On the Role of Data Balancing for Machine Learning-Based Code Smell Detection

Fabiano Pecorelli
University of Salerno
Italy
fpecorelli@unisa.it

Dario Di Nucci
Vrije Universiteit Brussel
Belgium
dario.di.nucci@vub.be

Coen De Roover
Vrije Universiteit Brussel
Belgium
coen.de.roover@vub.be

Andrea De Lucia
University of Salerno
Italy
adelucia@unisa.it

ABSTRACT

Code smells can compromise software quality in the long term by inducing technical debt. For this reason, many approaches aimed at identifying these design flaws have been proposed in the last decade. Most of them are based on heuristics in which a set of metrics (e.g., code metrics, process metrics) is used to detect smelly code components. However, these techniques suffer of subjective interpretation, low agreement between detectors, and threshold dependability. To overcome these limitations, previous work applied Machine Learning techniques that can learn from previous datasets without needing any threshold definition. However, more recent work has shown that Machine Learning is not always suitable for code smell detection due to the highly unbalanced nature of the problem. In this study we investigate several approaches able to mitigate data unbalancing issues to understand their impact on ML-based approaches for code smell detection. Our findings highlight a number of limitations and open issues with respect to the usage of data balancing in ML-based code smell detection.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Code Smells, Machine Learning, Data Balancing

ACM Reference Format:
Fabiano Pecorelli, Dario Di Nucci, Coen De Roover, and Andrea De Lucia. 2019. On the Role of Data Balancing for Machine Learning-Based Code Smell Detection. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE '19), August 27, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3340482.3342744

MaLTeSQuE '19, August 27, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-6855-1/19/08…$15.00
https://doi.org/10.1145/3340482.3342744

1 INTRODUCTION

During software development strict deadlines and new requirements could lead to the introduction of *technical debt* [8], namely a set of design issues that may negatively affect a system's maintainability in the future. *Code smells* [17] are one of the first indications of code technical debt, i.e., sub-optimal design solutions that developers apply to a software system.

Code smells has been investigated from several perspectives [3, 9]: their introduction [45, 46] and evolution [6, 31, 34], their impact on reliability [38, 39] and maintainability [23, 35], as well as the way developers perceive them [36, 44, 48] have been deeply analysed in literature and have revealed that code smells represent serious threats to source code maintenance and evolution.

For all these reasons, several techniques to automatically identify code smells in source code have been investigated [11, 30, 37]. These techniques rely on heuristics and discriminate code artefacts affected (or not) by a certain type of smell through the application of detection rules that compare the values of relevant metrics extracted from source code against empirically identified thresholds. The accuracy of such approaches has been empirically assessed and was found to be fairly high. Nevertheless, they share common limitations that hinder their adoption in practice [11, 49]. First, they could return code smell candidates that are not considered as actual problems by developers [13, 28]. Furthermore, the agreement between detectors is very low [12], which means that different detectors are required to detect the smelliness of different code components. Finally, the performance of most of the current detectors is strongly influenced by the thresholds needed to identify smelly and non-smelly instances [11].

To overcome these limitations, researchers recently adopted machine learning (ML) to avoid thresholds and decrease the false positive rate [14]: in this schema, a classifier is trained on previous releases of the source code by exploiting a set of independent variables (e.g., structural, historical, or textual metrics). The resulting model is employed to determine the presence of a smell or the degree of smelliness of a code element. Although the use of machine learning looks promising, previous work has observed contrasting results [10, 14, 41]. Heuristic-based approaches perform slightly better than machine learning approaches, thus indicating that Machine Learning is still unsuitable for code smell detection [41]. As code smell detection is a highly unbalanced problem [10, 41], data balancing is a key factor to improve the reliability of such models.

Data balancing can be introduced in several ways by transforming the training set or by using cost-sensitive classifiers.

In this paper, we propose a large-scale empirical study —that features 125 releases, 13 software systems, and five code smell types— in which we compare the performance of five data-balancing techniques for code smell detection and compare their performance with a *no-balancing* baseline. Our results suggest that models employing *SMOTE* realize the best performance but exhibit some limitations related to the model training. Moreover, data balancing does not dramatically improve the performance of the models. In conclusion, further work is needed to improve the current ML-based approaches to code smell detection, primarily to improve the quality of the training set on which the models are trained.

