A Preliminary Study on the Adequacy of Static Analysis Warnings with Respect to Code Smell Prediction

Savanna Lujan,[1*] Fabiano Pecorelli,[2*] Fabio Palomba,[2] Andrea De Lucia,[2] Valentina Lenarduzzi[3]
[1]Tampere University, Finland — [2]SeSa Lab, University of Salerno, Italy — [3]LUT University, Finland
savanna.lujan@tuni.fi,fpecorelli@unisa.it,fpalomba@unisa.it,adelucia@unisa.it,valentina.lenarduzzi@lut.fi

ABSTRACT

Code smells are poor implementation choices applied during software evolution that can affect source code maintainability. While several heuristic-based approaches have been proposed in the past, machine learning solutions have recently gained attention since they may potentially address some limitations of state-of-the-art approaches. Unfortunately, however, machine learning-based code smell detectors still suffer from low accuracy. In this paper, we aim at advancing the knowledge in the field by investigating the role of static analysis warnings as features of machine learning models for the detection of three code smell types. We first verify the potential contribution given by these features. Then, we build code smell prediction models exploiting the most relevant features coming from the first analysis. The main finding of the study reports that the warnings given by the considered tools lead the performance of code smell prediction models to drastically increase with respect to what reported by previous research in the field.

KEYWORDS

Code Smells, Static Analysis Tools, Machine Learning.

ACM Reference Format:
Savanna Lujan,[1*] Fabiano Pecorelli,[2*] Fabio Palomba,[2] Andrea De Lucia,[2] Valentina Lenarduzzi[3]. 2020. A Preliminary Study on the Adequacy of Static Analysis Warnings with Respect to Code Smell Prediction. In *MaLTeSQuE 2020: International Workshop on Machine Learning Techniques for Software Quality Evaluation, November 13, 2020 - Sacramento, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

During software maintenance and evolution, developers continuously modify source code to fix defects, enhance existing functionalities or adapt the system to new environments [22]. In such a context, the need of delivering the system in a timely manner often leads developers to set aside good design and implementation solutions and apply modifications that potentially cause the introduction of the so-called *technical debt* [8]: this is a metaphor introduced to explain, in more practical terms, the compromise between delivering fast and producing high-quality code. One of the most relevant forms of technical debt is represented by *code smells* [16], i.e., symptoms of the presence of sub-optimal implementation solutions. Complex classes or overly long methods are just two examples of code smells that often arise in practice [30]. Previous research has shown that code smells hinder program comprehensibility [1], increase source code change- and fault-proneness [19, 30], and increase maintenance effort [37]. These reasons have inspired the research effort around the definition of automatic solutions to detect code smells in source code [9]. While a number of heuristic-based techniques, relying on different types of software metrics, have been devised (e.g., [27, 29, 32]), a recent trend is represented by the use of machine learning approaches [4]. In particular, machine learning has the potential to address some common limitations of heuristic-based approaches: (1) the subjectivity with which their output is interpreted by developers [14, 28], (2) the need of defining thresholds for the detection [15], and (3) the low agreement among them [13]. Indeed, machine learning may be exploited to combine multiple metrics, learning code smell instances considered relevant by developers without the specification of any threshold [4].

Despite this potential, however, machine learning models for code smell detection have still low capabilities [11], especially due to (1) the little contributions given by the features investigated so far [34] and (2) the limited amount of code smell instances available to train a machine learner in an appropriate manner [33].

In this paper, we focus on the first problem: we aim at advancing the state of the art in machine learning for code smell detection by focusing on the contribution given by the warnings of automated static analysis tools to the classification capabilities. The motivation behind our study is twofold. On the one hand, static analysis tools provide indications about the quality of source code [43], hence being potentially useful to characterize code smell instances. On the other hand, their usage in practice is threatened by the high amount of false positives they output [18]: to deal with it, new instruments able to incorporate static analysis warnings within smarter solutions may represent an interesting use case to make static analysis tools more useful in practice.

Driven by these motivations, we first investigate the potential contribution given by individual types of warnings output by three static analysis tools, i.e., Checkstyle, FindBugs, and PMD, to the prediction of three code smell types, i.e., *God Class*, *Spaghetti Code*, and *Complex Class*. Then, we used the most relevant features coming from the first analysis to build and assess the capabilities of machine learning models when detecting the three considered smells. The key results of the study highlight promising results: models built using the warnings of individual static analysis tools score between 55% and 91% in terms of F-Measure. The warning types that contribute the most to the performance of the learners depend on the specific code smell considered.

