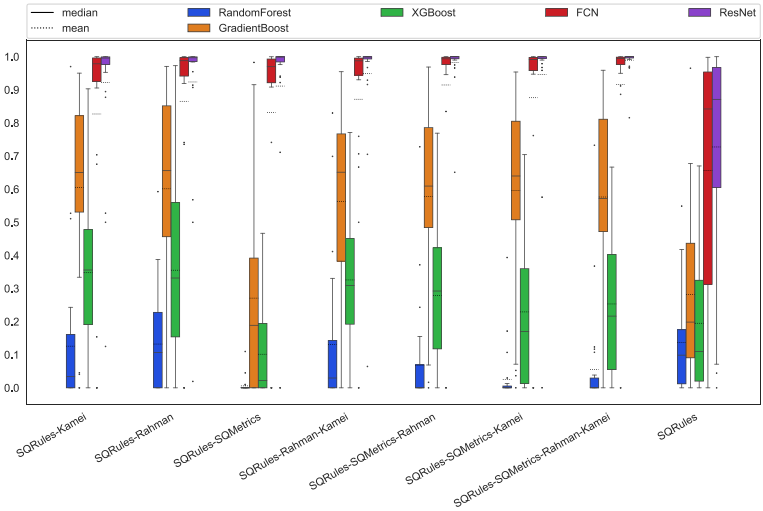


**Fig. 16** F-measure comparison among Machine Learning and Deep Learning models for SQ rules compared to software metrics ($RQ_3$)

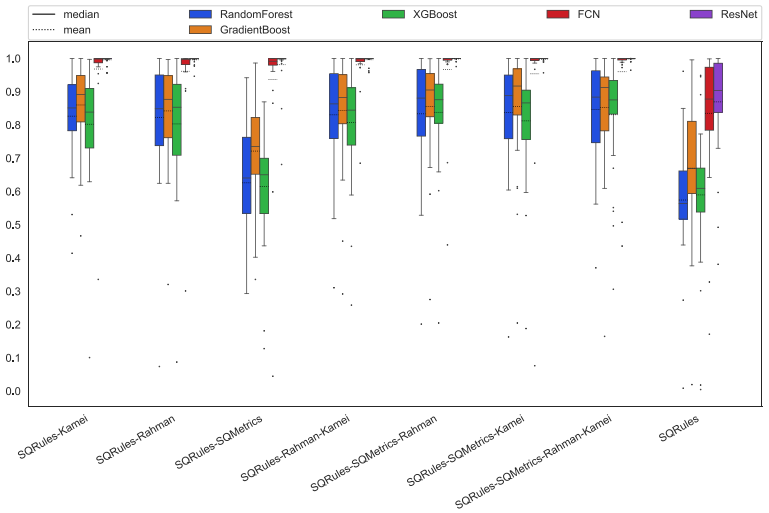


**Fig. 17** AUC comparison among Machine Learning and Deep Learning models for SQ rules type compared to software metrics ($RQ_3$)

– SonarQube rule types + SonarQube + Rahman and Devanbu (2013) metrics (Fig. 19a and b)
– SonarQube rule types + SonarQube + Rahman and Devanbu (2013) + Kamei et al. (2013) metrics (Fig. 20a and b)

Figures 19a and 20a show statistically significant differences (depicted in dark violet) in AUC values between Machine Learning and Deep Learning models. These results confirm the large positive effect that Deep Learning models provide to the two identified feature sets. On a similar note, Figs. 19b and 20b show, in terms of F-measure, the presence of

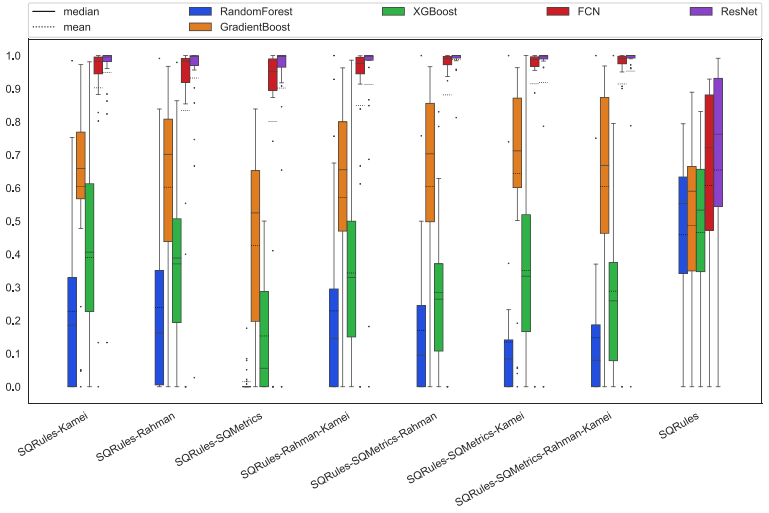


**Fig. 18** F-measure comparison among Machine Learning and Deep Learning models for SQ rules type compared to software metrics ($RQ_3$)

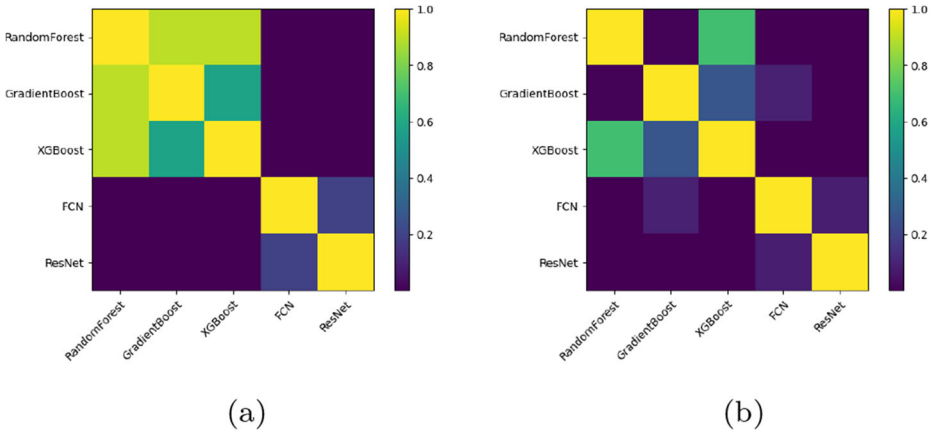


**Fig. 19** Nemenyi test $p$-values obtained for comparing the models trained on SQ rule types, SQ metrics and Rahman and Devanbu (2013) using the different models ($RQ_4$)

statistically significant differences in the same feature sets as for AUC. This further supports the contribution provided by the Deep Learning models. It can be further seen in Fig. 21a and b and Fig. 22a and b, that the *SonarQube rule types* + *SonarQube* + Rahman and Devanbu (2013) *metrics* and *SonarQube rule types* + *SonarQube* + Rahman and Devanbu (2013) + Kamei et al. (2013) *metrics* yield significantly better results when used as feature set for the *ResNet* model.

> ***Finding 5***. *Significance tests confirm the findings discovered during the qualitative analysis of the distributions by the means of box plots: SonarQube rule types + SonarQube + Rahman and Devanbu (2013) metrics and SonarQube rule types + SonarQube + Rahman and Devanbu (2013) + Kamei et al. (2012) metrics as feature set and ResNet as learning model.*

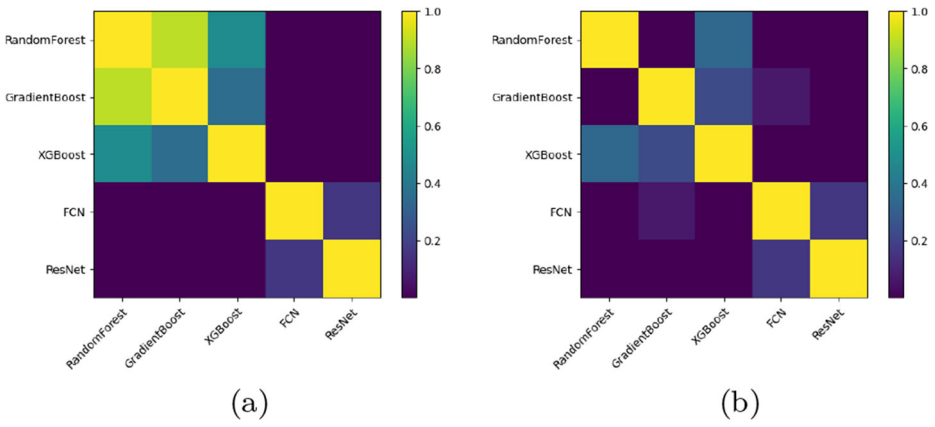


**Fig. 20** Nemenyi test $p$-values obtained for comparing the models trained on SQ rule types, SQ metrics, Rahman and Devanbu (2013) and Kamei et al. (2013) using the different models ($RQ_4$)