Structure of the paper. Section 2 discusses the literature related to ML-based code smell detection. Section 3 describes the design of the empirical study, while Section 4 analyses the achieved results. Section 5 sketches the possible threats affecting our findings. Finally, Section 6 concludes the paper.

2 RELATED WORK

Machine learning has been used in several recent works on code smell detection [3]. Kreimer [26] proposed a prediction model based on *Decision Trees* and code metrics to detect two code smells (i.e., *Blob* and *Long Method*). This model can lead to high values of accuracy. Later on, Amorim et al. [1] confirmed the previous findings on four medium-scale open-source projects. Vaucher et al. [47] studied *Blob*'s evolution relying on a *Naive Bayes* classifier, whereas Maiga et al. [27] proposed the use of *Support Vector Machine* (SVM). The use of *Bayesian Belief Networks* to detect *Blob*, *Functional Decomposition*, and *Spaghetti Code* instances on open-source programs, proposed by Khomh et al. [24] lead to an overall F-Measure close to 60%. Similarly, Hassaine et al. [20] defined an immune-inspired approach for the detection of *Blob* smells, while Oliveto et al. [32] used B-Splines to detect them. Arcelli Fontana et al. made the most relevant progress in this field [14–16]. In their work, they (i) theorised that ML might lead to a more objective evaluation of the smells' hazardousness [16], (ii) provided a ML method to assess code smell intensity [15], and (iii) compared 16 ML techniques for the detection of four code smell types [14] showing that ML can lead to F-Measure values close to 100%. Nevertheless, recently Di Nucci et al. [10] demonstrated that, in a real use-case scenario, the results achieved by Arcelli Fontana et al. [14] cannot be generalised, thus casting doubt on the actual effectiveness of machine learning for code smell detection. Finally, Pecorelli et al. [41] compared ML-based and heuristic metric-based approaches to assess the real capabilities of ML in the context of code smell detection showing that heuristic techniques for code smell detection still perform slightly better.

3 STUDY DESIGN

The purpose of this study is to understand the impact of data balancing techniques on the performance of Machine Learning (ML) algorithms in code smell detection. In particular, we aim to address the following research questions:

RQ1. Do data balancing techniques impact the performance of Machine Learning algorithms in code smell detection?

Table 1: Descriptive statistics for smells distribution

Code Smell	min	mean	median	max	total
God Class	0	5.5	4	24	509
Spaghetti Code	0	12.7	11	31	1443
Class Data Should Be Private	0	11.4	11	37	1150
Complex Class	0	6.4	4	20	669
Long Method	3	48.3	26	147	4763

RQ2. Which data balancing technique is the most effective at improving the performance of Machine Learning algorithms in code smell detection?

3.1 Context of the Study

The *context* of the study consisted of 125 releases of 13 open-source software systems [33]. We relied on the same dataset and the same lists of code smells that we used in our previous study [41] in which we compared heuristic- and ML-based techniques for code smell detection. The dataset is also available in our online appendix [40].

The projects are heterogenous since they have different sizes, lifetimes, and belong to different application domains. The main characteristics of the considered projects are reported in the online appendix [40], as well as in the previous study [41]. Note that the dataset is not composed of artificially crafted code smell instances but of *manually validated* ones (i.e., 8, 534). The distribution of code smells in the dataset is reported in Table 1. The low median number of code smells in each considered release clearly demonstrates that code smell detection is a highly unbalanced problem. We considered five different types of code smells defined by Fowler [17]:

- *God Class.* This smell characterises classes having a large size, poor cohesion, and several dependencies with other data classes of the system [17]. Previous work showed that this smell has a negative impact on both program comprehension and software maintainability [23, 33].
- *Spaghetti Code.* Classes affected by this smell declare a number of long methods without parameters [17]. For this smell too, the negative impact on comprehensibility and maintainability has previously been shown [23, 33].
- *Class Data Should be Private.* This smell appears in cases where a class exposes its attributes, thus violating the information hiding principle [17]; for this smell, previous work has shown that developers often do not recognise its presence and consider it as less harmful than others to maintainability [36, 44].
- *Complex Class.* Classes presenting a overly high cyclomatic complexity [29] are affected by this design flaw. As shown in the literature [23, 33, 36], it can worsen software maintainability and reduce the ability of developers to properly enhance the corresponding source code.
- *Long Method.* Methods implementing more than one functionality are affected by this smell [17]. It can lower program understanding and make the source code more change- and fault-prone [23, 33, 36].