*Lujan and Pecorelli must be both considered as first authors of the paper.

MaLTeSQuE 2020, November 13, 2020, Sacramento, USA

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 RESEARCH METHODOLOGY

In our study, we defined the following research questions ($\mathbf{RQ}_s$):

- $\mathbf{RQ}_1$ *Which warning categories contribute the most to the prediction of code smells?*
- $\mathbf{RQ}_2$ *To what extent can static analysis warnings output by different tools predict the presence of code smells?*

More specifically, $\mathbf{RQ}_1$ represents a preliminary research question in which we aim at quantifying whether and to what extent each warning category of the considered tools is relevant for the task of code smell prediction. In $\mathbf{RQ}_2$, instead, we assess the actual capabilities of a machine learner built using the relevant features coming from the previous research question when predicting the presence of code smells in source code; to this aim, we create individual models, i.e., one for each static analyzer considered.

Table 1: Software systems considered in the project.

Project	Description	# Classes	# Methods
Ant 1.8.3	Build system	1,218	11,919
Cassandra 1.1.0	Database Management System	727	7,901
Eclise JDT 3.4.0	Integrated Development Environment	5,736	51,008
HSQLDB 2.0.0	HyperSQL Database Engine	601	11,016
Xerces 2.0.0	XML Parser	542	6,126

2.1 Context of the Study

The *context* of the study is composed of software projects, static analysis tools, and code smells.

2.1.1 Selected Projects. The study considered five large open-source software projects, whose characteristics are reported in Table 1. The systems have different size and scope, hence allowing us to understand whether these factors have an influence on the results. It is important to point out that, we aim at providing preliminary insights into the adequacy of static analysis warnings for code smell detection: a larger-scale analysis is part of our future research agenda.

2.1.2 Selected Tools. To detect static analysis tool warnings, we selected three tools, namely Checkstyle, Findbugs, and PMD. The selection of these tools is driven by recent findings that showed that these are among the static analysis tools more employed in practice by developers [24, 41, 42]. In the following, we report a brief description of each tool.

Checkstyle. Checkstyle is an open-source developer tool that evaluates Java code according to a certain coding standard, which is configured according to a set of "checks". These checks are classified under 14 different categories, are configured according to the coding standard preference, and are grouped under two severity levels: error and warning. More information regarding the standard checks can be found from the Checkstyle web site.[1]

Findbugs. Findbugs is another commonly used static analysis tool for evaluating Java code, more precisely Java bytecode. The analysis is based on detecting "bug patterns", which arise for various reasons. Such bugs are classified under 9 different categories, and the severity of the issue is ranked from 1-20. Rank 1-4 is the *scariest* group, rank 5-9 is the *scary* group, rank 10-14 is the *troubling* group, and rank 15-20 is the *concern* group.[2]

PMD. PMD is an open-source tool that provides different standard rule sets for major languages, which can be customized by the users, if necessary. PMD categorizes the rules according to five priority levels (from P1 "Change absolutely required" to P5 "Change highly optional"). Rule priority guidelines for default and custom-made rules can be found in the PMD project documentation.[3]

2.1.3 Selected code smells. The study considers three class-level code smell types, such as:

- **God Class.** This smell generally appears when a class is large, poorly cohesive, and has a number of dependencies with other data classes of the system [16].
- **Spaghetti Code.** Instances of this code smell arise when a class does not properly use Object-Oriented programming principles (e.g., inheritance), declares at least one long method with no parameters, and uses instance variables [5].
- **Complex Class.** As the name suggests, instances of this smell affect classes that have high cyclomatic complexity [26] and that, therefore, may primarily make the testing of those classes harder [16].

The selection of these smells was driven by two main observations. Firstly, previous studies have connected them to an increase of change- and fault-proneness of source code [6, 19, 30] as well as maintenance effort [37]. Secondly, these smells are highly relevant for developers that, indeed, often recognize them as harmful for the evolvability of software projects [28, 38, 45, 46].

