Comparing Within- and Cross-Project Machine Learning Algorithms for Code Smell Detection

Manuel De Stefano
SeSa Lab - University of Salerno, Italy
madestefano@unisa.it

Fabiano Pecorelli
SeSa Lab - University of Salerno, Italy
fpecorelli@unisa.it

Fabio Palomba
SeSa Lab - University of Salerno, Italy
fpalomba@unisa.it

Andrea De Lucia
SeSa Lab - University of Salerno, Italy
adelucia@unisa.it

ABSTRACT

Code smells represent a well-known problem in software engineering, since they are a notorious cause of loss of comprehensibility and maintainability. The most recent efforts in devising automatic machine learning-based code smell detection techniques have achieved unsatisfying results so far. This could be explained by the fact that all these approaches follow a within-project classification, i.e., training and test data are taken from the same source project, which combined with the imbalanced nature of the problem, produces datasets with a very low number of instances belonging to the minority class (i.e., smelly instances). In this paper, we propose a cross-project machine learning approach and compare its performance with a within-project alternative. The core idea is to use transfer learning to increase the overall number of smelly instances in the training datasets. Our results have shown that cross-project classification provides very similar performance with respect to within-project. Despite this finding does not yet provide a step forward in increasing the performance of ML techniques for code smell detection, it sets the basis for further investigations.

CCS CONCEPTS

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

KEYWORDS

Code smells, Transfer Learning, Empirical Software Engineering.

ACM Reference Format:
Manuel De Stefano, Fabiano Pecorelli, Fabio Palomba, and Andrea De Lucia. 2021. Comparing Within- and Cross-Project Machine Learning Algorithms for Code Smell Detection. In *Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution (MaLTESQuE '21), August 23, 2021, Athens, Greece.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3472674.3473978

MaLTESQuE '21, August 23, 2021, Athens, Greece

ACM ISBN 978-1-4503-8625-8/21/08…$15.00
https://doi.org/10.1145/3472674.3473978

1 INTRODUCTION

Software maintenance and evolution is a complex activity of the software life cycle that requires developers to continually change the source code, in order to adapt it to new requirements or fix defects identified in production [25]. More and more often, developers are required to perform these modifications under strict deadlines, making them neglect good programming practices in favor of delivering the most appropriate product on time [6, 24, 37]. This common practice often leads to the introduction of *technical debt* [10], which can be generated by the presence of the so-called *code smells* [22], sub-optimal implementation choices that erode the maintainability and understandability of source code [1, 30, 40].

Over the last years, researchers have been proposing several approaches and techniques which are able to automatically detect code smells in code-bases [4, 15, 16, 28, 32, 33]. The majority of them are heuristics-based: they use a two-step method in which a collection of metrics is computed first, and then thresholds are applied to those metrics to distinguish between smelly and non-smelly groups. They mainly vary in terms of the algorithms used to detect code smells (e.g., a combination of metrics [28] or more sophisticated methodologies such as Relational Topic Modeling [4]) and the metrics used (e.g., based on code metrics [28], textual [33] or historical data [32]). While it has been demonstrated that such detectors have good detection accuracy, previous research has revealed a number of significant limitations that may prevent their practical usage [14, 43]: (i) the output given by these detectors may be subjectively interpreted by developers [18, 27], (ii) there is no consensus between them [17], and (iii) the majority of them require thresholds definition to differentiate smelly code components from non-smelly ones; choosing them well heavily impacts the accuracy of the final results [14].

These are the main reasons for a recent trend in using machine learning for code smell detection [19]. In such a scenario, a classifier exploits a set of independent variables (a.k.a. predictors) to determine the presence (or the absence) of a code smell in a specific code component. This approach should overcome both the threshold issue, as the detection rule is automatically deduced from training data (and not defined *a priori*), and the lack of consensus between heuristic predictors, since multiple metrics can be used as smelliness predictors.