3.2 Experimental Design

The goal of the experiment was to compare the performance of different data balancing techniques. To this aim we configured five different ML variants based on the Naive Bayes classifier [22] which in our previous study [41] performed the best in code smell detection. Before creating the model we applied a Feature Selection step by using Correlation-based Feature Selection (CFS) [19] to remove highly correlated independent variables. We tuned the hyper-parameters of the classifier by applying the Grid Search algorithm [5]. Thus, the only difference in the models consisted in the way in which we balanced the training set. Data balancing can be introduced by transforming the training set or by using meta-classifiers (e.g., cost-sensitive classifiers). In the latter case, the cost sensitivity can be introduced in a two-fold manner. On the one hand, the training instances can be re-weighted according to the total cost assigned to each class, i.e., the cost-sensitivity is considered during the training phase. On the other hand, the class with the minimum expected misclassification cost rather than the most likely class can be predicted, i.e., the cost-sensitivity is introduced in the testing phase. In this experiment, we employed five different techniques:

- *Class Balancer* [18] re-weights the instances of the training set so that the sum of the weights for each class of instances in the dataset is equal.
- *Resample* produces a random subsample of the dataset using either sampling with replacement or without replacement. In our experiment we sampled by replacing instances of the majority class (i.e., clean classes) with instances from the minority class (i.e., smelly classes) until obtaining an even number of instances for both classes.
- *Synthetic Minority Over-sampling TEchnique* [7] increases the number of instances from the minority class by generating new synthetic instances based on the nearest neighbours belonging to that class.
- *Cost Sensitive Classifier* [25] is a meta-classifier that renders a cost-sensitive version of the base classifier. Specifically, we relied on implementation provided by Weka [18].
- *One Class Classifiers* [21] are trained only on the samples belonging to the minority class to learn the unique features of this class and accurately identify an unseen sample of this class as distinct from a sample of any other class. All instances belonging to other classes are identified as outliers.

Finally, to answer **RQ1**, we trained the models without applying any data balancing technique (i.e., *No-balancing*).

Table 2: Full Names of the Considered Metrics.

Acronym	Full Name	Code Smells
ELOC	Effective Lines Of Code	God Class, Spaghetti Code
LCOM	Lack of COhesion in Methods	God Class
LOC_METHOD	Lines Of Code of METHOD	Long Method
NOA	Number Of Attributes	God Class
NOM	Number Of Methods	God Class
NOPA	Number Of Public Attributes	Class Data Should Be Private
NP	Number of Parameters	Long Method
NMNOPARAM	Number of Methods with NO PARAMeters	Spaghetti Code
WMC	Weighted Methods Count	God Class, Complex Class

As *independent variables* we considered only code metrics that are related to code features of the software instances (e.g., size, complexity). For the detection of each Code Smell, we exploited the set of metrics originally adopted by Moha et al. [30]. Table 2 reports the entire list of metrics used as independent variables and for which code smell have been used.

Since we were interested in detecting code smells, we set the presence/absence of a certain code smell as *dependent variable* of the machine learning model. This information was already available in the considered dataset.

To assess the capabilities of the machine learning model, we adopted 10-Fold Cross Validation [43]. This methodology randomly partitions the data into 10 folds of equal size, applying a stratified sampling (e.g., each fold has the same proportion of code smell instances). A single fold is used as test set, while the remaining ones are used as training set. The process was repeated 10 times, using each time a different fold as test set. The result of the process described above consisted of a confusion matrix for each code smell type, for each of the 125 releases of the considered projects, and for each experimented classifier. These matrices have been later analysed to measure the evaluation metrics described in the following parts of the section.

3.3 Evaluation Metrics

To assess the performance of the experimented detection techniques we computed four well-known metrics [4, 42], namely, *precision*, *recall*, *F-Measure*, and *Matthews Correlation Coefficient (MCC)*. Since we considered several releases of several systems, we needed to aggregate the results achieved for each release to have a clearer overview of the performance [2]. We discuss the results in terms of MCC because this metric provides a better overview with respect to the other metrics by considering all the confusion matrix. Therefore, we first computed the confusion matrix for each release on which we detected code smells. Then, we aggregated the confusion matrices ranging over all the releases of a software system and we computed the above metrics. Aggregate metrics are more robust than the mean, which is biased by the fact that datasets are unbalanced for different smell types in terms of smelly and non smelly instances.