2.2 Data Collection

The data collection phase aimed at gathering information related to independent and dependent variables of our study. These concern with the collection of static analysis warnings from the selected analyzer, which will represent the features to be used in the machine learner, and the labeling of code smell instances, namely the identification of real code smells affecting the considered systems.

2.2.1 Collecting static analysis tool warnings. This step differs based on the static analysis tool considered, as each of them has a different process to be executed.

Checkstyle. The `jar` file for the Checkstyle analysis was downloaded directly from the Checkstyle's website[4] in order to engage the analysis from the command line. The executable `JAR` file used in this case was `checkstyle-8.30-all.jar`. In addition to downloading the `JAR` executable, Checkstyle offers two different types of rule sets for the analysis. For each of the rule sets, the configuration file was downloaded directly from Checkstyle's website.[5] In order to start the analysis, the files `checkstyle-8.30-all.jar` and the configuration file in question were saved in the directory where all the projects resided.

[1] https://checkstyle.sourceforge.io

[2] http://findbugs.sourceforge.net/findbugs2.html

[3] https://pmd.github.io/latest/

[4] https://checkstyle.org/#Download

[5] https://github.com/checkstyle/checkstyle/tree/master/src/main/resources

Findbugs. FindBugs 3.0.1 was installed by running the `brew install findbugs` in the command line. Once installed, the GUI was then engaged by writing `spotbugs`. From the GUI, the analysis was executed through *File → New Project*. The classpath for the analysis was identified to be the location of the project directory. Moreover, the source directories were identified to be the project `JAR` executable. Once the class path and source directories were identified, the analysis was engaged by clicking Analyze in the GUI. Once the analysis finished, the results were saved through *File → Save as* using the XML file format. The main specifications were the "Classpath for analysis (jar, ear, war, zip, or directory)" and "Source directories (optional; used when browsing found bugs)" where the project directory and project `jar` file were added.

PMD. PMD 6.23.0 was downloaded from GitHub[6] as a zip file. After unzipping, the analysis was engaged by identifying several parameters: project directory, export file format, rule set, and export file name. In addition to downloading the zip file, PMD offers 32 different types of rule sets for Java.[7] All 32 rule sets were used during the configuration of the analysis.

Using these procedures, we ran the three static analysis tools against the source code of the considered systems. At the end of the analysis, these tools extracted a total of 60,904, 4,707, and 179,020 warnings for Checkstyle, FindBugs, and PMD respectively.

Table 2: Descriptive statistics about the number of code smell instances.

Code Smell	Min.	Median	Mean	Max.
God Class	1.00	6.00	8.60	25.00
Complex Class	0.00	4.00	13.40	54.00
Spaghetti	2.00	26.00	18.00	30.00

2.2.2 Collecting information on actual code smell instances. This stage consisted of identifying real code smells in the considered software projects. While some previous studies relied on automated mechanisms for this step, e.g., by using metric-based detectors [2, 20, 25], recent findings showed that such a procedure could threaten the reliability of the dependent variable and, as a consequence, of the entire machine learning model [10]. Hence, in our study we preferred a different solution, namely considering manually-validated code smell instances. In particular, for all the systems considered, there exist a publicly available dataset reporting actual code smell instances [31] which has been also used in more recent studies evaluating the performance of machine learning models for code smell detection [30, 33, 34]. For each code smell, Table 2 reports the distribution of the code smells in the dataset.

2.3 Data Analysis

In this section, we report the methodological steps conducted to address our research questions.

[6] https://github.com/pmd/pmd/releases/download/pmd_releases%2F6.23.0/pmd-bin-6.23.0.zip

[7] https://github.com/pmd/pmd/tree/master/pmd-java/src/main/resources/rulesets/java

2.3.1 ***RQ*$_1$***. Contribution of static analysis warnings in code smell prediction.* In the first RQ, we assessed the extent to which the various warning categories of the considered static analysis tools can potentially impact the performance of a machine learning-based code smell detector. To this aim, we employed an information gain algorithm [36], and particularly the *Gain Ratio Feature Evaluation* technique, to establish a ranking of the features according to their importance for the predictions done by the different models. Given a set of features F = $\{f_1, ..., f_n\}$ belonging to the model M, the *Gain Ratio Feature Evaluation* computes the difference, in terms of Shannon entropy, between the model including the feature f_i and the model that does not include f_i as independent variable. The higher the difference obtained by a feature f_i, the higher its value for the model. The outcome of the algorithm is represented by a ranked list, where the features providing the highest gain are put at the top. This ranking was used to address **RQ**$_1$.