Despite the expectations behind the use of machine learning for code smell detection, previous studies have found controversial results in terms of classification performance [13, 19]. More specifically, not only it has been shown that *within project* machine

learning models suffer from heavily imbalanced datasets, performing worse than heuristic-based detectors [35], but also that current balancing techniques do not improve models' performance [34].

For these reasons, this paper attempts to perform a step further by investigating whether the use of a transfer learning approach may boost the code smell classification performance. We exploit a validated dataset of code smells made up by 15 open source projects to train four well-known machine learning algorithms. We apply a leave-one-project-out validation to assess the performance of the models and compare cross-project techniques with within-project baselines trained on the individual projects of the dataset.

Our results show that the experimented approaches are very similar from a statistical perspective. Nonetheless, we point out that, in some cases, cross-project models might outperform within-project techniques. This may possibly indicate initial insights into the unexplored capabilities of transfer learning for code smell detection, encouraging to deepen the investigations on this matter.

Replicability. To enable full replicability and reproducibility of our results, we made both data and scripts used in the study publicly available in our online appendix [12].

2 RELATED WORK

Over the last years, many researchers have studied code smells in terms of their nature and their impact on code-bases [2, 11]. One of the two main research tracks in this field is related to the automatic or semi-automatic detection of code smells, either using heuristic-based approaches or machine-learning. In this section we put more focus on the machine learning-based, to better explain their *theoretical* advantages with respect to the heuristic-based ones, and to better explain their *real* limitations. The techniques which fall in this category generally exploit a supervised method, i.e., they use a set of independent variables (*predictors*), to predict the value of a dependent variable (i.e., the smelliness of a class) using a machine learning classifier. The model may be trained using data from the subject project (*within-project* strategy) or from other systems (*cross-project* strategy).

Arcelli Fontana *et al.* [19–21] achieved the most relevant results in this field. They not only hypothesized that machine learning might contribute to a more objective assessment of the severity of the code smells [21], but also provided a learning-based method to assess code smell intensity [20]. Finally, they conducted a benchmark study to compare 16 machine-learning-based techniques for the detection of four code smells, obtaining F-Measure values close to 100%. However, Di Nucci *et al.* [13] showed that the performance of these techniques strongly depends on the dataset exploited, hence questioning whether machine learning for code smell detection could be really used in a real-case scenario.

Recently, Pecorelli *et al.* [35] conducted an empirical study aimed at comparing the performance of heuristic-based (metric-based) approaches with machine learning-based ones. They considered five code smell types and contrasted machine learning models with DECOR [28], using as predictors the same set of metrics used by the tool. They showed that, despite DECOR generally performing better than the learning-based detectors, its precision is still too low to make it usable in practice. This is mainly caused by the imbalanced nature of the learning-based code smell detection problem.

The role of balancing in machine learning code smell detection was further investigated by Pecorelli *et al.* [34]: in this case, they took into account five approaches to mitigate data imbalance issues to understand their impact on machine learning-based approaches for code smell detection. They pointed out that avoiding balancing does not dramatically impact accuracy since existing data balancing techniques are not suited for code smell detection, possibly even deteriorating the classification performance. As a result of these findings, we conclude that the problem of classifying code smells with machine learning is still open: this paper poses the basis for a complementary viewpoint, namely the use of transfer learning for training effective code smell detection models.

3 EMPIRICAL STUDY DESIGN

The *goal* of this study is to compare a *cross-project* machine learning-based approach for code smell detection with a *within-project* one, with the *purpose* of having a preliminary indication of whether transfer learning can be considered a valid alternative for code smell detection. The *perspective* is of both researchers and practitioners: the former have an interest in recognizing the shortcomings of existing methods, while the latter are interested in testing the effectiveness of machine learning for code smell detection. In detail, we aim at addressing the following research question:

> **RQ.** How do machine learning-based *cross-project* approaches for code smell detection perform when compared to traditional *within-project* baselines?

To fulfill our goal, we conducted an empirical investigation involving 15 open-source projects and three code smells types. More details are reported in the following sections.

3.1 Context of the Study

The *context* of our study is made up of two fundamental elements: projects and code smells.