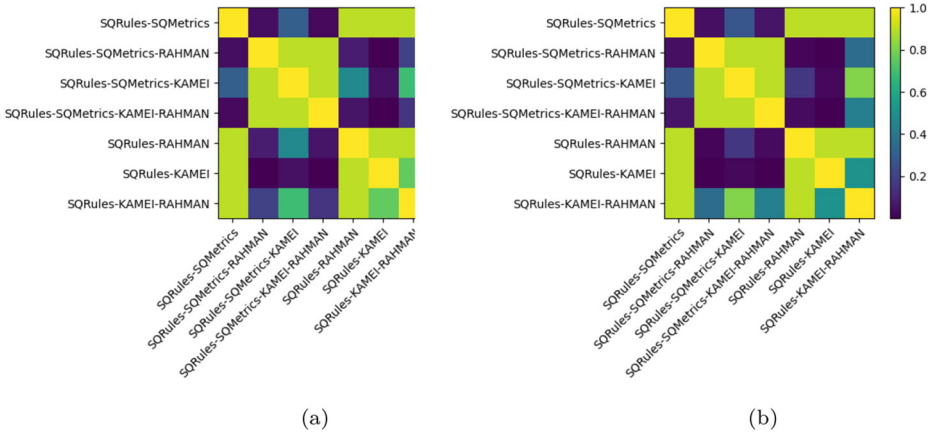


**Fig. 21** Nemenyi test $p$-values obtained for comparing the ResNet trained on the different feature combinations using the SQ rule types ($RQ_4$)

## 5 Discussion

In this Section, we discuss the results obtained according to the RQs. The results achieved revealed a number of insights that may lead to concrete implications for the software engineering research community.

**Only SonarQube Rules are Not Enough** One of the main outcomes of our study revealed the ability of SonarQube rules alone to predict faults only under certain conditions. In order to achieve the best performance, the analysis should be run considering Deep Learning models as classifiers. Unfortunately, Machine Learning models led to poorly accurate results and did not provide comparable values. The obtained values are lower, making the prediction similar to a "random guess". Adopting historical data instead of a single snapshot (as for Machine Learning models) can be better when the commit data is time-dependent. Even if

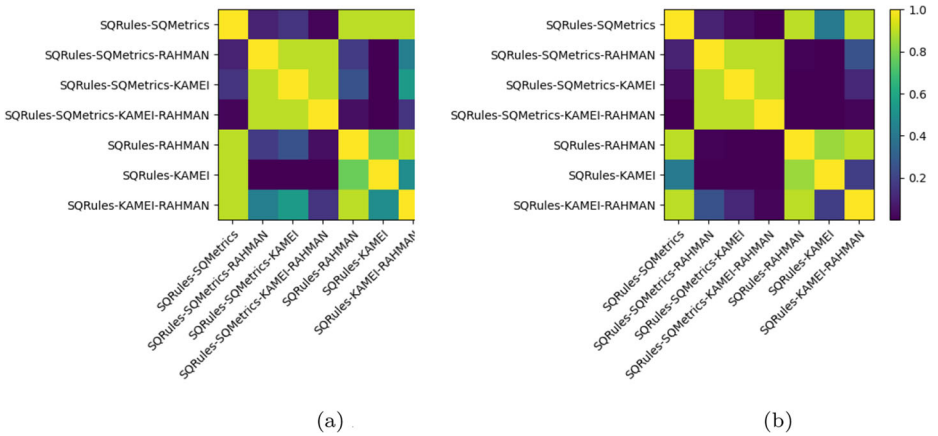


**Fig. 22** Nemenyi test $p$-values obtained for comparing the ResNet trained on the different feature combinations using the SQ rule ($RQ_4$)

these results with Machine Learning models are contrasting with the previous ones (Lenarduzzi et al. 2019b), they are more reliable and realistic because of the new preprocessing approach and the more accurate validation strategy.

However, in the latter case, when we considered the SonarQube rule types as predictors, we observed unsuspected results. We observed that Machine Learning models have benefited from the grouping, while Deep Learning models seem unaffected by it. We should notice that the benefit achieved with Machine Learning models is small but significant.

In the light of the facts, our suggestion is to equally include the rule types as predictors mainly because it is more simple to monitor the analysis since the number of variables is less than considering all the rules without grouping.

Our results, therefore, represent a call for further investigation regarding the role of static analysis tools for faults prediction. The different static analysis tools can classify and group similar rules differently or provide different classifications. It should be interesting to evaluate if the same trend observed with SonarQube could also be recoverable with other static analysis tools, such as Findbugs or Checkstyle. In particular, the focus should be reserved to the case where the rule types are considered as predictors to confirm or deny the results obtained with sonarQube. It should be important to determine if the negligible effect of the rules types achieved from Deep Learning is intrinsic of the adopted tool or can be generalizable. In particular, it is important to determine if the trend is attributable to the classifier and not to the static analysis tool.

**Product and Process Metrics. Which Ones?** The performances reached adopting process and product metrics as fault predictors are higher in terms of AUC and F-measure. This is particularly evident when considering Rahman and Devanbu (2013) and Kamei et al. (2012) metrics individually, confirming the previous study results (Kamei et al. 2012; Pascarella et al. 2019). However, when these two metrics sets are combined the performance decreases. This phenomenon deserves further and deeper investigation. Considering the third metric set provided by SonarQube, the performances are inferior; however, combined with Rahman and Devanbu (2013) or with Kamei et al. (2012) set the prediction accuracy increases, especially when combined with Kamei et al. (2012) metrics. Considering all the three metric sets together does not provide an evident improvement.

**SonarQube Rules, Product and Process Metrics. All Together?** Even if we achieved a higher accuracy when considering Rahman and Devanbu (2013) and Kamei et al. (2012) metrics, including SonarQube rules still improves the prediction. The accuracy metrics reached stunning values (more than 95%), better than expected. These results deserve further focus and a deep investigation in order to determine if it is an isolated case attributable only to SonarQube or can be generalizable to other static analysis tools. As for the previous case, when we considered only the SonarQube rules as features, we suggest to deeply investigate the role of the other static analysis tools in combination with the different software metrics.

**Machine Learning or Deep Learning?** We observed that the classifiers' choice between single snapshot (Machine Learning) and historical data (Deep Learning) and inside the single classifier categories has a significant impact on the resulting capabilities.

Considering the three Machine Learning models, we notice that, as expected, boosting methods performed better the faults detection accuracy, compared with traditional ensemble models such as Random Forest. We believe that the reason behind this is due to the boosting models' characteristics. Such characteristics allow to iteratively train a weak classifier on

subsequent training data, assigning a weight to each instance of the training set, and modifying it at each iteration, increasing the weight for the misclassified samples. Consequently, the boosting methods are focused more on misclassified samples, which results in better performances.

Results discriminated Machine Learning and Deep Learning models performance in terms of accuracy. Deep Learning models work better than Machine Learning ones, and the difference between the two Deep Learning models is negligible. The performances of the ResNet were expected, as similar results were also found in other time series classification tasks (Lomio et al. 2019). The better performance of the deep learning models can be attributed also to the fact that these can take into account the time dependency of the commits as this can bring additional useful information which should be considered (Saarimäki et al. 2022). Compared with Machine Learning models, Deep Learning increases the AUC rate, enables the correct fault identification, and decreases the probability of an incorrectly identification.

Regarding the preprossessing approach, we found that, independently from the classifier categories, when the dataset is imbalanced, the commits labeled as fault inducing represent a very small portion of the total number of commits. The inclusion of an oversampling step (e.g., SMOTE) improves the performance of the classifiers. Therefore, we recommend researcher to consider oversampling techniques in similar contexts.

## 6 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 (Yin 2009).

**Construct Validity** This threat concerns the relationship between theory and observation due to possible measurement errors. SonarQube is one of the most adopted static analysis tool by developers (Vassallo et al. 2019a; Avgeriou et al. 2021). Nevertheless, we cannot exclude the presence of false positives or false negatives in the detected warnings; further analyses on these aspects are part of our future research agenda. As for code smells, we employed a manually-validated oracle, hence avoiding possible issues due to the presence of false positives and negatives. 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 to understand the fault proneness of the rules adopted by the tools without modifying them to reflect the real impact that developers would have while using the tools. In future works, we plan to replicate this work manually validating a statistically significant sample of violations, to assess the impact of false positives on the achieved findings. In addition, it is worth mentioning that while SonarQube is a very well known and used static analysis tool, there are many others from which it differs for number and type of metrics. This could therefore lead to very different prediction results in terms of fault-proneness. For this reason, in the future we plan on further extending the analysis including and comparing static analysis tool beyond SonarQube. As for the analysis time frame, 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** This threat concerns internal factors related to the study that might have affected the results. As for the identification of the fault-inducing commits, we relied on the SZZ algorithm (Śliwerski et al. 2005). 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 locations than in the lines that induced them. Moreover, we are aware that the imbalanced data could have influenced the results (more than 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** Our study considered the 33 Java open-source software projects with different scope and characteristics included in the Technical Debt dataset. All the 29 Java projects are members of the Apache Software Foundations that incubates only certain systems that follow specific and strict quality rules. Our empirical 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. We analyzed commits until the end of 2015, considering all the faults raised until the end of March 2018. We are aware that recent data can provide different results.

We are aware that different programming languages, and projects at different maturity levels could provide different results. Our empirical 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.