2.3.2 ***RQ*$_2$***. The role of static analysis warnings in code smell prediction.* Once we had investigated which warning categories relate the most to the presence of code smells, in **RQ**$_2$ we proceeded with the definition of machine learning models.

Specifically, we defined a feature for each warning type raised by the tools, where each feature contained the number of violations of that type identified in a class. For instance, suppose that for a class C_i Checkstyle identifies seven violations to the warning type called *"Bad Practices"*: the machine learner is fed with the integer value "7" for the feature *"Bad Practices"* computed on the class C_i.

The dependent variable is, instead, given by the presence/absence of a certain code smell. This implies the construction of three models for each tool, i.e., for *God Class*, *Spaghetti Code*, and *Complex Class*, respectively. Overall, we therefore built nine models per project - one for each code smell/static analysis tool pair.

As for the supervised learning algorithm, the literature in the field still misses a comprehensive analysis of which algorithm works better in the context of code smell detection [4]. For this reason, we experimented with multiple classifiers such as *J48*, *Random Forest*, *Naive Bayes*, *Support Vector Machine*, and *JRip*. When training these algorithms, we followed the recommendations provided by previous research [4, 39] to define a pipeline dealing with some common issues in machine learning modeling. Particularly, we exploit the output of the Gain Information algorithm—used in the context of **RQ**$_1$—to discard unrelevant features that can bias the interpretation of the models [39]: we did that by excluding the features not providing any information gain. We also configured the hyper-parameters of the considered machine learners using the MultiSearch algorithm, which implements a multidimensional search of the hyper-parameter space to identify the best configuration of the model based on the input data. Finally, we considered the problem of data balancing: it has been recently explored in the context of code smell prediction [33] and the reported findings showed that data balancing may and may not be useful to improve the performance of a model. Hence, before deciding on whether to apply data balancing, we benchmarked (i) *Class Balancer*, which is an oversampling approach (ii) *Resample*, an undersampling method (iii) *Smote*, an approach including synthetic instances to oversample the minority class, and (iv) *NoBalance*, namely the application of no balancing methods.

After training the models, we proceeded with the evaluation of their performance. We applied a 10-fold cross-validation: with this strategy, the dataset (including the training set) was divided in 10 parts respecting the proportion between smelly and non-smelly elements. Then, we trained for ten times the models using 9/10 of the data, retaining the remaining fold for testing purpose—in this way, we allowed each fold to be the test set exactly once. For each test fold, we evaluated the models by computing a number of performance metrics, such as precision, recall, F-Measure, AUC-ROC, and Matthews Correlation Coefficient (MCC).

3 RESULTS AND DISCUSSION

In the following, we discuss the results addressing our research questions.

Table 3: Information Gain of our independent variables for each static analysis tool.

	Checkstyle		FindBugs		PMD	
Code Smell	**Metric**	**Mean**	**Metric**	**Mean**	**Metric**	**Mean**
God Class	Imports	0.42	Performance	0.10	Error Prone	0.05
	Blocks	0.34	Style	0.06	Design	0.03
	Javadoc	0.12	I18N	0.00	Code Style	0.03
	Design	0.11	Correctness	0.00	Multithreading	0.02
	Indentation	0.09	Experimental	0.00	Documentation	0.02
	0ming	0.08	Malicious Code	0.00	Performance	0.01
	Coding	0.04	Security	0.00	Best Practices	0.01
	Checks	0.03	MT Correctness	0.00	—	—
	Sizes	0.01	Bad Practice	0.00	—	—
	Whitespace	0.01	—	—	—	—
	Modifier	0.00	—	—	—	—
	Regexp	0.00	—	—	—	—
Complex Class	Regexp	0.45	Style	0.07	Design	0.06
	Checks	0.05	Performance	0.04	Error Prone	0.05
	Coding	0.03	Correctness	0.02	Code Style	0.04
	Blocks	0.03	I18N	0.00	Documentation	0.03
	Javadoc	0.02	Experimental	0.00	Performance	0.03
	Indentation	0.02	Malicious Code	0.00	Best Practices	0.02
	Design	0.01	Security	0.00	Multithreading	0.01
	0ming	0.01	MT Correctness	0.00	—	—
	Sizes	0.01	Bad Practice	0.00	—	—
	Modifier	0.00	—	—	—	—
	Imports	0.00	—	—	—	—
	Whitespace	0.00	—	—	—	—
Spaghetti Code	Javadoc	0.04	Security	0.40	Error Prone	0.04
	Design	0.04	Performance	0.06	Design	0.02
	Indentation	0.03	Style	0.06	Code Style	0.02
	Coding	0.03	I18N	0.03	Multithreading	0.02
	Checks	0.02	Experimental	0.00	Performance	0.01
	0mings	0.02	Correctness	0.00	Best Practices	0.01
	Imports	0.00	Malicious Code	0.00	Documentation	0.01
	Whitespace	0.00	MT Correctness	0.00	—	—
	Sizes	0.00	Bad Practice	0.00	—	—
	Modifier	0.00	—	—	—	—
	Regexp	0.00	—	—	—	—
	Blocks	0.00	—	—	—	—