Projects. For our study, we employed a publicly available code smell dataset [30], which provides manually validated instances of 13 code smell types pertaining to multiple releases of 30 Java open-source projects. Of these, we discarded the systems that were not present anymore on GitHub at the time of the study or that do not use release tags on GitHub because they were not suitable to perform a reproducible mining. This filter led us to 15 projects left. Since we aimed at creating *cross-project* models, we then considered a single release for each project: this was required to avoid possible bias in the training phase of the models caused by using project data coming from previous releases of the same projects. For this reason, we limited the study to the last releases of the 15 available projects. Table 1 reports the main characteristics of these systems.

Code Smells. While the code smell dataset contains data pertaining to 13 code smell types [30], in our work we took into account only three of them, namely Complex Class, God Class (or Large Class), and Spaghetti Code. Table 2 reports a description for each of them. The selection was driven by two main factors. On the one hand, these smells are among the most harmful for developers [31], as they heavily impact the maintainability of source code [1]. On the other hand, these have been investigated multiple times by the

Table 1: Projects considered in the context of study

Project Name	Release Tag	Classes	LOC
ant	rel/1.8.3	1163	169 510
ant-ivy	2.0.0-alpha2	388	36 665
cassandra	cassandra-1.0.0	644	68 160
elasticsearch	v0.19.0	2327	188 689
hadoop	release-0.6.0	297	40 129
hive	release-0.9.0	1268	161 239
hsqldb	2.2.8	589	182 898
karaf	karaf-2.3.0	549	42 420
lucene	releases/lucene-solr/3.6.0	3608	406 298
manifold-cf	release-0.6	729	119 998
nutch	release-1.4	296	30 024
pig	release-0.8.0	1345	206 603
qpid	0.14	1528	150 037
struts	STRUTS_2_3_4	1797	148 151
xerces2-j	Xerces-J_2_3_0	771	113 385

code smell research community and, for this reason, we decided to focus on them first in the context of this preliminary study.

3.2 Experimental Setup

The first step that we performed in our study was to mine the selected projects in order to extract the software metrics to use as predictors. We mined a set of well known object-oriented static metrics, i.e., the ones defined by Chidamber and Kemerer [7]. As a result, we obtained a dataset where each row represented a class belonging to the project and each column the value of a metric computed on that class. We then combined this dataset with the one reporting the code smells instances. Thus, we added three columns, one for each smell considered: the value '1' indicated the presence of a smell in the class, '0' otherwise.

Afterwards, we set up the machine learning-based classifiers to detect code smells. This entailed a number of steps, from the description of the dependent and independent variables to the preprocessing steps necessary to avoid issues like multi-collinearity and biased interpretation [29].

Dependent Variables. The selected dependent variable is an indicator of a presence (or absence) of each code smell considered, as indicated in the collected dataset.

Independent Variables. We used the code metrics previously mined as independent variables. In particular, Table 3 reports the predictors used for each of the considered smells. The choice of these metrics was driven by our willingness to use the same indicators as previous work [35]: in this way, we could have a direct comparison with the results reported in literature and possibly corroborate previous findings.

Machine Learning Models. To perform the classification phase, we configured and executed four well-known and widely used

Table 2: Code smells considered in the context of the study

Name	Description
Complex Class	A class having at least one method having a high cyclomatic complexity.
God Class (Large Class)	A Large Class implementing different responsibilities and centralizing most of the system processing.
Spaghetti Code	A class implementing complex methods interacting between them, with no parameters, using global variables.

Table 3: List of predictors used for each smell

Acronym	Full Name	Interested Smells
WMC	Weighted Methods Count	Complex Class
ELOC	Effective Lines of Code	Spaghetti Code, Large Class
NMNOPARAM	Number of Methods with NO PARAMeters	Spaghetti Code
NOM	Number Of Methods	Large Class
NOA	Number Of Attributes	Large Class
LCOM	Lack of COhesion in Methods	Large Class

ML Classifiers, i.e., Logistic Regression, Decision Tree, Naive Bayesian Classifier, and Random Forest. In particular, we relied on the implementations provided by the `Tidymodels` R package.[1] Before running each classifier, we performed some preprocessing steps to avoid possible biases due to an improper interpretation of their performance [26, 39, 42]. More specifically:

Hyper-parameter Tuning. Each of the selected classifiers has a number of parameters that have to be configured before running. We configured those hyper-parameters by exploiting the Grid Search algorithm [5] using AUC-ROC as an optimization target.