4 STUDY RESULTS

We report the results of the experiments for each code smell.

4.1 God Class

By and large, all the balancing techniques, except *One Class Classifier*, are able to detect most of the *God Class* instances, as inferable from the high values for recall (0,83 to 0,93). However the low precision indicates that, the considered balancing techniques do not help in reducing the high number of false positives. Indeed, only *SMOTE* has a slightly higher precision (only 1%) than *No-balancing*. We can notice similar results when analysing the MCC and the boxplot in Figure 1. It is important to note that the performance of *No-balancing* is very close to that obtained by *SMOTE* and clearly better than the other balancing techniques.

4.2 Complex Class

In case of *Complex Class*, the results are discordant. In particular, *ClassBalancer*, *Resample*, and *CostSensitiveClassifier* show very

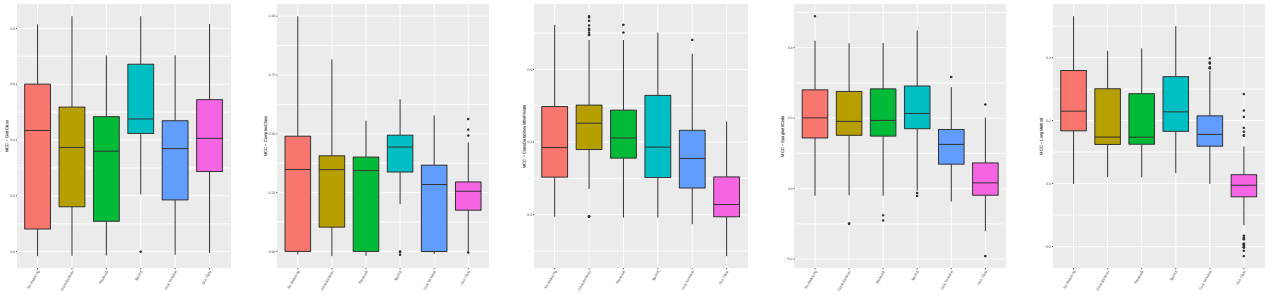

Figure 1: Boxplots representing the MCC values obtained by Naive Bayesian trained applying different balancing strategies for all the considered code smells

Table 3: Aggregate Results for God Class

	God Class			
	Precision	Recall	F-measure	MCC
No-balancing	0.25	0.83	0.39	0.45
ClassBalancer	0.18	0.92	0.30	0.41
SMOTE	**0.26**	**0.93**	**0.41**	**0.49**
Resample	0.17	0.90	0.28	0.39
CostSensitiveClassifier	0.19	0.85	0.31	0.39
OneClassClassifier	0.19	0.51	0.27	0.30

low results in terms of precision and high recall. Contrarily, *No-balancing* and *SMOTE* show slightly higher precision and lower recall while *OneClassClassifier* has bad precision and bad recall. As for *God Class*, *SMOTE* is the best techniques in terms of F-measure and MCC but the performance is close to the one obtained without applying any balancing. Figure 1 confirms that *SMOTE* achieves the best performance for *Complex Class*. Indeed, the boxplot for *SMOTE* has the highest median and the lowest Interquartile Range, which means that its results are more reliable.

Table 4: Aggregate Results for Complex Class

	Complex Class			
	Precision	Recall	F-measure	MCC
No-balancing	0.23	0.58	0.33	0.36
ClassBalancer	0.14	**0.85**	0.24	0.34
SMOTE	**0.26**	0.65	**0.37**	**0.4**
Resample	0.14	0.84	0.24	0.33
CostSensitiveClassifier	0.12	0.74	0.20	0.29
OneClassClassifier	0.06	0.65	0.11	0.19

4.3 Class Data Should Be Private

Table 5 reports the results for *Class Data Should Be Private*. Compared to the other smells, the moderately higher precision is countered by the general lower recall. *Class Balancer* shows the highest values for F-Measure and MCC and the boxplots in Figure 1 confirm that this technique should be applied for this kind of smell.

4.4 Spaghetti Code

The results show that this is the hardest smell to detect, with respect to the other four considered smells. The MCC values range between 0,03 (i.e., *OneClassClassifier*) and 0,22 (i.e., *SMOTE*). As for the other metrics (i.e., Precision, Recall, F-Measure), the results are very poor compared to the ones obtained on other code smells. Overall, the higher F-Measure and MCC of *SMOTE* suggest that it is the best balancing technique.

