Noname manuscript No.
(will be inserted by the editor)

On the Fault Proneness of SonarQube Technical Debt Violations: A comparison of eight Machine Learning Techniques

Valentina Lenarduzzi · Francesco Lomio · Davide Taibi[*] · Heikki Huttunen

the date of receipt and acceptance should be inserted later

Abstract *Background.* The popularity of tools for analyzing Technical Debt, and particularly that of SonarQube, is increasing rapidly. SonarQube proposes a set of coding rules, which represent something wrong in the code that will soon be reflected in a fault or will increase maintenance effort. However, while the management of some companies is encouraging developers not to violate these rules in the first place and to produce code below a certain technical debt threshold, developers are skeptical of their importance.
Objective. In order to understand which SonarQube violations are actually fault-prone and to analyze the accuracy of the fault-prediction model, we designed and conducted an empirical study on 21 well-known mature open-source projects.
Method. We applied the SZZ algorithm to label the fault-inducing commits. We compared the classification power of eight Machine Learning models (Logistic Regression, Decision Tree, Random Forest, Extremely Randomized Trees, AdaBoost, Gradient Boosting, XGBoost) to obtain a set of violations that are correlated with fault-inducing commits. Finally, we calculated the percentage of violations introduced in the fault-inducing commit and removed in the fault-fixing commit, so as to reduce the risk of spurious correlations.
Result. Among the 202 violations defined for Java by SonarQube, only 26 have a relatively low fault-proneness. Moreover, violations classified as "bugs" by SonarQube hardly never become a failure. Consequently, the accuracy of the fault-prediction power proposed by SonarQube is extremely low.
Conclusion. The rules applied by SonarQube for calculating technical debt should be thoroughly investigated and their harmfulness needs to be further confirmed. Therefore, companies should carefully consider which rules they really need to apply, especially if their goal is to reduce fault-proneness.

[*] Corresponding author.

Valentina Lenarduzzi, Francesco Lomio, Davide Taibi and Heikki Huttunen
Tampere University
E-mail: valentina.lenarduzzi@tuni.fi
E-mail: francesco.lomio@tuni.fi E-mail: davide.taibi@tuni.fi E-mail: heikki.huttunen@tuni.fi

Keywords Technical Debt, SonarQube, coding style, code smells, architectural smells, static analysis, machine learning

1 Introduction

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

Although SonarQube recommends customizing the out-of-the-box set of rules (named ”sonar way”)[4], practitioners are reluctant to customize it and commonly rely on the ”sonar way” standard rule-set [43]. Querying the SonarQube public instance APIs [5], we can see that more than 98% of the public projects (14,732 projects up to 14,957) uses the ”sonar way” rule set. However, even if developers are not sure about the usefulness of the rules [43], [12], they do pay attention to different rules categories and priorities and remove violations related to rules with high severity [43] in order to avoid the possible risk of faults [12]. Therefore, investing effort for tasks that do not bring the expected benefits.

Several studies have analyzed the impact of code smells [16] on faults [41], [45], [25][34]. At best, only a limited number of studies have considered SonarQube rule violations [14], [26], but they did not investigate the impact of the SonarQube violations considered as ”bugs” on faults.

The goal of this work is twofold:

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

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

We designed this study as a case study and analyzed 21 randomly selected mature Java projects from the Apache Software Foundation. A controlled experiment with practitioner as participants, would have allowed us to evaluate our hypotheses more accurately [41], [45], [25]. However, controlled experiments are very expensive and can hardly deal with multivariate problems (e.g., they can

[1] https://www.sonarqube.org

[2] https://sonarcloud.io/explore/projects

[3] SonarQube Rules: https://docs.sonarqube.org/display/SONAR/Rules Last Access:May 2018

[4] SonarQube Quality Profiles: https://docs.sonarqube.org/display/SONAR/Quality+Profiles Last Access:May 2018

[5] https://docs.sonarqube.org/display/DEV/API+Basics

observe a limited number of smell types [41], [46], or a large number of subjects (e.g., [41], [45]), or several maintenance activities (e.g., [41], [26]). We first analyzed all the commits of the projects with SonarQube. Then, we identified and labeled the commits responsible for the introduction of a fault with the SZZ algorithm [27]. Finally, we analyzed the fault proneness of the sq-violations by comparing the accuracy of the commonly used Logistic Regression [9] with seven Machine Learning algorithms (Decision Trees [6], Random Forest [5], Bagging [4], Extra Trees [20], Ada Boost [17], Gradient Boost [18], XG Boost [7]).

Results show that only a limited number of sq-violations can be considered fault-prone. Moreover, the introduction of sq-violations considered as "bug" by SonarQube hardly ever resulted in a fault.

The remainder of this paper is structured as follows. In Section 2.1 we introduce SonarQube and the sq-violations adopted in this work. In Section 2 we present the background of this work, introducing the SonarQube violations and the different Machine Learning algorithms applied in this work. In Section 3, we describe the case study design. Section 4 presents respectively the obtained results. Section 5 identifies Threats to validity while Section 6 describes Related Works. Finally, conclusions are drawn in Section 7.

2 Background

2.1 SonarQube

SonarQube is one of the most common Open Source static code analysis tools. SonarQube is provided as a service from the sonarcloud.io platform or can be downloaded and executed on a private server.

SonerQube 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. Moreover, it defines a set of thresholds ("quality gates") for each metric and rule. In case the analyzed source code violates a coding rule or if a metric is outside a predefined threshold (also named "gate"), SonarQube generates an "issue". The time needed to remove these issues (remediation effort) is used to calculate the remediation cost and the technical debt. SonarQube includes Reliability, Maintainability and Security rules. Moreover, SonarQube claims that zero false positives are expected from the Reliability and Maintainability rules while there could be some false positives [3].

Reliability rules, also named "bugs" create issues (code violations) that "represents something wrong in the code" and that will soon be reflected in a bug. "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 et al [16] but to a different set of rules. Fowler et al [16] consider code smells as "surface indication that usually corresponds to a deeper problem in the system" but they can be indicators of different problems (bugs, maintenance effort, code readability, ...) while rules classified by SonarQube as "Code Smells" are only referred to maintenance issues. Moreover, only four of the 22 smells proposed my Fowler et al are included in the rules classified as "Code Smells" by SonarQube (Duplicated Code, Long Method, Large Class, Long Parameter List).

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

SonarQube also classifies the rules into five *severity* levels [6] :

- *BLOCKER*: "Bug with a high probability to impact the behavior of the application in production: memory leak, unclosed JDBC connection." SonarQube recommends to immediately review this issue
- *CRITICAL*: "Either a bug with a low probability to impact the behavior of the application in production or an issue which represents a security flaw: empty catch block, SQL injection" SonarQube recommends to immediately review this issue
- *MAJOR*: "Quality flaw which can highly impact the developer productivity: uncovered piece of code, duplicated blocks, unused parameters"
- *MINOR*: "Quality flaw which can slightly impact the developer productivity: lines should not be too long, switch statements should have at least 3 cases, ..."
- *INFO*: "Neither a bug nor a quality flaw, just a finding."

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

SonarQube includes more than 200 rules for Java. In Appendix, Table 6 lists the 90 most common violations present in our dataset. Column "*squid*" represents the original rule-id (SonarQube ID) defined by SonarQube. We did not rename it, to ease the replicability of this work. In the remainder of this work, we will refer to the different sq-violations with their id (squid).

The complete list of violations can be found in the file "SonarQube-rules.xsls" in the online raw data (Section 3.5).

2.2 Machine Learning Techniques

In this Section, we describe the Machine Learning techniques adopted in this work to predict the fault-proneness of sq-violations. Due to the nature of the task, all the models used for this work were used for classification. We compared eight machine learning models. Among these, we used a generalized linear model: Logistic Regression [9], one tree based classifier: Decision Tree [6], and 6 *ensemble classifiers*: Bagging [4], Random Forest [5], Extremely Randomized Trees [20], AdaBoost [17], Gradient Boosting [18] and XGBoost [7], an optimized implementation of Gradient Boosting.

2.2.1 Logistic Regression

One of the most used algorithms in Machine Learning is *Logistic Regression* [9]. Contrary to the linear regression, which is used to predict a numerical value, Logistic Regression is used for predicting the category of a sample. Particularly, a binary Logistic Regression model is used to estimate the probability of a binary result (0 or 1) given a set of independent variables. Once the probabilities are known, these can be used to classify the inputs in one of the two classes, based on their probability to belong to either of the two.

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

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

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

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

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

2.2.2 Decision Tree

The second model used is a *decision tree* classifier [6]. This model utilizes a decision tree to return an output given a series of input variables. Its tree structures is characterized by a *root node* and multiple *internal nodes*, which are represented by the input variable, and *leaf*, corresponding to the output. The nodes are linked between one another through branches, representing a test. The output is given by the decision path taken.

A decision tree is structured as a if-then-else diagram: in this structure, given the value of the variable in the root node, it can lead to subsequent nodes through branches following the result of a test. This process is iterated for all the input variables (one for each node) until it reaches the output, represented by the leaves of the tree.