Data Balancing. Code smell detection suffers from data imbalance [13]. We applied the Synthetic Minority Oversampling Technique, a.k.a. SMOTE, since it has been proven to be the best performing data balancing technique when detecting code smells [34].

Validation Strategy. To assess the capabilites of the experimented machine learning models, we adopted two different validation strategies, according to the configuration under analysis (i.e., within- or cross-project). For within-project, we adopted the ten-fold cross validation [38]. For each of the selected projects, this strategy first randomly partitions the data into ten stratified folds (i.e., each fold has the same proportion of code smell instances) of equal size. Then, it iteratively selects a single fold to use as test set, while the remaining nine are used as training set. As for the cross-project configuration, we adopted the leave-one-out cross-validation [41]. This strategy iteratively selects one project as test set, while the remaining ones are used as training set.

It is worth remarking that the preprocessing steps were performed when training the models, hence not biasing the distribution of code smells in the test set.

[1] `Tidymodels`: https://www.tidymodels.org/

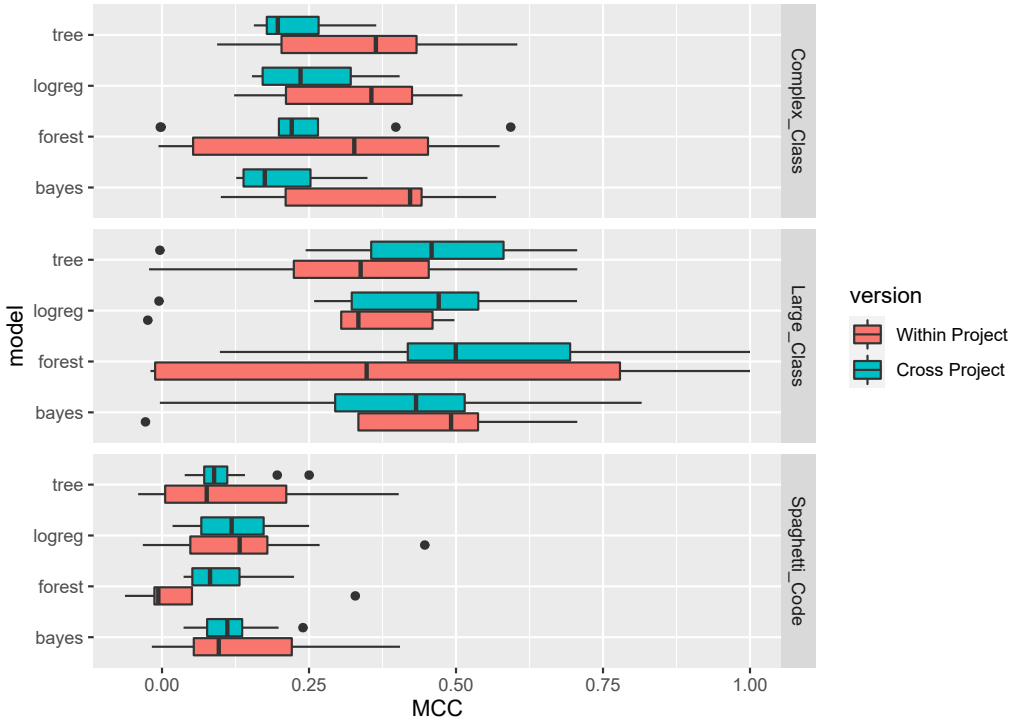

Figure 1: Box-plots comparing the performances of within- and cross-project approaches, plotted for each smell in terms of MCC. For Large Class detection the median values of the cross-project models are higher or similar than their within-project counterparts. In the other cases, there is no clear winner between the approaches.