3.1 RQ_1. Investigating the contribution of warning types.

Table 3 summarizes the information gain values obtained by the metrics composing the nine models built in our study. The first thing to notice is that, depending on the code smell type, the warning types have a different weight: this practically means that a machine learner for code smell identification should exploit different features depending on the target code smell rather than rely on a unique set of metrics to detect them all.

When analyzing the most powerful features of Checkstyle and PMD, we could notice that source code design-related features are constantly at the top of the ranked list for all the considered code smells. This is, for instance, the case of the *Regexp* warnings given by Checkstyle for *Complex Class* or the *Design* metrics output by PMD for *Spaghetti Code*. The most relevant warnings also seem to be strongly related to specific code smells: as an example, the complexity of regular expressions might strongly affect the likelihood to have a *Complex Class* smell; similarly, design-related issues are the most characterizing aspects of a *Spaghetti Code*. In other words, from this analysis we could delineate a relation between the most relevant features output by Checkstyle and PMD and the specific code smells considered in this paper.

A different discussion should be done for FindBugs: in this case, the most powerful metrics mostly relate to *Performance* or *Security*, which are supposed to cover different code issues than code smells. As such, we expect this static analysis tool to have lower performance when applied to code smell detection.

Finally, it is worth noting that in some cases the information gain of the considered features seems to be low, e.g., the *Error Prone* warning category of PMD in the case of *God Class*. On the one hand, this may potentially imply a low capability of the features when employed within a machine learning model. On the other hand, it may also be the case the such a little information would already be enough to characterize and predict the existence of code smell instances. Next section addresses this point further.

3.2 RQ_2. Assessing the models built using static analysis tools alone.

Table 4 reports the performance capabilities of the models built using the warnings given by Checkstyle, FindBugs, and PMD, respectively. For the sake of space limitations, we only discuss the overall results obtained with the best configuration of the models, namely the one considering *Random Forest* as classifier and *Class Balancer* as data balancing algorithm. The results for the other models are available in our online appendix.

We can immediately point out that the performance of the models built using the warnings of static analysis tools can drastically improve the capabilities of code smell prediction models previously reported in literature [11, 34]. As an example, Pecorelli et al. [34] reported that models built using code metrics of the Chidamber-Kemerer suite [7] work worst than a constant classifier that always considers an instance as non-smelly. Instead, our findings report that it is possible to achieve high classification values by relying on different sets of metrics that cover various aspects of source code quality. While the values of F-Measure and AUC-ROC vary depending on the specific models built, all of them range between 55% and 91% and from 78% and 98% when considering F-Measure and AUC-ROC, respectively.

The lowest performance was obtained by the model built using the output of FindBugs to predict the presence of *Spaghetti Code* instances. Based on the results obtained in RQ_1, we can reason on the motivations behind this result. Unlike the other static analysis tools, FindBugs has the specific goal to identify bug patterns rather than more generic design problems: despite containing a number of warning types analyzing the overall quality of a class, it often looks at individual lines of code trying to spot the existence of possible implementation errors. For example, this is the case of

Table 4: Results reporting the performance of the models built with the warning generated by the three static automatic tools.