Table 5: Aggregate Results for Class Data Should Be Private

	Class Data Should Be Private			
	Precision	Recall	F-measure	MCC
No-balancing	**0.3**	0.33	0.31	0.31
ClassBalancer	0.23	0.55	**0.33**	**0.35**
SMOTE	0.27	0.33	0.30	0.29
Resample	0.21	0.54	0.31	0.33
CostSensitiveClassifier	0.07	0.55	0.12	0.17
OneClassClassifier	0.01	**0.71**	0.03	0.05

Table 6: Aggregate Results for Spaghetti Code

	Spaghetti Code			
	Precision	Recall	F-measure	MCC
No-balancing	**0.16**	0.30	0.21	0.21
ClassBalancer	0.09	0.56	0.15	0.20
SMOTE	**0.16**	0.34	**0.22**	**0.22**
Resample	0.08	0.56	0.15	0.20
CostSensitiveClassifier	0.04	0.54	0.08	0.13
OneClassClassifier	0.01	**0.72**	0.02	0.03

4.5 Long Method

Long Method is a method-level code smell, so the number of analysed instances is higher than the other considered smells. This could explain the very low precision values shown in Table 7. Indeed, although the recall is quite high for all techniques, F-Measure and MCC are very low. Figure 1 confirms that *No-balancing* and *SMOTE* are the most accurate balancing techniques for this smell.

4.6 Discussion

The results show that the current ML-based approaches for code smell detection are quite limited independently from the balancing technique used ($MCC < 0,50$). Overall, *SMOTE* seems to have better performance, however, after analysing the results, we discovered a common but often not clearly stated limitation of this technique. It

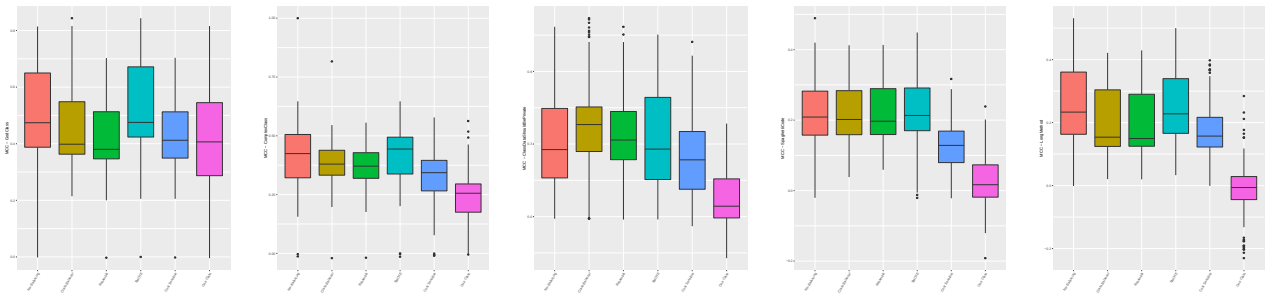

Figure 2: Boxplots representing the MCC values obtained by Naive Bayesian trained applying different balancing strategies for all the considered code smells and excluding NaN values

Table 7: Aggregate Results for Long Method

	Long Method			
	Precision	Recall	F-measure	MCC
No-balancing	**0.15**	0.56	**0.23**	**0.28**
ClassBalancer	0.05	0.74	0.10	0.19
SMOTE	0.12	0.58	0.20	0.26
Resample	0.05	0.74	0.10	0.19
CostSensitiveClassifier	0.06	0.71	0.11	0.19
OneClassClassifier	0.00	**0.80**	0.01	0.01

fails to balance the dataset when the number of smelly instances is small. In particular, *SMOTE* requires at least k smelly instances and if these are not available then the algorithm fails.

Aware of this limitation, we analysed the percentage of failures for all the techniques and noticed that in $\approx 11\%$ of cases *SMOTE* is not able to balance the training set. This represents a clear disadvantage with respect to the other techniques. Based on this observation, we removed all these "boundary" cases and re-computed the boxplots. Figure 2 shows that, in contrast to the previous analysis, *No-balancing* performs better than all the other techniques. Thus, the results suggest that the current data balancing techniques are not adequate for code smell detection, posing several questions on the feasibility of current ML-based approaches. An interesting outcome concerns the very bad performance of *OneClassClassifier* that usually provides good performance when applied on unbalanced problems. Our explanation is that in this case, it is not able to realize an effective training due to the extremely low number of "target" instances (i.e., smelly instances).