Evaluation metrics. To evaluate the performance achieved by the experimented techniques, we relied on four well-known information retrieval metrics [3, 36], namely precision, recall, F-measure, and Mattews Correlation Coefficient (MCC). Then, to statistically verify our results, we applied the non-parametric Wilcoxon Rank Sum Test [9], with $\alpha = 0.05$, on the distribution of MCC values achieved by within- and cross-project configurations. We also estimated the magnitude of the differences by exploiting the Cliff's Delta (or δ) non-parametric effect size measure [8]. We interpreted the effect size values as: negligible for $|d| < 0.10$, small for $|d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [8].

4 ANALYSIS OF THE RESULTS

Figure 1 shows the boxplots reporting the distributions of the MCC scores computed for both within- and cross-project models.

As a first consideration, we can see that, regardless of the model applied, the higher performance is achieved when classifying Large Class instances, whereas the worst is found when detecting Spaghetti Code. This result corroborates previous findings that reported the Large Class smell as the easiest to detect and the Spaghetti Code as the hardest to identify [28, 32, 35].

Looking at the comparison between classifiers, it seems that, generally, Logistic Regression achieves the highest performance for cross-project classification, while Naive Bayesian Classifier is the one performing best in the within-project scenario, achieving an overall performance that is perfectly in line with previous investigations conducted in the field [35].

Turning to the differences between within- and cross-project classification, the boxplots give a first indication that there is not

Table 4: Results for Wilcoxon Test and Cliff's Delta computed for each combination of model and smell MCC values. P-values lower than 0.05 are reported in bold

Smell Name	Model Name	p-value	*d*	Meaning
Complex Class	Naive Bayesian	0.181	-0.444	medium
Complex Class	Random Forest	0.863	-0.074	negligible
Complex Class	Logistic Regression	0.388	-0.296	small
Complex Class	Decision Tree	0.327	-0.333	medium
Large Class	Naive Bayesian	0.878	-0.066	negligible
Large Class	Random Forest	0.695	0.159	small
Large Class	Logistic Regression	0.278	0.366	medium
Large Class	Decision Tree	0.520	0.25	small
Spaghetti Code	Naive Bayesian	0.978	-0.012	negligible
Spaghetti Code	**Random Forest**	**0.019**	**0.607**	**large**
Spaghetti Code	Logistic Regression	1	0	negligible
Spaghetti Code	Decision Tree	0.686	0.103	negligible

a clear difference between the performance achieved by the two approaches, with some cross-project models performing better (e.g., Large Class), and some performing worse (e.g., Complex Class) than their within-project counterparts.

To support this last observation, let us consider the results of the statistical tests (i.e., Wilcoxon and Cliff's Delta) shown in Table 4, that report the differences in the distribution of MCC values for each smell and model. These results clearly indicate that there is no statistically significant difference between the two approaches, thus confirming our intuitions. This is also further strengthened when considering that the Cliff's delta values are in most cases "small" or "negligible". The only exception is represented by the application of Random Forest for Spaghetti Code. In this case, the results indicate statistically significant differences (i.e., a p-value lower than 0.05) with a large effect size in favor of cross-project—looking at the boxplots in Figure 1, this was the only case in which there was a clearly visible difference in the distributions achieved.

To broaden the scope of the discussion, we can observe that, despite the presence of some promising models, e.g., Random Forest for Large Class detection, no median value goes beyond 0.5, meaning that the overall classification performance of the experimented models is fairly low. Our results confirm once again that the problem of classifying code smells using machine learning is still far from being solved. As a matter of fact, the application of transfer learning does not provide immediate benefits: indeed, increasing the number of instances of the minority class in the training sets did not overcome the limitations pointed out in literature [34].

However, it is important to note that, while within-project models have been analyzed from different perspectives, cross-project ones are still unexplored: as such, we cannot exclude that this approach may lead to a performance improvement in the future. Moreover, the fact that cross-project approach does not perform worse than within-project one paves the way for further, deeper investigations.