	Checkstyle					FindBugs					PMD				
	Prec.	Recall	FM	AUC-ROC	MCC	Prec.	Recall	FM	AUC-ROC	MCC	Prec.	Recall	FM	AUC-ROC	MCC
God Class	0.91	0.91	0.91	0.99	0.91	0.83	0.60	0.70	0.92	0.70	0.91	0.70	0.79	0.99	0.80
Complex Class	1.00	0.40	0.57	0.90	0.63	0.84	0.63	0.72	0.92	0.72	0.91	0.59	0.71	0.99	0.73
Spaghetti Code	0.91	0.70	0.79	0.98	0.80	0.78	0.43	0.55	0.78	0.56	0.97	0.66	0.79	0.97	0.80

"Method call passes null to a non-null parameter", that is a type of warning that validates the exchange of information between methods. The granularity of these warnings is lower than the other tools, hence influencing the ability of the model to characterize the overall quality of a class affected by *Spaghetti Code*. Hence, we may conclude that the class-level warning types of FindBugs are not enough to identify code smells—this is the tool performing worst also when considering the other code smells.

4 THREATS TO VALIDITY

Construct Validity. This threat concerns the relationship between theory and observation due to possible measurement errors. The selected static analysis tools are among the most reliable static analysis tools and most adopted by developers [41]. 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.

Internal Validity. This threat concerns internal factors related to the study that might have affected the results. When assessing the role of static analysis tools for code smell detection, we considered three tools to increase our knowledge on the matter. Yet, we recognize that other tools might consider different, more powerful warnings that may affect the performance of the learners. Also in this case, further analyses are part of our future research agenda.

External Validity. As for the generalizability of the results, our empirical study should be considered as a preliminary study conducted on five open-source software projects with different scope and characteristics. We plan to conduct a larger scale analysis as future work.

Conclusion Validity. This threat concerns the relationship between the treatment and the outcome. We adopted different machine learning techniques to reduce the bias of the low prediction power that a single classifier could have. We also addressed possible issues due to multi-collinearity, missing hyper-parameter configuration, and data unbalance. We recognize, however, that other statistical or machine learning techniques (e.g. deep learning) might have yielded similar or even better accuracy than the techniques we used.

5 RELATED WORK

The use of machine learning techniques for code smell detection is recently gaining attention, as proved by the amount of publications in the last years. A complete overview of the research done in the field is available in the survey by Azeem et al. [4].

Specifically, while machine learning has been originally applied to detect individual code smell types, e.g., [20, 21, 44], some effort has recently been made to generalize its usage. Arcelli Fontana et al. were among the most active researchers in the field and applied machine learning techniques to detect multiple code smell types [2], estimate their harmfulness [2], and compute their intensity [3], showing the potential usefulness of these techniques. Pecorelli et al. [35] investigated the adoption of machine learning to classify code smells according to their perceived criticality. Nonetheless, Di Nucci et al. [10] reported that the composition of the training data can notably influence the performance of machine learning-based code smell detection methods: in particular, this is due to the small amount of actual smelly instances that can be retrieved in a software system which does not allow a learner to properly characterize code smells [33]. In addition, the features exploited so far (e.g., the CK metrics [7]) are not able to properly describe code smells and, as a consequence, these techniques do not perform better than simpler constant baselines [34]. The works discussed above represent the main motivation leading to our study. Indeed, we aimed at advancing the state of the art by understanding the value of the warnings of static analysis tools as features of a machine learning-based code smell detector.

On a different note, a few works have applied machine learning techniques to analyze static analysis warnings and, particularly, to evaluate change- and fault-proneness of SonarQube violations [12, 17, 40]. Tollin et al. [17], analyzed in the context of two industrial projects, analyzed whether the warnings given by the tool are associated to classes with higher change-proneness, confirming the relation. Falessi et al. [12] analyzed 106 SonarQube violations in an industrial project: the results demonstrated that 20% of faults were preventable should these violations have been removed. Lenarduzzi et al. [40] assessed the fault-proneness of SonarQube violations on 21 open-source systems, showing that violations classified as *"bugs"* hardly lead to a failure. In another work, Lenarduzzi et al. [23] showed that technical debt cannot be predicted using standard software metrics. Our work is complementary to those discussed above, since our goal is to exploit the outcome of different static analysis tools in order to improve the accuracy of code smell detection.