**Conclusion Validity** This threat concerns the relationship between the treatment and the outcome. We adopted different machine learning and deep learning models to reduce the bias of the low prediction power that a single classifier could have. We also addressed possible issues due to multicollinearity, missing hyper-parameter configuration, and data imbalance. We recognize, however, that other statistical or machine learning techniques might have yielded similar or even better accuracy than the techniques we used. It is not to be excluded that the results might differ slightly when considering a within-project validation. Unfortunately, due to the nature of the data, having less than 5% of samples belonging to the positive class, the only way to have enough samples of both classes is to consider all projects together, using, therefore, a cross-project validation setting. We tried using a within-project validation, but this unfortunately would "break" the algorithms used since there are many data "splits" in which there are no inducing commits. For this reason we chose to use a cross-project validation.

## 7 Related Work

Software defect prediction is one of the most active research areas in software engineering. Faults prediction has been deeply investigated in the last years, where research focused mainly on improving the predictions granularity (Pascarella et al. 2019) such as method or

file (Menzies et al. 2010; Kim et al. 2011; Bettenburg et al. 2012; Prechelt and Pepper 2014), adding features, e.g., code review (McIntosh and Kamei 2018), change context (Kondo et al. 2019), or applying machine and deep learning models (Hoang et al. 2019; Lenarduzzi et al. 2020e).

As factors to predict bug-inducing changes some authors adopted change based metrics (McIntosh and Kamei 2018), including size (Kamei et al. 2013), the history of a change as well as developer experience (Kamei et al. 2013), or churn metrics (Tan et al. 2015). Another study included code review metrics for the predictive models (McIntosh and Kamei 2018). One aspect investigated was also the decreasing of the effort required to diagnose a defect (Pascarella et al. 2019). Researchers included several other software properties, like structural (Basili et al. 1996; Chidamber and Kemerer 1994), historical (D'Ambros et al. 2012; Graves et al. 2000), and alternative (Bird et al. 2011; Pascarella et al. 2020; Palomba et al. 2017) metrics. The achieved results considering software properties, product and process metrics are the most promising ones (Pascarella et al. 2020).

In the recent years, researchers investigated mainly shorter-term defects analysis, since this better fits the developers' needs (Pascarella et al. 2018b). Moreover, developers can immediately identify defects in their code adopting shorter-term approaches (Yang et al. 2016).

Two studies included as factors static analysis warnings (Querel and Rigby 2018; Trautsch et al. 2020) for building just-in-time defect prediction models. According to their results, they can improve the predictive models accuracy (Querel and Rigby 2018). Moreover, both code metrics and static analysis warnings are correlated with bugs and that they can improve the prediction (Trautsch et al. 2020).

The most adopted approaches are based on supervised (Graves et al. 2000; Hall et al. 2012; Jing et al. 2014) and unsupervised models (Fu and Menzies 2017; Li et al. 2020). These models consider features such as product (e.g., CK metrics Chidamber and Kemerer 1994) or process features (e.g., entropy of the development process Hassan 2009b).

Significant milestones for just-in-time defect prediction are represented by the works made by Kamei et al. (2012, 2016). They proposed a just-in-time prediction model to predict whether or not a change will lead to a defect with the aim of reducing developers and reviewers' effort. In particular, they applied logistic regression considering different change measures such as diffusion, size, and purpose, obtaining an average accuracy of 68% and an average recall of 64%. More recently, Pascarella et al. (2019) complemented their results considering the attributes necessary to filter only those files that are defect-prone. The reduced granularity is justified by the fact that 42% of defective commits are partially defective, i.e., composed of both files that are changed without introducing defects and files that are changed introducing defects. Furthermore, in almost 43% of the changed files a defect is introduced, while the remaining files are defect-free.

Faults prediction were investigated adopting Machine learning models focusing on the features role such as change size or changes history, that can represent a code change, and using them as predictors (Kamei et al. 2013; Pascarella et al. 2018a, 2019).

Machine learning techniques were also largely applied in detection of technical issues in the code, such as code smells (Arcelli Fontana et al. 2016; Di Nucci et al. 2018; Pecorelli et al. 2020b; Lujan et al. 2020). While machine learning has been mainly applied to detect different code smell types (Khomh 2009; Khomh et al. 2011), unfortunately, only few studies applied machine learning techniques to investigate static analysis tool rules, such as SonarQube (Falessi et al. 2017; Tollin et al. 2017; Lenarduzzi et al. 2020e) or PMD (Lenarduzzi et al. 2021c).

Considering defect prediction Yang et al. (2017) proposes a novel approach TLEL composed by a two layer ensemble learning technique. In the inner layer, we adopted bagging based on decision tree to build a Random Forest model. In the outer layer, they ensembled many different Random Forest models.

Machine learning techniques were applied to detect multiple code smell types (Arcelli Fontana et al. 2016), estimate their harmfulness (Arcelli Fontana et al. 2016), determine the intensity (Arcelli Fontana and Zanoni 2017), and to classify code smells according to their perceived criticality (Pecorelli et al. 2020b). The training data selection can influence the performance of machine learning-based code smell detection approaches (Di Nucci et al. 2018) since the code smells detected in the code are generally few in terms of number of occurrences (Pecorelli et al. 2020a).

Moreover, machine learning algorithms were successfully applied to derive code smells from different software metrics (Maneerat and Muenchaisri 2011).

Considering the detection of static analysis tool rules, SonarQube was the tool mainly investigated, focusing on the effect of the presence of its rules on fault-proneness (Falessi et al. 2017; Lenarduzzi et al. 2020e) or the change-proneness (Tollin et al. 2017).

Machine learning approaches were successfully applied since results showed that 20% of faults were avoidable if the SonarQube-related issues would have been removed (Falessi et al. 2017), however, the harmfulness of the SonarQube rules is very low (Lenarduzzi et al. 2020e). Positive results application were collected also considering class change-proneness (Tollin et al. 2017).

Machine learning approaches were also used to determine if the SonarQube technical debt was be predicted based also on software metrics (Lenarduzzi et al. 2019a). Results demonstrated the impossibility to have positive prediction. Another point of view which has benefited from machine learning was the evaluation of the remediation effort calculated by SonarQube (Saarimaki et al. 2019; Baldassarre et al. 2020). Results highlighted the model overestimation of the time to fix the Technical Debt-related issues.

In order to satisfy computer performance that are fastly increasing in the last years, Deep Learning is becoming popular in many domains (Hinton and Salakhutdinov 2006) such as image classification (Krizhevsky et al. 2017) or natural language processing (Sarikaya et al. 2014). There also many existing studies that leverage deep learning techniques to address other problems in software engineering (White et al. 2015; Lam et al. 2015; Gu et al. 2016, 2018; Guo et al. 2017). Since the promising results, Deep Learning could be a valid approach to adopt also in bug prediction in order to improve the performance of just-in-time defect prediction.

Deep learning can be useful to improve the logistic regression weaknesses when the study should combine features to generate new ones. This approach was successfully applied in Yang et al. (2015) considering 14 traditional change level features in order to predict bugs.

The benefit of using Deep Learning instead of machine Learning to improve the performance of just-in-time defect prediction is still under investigation (Yang et al. 2015; Abozeed et al. 2020; Ferenc et al. 2020; Wang et al. 2020). The results achieved until now demonstrates a promising improvement in the bug prediction accuracy compared with other approaches (32.22% more bugs detected) (Yang et al. 2015) especially for small dataset and in the feature selection (Abozeed et al. 2020), and to predict the presence of bugs in classes from static source code metrics (Ferenc et al. 2020).

Ones of the most adopted Deep Learning models to automate feature learning for defect prediction are Long Short Term Memory (Dam et al. 2021) and Convolutional Neural Network (Li et al. 2017). Another models well-known is Deep Belief Network (Wang et al. 2020).

Deep learning was applied in the context of defect prediction (Yang et al. 2015). Yang et al. (2015) proposed a Deeper approach to predict defect-prone changes obtaining promising results in terms of detection power and accuracy compared with traditional approaches such as Kamei et al. (2013).

## 8 Conclusion

In this paper, we investigated the fault-proneness of SonarQube rules and product and process metrics proposed by Rahman and Devanbu (2013), Kamei et al. (2012), and SonarQube 7.5 suite. We adopted five models, three Machine Learning and two Deep Learning ones.

In our previous work, on a reduced dataset (Lenarduzzi et al. 2020e), we found that SonarQube rules considered fault-inducing were not correctly classified. However, even if we obtained a good prediction accuracy, we could not accurately detect the impact of each rule on the fault-proneness. Results were also confirmed by our next work (Lenarduzzi et al. 2020b) on an extended dataset (the same considered in this work) where we applied statistical techniques to detect if the violation of any SonarQube rule impacted the fault-proneness.

In order to corroborate our previous results, and to clearly identify the impact of each different SonarQube rule, and the three sets of product and process metrics, in this work, we better preprocessed the data to avoid multicollinearity and to model an unbalanced dataset, and we adopted a more accurate data validation strategy.

Our work clearly identified best practices in terms of features, models, preprocessing.