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

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

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

2.2.3 Random Forest

To overcome the problems related to overfitting linked to the decision tree, we also tested a *Random Forest* model [5]. This is the first of the ensemble methods presented before. The term ensemble indicates that these models use a set of simpler models to solve the task assigned. In this case, Random Forest uses an ensemble of decision trees.

An arbitrary number of decision trees is generated considering a randomly chosen subset of the samples of the original dataset [4]. This subset is created with replacement, hence a sample can appear multiple times. Moreover, in order to reduce the correlation between the individual decision trees a random subset of the features of the original dataset. In this case, the subset is created without replacement. Each tree is therefore trained on its subset of the data, and it is able to give a prediction on new unseen data. The Random Forest classifier uses the results of all these trees and averages them to assign a label to the input.

By randomly generating multiple decision trees, and averaging their results, the Random Forest classifier is able to better generalize the data. Moreover, using the random subspace method, the individual trees are not correlated between one another. This is particularly important when dealing with a dataset with many features, as the probability of them being correlated between each other increases.

2.2.4 Bagging

Beside the Random Forest classifier, we decided to use also the more classical implementation of Bagging [4]. Exactly like the Random Forest model, the Bagging classifier is applied to an arbitrary number of decision tree constructed choosing a subset of the samples of the original dataset.

The difference with the Random Forest classifier is in the way in which the split point is decided: while in the Random Forest algorithm the splitting point is decided base on a random subset of the variables, the Bagging algorithm is allowed to look at the full set of variable to find the point which minimize the error. This translates in structural similarities between the trees which don't resolve the overfitting problem related to the single decision tree.

This model was included as a mean of comparison with newer and better performing models.

2.2.5 Extremely Randomized Trees

The *extremely randomized trees* (ExtraTrees) model [20], provides a further randomization degree to the Random Forest. For the Random Forest model, the individual trees are created by randomly choosing subsets of the dataset features. In the ExtraTrees model the way each node in the individual decision trees are split is also randomized. Instead of using the metrics seen before to find the optimal split for each node (Gini impurity and Information gain), the cut-off choice for each node is completely randomized, and the resulting splitting rule is decided based on the best random split.

Due to its characteristics, especially related to the way the splits are made at the node level, the ExtraTrees model is less computationally expensive than the Random Forest model, while retaining a higher generalization capability compared to the single decision trees.

2.2.6 AdaBoost

AdaBoost [17] is another ensemble algorithm based on *boosting* [40]. This term indicates an algorithm capable of creating a strong classifier using an ensemble of weak classifiers, created sequentially.

In the AdaBoost algorithm, the individual decision trees are grown sequentially. Moreover, a weight is assigned to each sample of the training set. Initially, all the samples are assigned the same weight. The model trains the first tree in order to minimize the classification error, and after the training is over, it increases the weights to those samples in the training set which were misclassified. Moreover, it grows another tree and the whole model is trained again with the new weights. This whole process continues until a predefined number of trees has been generated or the accuracy of the model cannot be improved anymore.

Due to the many decision trees, as for the other ensemble algorithms, AdaBoost is less prone to overfitting and can, therefore, generalize better the data. Moreover, it automatically selects the most important features for the task it is trying to solve. However, it can be more susceptible to the presence of noise and outliers in the data.

2.2.7 Gradient Boosting

Similarly to the other ensemble methods, the *Gradient Boosting* algorithm [18] uses an ensemble of individual decision trees which are generated sequentially, like for the AdaBoost.

The Gradient Boosting trains at first only one decision tree and, after each iteration, grows a new tree in order to minimize the loss function. Similarly to the AdaBoost, the process stops when the predefined number of trees has been created or when the loss function no longer improves.

2.2.8 XGBoost

The last model used for this work is the XGBoost [7]. This model can be viewed as a better performing implementation of the Gradient Boosting algorithm, as it allows for faster computation and parallelization. For this reason it can yield better performance compared to the latter, and can be more easily scaled for the use with high dimensional data.

3 Case Study Design

We designed our empirical study as a case study based on the guidelines defined by Runeson and Höst. [39]. 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

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

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

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

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

3.2 Study Context

We selected projects for this study based on a ”criterion sampling”[37]. The selected projects must fulfill all the following criteria:

- Developed in Java
- Older than three years
- More than 500 commits
- More than 100 classes
- Using Github for source code versioning
- Projects reporting faults in Jira
- Projects where, in case of fault-fixing activities, developers report the fault-id in the commit message

Moreover, as recommended by Nagappan et al. [32], we also tried to maximize diversity and representativeness, considering a comparable number of projects with respect to project age (number of years from the project creation), size (number of LOC of the last version), and application type (eg. web server, library, IDE, ...).

Based on these criteria, we selected 21 Java projects from the Apache Software Foundation (ASF) repository[7]. The repository includes some of the most widely used software solutions. The available projects can be considered industrial and mature, due the strict review process required by the ASF. Moreover, the included projects have to keep on reviewing their code and follow a strict quality process[8]. All the selected projects are available in a Git repository, and track their issues with Jira[9].

Faults are commonly discovered after the code has already been committed. Therefore, in order to ensure that we would find the vast majority of faults related to commits, we analyzed the projects from their first commit until the end of 2015, considering all the faults raised until the end of March 2018. Therefore, we can ensure that the vast majority of the faults introduced in the commits should have been discovered after more than two years.

In Table 1, we report the list of the 21 projects we considered together with the number of analyzed commits, the project size (LOC) of the last analyzed commits, the number of faults identified in the selected commits, and the total number of sq-violations.

3.3 Data Collection

In this section, we describe the approach we adopted to collect the data. For each project reported in Table 1, we first ran SonarQube to collect the SQ-Violations and then we collected faults from Jira.

3.3.1 SonarQube Violations Detection

We cloned the 21 Git repositories. Then we analyzed the entire commit history of each repository by means of SonarQube, with the purpose of identifying the violations introduced.

[7] http://apache.org
[8] https://incubator.apache.org/policy/process.html
[9] https://issues.apache.org/jira/

Table 1 The selected projects

Project Name	**Analyzed commits**	**Last commit LOC**	**Faults**	**SonarQube Violations**
Ambari	9727	396775	3005	42348
Bcel	1255	75155	41	8420
Beanutils	1155	72137	64	5156
Cli	861	12045	59	37336
Codec	1644	34716	57	2002
Collections	2847	119208	103	11120
Configuration	2822	124892	153	5598
Dbcp	1564	32649	100	3600
Dbutils	620	15114	21	642
Deamon	886	3302	4	393
Digester	2132	43177	23	4945
FileUpload	898	10577	30	767
Io	1978	56010	110	4097
Jelly	1914	63840	45	5057
Jexl	1499	36652	58	34802
Jxpath	596	40360	43	4951
Net	2078	60049	160	41340
Ognl	608	35085	15	4945
Sshd	1175	139502	222	8282
Validator	1325	33127	63	2048
Vfs	1939	59948	129	3604
Sum	**39.518**	**1.464.320**	**4.505**	**231.453**

We identified code smells by applying the standard set of rules defined by SonarQube, using the SonarQube quality model called "Sonar Way".

To understand the contribution of the injection of a sq-violation in a commit instead of the contribution of all the violations present at commit time, we only considered the violations introduced in a specific commit, taking into account the exact location of the violation in each file.

3.3.2 Faults Collection

We extracted faults from Jira, considering issues tagged as "bug" with resolution=fixed and status=closed. All the projects in the ASF have to tag faults with the tag "bug" and must report the issue-id in the commit message. Therefore, we relied on this mechanism to map the commits where developers had removed faults.

Results were saved in a CSV file and are available in our replication package (see Section 3.5).

3.4 Data Analysis

To understand which sq-violations are more fault-prone (**RQ1**), we first labeled all the commits by reporting which commit induced a fault using the SZZ algorithm [27].

Then, we applied the techniques reported in Section 2.2, **together with the drop-column** mechanism [42], and we compared their accuracy.

Next, to understand whether sq-violations classified as "bugs" are more fault-prone than other sq-violations (**RQ2**), we compared their importance (from the most accurate technique) between them and all the other rules (”code smells” and ”vulnerabilities;”). Finally, we calculated the fault-prediction accuracy of the SonarQube model (**RQ3**).