6 CONCLUSION

In this paper, we assessed the adequacy of static analysis warnings in the context of code smell prediction. We started by analyzing the contribution given by each warning type to the prediction of three code smell types. Then, we measured the performance of machine learning models using static analysis warnings as features and aiming at identifying the presence of code smells. To sum up, the main contributions provided by this paper are:

(1) An investigation into the role of static analysis warnings for machine learning-based code smell detection;

(2) An empirical study of how static analysis warnings contribute to the accuracy of existing machine learning approaches for code smell detection;
(3) An online appendix [] reporting all data and scripts used to conduct our study.

Our future research agenda includes a larger scale evaluation of the devised models as well as the definition of a combined model able to exploit warnings coming from different static analysis tools to improve the overall code smell identification performance.

REFERENCES

[1] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. 2011. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *2011 15th European Conference on Software Maintenance and Reengineering*. IEEE, 181–190.

[2] Francesca Arcelli Fontana, Mika V. Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and Experimenting Machine Learning Techniques for Code Smell Detection. *Empirical Softw. Engg.* 21, 3 (June 2016), 1143–1191.

[3] Francesca Arcelli Fontana and Marco Zanoni. 2017. Code Smell Severity Classification Using Machine Learning Techniques. *Know.-Based Syst.* 128, C (July 2017), 43–58.

[4] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. 2019. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology* 108 (2019), 115–138.

[5] William J Brown, Raphael C Malveau, Hays W McCormick III, and Thomas J Mowbray. 1998. Refactoring software, architectures, and projects in crisis.

[6] Gemma Catolino, Fabio Palomba, Francesca Arcelli Fontana, Andrea De Lucia, Andy Zaidman, and Filomena Ferrucci. 2020. Improving change prediction models with code smell-related information. *Empirical Software Engineering* 25, 1 (2020).

[7] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on software engineering* 20, 6 (1994), 476–493.

[8] W. Cunningham. 1992. The WyCash Portfolio Management System *(OOPSLA-92)*.

[9] Elder Vicente de Paulo Sobrinho, Andrea De Lucia, and Marcelo de Almeida Maia. 2018. A systematic literature review on bad smells—5 W's: which, when, what, who, where. *IEEE Transactions on Software Engineering* (2018).

[10] Dario Di Nucci, Fabio Palomba, Damian Tamburri, Alexander Serebrenik, and Andrea De Lucia. 2018. Detecting Code Smells using Machine Learning Techniques: Are We There Yet?. In *Int. Conf. on Software Analysis, Evolution, and Reengineering*.

[11] Dario Di Nucci, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Andrea De Lucia. 2018. Detecting code smells using machine learning techniques: are we there yet?. In *26th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 612–621.

[12] D. Falessi, B. Russo, and K. Mullen. 2017. What if I Had No Smells? *ESEM* (2017).

[13] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. 2012. Automatic detection of bad smells in code: An experimental assessment. *J. Object Technol.* 11, 2 (2012), 5–1.

[14] Francesca Arcelli Fontana, Jens Dietrich, Bartosz Walter, Aiko Yamashita, and Marco Zanoni. 2016. Antipattern and code smell false positives: Preliminary conceptualization and classification. In *23rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 1. IEEE, 609–613.

[15] Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, and Aiko Yamashita. 2015. Automatic metric thresholds derivation for code smell detection. In *6th International Workshop on Emerging Trends in Software Metrics*. IEEE, 44–53.

[16] M. Fowler and K. Beck. 1999. Refactoring: Improving the Design of Existing Code. *Addison-Wesley Longman Publishing Co., Inc.* (1999).

[17] F. Arcelli Fontana I. Tollin, M. Zanoni, and R. Roveda. 2017. Change Prediction Through Coding Rules Violations. *Int. Conf. on Evaluation and Assessment in Software Engineering*, 61–64.

[18] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *35th International Conference on Software Engineering (ICSE)*. IEEE, 672–681.

[19] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2012. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering* 17, 3 (2012), 243–275.

[20] F. Khomh, S. Vaucher, Y.-G. Gueheneuc, and H. Sahraoui. 2009. A Bayesian Approach for the Detection of Code and Design Smells. In *Int. Conf. on Quality Software (QSIC '09)*. IEE, Jeju, Korea, 305–314.