Main findings for RQ

Cross-project machine learning for code-smell detection shows no statistically significant differences with respect to within-project approaches. However, the early results achieved pave the way for further improvement and investigations.

5 THREATS TO VALIDITY

In the following, we describe how we assessed and mitigated potential threats to the validity of our study.

Threats to Construct Validity. This group of threats is concerned with the relationship between hypothesis and observation. A first, possible threat could be represented by the dataset used for our empirical analysis. We based the selection of the dataset on a number of considerations, including heterogeneity of the data and the provision of a manual validation. However, we must keep in mind that they might include discrepancies or inaccuracies, such as labeling errors or some positive instances that might have been overlooked. Another potential threat might be represented by the machine learning models we adopted, since their construction needed to take into account several aspects that might have influenced the obtained results, e.g., the predictors used. However, since we applied well-known and established rule of thumbs, that have been employed also in previous studies (e.g., [13, 34, 35]), we can consider them reliable for our study.

Threats to Conclusion Validity. Concerns about the relationship between treatment and outcome were addressed by using a collection of commonly used indicators to assess the tested techniques (i.e., precision, recall, F-measure, MCC). Furthermore, we supported our results with sufficient statistical tests (Wilcoxon and Cliff's delta). In the case of the within-project machine learning model, the use of 10-fold cross validation may have introduced a bias in the interpretation of the results. This strategy randomly partitions the dataset into training and test sets; this randomness may have resulted in skewed sets, which under- or over-estimate model performance. To account for this, we conducted a further analysis, as suggested by Hall et al.[23], in which we ran the experimented model several times to determine how robust it is in relation to the random splits performed by the validation strategy. As a result, we conducted a 10 times 10-fold cross validation and then tested the instability of the model's predictions; we discovered that the predictions do not change in 97% of the cases. According to these findings, we can assume that the model's outcomes are not affected by the randomness of the validation strategy.

Threats to External Validity. To what concerns the external validity of our study, we considered a dataset of 15 java project versions. While this number seems to be small, several factors need to be considered that are relevant for the problem. In the context of within-project approaches, the number of considered projects or releases does not affect the external validity, since all the data must come from the same source, i.e., the same project. In this case, the number of positive instances is limited by nature and the only applicable solutions are represented by the applied balancing techniques. Moreover, since the obtained model could be used only to detect new code smells emerging in the same project, within-project approaches require only validated data coming from a single release of a project. On the other hand, to what concerns transfer learning, which uses data coming from multiple sources, it might be affected by the number of projects considered for model training. This limits the possibility of a larger-scale empirical study. Nonetheless, further studies involving a greater number of projects is on our research agenda, whenever new data will be available. Another aspect to take into account was the choice of the smells that we made. We chose a set of smells to reflect a wide range of design issues (e.g., smells related to complexity or excessive coupling between objects), that have been already object of several studies. This helped us better understand the potential of transfer learning techniques for detecting code smells, as well as their drawbacks in comparison to "traditional" approaches.

6 CONCLUSION AND FUTURE WORK

In this paper we have compared two approaches for machine learning-based code smell detection, i.e., within- and cross-project, with the aim of discovering if a transfer learning approach (cross-project) could bring some benefits in machine learning code smell detection research. Our preliminary results show that there is no statistically

significant difference between the two approaches. However, since the transfer learning approach does not perform worse than the "traditional" one, it may be worth to keep investigating on this topic, for instance by accurately tuning the cross-project machine learning pipeline or by defining *ad hoc* software engineering for AI techniques. In our future research agenda we plan to conduct a larger study, involving other machine-learning models that we have not employed so far. Moreover, whenever a new and larger code smell dataset will be available, we plan to replicate this study in order to deepen our knowledge on this topic.

ACKNOWLEDGEMENT

Fabio gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Projects No. PZ00P2_186090.

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] 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.

[3] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. 1999. *Modern information retrieval*. Vol. 463. ACM press New York.