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

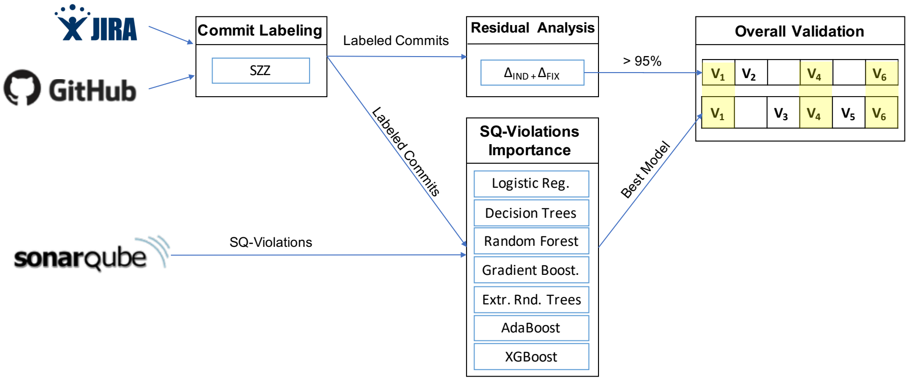
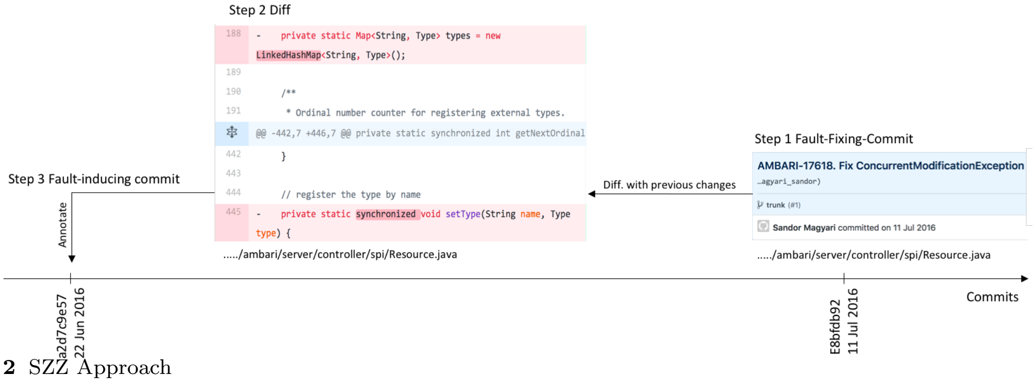


Fig. 1 The Data Analysis Process


Step 2 Diff
Step 3 Fault-inducing commit
Annotate
Diff. with previous changes
Step 1 Fault-Fixing-Commit
Commits
a2d7c9e57
22 Jun 2016
E8bfdb92
11 Jul 2016

```
-    private static Map<String, Type> types = new
LinkedHashMap<String, Type>();

     /**
      * Ordinal number counter for registering external types.
@@ -442,7 +446,7 @@ private static synchronized int getNextOrdinal
     }

     // register the type by name
-    private static synchronized void setType(String name, Type
type) {
```

...../ambari/server/controller/spi/Resource.java

AMBARI-17618. Fix ConcurrentModificationException
…agyari_sandor)
trunk (#1)
Sandor Magyari committed on 11 Jul 2016

...../ambari/server/controller/spi/Resource.java

Fig. 2 SZZ Approach

3.4.1 Commit Labeling: Fault-Inducing Commit Identification

In order to label the commits that induced a fault, we adopted the SZZ algorithm [27]. The SZZ algorithm is based on the annotation/blame features of commits in GitHub. The SZZ algorithm has been adopted in more than 200 empirical studies [19] [8].

As depicted in Figure 2, in Step 1 we identified bug-fixing commits, i.e., commits that are known to have fixed a bug. This step is ensured by the ASF foundation policies [10]. In ASF foundation, developers must report the fault-id in the commit messages whenever they fix it. Therefore, the commits filtering is ensured by the ASF policies.

[10] https://www.apache.org/foundation/policies/

Moreover, we also tagged each commit found with the information about the fault retrieved from the SZZ algorithm. Commits could be of one of three fault types: inducing (I), fixing (F), or not related to faults (N).

Step 2 shows bug fixes that involved updating a data structure in the file Resource.java in order to avoid some concurrency problems. For each identified bug-fixing change, SZZ analyzes the lines of code that were updated. For instance, Step 2 shows the differences between the commit #e8bfdb and its predecessor (#300a7e) in the Resource.java file. In this case, in order to fix the bug, the data structure at line 188 was changed. Therefore, SZZ identifies the changes that introduced bug AMBARI-17618 through the history of the source configuration management system (GitHub).

Step 3 shows commit *#a2d7c9* being flagged as a potential bug-introducing change by SZZ. We first labeled each commit as:

- Not a fault-Inducing commit
- Fault-Inducing Commit

3.4.2 Machine Learning Execution

In this Section we aim at comparing fault-proneness prediction power of SQ-Violations by applying the eight machine learning models described in Section 2.2 in order to confirm or reject the results obtained in the residual analysis.

Therefore we aim at predicting the fault proneness of a commit (labeled in Section 3.4.1) by means of the SQ-Violations introduced in the same commit. We used the SQ-Violations introduced in each commits as independent variables (a.k.a. predictors) to determine if a commit is fault-inducing (dependent variable).

After training the eight models described in Section 2.2, we performed a second analysis re-training the models using a *drop-column mechanism*[42]. This mechanism is a simplified variant of the exhaustive search [48], which iteratively tests every subset of features for their classification performance. The full exhaustive search is very time-consuming requiring 2^P train-evaluation steps for a P-dimensional feature space. Instead, we look only at dropping individual features one at a time, instead of all possible groups of features.

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

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

3.4.3 Accuracy Comparison

Apart from ranking the sq-violations by their importance, we first need to confirm the validity of the prediction model. If the predictions obtained from the ML techniques are not accurate, the feature ranking would also become questionable. To assess the prediction accuracy, we performed a 10-fold cross-validation, dividing the data in 11 parts, *i.e.,* we trained the models ten times always using 1/11 of the data as a testing fold. For each fold, we evaluated the classifiers by calculating a number of accuracy metrics (see below). The data related to each project have been split in 11 sequential parts, thus respecting the temporal order, and the proportion of data for each project. The models have been trained iteratively on group of data preceding the test set. The temporal order was also respected for the groups included in the training set: as an example, in fold 1 we used group 1 for training and group 2 for testing, in fold 2 groups 1 and 2 were used for training and group 3 for testing, and so on for the remaining folds.

As accuracy metrics, we first calculated precision and recall. However, as suggested by [38], 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 [38], the Matthews Correlation Coefficient (MCC) should be also considered to understand possible disagreement between actual values and predictions as it involves all the four quadrants of the contingency matrix.

From the contingency matrix, we retrieved the measure of *true negative rate* (TNR), which measures the percentage of negative sample correctly categorized as negative, *false positive rate* (FPR) which measures the percentage of negative sample misclassified as positive, and *false negative rate* (FNR), measuring the percentage of positive samples misclassified as negative. The measure of *true positive rate* is left out as equivalent to the recall. The way these measures were calculated can be found in Table 2.

Table 2 Accuracy Metrics formulae

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

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

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

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

3.4.4 Sq-violations Residual Analysis

The results from the previous ML techniques show a set of sq-violations related with fault-inducing commits. However, the relations obtained in the previous analysis do not imply causation between faults and sq-violations.

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

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

$$Residual = \mathit{\Delta}_{IND} + \mathit{\Delta}_{FIX}$$

where $\mathit{\Delta}_{IND}$ and $\mathit{\Delta}_{FIX}$ are calculated as:

$$\mathit{\Delta}_{IND} = \text{\#sq-violations introduced in the fault-inducing commit}$$

$$\mathit{\Delta}_{FIX} = \text{\#sq-violations removed in the fault-fixing commit}$$

Figure 3 shows the possible cases of introduced and removed violations while Figure 4 schematizes the residual analysis.

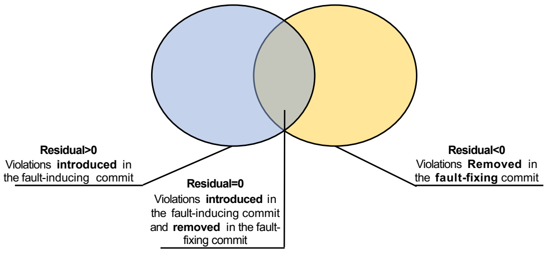

Residual>0
Violations **introduced** in the fault-inducing commit
Residual=0
Violations **introduced** in the fault-inducing commit and **removed** in the fault-fixing commit
Residual<0
Violations **Removed** in the **fault-fixing** commit

Fig. 3 Possible cases of introduced and removed violations

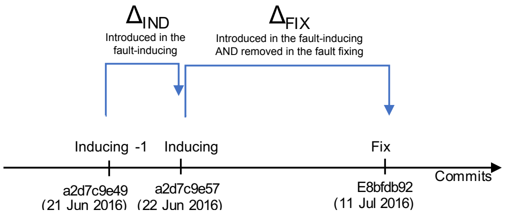


Fig. 4 Residuals Analysis

We calculated the residuals for each sq-violation V_i present in each fault. If $\mathit{\Delta}_{IND}$ was lower than zero, no sq-violations were introduced in the fault-inducing commit. Therefore, we tagged such a commit as not related to faults.

For each violation, the analysis of the residuals led us to two group of commits (Figure 3):

- *Residual¿0*: The sq-violations introduced in the fault-inducing commits were not removed during the fault-fixing.
- *Residual¡=0*: All the sq-violations introduced in the fault-inducing commits were removed during the fault-fixing. If Residual ¡0, other sq-violations of the same type already present in the code before the bug-inducing commit were also removed.