5 THREATS TO VALIDITY

Possible threats to validity are within *Construct Validity*, *External Validity*, and *Conclusion Validity*.

Construct Validity. The dataset choice is a threat. We relied on a dataset from a previous study [41] that was created considering several factors such as heterogeneity. The dataset has been manually-validated, but we have to consider that it may be incomplete as well as imprecise. Another threat is the construction of the machine-learning models, for which we took several aspects into account that could have possibly influenced the study, i.e., which features to consider, how to train the classifier, etc. However, the procedures followed in this respect are precise enough to ensure the validity of the study.

External Validity. We considered a large dataset consisting of 125 releases of 13 open source systems belonging to different application domains and having different characteristics. As for the code smells, we selected five smells that represent a large variety of design issues. Although in a previous study [41] Naive Bayes outperformed the other ML algorithms, the choice of this technique could be a possible threat to validity. For both previous threats, further experiments on different datasets and techniques would be desirable and are already part of our future research agenda.

Conclusion Validity. We exploited a set of widely-used metrics to evaluate the experimented techniques (i.e., precision, recall, F-measure, MCC). As for the machine learning model, a possible bias might have been due to the usage of 10-fold cross validation. This strategy randomly partitions the set of data to create training and test sets: such randomness might have possibly led to the creation of biased training/test sets that have the consequence of under- or over-estimating the model performance.

6 CONCLUSION

In this paper, we have reported on a large-scale empirical comparison between five different balancing techniques for ML-based code smell detection. The study considered five code smell types in a manually-validated dataset comprising 125 releases belonging to 13 open source systems.

The results suggest that ML models relying on *SMOTE* realize the best performance. However, its training phase is not always feasible in practice. Furthermore, avoiding balancing does not dramatically impact the performance. Existing data balancing techniques are therefore inadequate for code smell detection. This hinders the feasibility of the current ML-based approaches.

Our future work includes devising new techniques for data balancing, as well as understanding how other configurations (e.g., Parameter Tuning, Feature Selection) impact the quality of the predictions. Finally, we aim to investigate novel techniques to compose the training sets on which to train the models.

ACKNOWLEDGMENTS

This work was partially supported by the Excellence of Science Project SECO-Assist (0015718F, FWO - Vlaanderen and F.R.S.-FNRS).

REFERENCES

[1] L. Amorim, E. Costa, N. Antunes, B. Fonseca, and M. Ribeiro. Experience report: Evaluating the effectiveness of decision trees for detecting code smells. In *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pages 261–269. IEEE, 2015.

[2] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE transactions on software engineering*, 28(10):970–983, 2002.

[3] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, page in press., 2019.

[4] R. Baeza-Yates, B. d. A. N. Ribeiro, et al. *Modern information retrieval*. New York: ACM Press; Harlow, England: Addison-Wesley,, 2011.

[5] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[6] A. Chatzigeorgiou and A. Manakos. Investigating the evolution of bad smells in object-oriented code. In *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, pages 106–115. IEEE, 2010.

[7] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[8] W. Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1993.

[9] E. V. de Paulo Sobrinho, A. De Lucia, and M. de Almeida Maia. A systematic literature review on bad smells—5 w's: which, when, what, who, where. *IEEE Transactions on Software Engineering*, 2018.

[10] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia. Detecting code smells using machine learning techniques: are we there yet? In *25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER2018): REproducibility Studies and NEgative Results (RENE) Track*. Institute of Electrical and Electronics Engineers (IEEE), 2018.

[11] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo. A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, page 18. ACM, 2016.

[12] F. A. Fontana, P. Braione, and M. Zanoni. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11(2):5–1, 2012.

[13] F. A. Fontana, J. Dietrich, B. Walter, A. Yamashita, and M. Zanoni. Antipattern and code smell false positives: Preliminary conceptualization and classification. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 609–613. IEEE, 2016.

[14] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3):1143–1191, 2016.

[15] F. A. Fontana and M. Zanoni. Code smell severity classification using machine learning techniques. *Knowledge-Based Systems*, 128:43–58, 2017.