[4] Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. 2013. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering* 40, 7 (2013), 671–694.

[5] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of machine learning research* 13, 2 (2012).

[6] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, et al. 2010. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 47–52.

[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] Norman Cliff. 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin* 114, 3 (1993), 494.

[9] William Jay Conover. 1999. *Practical nonparametric statistics* (3. ed ed.). Wiley, New York, NY [u.a.].

[10] Ward Cunningham. 1993. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger* 4, 2 (1993), 29–30.

[11] 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).

[12] Manuel De Stefano, Fabiano Pecorelli, Fabio Palomba, and Andrea De Lucia. 2021. Comparing Within- and Cross-Project Machine Learning Algorithms for Code Smell Detection - Online Appendix. https://bit.ly/3vICTxO.

[13] 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 *2018 ieee 25th international conference on software analysis, evolution and reengineering (saner)*. IEEE, 612–621.

[14] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. 2016. A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*. 1–12.

[15] Marios Fokaefs, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. 2007. Jdeodorant: Identification and removal of feature envy bad smells. In *2007 ieee international conference on software maintenance*. IEEE, 519–520.

[16] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. 2011. JDeodorant: identification and application of extract class refactorings. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 1037–1039.

[17] 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.

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

[19] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering* 21, 3 (2016), 1143–1191.

[20] Francesca Arcelli Fontana and Marco Zanoni. 2017. Code smell severity classification using machine learning techniques. *Knowledge-Based Systems* 128 (2017), 43–58.

[21] Francesca Arcelli Fontana, Marco Zanoni, Alessandro Marino, and Mika V Mäntylä. 2013. Code smell detection: Towards a machine learning-based approach. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 396–399.

[22] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[23] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. 2011. Developing fault-prediction models: What the research can show industry. *IEEE software* 28, 6 (2011), 96–99.

[24] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. 2012. Technical debt: From metaphor to theory and practice. *Ieee software* 29, 6 (2012), 18–21.

[25] Meir M Lehman. 1980. Programs, life cycles, and laws of software evolution. *Proc. IEEE* 68, 9 (1980), 1060–1076.

[26] Zaheed Mahmood, David Bowes, Peter CR Lane, and Tracy Hall. 2015. What is the impact of imbalance on software defect prediction performance?. In *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*. 1–4.

[27] Mika V Mäntylä and Casper Lassenius. 2006. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering* 11, 3 (2006), 395–431.

[28] Naouel Moha, Yann-Gaël Guéhéneuc, 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.

[29] Robert M O'brien. 2007. A caution regarding rules of thumb for variance inflation factors. *Quality & quantity* 41, 5 (2007), 673–690.

[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 (2018), 1188–1221.

[31] 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 *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 101–110.

[32] 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.

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

[34] 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* 169 (2020), 110693.

[35] 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 *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 93–104.

[36] David MW Powers. 2020. Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. *arXiv preprint arXiv:2010.16061* (2020).

[37] Forrest Shull, Davide Falessi, Carolyn Seaman, Madeline Diep, and Lucas Layman. 2013. Technical debt: Showing the way for better transfer of empirical results. In *Perspectives on the Future of Software Engineering*. Springer, 179–190.

[38] Mervyn Stone. 1974. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society: Series B (Methodological)* 36, 2 (1974), 111–133.

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

[40] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2017. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering* 43, 11 (2017), 1063–1088.

[41] Aki Vehtari, Andrew Gelman, and Jonah Gabry. 2017. Practical Bayesian model evaluation using leave-one-out cross-validation and WAIC. *Statistics and computing* 27, 5 (2017), 1413–1432.

[42] Zhou Xu, Jin Liu, Zijiang Yang, Gege An, and Xiangyang Jia. 2016. The impact of feature selection on defect prediction performance: An empirical comparison. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 309–320.

[43] Min Zhang, Tracy Hall, and Nathan Baddoo. 2011. Code bad smells: a review of current knowledge. *Journal of Software Maintenance and Evolution: research and practice* 23, 3 (2011), 179–202.