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

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

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

We calculated the percentage of residuals equal to zero as:

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

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

As a ,.mfinal step to analyze RQ1, we combined the results obtained from the best ML technique and from the residual analysis.

Therefore, if a violation has a high correlation with faults but the percentage of the residual is very low, we can discard it from our model, since it will be valuable only in a limited number of cases. As we cannot claim a cause-effect relationship without a controlled experiment, the results of the residual analysis are a step towards the identification of this relationship and the reduction of spurious correlations.

3.4.5 Relation Between sq-violations classified as "bugs" and faults (RQ3)

Since SonarQube considers every sq-violation tagged as a "bug" as "something wrong in the code that will soon be reflected in a bug", we also analyzed the accuracy of the model provided by SonarQube.

In order to answer our RQ3, we calculated the percentage of sq-violations classified as "bugs" that resulted in being highly fault-prone according to the previous analysis. To answer RQ3.1, we first labeled every commit, considering as "sq-faulty" every commit where a "bug" violation was introduced. Then we analyzed the accuracy of the model calculated the contingency matrix, precision and recall, and the Mathews correlation coefficient to compare the results with the commit labeled as fault-inducing by the SZZ.

3.5 Replicability

In order to allow the replication of our study, we published the raw data in the replication package [11].

4 Results

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

The data collection required three months of computation time on a Linux Ubuntu server with 15 cores and 64 GB RAM, while the data analysis required 6 days on the same machine.

All the 202 rules available in SonarQube for Java were found in the analyzed projects. For reasons of space limitations, we will refer to the sq-violations only with their SonarQube id number (SQUID). Table 6 report the description and type of the most important violations resulted in this work while the complete list of rules, together with their description is reported in the online replication package (file SonarQube-rules.xlsx). We should notice that in coloumn "Type" MA menas Major, Mi means Miniro, CR means Critical and BL means Blocker.

[11] Replication Package: http://www.taibi.it/raw-data/SQJ2019.zip - The link is temporary. Raw data will be moved

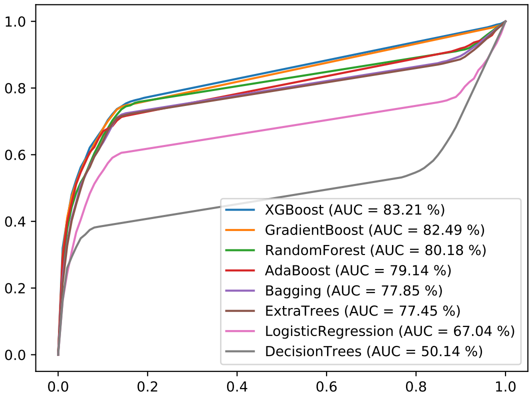


Fig. 5 ROC Curve (Average between 10-fold validation models)

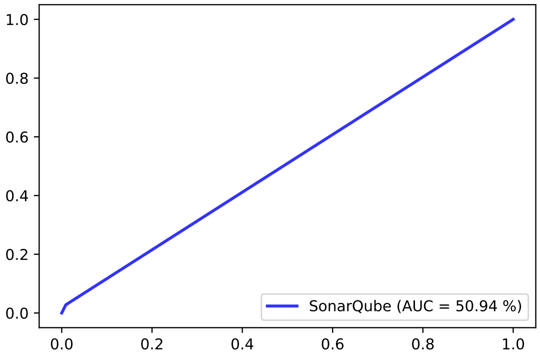


Fig. 6 ROC Curve of the fault-proneness of SonarQube violations classified as ”bugs”

4.1 RQ1: Which sq-violations are more fault-prone?

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

4.1.1 SQ-Violations Importance Analysis

As shown in Figure 5, XGBoost resulted in the most accurate model among the eight Machine Learning techniques applied to the dataset. The 10-fold cross-validation reported an average AUC of 0.83. Table 3 (column RQ1) reports average reliability measures for the eight models.

Despite the different measures have different strength and weaknesses (see Section 3.4.3), all the measures are consistently showing that XG Boost is the most accurate technique.

The ROC curves of all models are depicted in Table 3 while the reliability results of all the 10-folds models are available in the online replication package.

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

23 sq-violations have been ranked with an importance higher than zero by the XGBoost. In Table 5, we report the sq-violations with an importance higher or equal than 0.01 % (coloumn ”Intr. & Rem. (%)” reports the number of violations introduced in the fault-inducing commits AND removed in the fault-fixing commits). The remaining sq-violations are reported in the raw data for reasons of space. coloumn ”Intr. & Rem. (%)” means

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

to a permanent repository in case of acceptance

Table 3 Model Reliability

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

4.1.2 Model Accuracy Validation

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

Table 5 reports the descriptive statistics of residuals, together with the percentage residuals =0 (number of sq-violations introduced during fault-inducing commits and removed during fault-fixing commits).

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

Figure 7 compares the number of violations introduced in fault-inducing commits, and the number of violations removed in the fault-fixing commits.

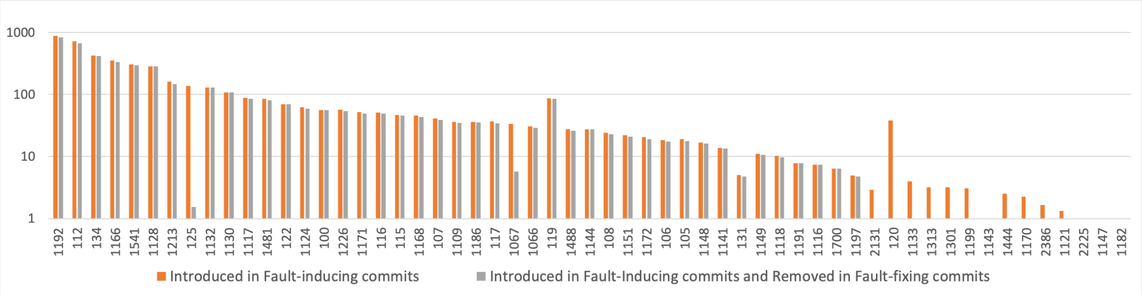


Fig. 7 Comparison of Violations introduced in fault-inducing commits and removed in fault-fixing commits

4.2 Manual Validation of the Results

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

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

We also noticed that violations related to duplicated code and empty statements (eg. "method should not be empty") always generated a fault (in the randomly selected cases). When committing an empty method (often containing only a "TODO" note), developers often forgot to implement it and then used it without realizing that the method did not return the expected value. An extensive application of unit test could definitely reduce this issue. However, we are aware that is is a very common practice in several projects. Moreover, sq-violations such as 1481 (unused private variable should be removed) and 1144(unused private methods should be removed) unexpectedly resulted to be an issue. In several cases, we discovered methods not used, but expected to be used in other methods, resulted in a fault. As example, if a method A calls another method B to compose a result message, not calling the method B results in the loss of the information provided by B.

4.3 RQ2: Are sq-violations classified as "bugs" by SonarQube more fault-prone than other rules?

Out of the 57 violations classified as "bugs" by SonarQube, only three (squid 1143, 1147, 1764) were considered fault-prone with a very low importance from the XGBoost and with residuals higher than 95%. However, rules classified as "code smells" were frequently violated in fault-inducing commits. However, is considering all the sq-violations, out of 40 the sq-violations that we identified as fault-prone, 37 are classified as "code smells" and one as security "vulnerability". When comparing severity with fault proneness of the sq-violations, only three sq-violations (squid 1147, 2068, 2178) were associated with the highest severity level (blocker). However, the fault-proneness of this rule is extremely low (importance <= 0.14%). Looking at the remaining violations, we can see that the severity level is not related to the importance reported by the XGBoost algorithm since the rules of different level of severity are distributed homogeneously across all importance levels.

4.4 RQ3: Fault prediction accuracy of the SonarQube model

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

All the accuracy measures (Table 3, column "RQ2") confirm the very low prediction power of "bug" violations. The vast majority of "bug" violation never become a filure. Results are also confirmed by the extremely low AUC (50.95%) also depicted in the ROC curve in Figure 6 and by the contingency matrix (Table 4). The results of the SonarQube model also confirm the results obtained in RQ2. Violations classified as "bugs" should be classified differently since they are hardly ever injected in fault-inducing commits.