Our results revealed unexpectedly that SonarQube rules are good fault predictors considering the historical data (Deep Learning models). The performance reached with Machine Learning models are lower than in the previous studies, but more realistic with the adopted preprocessing approach.

Moroever, product and process metrics Rahman and Devanbu (2013) and Kamei et al. (2012) are good fault predictors, confirming the previous founding on this last set of metrics. However, including the SonarQube metrics does provide an impressive accurate performance.

Therefore, we identified a clear set of metrics that provided a significantly higher accurate fault prediction (more than 95%). This result might enable developers to save time to manually verify each SonarQube rule and, therefore, only focus on fault prone features.

Considering the models and preprocessing that can achieve the higher accuracy performance achieved: Deep Learning models and the adoption of oversampling techniques (in particular, for Deep Learning) are the better solution. Compared with Machine Learning models, Deep Learning increases the AUC rate, enables the correct fault identification, and decreases the probability of an incorrectly identification.

Future works might consider the adoption of time series analysis and anomaly detection techniques, since in our work, the data present two main characteristics: unbalanced data and time dependency of the commit data. To overcome these two "aspects" we opted to include Synthetic Minority Oversampling Technique (SMOTE) and Deep Learning Models in our data preprocessing and data analysis protocol to corroborate Machine Learning ones. A further alternative to confirm the results can be using time series analysis for time dependency of the commit data and anomaly detection for data unbalanced. Another possible future work could be to investigate whether other static analysis tools, such as FindBugs or Checkstyle (Pecorelli et al. 2022; Lenarduzzi et al. 2021b), can be complementary to SonarQube and can provide similar or different results, also considering other dataset (Nguyen et al. 2022).

# Appendix

**Table 15** Metrics accuracy (%) comparison *Synthetic Minority Oversampling Technique* for SonarQube rules ($RQ_1$)

| SQ rules | Machine learning | | | Deep learning | |
|---|---|---|---|---|---|
| | Gradient boost | Random forest | XG boost | FCNN | RN |
| AUC | 59.64 | 54.78 | 58.26 | 64.63 | 63.15 |
| F-measure | 1.61 | 0.09 | 0.42 | 3.20 | 4.10 |
| Precision | 4.74 | 0.09 | 2.21 | 2.26 | 2.75 |
| Recall | 1.46 | 0.09 | 0.24 | 15.09 | 19.90 |
| MCC | 2.04 | −0.03 | 0.66 | 3.26 | 4.73 |
| FPR | 0.22 | 0.07 | 0.02 | 6.41 | 6.38 |
| TNR | 99.78 | 99.93 | 99.98 | 93.59 | 93.62 |
| FNR | 98.54 | 99.91 | 99.76 | 84.91 | 80.10 |

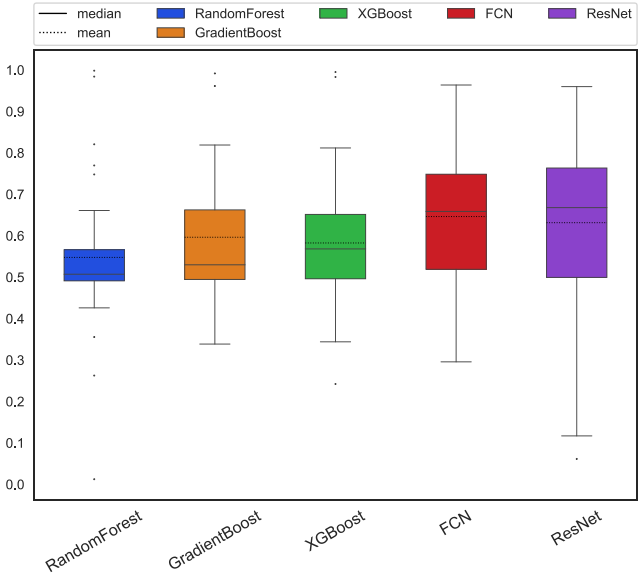


**Fig. 23** AUC comparison among Machine Learning and Deep Learning models for software metrics without *Synthetic Minority Oversampling Technique* for SonarQube rules ($RQ_1$)

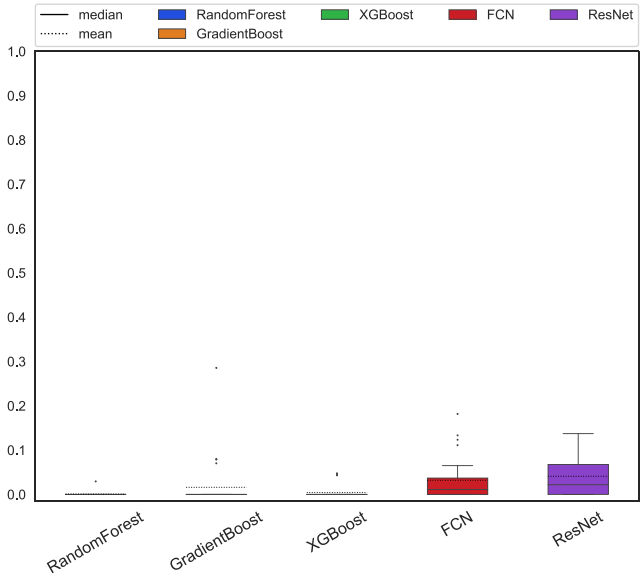


**Fig. 24** F-measure comparison among Machine Learning and Deep Learning models for software metrics without *Synthetic Minority Oversampling Technique* for SonarQube rules ($RQ_1$)

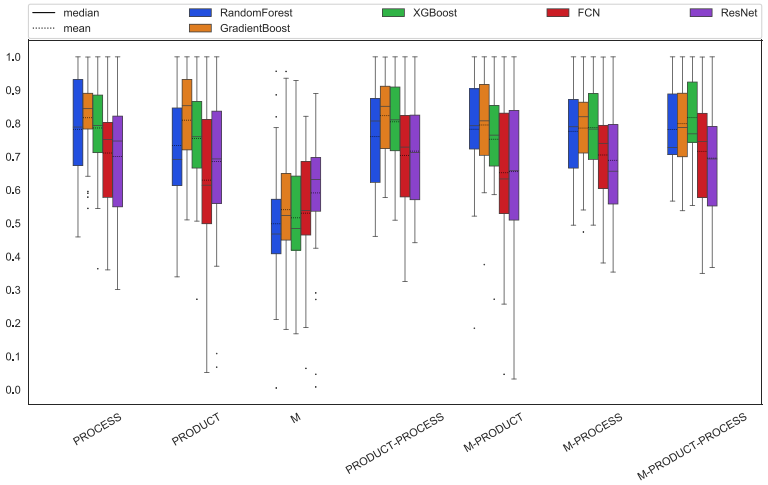


**Fig. 25** AUC comparison among Machine Learning and Deep Learning models for software metrics without *Synthetic Minority Oversampling Technique* for software metrics ($RQ_2$)

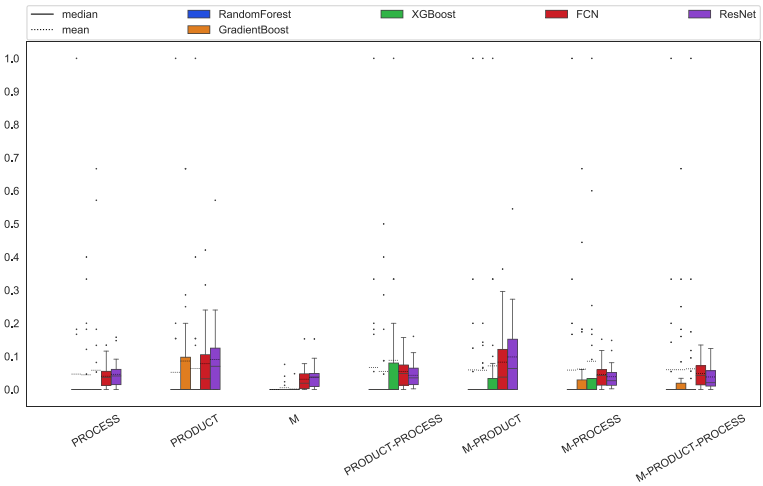


**Fig. 26** F-measure comparison among Machine Learning and Deep Learning models for software metrics without *Synthetic Minority Oversampling Technique* for software metrics ($RQ_2$)

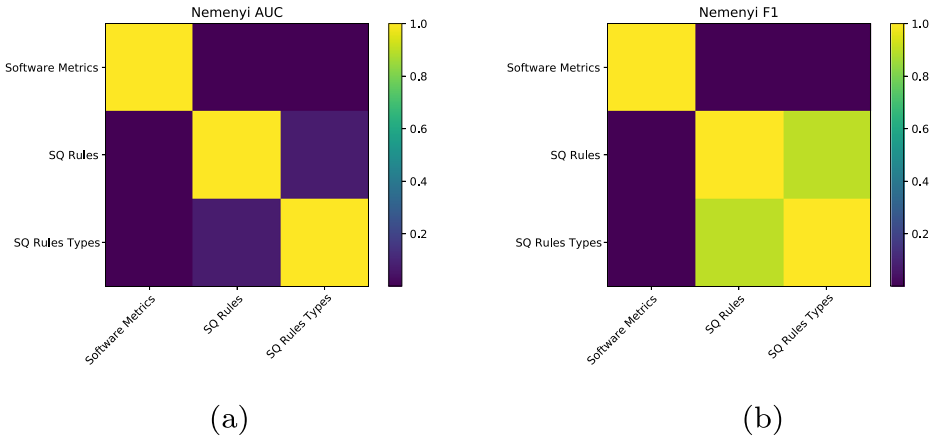


**Fig. 27** Nemenyi test for comparing the different the results obtained using SQ metrics, and its combinations with SQ Rules and SQ Rules Type ($RQ_3$)