[16] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mantyla. Code smell detection: Towards a machine learning-based approach. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 396–399. IEEE, 2013.

[17] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[18] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[19] M. A. Hall. Correlation-based feature selection for machine learning. Technical report, 1998.

[20] S. Hassaine, F. Khomh, Y.-G. Guéhéneuc, and S. Hamel. Ids: An immune-inspired approach for the detection of software design smells. In *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, pages 343–348. IEEE, 2010.

[21] K. Hempstalk, E. Frank, and I. H. Witten. One-class classification by combining density and class probability estimation. In *Proceedings of the 12th European Conference on Principles and Practice of Knowledge Discovery in Databases and 19th European Conference on Machine Learning, ECMLPKDD2008*, volume Vol. 5211 of *Lecture Notes in Computer Science*, pages 505–519, Berlin, September 2008. Springer.

[22] G. H. John and P. Langley. Estimating continuous distributions in bayesian classifiers. In *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*, pages 338–345. Morgan Kaufmann Publishers Inc., 1995.

[23] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, 2012.

[24] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui. Bdtex: A gqm-based bayesian approach for the detection of antipatterns. *Journal of Systems and Software*, 84(4):559–572, 2011.

[25] S. Kotsiantis, D. Kanellopoulos, P. Pintelas, et al. Handling imbalanced datasets: A review. *GESTS International Transactions on Computer Science and Engineering*, 30(1):25–36, 2006.

[26] J. Kreimer. Adaptive detection of design flaws. *Electronic Notes in Theoretical Computer Science*, 141(4):117–136, 2005.

[27] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Gueheneuc, and E. Aimeur. Smurf: A svm-based incremental anti-pattern detection approach. In *Reverse engineering (WCRE), 2012 19th working conference on*, pages 466–475. IEEE, 2012.

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

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

[30] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Trans. on Software Engineering*, 36(1):20–36, 2010.

[31] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka. The evolution and impact of code smells: A case study of two open source systems. In *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, pages 390–400. IEEE, 2009.

[32] R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc. Numerical signatures of antipatterns: An approach based on b-splines. In *Software maintenance and reengineering (CSMR), 2010 14th European Conference on*, pages 248–251. IEEE, 2010.

[33] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, pages 1–34, 2017.

[34] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia. A large-scale empirical study on the lifecycle of code smell co-occurrences. *Information and Software Technology*, 99:1–10, 2018.

[35] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, 23(3):1188–1221, 2018.

[36] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia. Do they really smell bad? a study on developers' perception of bad code smells. In *Software maintenance and evolution (ICSME), 2014 IEEE international conference on*, pages 101–110. IEEE, 2014.

[37] F. Palomba, A. De Lucia, G. Bavota, and R. Oliveto. Anti-pattern detection: Methods, challenges, and open issues. In *Advances in Computers*, volume 95, pages 201–238. Elsevier, 2014.

[38] F. Palomba and A. Zaidman. Does refactoring of test smells induce fixing flaky tests? In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 1–12. IEEE, 2017.

[39] F. Palomba and A. Zaidman. The smell of fear: On the relation between test smells and flaky tests. *Empirical Software Engineering Journal*, page in press., 2019.

[40] F. Pecorelli, D. Di Nucci, C. De Roover, and A. De Lucia. On the role of data balancing for machine learning-based code smell detection - online appendix https://figshare.com/articles/_/8247509, 2019.

[41] F. Pecorelli, F. Palomba, D. Di Nucci, and A. De Lucia. Comparing heuristic and machine learning approaches for metric-based code smell detection. In *Proceedings of the 27th International Conference on Program Comprehension*, pages 93–104. IEEE Press, 2019.

[42] D. M. Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. 2011.

[43] M. Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of the royal statistical society. Series B (Methodological)*, pages 111–147, 1974.

[44] D. Taibi, A. Janes, and V. Lenarduzzi. How developers perceive smells in source code: A replicated study. *Information and Software Technology*, 92:223–235, 2017.

[45] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 4–15. ACM, 2016.

[46] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering*, 2017.

[47] S. Vaucher, F. Khomh, N. Moha, and Y.-G. Guéhéneuc. Tracking design smells: Lessons from a study of god classes. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pages 145–154. IEEE, 2009.

[48] A. Yamashita and L. Moonen. Do code smells reflect important maintainability aspects? In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 306–315, Sept 2012.

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