Table 4 SonarQube Contingency Matrix (Prediction model based on sq-violations considered as "Bug" by SonarQube)

Predicted	**Actual**	
	IND	NOT IND
IND	32	342
NOT IND	1124	38020

Table 5 Summary of the SonarQube Violations Related to Faults (XGBoost Importance > 0.03%)

SonarQube				SZZ		Residuals					XG Boost	Res.> 95%
SQUID	Severity	Type	# Occ.	Intr. & Rem. (%)	Intr. in fault-ind	Mean	Max	Min	Stdev	RSS	Imp.	
S1192	CRITICAL	CS	1815	50,87	95,10	245,60	-861	2139	344,42	1726	0,66	✓
S1444	MINOR	CS	96	2,69	97,92	4,59	-7	73	10,34	94	0,62	✓
Useless Import Check	MAJOR	CS	1026	28,76	97,27	33,37	-170	351	61,58	998	0,41	✓
S00105	MINOR	CS	263	7,37	97,72	1,96	-13	32	10,22	257	0,41	✓
S1481	MINOR	CS	568	15,92	95,25	10,41	-6	83	14,60	541	0,39	✓
S1181	MAJOR	CS	200	5,61	97,00	8,87	0	88	13,43	194	0,31	✓
S00112	MAJOR	CS	1644	46,08	94,77	188,26	-279	1529	270,34	1558	0,29	
S1132	MINOR	CS	704	19,73	93,75	121,75	-170	694	134,91	660	0,24	
Hidden Field	MAJOR	CS	584	16,37	92,98	26,96	-12	143	29,42	543	0,23	
S134	CRITICAL	CS	1272	35,65	94,65	70,66	-66	567	88,07	1204	0,20	
S1068	MAJOR	CS	471	13,20	97,24	7,07	-39	77	13,17	458	0,17	✓
S1186	CRITICAL	CS	369	10,34	94,85	12,72	-7	64	12,77	350	0,17	
S106	MAJOR	CS	267	7,48	92,51	7,25	-172	106	38,13	247	0,16	
S00108	MAJOR	CS	302	8,46	94,04	18,54	-19	149	31,06	284	0,16	
Redundant Throws Declaration	MAJOR	CS	639	17,91	94,84	93,28	-265	593	114,34	606	0,15	
S1147	BLOCKER	BUG	35	0,98	100,00	29,23	4	66	24,86	35	0,14	✓
S00119	MAJOR	CS	24	0,67	0,00	26,00	26	26	0,00	0	0,14	✓
S1172	MAJOR	CS	272	7,62	98,90	8,13	-3	101	11,81	269	0,13	✓
S00115	MINOR	CS	419	11,74	95,47	31,81	-53	166	36,21	400	0,13	✓
S00116	MAJOR	CS	433	12,14	97,46	37,49	-1681	1881	377,83	422	0,13	✓
S1199	MINOR	CS	107	3,00	96,26	11,53	-223	259	51,37	103	0,12	✓
Empty Statement Usage	MAJOR	CS	168	4,71	94,05	2,88	-3	11	2,27	158	0,11	
S1160	MAJOR	CS	224	6,28	93,30	11,31	0	59	14,41	209	0,11	
S2178	BLOCKER	CS	82	2,30	97,56	1,37	-1	5	1,37	80	0,09	✓
S1764	MAJOR	BUG	90	2,52	100,00	0,21	0	1	0,41	90	0,06	✓
S00122	MAJOR	CS	507	14,21	97,24	45,43	-378	430	64,42	493	0,06	✓
S2068	BLOCKER	VULN.	135	3,78	93,33	8,67	-1	24	6,57	126	0,06	
S1141	MAJOR	CS	229	6,42	94,32	5,27	1	28	6,09	216	0,06	
Commented Out Code Line	MAJOR	CS	718	20,12	95,54	12,28	-209	346	88,90	686	0,05	✓
S1158	MINOR	CS	4	0,11	100,00	2,00	0	3	1,41	4	0,05	✓
Object Finalize Overriden	MAJOR	CS	8	0,22	0,00	1,00	1	1	0,00	0	0,04	✓
S1488	MINOR	CS	324	9,08	94,44	8,92	-5	50	9,84	306	0,04	
S1118	MAJOR	CS	195	5,47	96,92	4,60	-10	22	4,66	189	0,03	✓
S1185	MINOR	CS	94	2,63	96,81	2,29	-1	14	2,55	91	0,03	✓
S1168	MAJOR	CS	414	11,60	96,38	11,92	-75	128	20,79	399	0,03	✓
S1067	CRITICAL	CS	352	9,87	96,59	9,34	-19	59	9,61	340	0,02	✓
S1143	MAJOR	BUG	8	0,22	100,00	0,75	0	2	0,71	8	0,02	✓
S1312	MINOR	CS	467	13,09	95,29	20,67	-38	111	23,78	445	0,02	✓
S1171	MAJOR	CS	443	12,42	94,13	64,82	-13	422	69,48	417	0,02	
S2386	MINOR	CS	77	2,16	100,00	7,52	0	27	9,01	77	0,01	✓

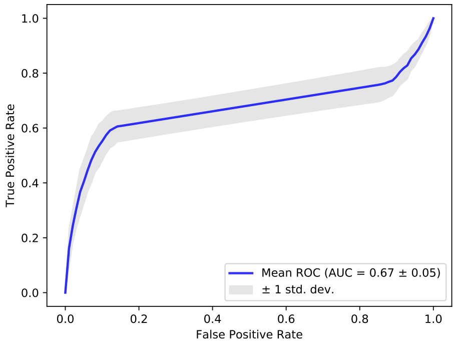


Fig. 8 LogisticRegression ROC Curve

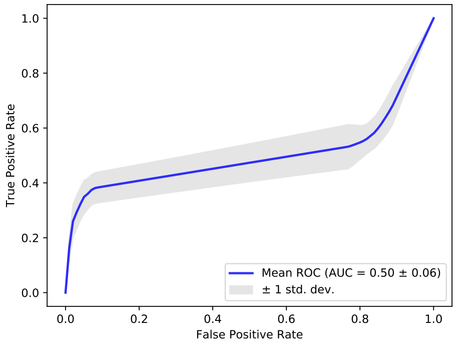


Fig. 9 DecisionTrees ROC Curve

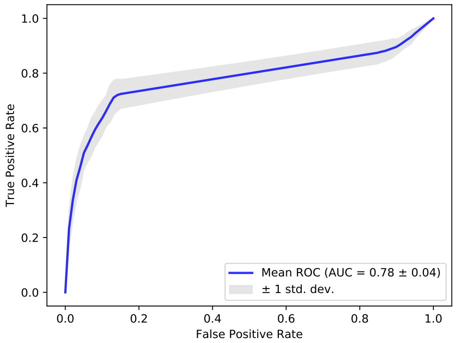


Fig. 10 Bagging ROC Curve

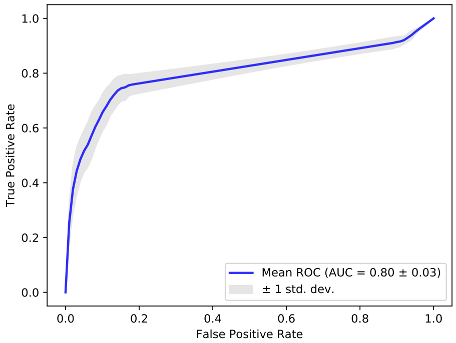


Fig. 11 RandomForest ROC Curve

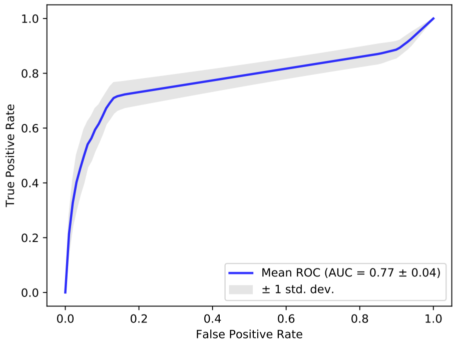


Fig. 12 ExtraTrees ROC Curve

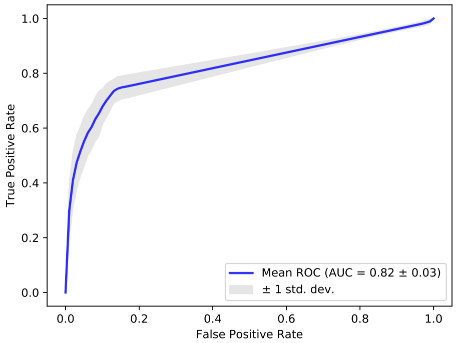


Fig. 13 Gradient Boost ROC Curve

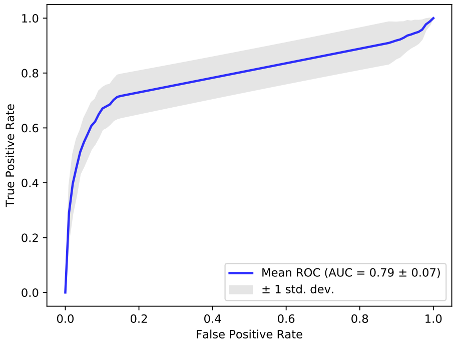


Fig. 14 AdaBoost ROC Curve

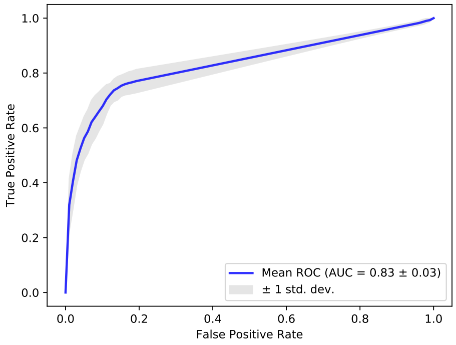


Fig. 15 XGBoost ROC Curve

Table 6 SonarQube Violation Names (90 most important)