**Table 16** Metrics accuracy (%) comparison without *Synthetic Minority Oversampling Technique* for software metrics ($RQ_2$)

| Metrics | Machine learning | | | Deep learning | |
|---|---|---|---|---|---|
| | Gradient boost | Random forest | XG boost | FCNN | RN |
| SonarQube metrics | | | | | |
| AUC | 54.07 | 49.84 | 51.64 | 53.04 | 59.12 |
| F-measure | 0.52 | 0.00 | 0.16 | 3.10 | 3.64 |
| Precision | 0.28 | 0.00 | 0.16 | 1.77 | 1.99 |
| Recall | 6.28 | 0.00 | 0.16 | 47.93 | 57.82 |
| MCC | -0.18 | -0.04 | 0.04 | 1.62 | 3.23 |
| FPR | 6.76 | 0.05 | 0.13 | 43.24 | 45.84 |
| TNR | 93.24 | 99.95 | 99.87 | 56.76 | 54.16 |
| FNR | 93.72 | 100.00 | 99.84 | 52.07 | 42.18 |
| Process metrics | | | | | |
| AUC | 81.70 | 78.23 | 78.63 | 71.10 | 70.06 |
| F-measure | 4.42 | 4.65 | 5.82 | 4.00 | 4.59 |
| Precision | 17.82 | 8.62 | 11.04 | 2.19 | 2.62 |
| Recall | 2.63 | 4.14 | 4.14 | 58.11 | 60.71 |
| MCC | 6.32 | 5.24 | 6.44 | 4.67 | 5.46 |
| FPR | 0.08 | 0.02 | 0.07 | 36.04 | 36.90 |
| TNR | 99.92 | 99.98 | 99.93 | 63.96 | 63.10 |
| FNR | 97.37 | 95.86 | 95.86 | 41.89 | 39.29 |
| Product metrics | | | | | |
| AUC | 80.98 | 73.35 | 75.45 | 62.96 | 68.55 |
| F-measure | 8.59 | 5.20 | 6.35 | 7.79 | 9.04 |
| Precision | 14.61 | 10.69 | 8.39 | 10.51 | 12.35 |
| Recall | 8.08 | 4.45 | 6.17 | 20.90 | 21.34 |
| MCC | 9.62 | 6.08 | 6.64 | 9.51 | 10.96 |
| FPR | 0.06 | 0.02 | 0.05 | 2.62 | 1.53 |
| TNR | 99.94 | 99.98 | 99.95 | 97.38 | 98.47 |
| FNR | 91.92 | 95.55 | 93.83 | 79.10 | 78.66 |
| Product + process metrics | | | | | |
| AUC | 82.33 | 76.02 | 80.48 | 70.41 | 71.68 |
| F-measure | 5.48 | 6.68 | 8.82 | 5.44 | 4.29 |
| Precision | 13.60 | 16.67 | 25.29 | 3.13 | 2.44 |
| Recall | 5.81 | 5.32 | 6.75 | 42.21 | 66.52 |
| MCC | 6.81 | 8.29 | 11.16 | 6.70 | 5.17 |
| FPR | 0.13 | 0.01 | 0.05 | 14.97 | 41.38 |
| TNR | 99.87 | 99.99 | 99.95 | 85.03 | 58.62 |
| FNR | 94.19 | 94.68 | 93.25 | 57.79 | 33.48 |
| SonarQube + process metrics | | | | | |
| AUC | 78.60 | 77.63 | 78.76 | 70.51 | 68.92 |
| F-measure | 6.16 | 5.86 | 8.54 | 4.49 | 3.85 |
| Precision | 11.66 | 13.79 | 17.04 | 2.62 | 2.29 |
| Recall | 6.90 | 4.83 | 7.81 | 53.09 | 70.75 |

**Table 16** (continued)

| Metrics | Machine learning | | | Deep learning | |
|---|---|---|---|---|---|
| | Gradient boost | Random forest | XG boost | FCNN | RN |
| MCC | 6.66 | 7.09 | 9.40 | 5.66 | 4.16 |
| FPR | 1.73 | 0.03 | 0.19 | 27.06 | 52.53 |
| TNR | 98.27 | 99.97 | 99.81 | 72.94 | 47.47 |
| FNR | 93.10 | 95.17 | 92.19 | 46.91 | 29.25 |
| SonarQube + product metrics | | | | | |
| AUC | 79.53 | 78.26 | 75.25 | 65.18 | 65.51 |
| F-measure | 5.82 | 5.90 | 7.13 | 8.26 | 9.83 |
| Precision | 8.87 | 10.46 | 14.48 | 9.45 | 12.83 |
| Recall | 5.87 | 5.11 | 6.10 | 22.51 | 19.23 |
| MCC | 6.08 | 6.54 | 7.94 | 9.90 | 11.43 |
| FPR | 0.34 | 0.04 | 0.09 | 2.35 | 1.18 |
| TNR | 99.66 | 99.96 | 99.91 | 97.65 | 98.82 |
| FNR | 94.13 | 94.89 | 93.90 | 77.49 | 80.77 |
| SonarQube + product + process metrics | | | | | |
| AUC | 79.93 | 78.19 | 81.73 | 71.62 | 69.58 |
| F-measure | 5.96 | 5.97 | 6.24 | 4.86 | 3.78 |
| Precision | 11.33 | 12.64 | 14.80 | 2.86 | 2.17 |
| Recall | 6.25 | 4.94 | 5.39 | 48.83 | 69.85 |
| MCC | 6.62 | 7.03 | 7.23 | 6.15 | 3.54 |
| FPR | 0.73 | 0.02 | 0.13 | 21.37 | 54.10 |
| TNR | 99.27 | 99.98 | 99.87 | 78.63 | 45.90 |
| FNR | 93.75 | 95.06 | 94.61 | 51.17 | 30.15 |

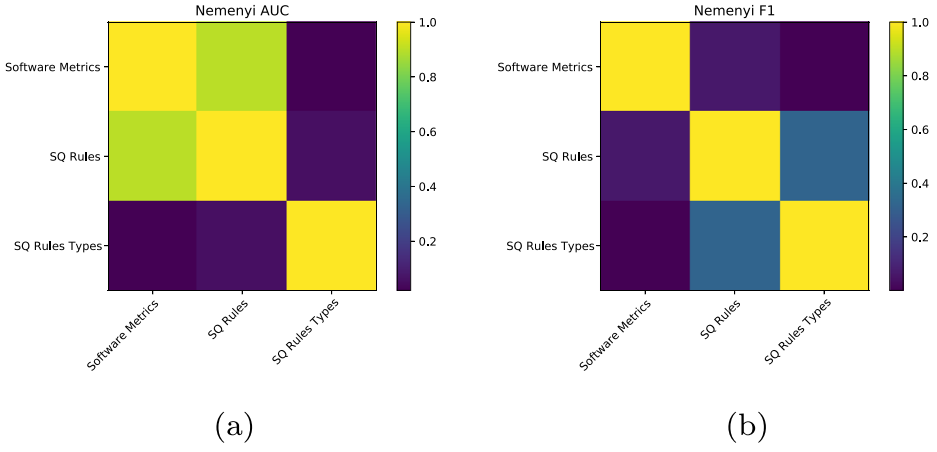


**Fig. 28** Nemenyi test for comparing the different the results obtained using SQ + Rahman and Devanbu (2013) metrics, and their combinations with SQ Rules and SQ Rules Type ($RQ_3$)

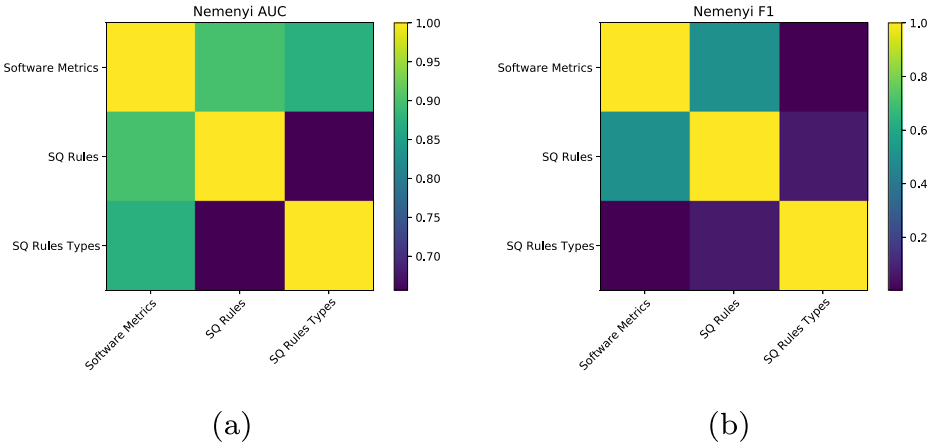


**Fig. 29** Nemenyi test for comparing the different the results obtained using SQ + Kamei et al. (2012) metrics, and their combinations with SQ Rules and SQ Rules Type ($RQ_3$)