[21] Foutse Khomh, Stephane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. 2011. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. *Journal of Systems and Software* 84, 4 (2011), 559–572.

[22] Manny M Lehman. 1996. Laws of software evolution revisited. In *European Workshop on Software Process Technology*. Springer, 108–124.

[23] Valentina Lenarduzzi, Antonio Martini, Davide Taibi, and Damian Andrew Tamburri. 2019. 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)*. 37–42.

[24] V. Lenarduzzi, A. Sillitti, and D. Taibi. 2020. A Survey on Code Analysis Tools for Software Maintenance Prediction. In *6th International Conference in Software Engineering for Defence Applications*. Springer International Publishing, 165–175.

[25] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabané, Yann-Gaël Guéhéneuc, and Esma Aimeur. 2012. Smurf: A svm-based incremental anti-pattern detection approach. In *Working Conference on Reverse Engineering*. 466–475.

[26] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[27] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. 2009. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering* 36, 1 (2009), 20–36.

[28] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. 2014. Do they really smell bad? a study on developers' perception of bad code smells. In *International Conference on Software Maintenance and Evolution*. IEEE, 101–110.

[29] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2014. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering* 41, 5 (2014), 462–489.

[30] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering* 23, 3 (01 Jun 2018), 1188–1221.

[31] Fabio Palomba, Dario Di Nucci, Michele Tufano, Gabriele Bavota, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2015. Landfill: An open dataset of code smells with public evaluation. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 482–485.

[32] Fabio Palomba, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Andy Zaidman. 2016. A textual-based technique for smell detection. In *24th international conference on program comprehension (ICPC)*. IEEE, 1–10.

[33] Fabiano Pecorelli, Dario Di Nucci, Coen De Roover, and Andrea De Lucia. 2020. A large empirical assessment of the role of data balancing in machine-learning-based code smell detection. *Journal of Systems and Software* (2020), 110693.

[34] Fabiano Pecorelli, Fabio Palomba, Dario Di Nucci, and Andrea De Lucia. 2019. Comparing heuristic and machine learning approaches for metric-based code smell detection. In *27th International Conference on Program Comprehension (ICPC)*. IEEE, 93–104.

[35] Fabiano Pecorelli, Fabio Palomba, Foutse Khomh, and Andrea De Lucia. 2020. Developer-Driven Code Smell Prioritization. In *International Conference on Mining Software Repositories*.

[36] J. Ross Quinlan. 1986. Induction of decision trees. *Machine learning* 1, 1 (1986), 81–106.

[37] Dag IK Sjøberg, Aiko Yamashita, Bente CD Anda, Audris Mockus, and Tore Dybå. 2012. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering* 39, 8 (2012), 1144–1156.

[38] Davide Taibi, Andrea Janes, and Valentina Lenarduzzi. 2017. How developers perceive smells in source code: A replicated study. *Information and Software Technology* 92 (2017), 223–235.

[39] Chakkrit Tantithamthavorn and Ahmed E Hassan. 2018. An experience report on defect modelling in practice: Pitfalls and challenges. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. 286–295.

[40] F. Lomio V. Lenarduzzi, H. Huttunen, and D. Taibi. 2019. Are SonarQube Rules Inducing Bugs? *27th International Conference on Software Analysis, Evolution and Reengineering (SANER) (preprint arXiv:1907.00376)* (2019).

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

[42] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. C. Gall. 2018. Context is king: The developer perspective on the usage of static analysis tools. *26th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2018).

[43] Fadi Wedyan, Dalal Alrmuny, and James M Bieman. 2009. The effectiveness of automated static analysis tools for fault detection and refactoring prediction. In *International Conference on Software Testing Verification and Validation*. 141–150.

[44] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Int. Conf. on Automated Software Engineering (ASE)*. 87–98.

[45] Aiko Yamashita and Leon Moonen. 2012. Do code smells reflect important maintainability aspects?. In *2012 28th IEEE international conference on software maintenance (ICSM)*. IEEE, 306–315.

[46] Aiko Yamashita and Leon Moonen. 2013. Do developers care about code smells? An exploratory survey. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 242–251.