Sq	Type	Name	Sq	Type	Name
100	MI	Method names should comply with a naming convention	1313	MI	IP addresses should not be hardcoded
105	MI	Tabulation characters should not be used	1481	MI	Unused local variables should be removed
106	MA	Standard outputs should not be used directly to log anything	1541	MA	Methods should not be too complex
108	MA	Nested blocks of code should not be left empty	1609	CR	@FunctionalInterface annotation should be used to flag Single Abstract Method interfaces
112	MA	Generic exceptions should never be thrown	1640	MI	Maps with keys that are enum values should be replaced with EnumMap
115	MI	”switch case” clauses should not have too many lines of code	1641	MI	Sets with elements that are enum values should be replaced with EnumSet
119	MA	Classes from ”sun.*” packages should not be used	1643	MI	Strings should not be concatenated using ’+’ in a loop
116	MA	Field names should comply with a naming convention	1698	MI	’==” and ”!=” should not be used when ”equals” is overridden
120	MA	Package names should comply with a naming convention	1700	MA	A field should not duplicate the name of its containing class
122	MA	Statements should be on separate lines	1764	MA	Identical expressions should not be used on both sides of a binary operator
124	MA	Track comments matching a regular expression	1850	MA	’instanceof” operators that always return ”true” or ”false” should be removed
125	MA	Sections of code should not be com- mented out	1854	MA	Dead stores should be removed
128	BL	Switch cases should end with an unconditional ”break” statement	1858	MI	’toString()” should never be called on a String object
131	MA	’switch” statements should end with ”default” clauses	1871	MA	Two branches in a conditional structure should not have exactly the same implementation
134	CR	Control flow statements if for while switch and try should not be nested too deeply	1943	MI	Classes and methods that rely on the default system encoding should not be used
1066	MA	Collapsible ”if” statements should be merged	1994	CRI	’for” loop increment clauses should modify the loops’ counters
1067	MA	Expressions should not be too complex	2047	MA	The names of methods with boolean return values should start with ”is” or ”has”
1075	MI	URIs should not be hardcoded	2070	CR	SHA-1 and Message-Digest hash algorithms should not be used
1109	MA	A close curly brace should be located at the beginning of a line	2077	BL	SQL binding mechanisms should be used
1116	MA	Empty statements should be removed	2096	BL	’main” should not ”throw” anything
1118	MA	Utility classes should not have public constructors	2112	MA	’URL.hashCode” and ”URL.equals” should be avoided
1124	MA	Modifiers should be declared in the cor- rect order	2129	MA	Constructors should not be used to instantiate ”String” and primitive-wrapper classes
1128	MA	Useless imports should be removed	2130	MA	Parsing should be used to convert ”Strings” to primitives
1130	MA	throws declara- tions should not be superfluous	2131	MA	Primitives should not be boxed just for ”String” conversion
1133	INFO	Deprecated code should be removed	2133	MA	Objects should not be created only to ”getClass”
1144	MA	Unused ”private” methods should be removed	2140	MI	Methods of ”Random” that return floating point values should not be used in random integer generation
1147	BL	Exit methods should not be called	2160	MI	Subclasses that add fields should override ”equals”
1148	MI	Throwable.printStackTrace(...) should not be called	2178	BL	Short-circuit logic should be used in boolean contexts
1149	MA	Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used	2184	MI	Math operands should be cast before assignment
1155	MI	Collection.isEmpty() should be used to test for emptiness	2185	MA	Silly math should not be performed
1157	MI	Case insensitive string comparisons should be made without intermediate upper or lower casing	2189	BL	Loops should not be infinite
1158	MI	Primitive wrappers should not be instantiated only for ”toString” or ”compareTo” calls	2197	BL	Modulus results should not be checked for direct equality
1161	MA	’@Override” should be used on overriding and implementing methods	2225	MA	”toString()” and ”clone()” methods should not return null
1163	CR	Exceptions should not be thrown in finally blocks	2232	MA	’ResultSet.isLast()” should not be used
1166	MA	Exception handlers should preserve the original exceptions	2250	MI	Collection methods with O(n) performance should be used carefully
1186	CR	Methods should not be empty	2273	MI	’wait”, ”notify” and ”notifyAll” should only be called when a lock is obviously held on an object
1170	MI	Public constants and fields initialized at declaration should be ”static final” rather than merely ”final”	2274	CR	’Object.wait(...)” and ”Condition.await(...)” should be called inside a ”while” loop
1171	MA	Only static class initializers should be used	2275	BL	Printf-style format strings should not lead to unexpected behavior at runtime
1172	MA	Unused method parameters should be removed	2276	BL	’wait(...)” should be used instead of ”Thread.sleep(...)” when a lock is held
1186	CR	Methods should not be empty	2278	BL	Neither DES (Data Encryption Standard) nor DESede (3DES) should be used

5 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 [47].

Construct Validity. As for construct validity, the results might be biased regarding the mapping between faults and commits. We relied on the ASF practice of tagging commits with the issue ID. However, in some cases, developers could have tagged a commit differently. Moreover, the results could also be biased due to detection errors of the code smell identification tool we adopted. Moreover, we analyzed commits until the end of 2015, considering all the faults raised until the end of March 2018. We expect that the vast majority of the faults should have been fixed. However, it could be possible that some of these faults were still not identified and fixed.

Internal Validity. Threats can be related to the causation between sq-violations and fault-fixing activities. As for the identification of the fault-inducing commits, we relied on the SZZ algorithm [27]. We are aware that in some cases, the SZZ algorithm might not have identified fault-inducing commits correctly because of the limitations of the line-based diff provided by git, and also because in some cases bugs can be fixed modifying code in other location than in the lines that induced them. Moreover, we are aware that the imbalanced data could have influenced the results (approximately 90% of the commits were non-fault-inducing). However, the application of solid machine learning techniques, commonly applied with imbalanced data could help to reduce this threat.

External Validity. We selected 21 projects from the Apache Software Foundation, which incubates only certain systems that follow specific and strict quality rules. Our case study was not based only on one application domain. This was avoided since we aimed to find general mathematical models for the prediction of the number of bugs in a system. Choosing only one or a very small number of application domains could have been an indication of the non-generality of our study, as only prediction models from the selected application domain would have been chosen. The selected projects stem from a very large set of application domains, ranging from external libraries, frameworks, and web utilities to large computational infrastructures. The application domain was not an important criterion for the selection of the projects to be analyzed, but in any case we tried to balance the selection and pick systems from as many contexts as possible.

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

6 Related Work

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

Falessi et al. [14] studied the distribution of 16 metrics and 106 sq-violations in an industrial project. They applied a *What-if* approach with the goal of investigating what could happen if a specific sq-violation would not have been introduced in the code and if the number of faulty classes decrease in case the violation is not introduced. They compared four ML techniques (Bagging, BayesNet, J48, and Logistic Regression) on the project and then they applied the same techniques on a modified version of the code where they manually removed sq-violations. Results showed that 20% of faults were avoidable if the code smells would have been removed.

Tollin et al. [26] used machine learning to predict the change-proneness of classes based on SonarQube violations and their evolution. They investigated if sq-violations introduced would led to an increase in the number of changes (code churns) in the next commits. The study was applied on two different industrial projects, written in C# and JavaScript. They compared the prediction accuracy of Decision Tres, Random Forest, and Naive Bayes. They reported that classes affected by more sq-violations have a higher change pronenenss. However they did not prioritize or classified the most change prone sq-violations.

Digkas et al. [11] studied weekly snapshots of 57 random-chosen Java open-source software projects by the Apache Software Foundation with the following requirements: the projects had to be written in Java programming language, had to have at least 100 classes, at least two years of history, at least 1000 commits and had to be still active in the year the project was conducted. Their goal was to find out how much technical debt was paid back over the course of the projects and what kind of issues were fixed. They considered sq-violations with severity marked as *Blocker, Critical and Major*. The results showed that only a small subset of all issue types was responsible for the largest percentage of technical debt repayment. Their results thus confirm our initial assumption that there is no need to fix all issues. Rather, by targeting particular violations, the development team can achieve higher benefits. However, their work does not consider how the issues actually related to faults.

Falessi and Reichel [13] developed an open-source tool called MIND, which reports the technical debt interest occurring due to violations of quality rules. Interest is measured by means of various metrics related to fault-proneness. MIND checks compliance of the code against the SonarQube quality rules and uses linear regression to estimate the defect-proneness of classes. The aim of MIND is to answer developers' questions like: is it worth to re-factor this piece of code? MIND was tested and evaluated on a project with millions of LOC. Again, the actual type of issue causing the defect was not considered.