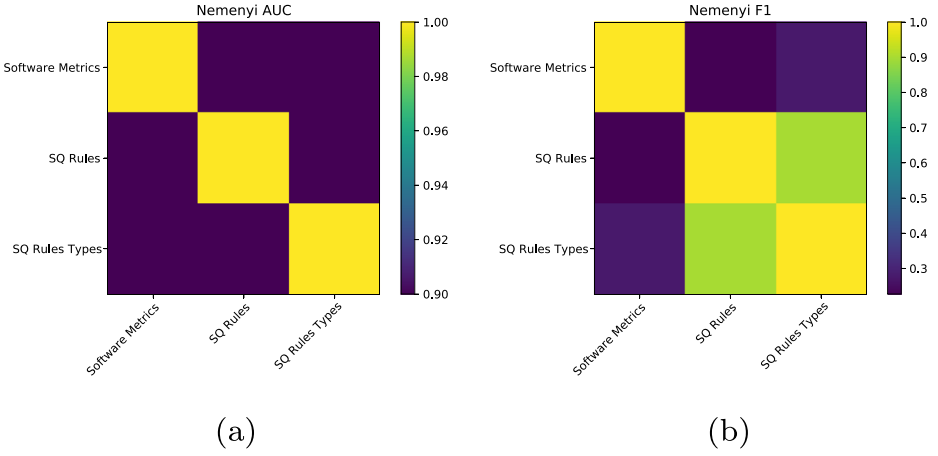


**Fig. 30** Nemenyi test for comparing the different the results obtained using Rahman and Devanbu (2013) metrics, and its combinations with SQ Rules and SQ Rules Type ($RQ_3$)

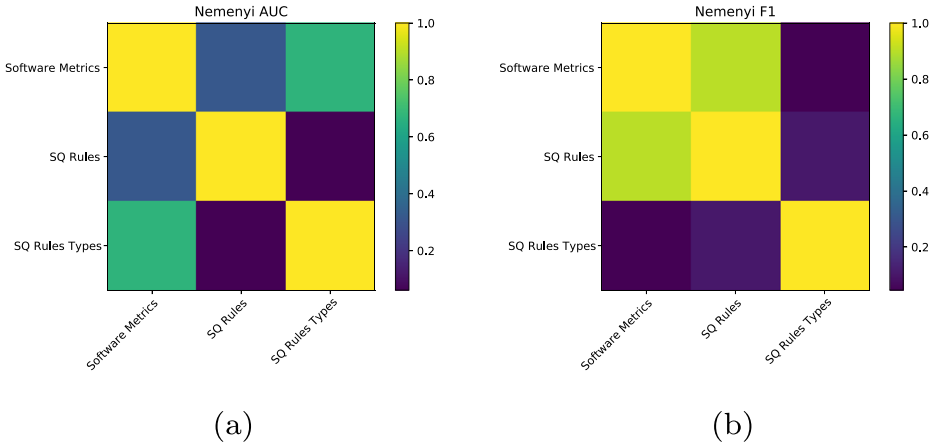


**Fig. 31** Nemenyi test for comparing the different the results obtained using Kamei et al. (2012) metrics, and its combinations with SQ Rules and SQ Rules Type ($RQ_3$)

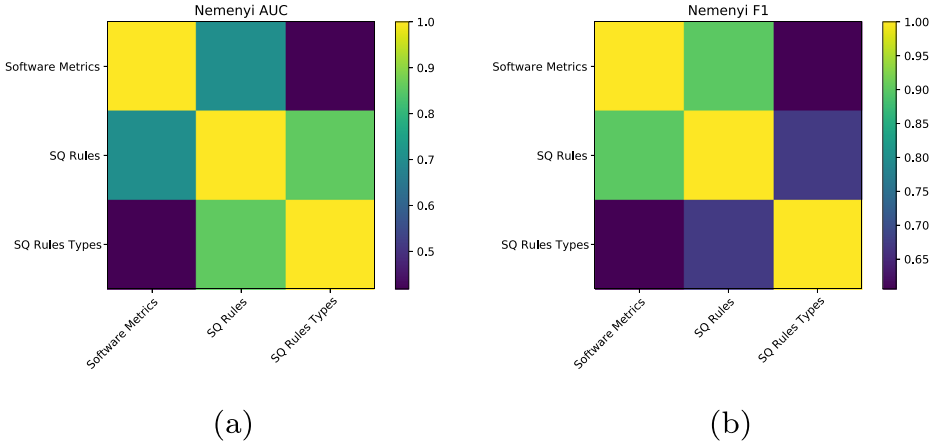


**Fig. 32** Nemenyi test for comparing the different the results obtained using Rahman and Devanbu (2013) and Kamei et al. (2012) metrics, and their combinations with SQ Rules and SQ Rules Type ($RQ_3$)

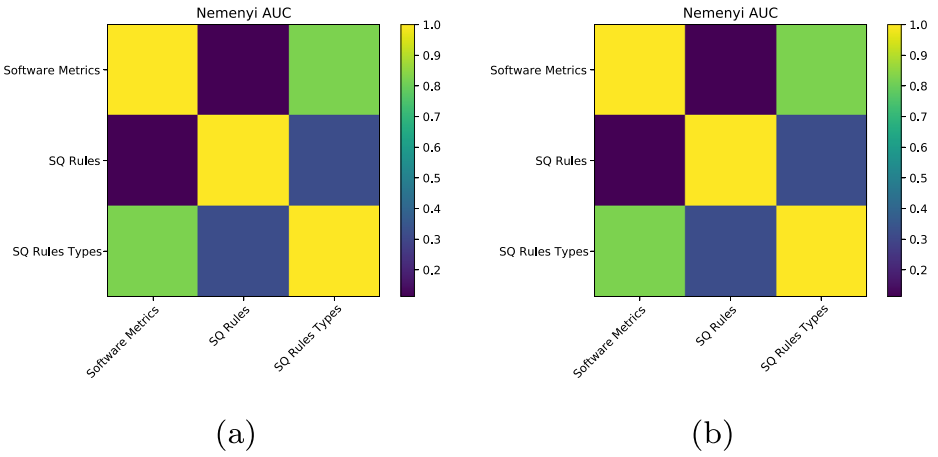


**Fig. 33** Nemenyi test for comparing the different the results obtained using SQ + Rahman and Devanbu (2013) + Kamei et al. (2012) metrics, and their combinations with SQ Rules and SQ Rules Type ($RQ_3$)

## Declarations

**Conflict of Interest** Authors have no conflict of interest



## References

Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M, Ghemawat S, Goodfellow I, Harp A, Irving G, Isard M, Jia Y, Jozefowicz R, Kaiser L, Kudlur M, Levenberg J, Mané D, Monga R, Moore S, Murray D, Olah C, Schuster M, Shlens J, Steiner B, Sutskever I, Talwar K, Tucker P, Vanhoucke V, Vasudevan V, Viégas F, Vinyals O, Warden P, Wattenberg M, Wicke M, Yu Y, Zheng X (2015) Tensorflow: Large-scale machine learning on heterogeneous systems. https://www.tensorflow.org/, software available from tensorflow.org

Abozeed SM, ElNainay MY, Fouad SA, Abougabal MS (2020) Software bug prediction employing feature selection and deep learning. In: International conference on advances in the emerging computing technologies (AECT), pp 1–6

Arcelli Fontana F, Zanoni M (2017) Code smell severity classification using machine learning techniques. Know-Based Syst 128(C):43–58

Arcelli Fontana F, Mäntylä MV, Zanoni M, Marino A (2016) Comparing and experimenting machine learning techniques for code smell detection. Empir Softw Eng 21(3):1143–1191

Avgeriou PC, Taibi D, Ampatzoglou A, Arcelli Fontana F, Besker T, Chatzigeorgiou A, Lenarduzzi V, Martini A, Moschou N, Pigazzini I, Saarimaki N, Sas DD, de Toledo SS, Tsintzira AA (2020) An overview and comparison of technical debt measurement tools. IEEE Softw

Avgeriou P, Taibi D, Ampatzoglou A, Arcelli Fontana F, Besker T, Chatzigeorgiou A, Lenarduzzi V, Martini A, Moschou N, Pigazzini I, Saarimäki N, Sas D, Soares de Toledo S, Tsintzira A (2021) An overview and comparison of technical debt measurement tools. IEEE Softw