Codabux and Williams [49] propose a predictive model to help prioritize technical debt. They extracted class-level metrics for defect- and change-prone classes using *Scitool Understanding* and *Jira Extracting Tool* from Apache Hive open-source project and determined significant independent variables for defect- and change-prone classes, respectively. Then they used a Bayesian approach to build a prediction model to determine the ”technical debt proneness” of each class. However, their model requires the identification of ”technical debt items”, which requires manual input. These items are ultimately ranked and given a risk probability by the predictive framework.

Other works investigated the fault-proneness of different type of code smells such as [16] or MVC smells [1] or testing smells [3] or android smells [28].

Fontana et al. [15] developed the JCodeOdor tool, which exploits the Eclipse JDT to analyze Java systems. JCodeOdor uses a set of metrics to calculate the presence of a set of code smells and calculates an Intensity index. JCodeOdor calculated the Intensity Index for six different code smells: *God Class, Data Class, Brain Method, Shotgun Surgery, Dispersed Coupling, Message Chain* in 74 different software systems of the Qualitas Corpus. The index is a value between zero and ten, where 10 is the most critical value. Their results show that only 10 % of the code smells found had high Intensity, which can be useful for prioritizing which segments of the code to inspect. Their work, however, does not relate the smells to actual faults.

Vidal et al. present the tool JSpiRIT [44] (Java Smart Identification of Refactoring opportunI-Ties) for detecting code smells from Java code and prioritizing technical debt based on the smells. Moreover, JSpiRIT allows developers to configure the tool to ranking smells according to different criteria. In the study, an usage of tool is exposed using as prioritization criteria modifiability vs code smells, the importance of smells type and likely the smelled code will be changed in the future. However, JSpiRIT currently only supports ten code smells (Brain Class, Brain Method, Data Class, Dispersed Coupling, Feature Envy, God Class, Intensive Coupling, Refused Parent Bequest, Shotgun Surgery, Tradition Breaker) and the prioritization strategy must be defined by the developers instantiating the tool.

Other approaches based on support vector machines were applied on two open-source programs [30] and also considering practitioners' feedback [29] to three systems in order to detect four anti-patterns (Blob, Functional Decomposition, Spaghetti Code and Swiss Army Knife).

Regarding other code quality rules detection, 7 different machine learning approaches (Random Forest, Naive Bayes, Logistic regression, IBl, IBk, VFI and J48) [31] were successfully applied on 6 code smells (Lazy Class, Feature Envy, Middle Man Message Chains, Long Method, Long Parameter Lists and Switch Statement) and 27 software metrics (including Basic, Class Employment, Complexity, Diagrams, Inheritance, MOOD) as independent variables.

Code smells detection was also investigated from the point of view of how the severity of code smells can be classified through machined learning models [2] such as J48, JRip, Random Forest, Naive Bayes, SMO and LibSVM with best agreement to detection 3 code smells (God Class, Large Class and Long Parameter List).

Panichella et al. [35], characterized defect prediction models and machine learning methods that investigated fault proneness of code entities in order to evaluate the prediction power whether they identified different defect-prone classes. They considered six machine learning techniques (such as linear regression, logistic regression, or classification trees) involving 10 open-source Java projects. The results did not show which classifier has superior prediction compared with the other ones. They suggested to complement each other during the prediction.

Also Di Nucci et al. [33], compared different predict software defect techniques in order to emphasize the role of human-related factors in the bugs introduction. They evaluated four techniques on 26 systems. The results confirmed that combining different techniques, the prediction accuracy increases.

Since the results provided by tools can be subjective and related with the detection process, also Di Nucci et al. [10], applied Machine-Learning (ML) techniques for code smell detection. They analyzed the metric distribution of smelly and non-smelly code elements and they found some open issues to be solved related to detecting code smells using machine learning models.

Herbold et al. [24], performed a benchmark for the comparison of existing Cross-Project Defect Prediction (CPDP) approaches for software projects quality assurance. They evaluated 24 different approaches proposed between 2008 and 2015 analyzing their performance on five different data sets. The results showed that CPDP approaches still have some open issues regarding performances and need to be more empirically validated.

Hassan [23], proposed a set of complexity metrics based on the code change process, validating it with a case study based on history data on six large open source projects. They found that the proposed change complexity metrics are better fault predictors, comparing these with other well-known historical fault predictors such as prior modifications or prior faults.

Habib and Pradel [22], investigated the bud prediction efficiency of three bug detectors tools on 15 Java projects with 594 well-known bugs. They applied a novel methodology combining automatic analysis with manual validation of detected bugs. The results showed that static bug detectors find a non-negligible amount of bugs missing he large majority of the well-known bugs.

Pascarella et al. [36] applied method-level bug prediction proposed by Ginger et al. [21] on different systems. The results showed promising performance of the method-level bug prediction models.

At the best of our knowledge, this our work is the first study that ranks the sq-violations based on their fault proneness.

7 Discussion and Conclusion

In this work, we performed a large case study with the goal of analyzing the fault-proneness of SonarQube violations and the prediction power of the default SonarQube quality model used by SonarQube to calculate the technical debt of Java code. The SonarQube model is composed of 202 rules that, when violated, generate issues that should be addressed (removed from the code) by the developers.

Moreover, among these rules, SonarQube classifies 57 as ”bugs”, claiming that they are the root causes of faults in 100% of cases. In order to validate this statement, we analyzed the presence of all 202 SonarQube detected violations in the complete project history of 21 well-known and active open-source projects from the Apache Software Foundation, analyzing all the commits from the beginning of the projects until the end of 2015. We identified and labeled the fault-inducing commits, mapping the faults reported in the Jira issue tracker by means of the SZZ algorithm.

The study considered 39,518 commits, including more than 38 billion lines of code, 1.4 million violations, and 4,505 faults mapped to the commits.

To understand which sq-violations have the highest fault proneness, we first applied eight Machine Learning approach to identify the sq-violations that are common in commits labeled as fault-inducing. As for the application of the different Machine Learning approaches, we can see an important difference in their accuracy, with a difference of more than 53% from the worst model (Decision Trees AUC=47.3%±3%) and the best model (XGBoost AUC=83.32%±10%). This confirms also what reported in section 2.2: ensemble models, like the XGBoost, can generalize better the data compared to Decision Trees, hence it results to be more scalable. The use of many *weak* classifiers, yields an overall better accuracy, as it can be seen by the fact that the *boosting* algorithms (AdaBoost, GradientBoost, and XGBoost) are the best performers for this classification task, followed shortly by the Random Forest classifier and the ExtraTrees.

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

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

When we looked at which of the sq-violations resulted fault-prone from the previous step, only four of them are also classified as (”bugs”) by SonarQube. The remaining fault-prone sq-violations are mainly classified as ”code smells” (SonarQube claims that ”code smells” increase maintenance effort but do not create faults). The analysis of the accuracy of the fault prediction power of the

SonarQube model based on "bugs" showed an extremely low fitness, with an AUC of 50.94%, confirming that violations classified as "bugs" almost never resulted in a fault.

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

Lesson 1: SonarQube violations are not good fault-proneness predictors if considered individually, but can be good predictors if considered together.

Lesson 2: SonarQube violations classified as "bug" does not seem to be the root cause of faults.

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

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

The lessons learned confirm our initial hypothesis about the fault-proneness of the SonarQube violations. However, we are not claiming that SonarQube violations are not harmful in general. We are aware that some violations could be more prone to changes [14], decrease code readability or increase the maintenance effort.

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

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

Our future work will include replication of this work on the fault-proneness of SonarQube violations and the correlations between SonarQube severity levels and the importance of the rules. We are currently working on an analysis of all the commits of the Java projects in the Apache Software Foundation repository.

As for our future research agenda, we will focus on the definition of recommender able to alert developers about the presence of potential problematic classes based on their (evolution of) change- and fault-proneness and rank them based on the potential benefits provided by their removal.