Baldassarre MT, Lenarduzzi V, Romano S, Saarimaki N (2020) On the diffuseness of technical debt items and accuracy of remediation time when using sonarqube. In: Information software system
Basili VR, Briand LC, Melo WL (1996) A validation of object-oriented design metrics as quality indicators. IEEE Trans Softw Eng 22(10):751–761
Beller M, Spruit N, Spinellis D, Zaidman A (2018) On the dichotomy of debugging behavior among programmers. In: 40th International conference on software engineering, ICSE '18, pp 572–583
Bettenburg N, Nagappan M, Hassan AE (2012) Think locally, act globally: improving defect and effort prediction models. In: Working conference on mining software repositories (MSR), pp 60–69
Bird C, Nagappan N, Murphy B, Gall H, Devanbu P (2011) Don't touch my code! Examining the effects of ownership on software quality. In: 13th European conference on foundations of software engineering, pp 4–14
Breiman L (1996) Bagging predictors. Mach Learn 24(2):123–140
Breiman L (2001) Random forests. Mach Learn 45(1):5–32
Breiman L, Friedman J, Stone CJ, Olshen R (1984) Classification and regression trees Regression trees. Chapman and Hall, New York
Carver J (2010) Towards reporting guidelines for experimental replications: a proposal. In: 1st International workshop on replication in empirical software engineering research (RESER 2010)
Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP (2002) Smote: synthetic minority over-sampling technique. J Artif Intell Res 16:321–357
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
Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20
Chollet F et al (2015) Keras. https://keras.io
Dam HK, Tran T, Pham T, Ng SW, Grundy J, Ghose A (2021) Automatic feature learning for predicting vulnerable software components. IEEE Trans Softw Eng 47(1):67–85
D'Ambros M, Lanza M, Robbes R (2010) An extensive comparison of bug prediction approaches. In: IEEE Working conference on mining software repositories (MSR 2010), pp 31–41
D'Ambros M, Lanza M, Robbes R (2012) Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empir Softw Eng 17(4):531–577
Di Nucci D, Palomba F, Tamburri D, Serebrenik A, De Lucia A (2018) Detecting code smells using machine learning techniques: are we there yet?
Falessi D, Russo B, Mullen K (2017) What if i had no smells? In: International symposium on empirical software engineering and measurement (ESEM), pp 78–84
Fawaz HI, Forestier G, Weber J, Idoumghar L, Muller P (2019) Deep learning for time series classification: a review. Data Min Knowl Disc 33(4):917–963
Ferenc R, Bán D, Grósz T, Gyimóthy T (2020) Deep learning in static, metric-based bug prediction. Array 6:100021
Fowler M, Beck K (1999) Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co, Inc
Freund Y, Schapire RE (1997) A decision-theoretic generalization of on-line learning and an application to boosting. J Comput Syst Sci 55(1):119–139
Friedman JH (2001) Greedy function approximation: a gradient boosting machine. Ann Stat 29:1189–1232
Fu W, Menzies T (2017) Revisiting unsupervised learning for defect prediction. In: 11th Joint meeting on foundations of software engineering, ESEC/FSE 2017, pp 72–83
Gatrell M, Counsell S (2015) The effect of refactoring on change and fault-proneness in commercial c# software. Sci Comput Program 102(C):44–56
Geurts P, Ernst D, Wehenkel L (2006) Extremely randomized trees. Mach Learn 63(1):3–42
Graves TL, Karr AF, Marron JS, Siy H (2000) Predicting fault incidence using software change history. IEEE Trans Softw Eng 26(7):653–661
Gu X, Zhang H, Zhang D, Kim S (2016) Deep api learning. In: International symposium on foundations of software engineering, FSE 2016, pp 631–642
Gu X, Zhang H, Kim S (2018) Deep code search. In: International conference on software engineering (ICSE), pp 933–944
Guo J, Cheng J, Cleland-Huang J (2017) Semantically enhanced software traceability using deep learning techniques. In: International conference on software engineering (ICSE), pp 3–14
Gyimothy T, Ferenc R, Siket I (2005) Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Trans Softw Eng 31(10):897–910
Hall T, Beecham S, Bowes D, Gray D, Counsell S (2012) A systematic literature review on fault prediction performance in software engineering. IEEE Trans Softw Eng 38
Hassan AE (2009a) Predicting faults using the complexity of code changes. In: 31st International conference on software engineering, ICSE '09, pp 78–88

Hassan AE (2009b) Predicting faults using the complexity of code changes. In: International conference on software engineering. IEEE, pp 78–88
Hassan AE, Holt RC (2005) The top ten list: dynamic fault prediction. In: 21st International conference on software maintenance (ICSM'05), pp 263–272
He K, Zhang X, Ren S, Sun J (2016) Deep residual learning for image recognition. In: IEEE Conference on computer vision and pattern recognition, pp 770–778
Hinton GE, Salakhutdinov RR (2006) Reducing the dimensionality of data with neural networks. Science 313(5786):504–507
Hoang T, Khanh Dam H, Kamei Y, Lo D, Ubayashi N (2019) Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In: 16th International conference on mining software repositories (MSR), pp 34–45
Ioffe S, Szegedy C (2015) Batch normalization: accelerating deep network training by reducing internal covariate shift. arXiv:150203167
Jing XY, Ying S, Zhang ZW, Wu SS, Liu J (2014) Dictionary learning based software defect prediction. In: International conference on software engineering, ICSE 2014, pp 414–423
Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N (2012) A large-scale empirical study of just-in-time quality assurance. IEEE Trans Softw Eng 39
Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N (2013) A large-scale empirical study of just-in-time quality assurance. IEEE Trans Softw Eng 39(6):757–773
Kamei Y, Fukushima T, McIntosh S, Yamashita K, Ubayashi N, Hassan AE (2016) Studying just-in-time defect prediction using cross-project models. Empir Softw Eng 21
Khomh F (2009) Squad: software quality understanding through the analysis of design. In: WCRE '09. IEEE Computer Society, Washington, pp 303–306
Khomh F, Vaucher S, Guéhéneuc YG, Sahraoui H (2011) Bdtex: a gqm-based bayesian approach for the detection of antipatterns. J Syst Softw 84(4):559–572
Kim S, Zimmermann T, Whitehead EJ Jr, Zeller A (2007) Predicting faults from cached history. In: 29th International conference on software engineering (ICSE'07), pp 489–498
Kim S, Zhang H, Wu R, Gong L (2011) Dealing with noise in defect prediction. In: International conference on software engineering, ICSE '11, pp 481–490
Kondo M, Germán D, Mizuno O, Choi EH (2019) The impact of context metrics on just-in-time defect prediction. Empir Softw Eng 25:890–939
Krizhevsky A, Sutskever I, Hinton GE (2017) Imagenet classification with deep convolutional neural networks. Commun ACM 60(6):84–90
Lam AN, Nguyen AT, Nguyen HA, Nguyen TN (2015) Combining deep learning with information retrieval to localize buggy files for bug reports (n). In: International conference on automated software engineering (ASE), pp 476–481
Lenarduzzi V, Sillitti A, Taibi D (2017) Analyzing forty years of software maintenance models. In: 39th International conference on software engineering companion, ICSE-c '17
Lenarduzzi V, Martini A, Taibi D, Tamburri DA (2019a) Towards surgically-precise technical debt estimation: Early results and research roadmap. In: 3rd international workshop on machine learning techniques for software quality evaluation, maLTeSQue 2019, pp 37–42
Lenarduzzi V, Saarimäki N, Taibi D (2019b) The technical debt dataset. In: 15th Conference on predictive models and data analycs in software engineering, PROMISE '19
Lenarduzzi V, Palomba F, Taibi D, Tamburri DA (2020a) Openszz: a free, open-source, web-accessible implementation of the szz algorithm. In: International conference on program comprehension, ICPC '20, pp 446–450
Lenarduzzi V, Saarimäki N, Taibi D (2020b) Some sonarqube issues have a significant but small effect on faults and changes. A large-scale empirical study. J Syst Softw 170:110750
Lenarduzzi V, Sillitti A, Taibi D (2020c) A survey on code analysis tools for software maintenance prediction. In: 6th International conference in software engineering for defence applications. Springer International Publishing, pp 165–175
Lenarduzzi V, Sillitti A, Taibi D (2020d) A survey on code analysis tools for software maintenance prediction. In: 6th International conference in software engineering for defence applications. Springer International Publishing, pp 165–175
Lenarduzzi V, Lomio F, Huttunen H, Taibi D (2020e) Are sonarqube rules inducing bugs? In: 27th International conference on software analysis, evolution and reengineering (SANER), pp 501–511
Lenarduzzi V, Besker T, Taibi D, Martini A, Arcelli Fontana F (2021a) A systematic literature review on technical debt prioritization: strategies, processes, factors, and tools. J Syst Softw 171:110827
Lenarduzzi V, Lujan S, Saarimaki N, Palomba F (2021b) A critical comparison on six static analysis tools: detection, agreement, and precision. arXiv:2101.08832

Lenarduzzi V, Nikkola V, Saarimäki N, Taibi D (2021c) Does code quality affect pull request acceptance? An empirical study. J Syst Softw 171
Li J, He P, Zhu J, Lyu MR (2017) Software defect prediction via convolutional neural network. In: International conference on software quality, reliability and security (QRS), pp 318–328
Li W, Zhang W, Jia X, Huang Z (2020) Effort-aware semi-supervised just-in-time defect prediction. Inf Softw Technol 126:106364
Lin M, Chen Q, Yan S (2013) Network in network. arXiv:13124400
Lomio F, Skenderi E, Mohamadi D, Collin J, Ghabcheloo R, Huttunen H (2019) Surface type classification for autonomous robot indoor navigation. In: Workshop at 27th European signal processing conference (EUSIPCO)
Lomio F, Moreschini S, Lenarduzzi V (2022) A machine and deep learning analysis among sonarqube rules, product, and process metrics for faults prediction. https://figshare.com/s/a65ea232162c7352c879
Long J, Shelhamer E, Darrell T (2015) Fully convolutional networks for semantic segmentation. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 3431–3440
Lujan S, Pecorelli F, Palomba F, De Lucia A, Lenarduzzi V (2020) A preliminary study on the adequacy of static analysis warnings with respect to code smell prediction, pp 1–6
Maneerat N, Muenchaisri P (2011) Bad-smell prediction from software design model using machine learning techniques. In: 8th International joint conference on computer science and software engineering (JCSSE), pp 331–336
McIntosh S, Kamei Y (2018) Are fix-inducing changes a moving target? A longitudinal case study of just-in-time defect prediction. IEEE Trans Softw Eng 44(5):412–428
Menzies T, Milton Z, Turhan B, Cukic B, Jiang Y, Bener A (2010) Defect prediction from static code features: current results, limitations, new approaches. Autom Softw Eng 1:375–407
Moser R, Pedrycz W, Succi G (2008) A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: 30th International conference on software engineering, pp 181–190
Murphy-Hill E, Zimmermann T, Bird C, Nagappan N (2015) The design space of bug fixes and how developers navigate it. IEEE Trans Softw Eng 41:65–81
Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density. In: 27th International conference on software engineering, 2005. ICSE 2005, pp 284–292
Nagappan N, Ball T, Zeller A (2006) Mining metrics to predict component failures. In: Proceedings of the 28th international conference on software engineering, ICSE '06, pp 452–461
Nair V, Hinton GE (2010) Rectified linear units improve restricted boltzmann machines. In: Proceedings of the 27th international conference on machine learning (ICML-10), pp 807–814
Nemenyi P (1962) Distribution-free multiple comparisons. In: Biometrics, vol 18, p 263
Nguyen H, Lomio F, Pecorelli F, Lenarduzzi V (2022) PANDORA: continuous mining software repository and dataset generation. In: EEE International conference on software analysis, evolution and reengineering (SANER2022). IEEE
O'Brien RM (2007) A caution regarding rules of thumb for variance inflation factors. Qual Quant 41(5):673–690
Osman H, Ghafari M, Nierstrasz O, Lungu M (2017) An extensive analysis of efficient bug prediction configurations. In: Proceedings of the 13th international conference on predictive models and data analytics in software engineering. Association for Computing Machinery, PROMISE, pp 107–116
Ostrand TJ, Weyuker EJ, Bell RM (2005) Predicting the location and number of faults in large software systems. IEEE Trans Softw Eng 31(4):340–355
Palomba F, Zanoni M, Fontana FA, De Lucia A, Oliveto R (2017) Toward a smell-aware bug prediction model. IEEE Trans Softw Eng 45(2):194–218
Palomba F, Bavota G, Penta MD, Fasano F, Oliveto R, Lucia AD (2018) On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empir Softw Eng 23(3):1188–1221
Pan K, Kim S, Whitehead EJ (2009) Toward an understanding of bug fix patterns. Empir Softw Eng 14(3):286–315
Pascarella L, Palomba F, Bacchelli A (2018a) Re-evaluating method-level bug prediction. In: 2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER), pp 592–601
Pascarella L, Spadini D, Palomba F, Bruntink M, Bacchelli A (2018b) Information needs in contemporary code review. ACM Hum-Comput Interact 2(CSCW):1–27
Pascarella L, Palomba F, Bacchelli A (2019) Fine-grained just-in-time defect prediction. J Syst Softw 150:22–36
Pascarella L, Palomba F, Bacchelli A (2020) On the performance of method-level bug prediction: a negative result. J Syst Softw 161
Patton M (2002) Qualitative evaluation and research methods. Sage, Newbury Park
Pecorelli F, Di Nucci D, De Roover C, De Lucia A (2020a) A large empirical assessment of the role of data balancing in machine-learning-based code smell detection. J Syst Softw 110693

Pecorelli F, Palomba F, Khomh F, De Lucia A (2020b) Developer-driven code smell prioritization. In: International conference on mining software repositories
Pecorelli F, Lujan S, Lenarduzzi V, Palomba F, De Lucia A (2022) On the adequacy of static analysis warnings with respect to code smell prediction. Empir Softw Eng
Powers DMW (2011) Evaluation: from precision, recall and f-measure to roc., informedness, markedness & correlation. J Mach Learn Technol 2(1):37–63
Prechelt L, Pepper A (2014) Why software repositories are not used for defect-insertion circumstance analysis more often: a case study. Inf Softw Technol 56(10)
Querel LP, Rigby PC (2018) 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, ESEC/FSE 2018, pp 892–895
Rahman F, Devanbu P (2013) How, and why, process metrics are better. In: International conference on software engineering. IEEE Press, pp 432–441
Runeson P, Höst M (2009) Guidelines for conducting and reporting case study research in software engineering. Empir Softw Engg 14(2):131–164
Saarimaki N, Baldassarre M, Lenarduzzi V, Romano S (2019) On the accuracy of sonarqube technical debt remediation time. In: SEAA Euromicro 2019
Saarimäki N, Lenarduzzi V, Taibi D (2019) On the diffuseness of code technical debt in open source projects of the apache ecosystem. In: International conference on technical debt (techdebt 2019)
Saarimäki N, Moreschini S, Lomio F, Penaloza R, Lenarduzzi V (2022) Towards a robust approach to analyze time-dependent data in software engineering
Saboury A, Musavi P, Khomh F, Antoniol G (2017) An empirical study of code smells in javascript projects. In: International conference on software analysis, evolution and reengineering (SANER 2017), pp 294–305
Sarikaya R, Hinton GE, Deoras A (2014) Application of deep belief networks for natural language understanding. IEEE/ACM Trans Audio Speech Lang Process 22(4):778–784
Śliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? In: International workshop on mining software repositories, MSR '05. ACM, New York, pp 1–5
Subramanyam R, Krishnan MS (2003) Empirical analysis of ck metrics for object-oriented design complexity: implications for software defects. IEEE Trans Softw Eng 29(4):297–310
Tan M, Tan L, Dara S, Mayeux C (2015) Online defect prediction for imbalanced data. In: IEEE International conference on software engineering, vol 2, pp 99–108
Tantithamthavorn C, Hassan AE (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
Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2017) An empirical comparison of model validation techniques for defect prediction models. IEEE Trans Softw Eng (TSE) (1)
Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2018) The impact of automated parameter optimization for defect prediction models. IEEE Trans Softw Eng (TSE)
Tollin I, Arcelli Fontana F, Zanoni M, Roveda R (2017) Change prediction through coding rules violations. In: Proceedings of the 21st international conference on evaluation and assessment in software engineering, pp 61–64
Trautsch A, Herbold S, Grabowski J (2020) 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)
Vassallo C, Panichella S, Palomba F, Proksch S, Zaidman A, Gall HC (2018) Context is king: the developer perspective on the usage of static analysis tools. In: 25th International conference on software analysis, evolution and reengineering (SANER)
Vassallo C, Panichella S, Palomba F, Proksc S, Gall H, Zaidman A (2019a) How developers engage with static analysis tools in different contexts. Empir Softw Eng
Wang Z, Yan W, Oates T (2017) Time series classification from scratch with deep neural networks: a strong baseline. In: 2017 International joint conference on neural networks (IJCNN), pp 1578–1585
Wang S, Liu T, Nam J, Tan L (2020) Deep semantic feature learning for software defect prediction. IEEE Trans Softw Eng 46(12):1267–1293
White M, Vendome C, Linares-Vasquez M, Poshyvanyk D (2015) Toward deep learning software repositories. In: 12th Working conference on mining software repositories, pp 334–345
Yang X, Lo D, Xia X, Zhang Y, Sun J (2015) Deep learning for just-in-time defect prediction. In: IEEE International conference on software quality, reliability and security, pp 17–26

Yang Y, Zhou Y, Liu J, Zhao Y, Lu H, Xu L, Xu B, Leung H (2016) Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In: International symposium on foundations of software engineering, pp 157–168

Yang X, Lo D, Xia X, Sun J (2017) Tlel: a two-layer ensemble learning approach for just-in-time defect prediction. Inf Softw Technol 87:206–220

Yin R (2009) Case study research: design and methods, 4th edn (Applied social research methods, vol 5). SAGE Publications, Inc

Zeiler MD (2012) Adadelta: an adaptive learning rate method. arXiv:1212.5701

Zeller A (2009) How failures come to be. In: Why programs fail. 2nd edn. Morgan Kaufmann, pp 1–23