References

1. Aniche, M., Bavota, G., Treude, C., Gerosa, M.A., Van Deursen, A.: Code smells for model-view-controller architectures. Empirical Software Engineering **23**(4), 2121–2157 (2018). DOI 10.1007/s10664-017-9540-2. URL https://doi.org/10.1007/s10664-017-9540-2
2. Arcelli Fontana, F., Zanoni, M.: Code smell severity classification using machine learning techniques. Know.-Based Syst. **128**(C), 43–58 (2017). DOI 10.1016/j.knosys.2017.04.014. URL https://doi.org/10.1016/j.knosys.2017.04.014
3. Bavota, G., Qusef, A., Oliveto, R., Lucia, A., Binkley, D.: Are test smells really harmful? an empirical study. Empirical Softw. Engg. **20**(4), 1052–1094 (2015). DOI 10.1007/s10664-014-9313-0. URL http://dx.doi.org/10.1007/s10664-014-9313-0
4. Breiman, L.: Bagging predictors. Machine Learning **24**(2), 123–140 (1996). DOI 10.1007/BF00058655. URL http://link.springer.com/10.1007/BF00058655
5. Breiman, L.: Random forests. Machine learning **45**(1), 5–32 (2001)
6. Breiman, L., Friedman, J., Stone, C.J., Olshen, R.: Classification and regression trees Regression trees (1984)
7. Chen, T., Guestrin, C.: XGBoost: A Scalable Tree Boosting System. pp. 785–794. ACM Press, New York, New York, USA (2016). DOI 10.1145/2939672.2939785. URL http://dl.acm.org/citation.cfm?doid=2939672.2939785

8. da Costa, D.A., McIntosh, S., Shang, W., Kulesza, U., Coelho, R., Hassan, A.E.: A framework for evaluating the results of the szz approach for identifying bug-introducing changes. IEEE Transactions on Software Engineering **43**(7), 641–657 (2017)
9. Cox, D.R.: The regression analysis of binary sequences. Journal of the Royal Statistical Society. Series B (Methodological) **20**(2), 215–242 (1958). URL `http://www.jstor.org/stable/2983890`
10. Di Nucci, D., Palomba, F., Tamburri, D., Serebrenik, A., De Lucia, A.: Detecting code smells using machine learning techniques: Are we there yet? (2018). DOI 10.1109/SANER.2018.8330266
11. Digkas, G., Lungu, M., Avgeriou, P., Chatzigeorgiou, A., Ampatzoglou, A.: How do developers fix issues and pay back technical debt in the apache ecosystem? pp. 153–163 (2018)
12. D.Taibi, A.Janes, Lenarduzzi, V.: How developers perceive smells in source code: A replicated study. Information and Software Technology **92**, 223 – 235 (2017)
13. Falessi, D., Reichel, A.: Towards an open-source tool for measuring and visualizing the interest of technical debt. pp. 1–8 (2015)
14. Falessi, D., Russo, B., Mullen, K.: What if i had no smells? 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) pp. 78–84 (2017). DOI 10.1109/ESEM.2017.14
15. Fontana, F.A., Ferme, V., Zanoni, M., Roveda, R.: Towards a prioritization of code debt: A code smell intensity index. pp. 16–24 (2015)
16. Fowler, M., Beck, K.: Refactoring: Improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc. (1999)
17. Freund, Y., Schapire, R.E.: A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. Journal of Computer and System Sciences **55**(1), 119–139 (1997). DOI 10.1006/JCSS.1997.1504. URL `https://www.sciencedirect.com/science/article/pii/S002200009791504X?via%3Dihub`
18. Friedman, J.H.: Greedy Function Approximation: A Gradient Boosting Machine. DOI 10.2307/2699986. URL `https://www.jstor.org/stable/2699986`
19. G. Rodriguez-Perez, G.R., González-Barahona, J.M.: Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm. Information and Software Technology **99**, 164 – 176 (2018)
20. Geurts, P., Ernst, D., Wehenkel, L.: Extremely randomized trees. Machine Learning **63**(1), 3–42 (2006). DOI 10.1007/s10994-006-6226-1. URL `http://link.springer.com/10.1007/s10994-006-6226-1`
21. Giger, E., D'Ambros, M., Pinzger, M., Gall, H.C.: Method-level bug prediction. In: Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 171–180 (2012). DOI 10.1145/2372251.2372285
22. Habib, A., Pradel, M.: How many of all bugs do we find? a study of static bug detectors. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, pp. 317–328. ACM, New York, NY, USA (2018). DOI 10.1145/3238147.3238213. URL `http://doi.acm.org/10.1145/3238147.3238213`
23. Hassan, A.E.: Predicting faults using the complexity of code changes. In: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pp. 78–88. IEEE Computer Society, Washington, DC, USA (2009). DOI 10.1109/ICSE.2009.5070510. URL `http://dx.doi.org/10.1109/ICSE.2009.5070510`
24. Herbold, S., Trautsch, A., Grabowski, J.: A comparative study to benchmark cross-project defect prediction approaches. IEEE Transactions on Software Engineering **44**(9), 811–833 (2018). DOI 10.1109/TSE.2017.2724538
25. I. Deligiannis, I.S., L. Angelis, M.R., Shepperd, M.: A controlled experiment investigation of an object-oriented design heuristic for maintainability. Journal of Systems and Software **72**(2), 129 – 143 (2004)
26. I. Tollin, F.A.F., Zanoni, M., Roveda, R.: Change prediction through coding rules violations. EASE'17, pp. 61–64. ACM, New York, NY, USA (2017)
27. J. Śliwerski T. Zimmermann, a.A.Z.: When do changes induce fixes? MSR '05, pp. 1–5. ACM, New York, NY, USA (2005)
28. Kessentini, M., Ouni, A.: Detecting android smells using multi-objective genetic programming. pp. 122–132 (2017). DOI 10.1109/MOBILESoft.2017.29
29. Maiga, A., Ali, N., Bhattacharya, N., Sabané, A., Guéhéneuc, Y.G., Aimeur, E.: Smurf: A svm-based incremental anti-pattern detection approach. pp. 466–475 (2012). DOI 10.1109/WCRE.2012.56
30. Maiga, A., Ali, N., Bhattacharya, N., Sabané, A., Guéhéneuc, Y.G., Antoniol, G., Aïmeur, E.: Support vector machines for anti-pattern detection. ASE 2012, pp. 278–281. ACM, New York, NY, USA (2012). DOI 10.1145/2351676.2351723. URL `http://doi.acm.org/10.1145/2351676.2351723`
31. Maneerat, N., Muenchaisri, P.: Bad-smell prediction from software design model using machine learning techniques. pp. 331–336 (2011). DOI 10.1109/JCSSE.2011.5930143
32. Nagappan, M., Zimmermann, T., Bird, C.: Diversity in software engineering research. ESEC/FSE 2013, pp. 466–476. ACM, New York, NY, USA (2013). DOI 10.1145/2491411.2491415. URL `http://doi.acm.org/10.1145/2491411.2491415`

33. Nucci, D.D., Palomba, F., Rosa, G.D., Bavota, G., Oliveto, R., Lucia, A.D.: A developer centered bug prediction model. IEEE Transactions on Software Engineering **44**(1), 5–24 (2018). DOI 10.1109/TSE.2017.2659747
34. Palomba, F., Zanoni, M., Fontana, F.A., De Lucia, A., Oliveto, R.: Toward a smell-aware bug prediction model. IEEE Transactions on Software Engineering **45**(2), 194–218 (2019). DOI 10.1109/TSE.2017.2770122
35. Panichella, A., Oliveto, R., Lucia, A.D.: Cross-project defect prediction models: L'union fait la force. In: 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), pp. 164–173 (2014). DOI 10.1109/CSMR-WCRE.2014.6747166
36. Pascarella, L., Palomba, F., Bacchelli, A.: Re-evaluating method-level bug prediction. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 592–601 (2018). DOI 10.1109/SANER.2018.8330264
37. Patton, M.: Qualitative Evaluation and Research Methods. Sage, Newbury Park (2002)
38. Powers, D.M.W.: Evaluation: From precision, recall and f-measure to roc., informedness, markedness & correlation. Journal of Machine Learning Technologies **2**(1), 37–63 (2011)
39. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empirical Softw. Engg. **14**(2), 131–164 (2009)
40. Schapire, R.E.: The Strength of Weak Learnability. Machine Learning **5**(2), 197–227 (1990). DOI 10.1023/A:1022648800760. URL `http://link.springer.com/10.1023/A:1022648800760`
41. Sjoberg, D.I.K., Yamashita, A., Anda, B.C.D., Mockus, A., Dyba, T.: Quantifying the effect of code smells on maintenance effort. IEEE Transactions on Software Engineering **39**(8), 1144–1156 (2013)
42. Terence, P., Kerem, T., Christopher, C., Jeremy, H.: Beware default random forest importances. `http://explained.ai/rf-importance/index.html`. Accessed: 2018-07-20
43. Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Zaidman, A., Gall, H.C.: Context is king: The developer perspective on the usage of static analysis tools. 25th International Conference on Software Analysis, Evolution and Reengineering (SANER) (2018)
44. Vidal, S., Vazquez, H., Diaz-Pace, J.A., Marcos, C., Garcia, A., Oizumi, W.: Jspirit: a flexible tool for the analysis of code smells. pp. 1–6 (2015)
45. Yamashita, A.: Assessing the capability of code smells to explain maintenance problems: An empirical study combining quantitative and qualitative data. Empirical Softw. Engg. **19**(4), 1111–1143 (2014)
46. Yamashita, A., Moonen, L.: To what extent can maintenance problems be predicted by code smell detection? - an empirical study. Inf. Softw. Technol. **55**(12), 2223–2242 (2013)
47. Yin, R.: Case Study Research: Design and Methods, 4th Edition (Applied Social Research Methods, Vol. 5), 4th edn. SAGE Publications, Inc (2009)
48. Yoon, H., Yang, K., Shahabi, C.: Feature subset selection and feature ranking for multivariate time series. IEEE transactions on knowledge and data engineering **17**(9), 1186–1198 (2005)
49. Z. Codabux, B.W.: Technical debt prioritization using predictive analytics. ICSE '16, pp. 704–706. ACM, New York, NY, USA (2016)