Automatic human-like detection of code smells

Soomlek, S.; Rijn, J.N. van, Bonsangue, M.M.; Soares, C.; Torgo, L.

Automatic Human-Like Detection of Code Smells

Chitsutha Soomlek[1(✉)], Jan N. van Rijn[2], and Marcello M. Bonsangue[2]

[1] Department of Computer Science, Khon Kaen University, Khon Kaen, Thailand
chitsutha@kku.ac.th
[2] Leiden Institute of Advanced Computer Science, Leiden University, Leiden, The Netherlands
{j.n.van.rijn,m.m.bonsangue}@liacs.leidenuniv.nl

Abstract. Many code smell detection techniques and tools have been proposed, mainly aiming to eliminate design flaws and improve software quality. Most of them are based on heuristics which rely on a set of software metrics and corresponding threshold values. Those techniques and tools suffer from subjectivity issues, discordant results among the tools, and the reliability of the thresholds. To mitigate these problems, we used machine learning to automate developers' perception in code smells detection. Different from other existing machine learning used in code smell detection we trained our models with an extensive dataset based on more than 3000 professional reviews on 518 open source projects. We conclude by an empirical evaluation of the performance of the machine learning approach against PMD, a widely used metric-based code smell detection tool for Java. The experimental results show that the machine learning approach outperforms the PMD classifier in all evaluations.

Keywords: Code smells · Machine learning · Software engineering

1 Introduction

Code smells are properties of the source code that may indicate either flaws in its design or some poor implementation choices. Differently from a bug, a code smell does not necessarily affect the technical correctness of a program, but rather it may be a symptom of a bad design pattern affecting the quality of a software system. Also, the experimental evaluation shows a direct correlation between code smells and software evolution issues, design vulnerabilities, and software failure in the long run [6,25,30]. Even in well-managed and designed projects, code smells could be inadvertently added into the code by inexperienced developers, and as such it is very important to detect them early in the design process [18,28].

Typically, code refactoring is a solution to the design problem coming from code smells [10,11]. Due to the subjectivity of their definition, detection of code smells, and the associated refactoring, are non-trivial tasks. The manual detection process requires tremendous efforts and is infeasible for large-scale software.

C. Soares and L. Torgo (Eds.): DS 2021, LNAI 12986, pp. 19–28, 2021.
https://doi.org/10.1007/978-3-030-88942-5_2

Commonly used automated approaches in tools and academic prototypes are search-based, metric-based, symptom-based, visualization-based, probabilistic-based, and machine learning [2,13,24]. The metric-based approach defines code smells systematically using a fixed set of metrics and corresponding threshold values. It is the most commonly used approach in both open-source and commercial tools and the idea has been adopted for more than a decade. The major problems of the metric-based approach are: (1) matching subjective perception of developers who often perceive code smells differently than the metrics classification and (2) reliability of the threshold values. Currently, many well-known code smell detectors adopt the metrics and their threshold values from Lanza and Marinescu's work [15] in 2006 as reference points. However, finding the best-fit threshold values for a certain type of code smell requires significant efforts on data collection and calibration. For example, Lanza and Marinescu's analysis [15] is based on their manual review of few dozen mid-size projects. Moreover, the concept of code smells was introduced and cataloged more than 20 years ago. During this period, programming languages have been evolving to today's modern programming language which comprises both functional and advanced object-oriented features. To obtain more reliable code smell detection results, human perceptions on design issues should be integrated into an automated analysis. Machine learning is one of the promising solutions for this case because it enables a machine to mimic the intelligence and capabilities of humans to perform many functions.

Following this direction, we define the following research questions:

- `RQ1`: Can we mimic a developer's perception of a code smell?
- `RQ2`: How does machine learning perform when comparing to existing tools?

We use a large dataset of industry projects reviewed by developers [17], we clean and prepare the data so to be utilized in training a machine learning model, and finally, we compare the results with those coming from a modern tool using a metric-based approach. This includes the validation of the two approaches concerning the perception by human experts. We make the dataset publicly available on OpenML [29]. Each of the above steps can be considered as a contribution to our work on its own. From the experimental results, we can conclude a better performance of the machine learning approach for code smells detection, compared to the tools based on static rules.

2 Related Work

While there is no general agreement on the definition of code smells or of their symptoms, many approaches have been introduced in the literature to automate code smell identification. There exist both commercial and open-source tools. Detection approaches can be classified from guided manual inspection to fully automated: manual, symptom-based, metric-based, probabilistic, visualization-based, search-based, and cooperative-based [12]. The metric-based approach is the most used technique in both research and tools. In this case, the generic code

smell identification process involves source code analysis and matching the examining source code to the code smell definition and specification by using specific software metrics [24]. Object-oriented metrics suite [4] and their threshold values are commonly used in the detection process. The accuracy of the metric-based approach depends on (i) the metric selection, (ii) choosing the right threshold values, and (iii) on their interpretation.

In recent years, many studies adopted artificial intelligence and machine learning algorithms for code smell identification [27], classification [5,8,13,16], and prioritization [9,20]. Machine learning techniques provide a suitable alternative to detect code smells without human intervention and errors, solving the difficulty in finding threshold values of metrics for identification of bad smells, lack of consistency between different identification techniques, and developers' subjectiveness in identifying code smells. Those techniques differ in the type of code smell detected, the algorithms used, the size of the dataset used for training, and the evaluation approaches and performance metrics.

Originally, Kreimer [14] proposed a prediction model based on decision trees and software metrics to detect blobs and long methods in Java code. The results were evaluated experimentally with regard to accuracy. The efficiency in using decision trees has been confirmed in mid-size open-source software projects [1]. For an extensive review of the existing approaches and their comparison, we refer the reader to [13].

Two closest works to ours are [8] and [5]. They both applied machine learning techniques to detect data class, blob, feature envy, and long method. In [8] the results of 16 supervised machine learning algorithms were evaluated and compared using 74 software systems from Qualitas Corpus. A large set of object-oriented metrics were extracted and computed from this training data by using deterministic rules in the sampling and manual labeling process. Labeling results were confirmed by master students. Due to their limited software engineering experience, relying on them can be considered a limitation of this work. The authors of [5] repeated the work of the authors of [8] to reveal critical limitations. In contrast, our work is based on a dataset constructed by more than 3000 reviews by human experts on more than 500 mid- to large-size software projects.

3 Dataset Construction and Pre-processing

To conduct our empirical study, we need to collect (1) data or a reliable and up-to-date dataset reporting human perceptions on a set of code smells that is large enough to train a machine learning model, and (2) software metrics of the software projects matching to (1). To achieve this, we first surveyed existing datasets and discussed with their contributors the scientific definitions and data collection process. From this study, we selected the dataset that fitted our requirements best. Finally, we used the selected dataset to define (1) a set of software projects, (2) the types of code smells we were interested in and their corresponding detectors, and (3) a set of software metrics to be extracted from the set of software projects.

The MLCQ Dataset. Madeyski and Lewowski contributed the MLCQ data set which contains industry-relevant code smells with severity levels, links to code samples, the location of each code smell in the Java project, and background information of the experts [17]. More specifically, they collaborated with 26 professional software developers to review the code samples with regard to four types of code smells both at class and function levels: blob, data class, feature envy, and long method. The reviews are based on four severity levels, i.e., *critical*, *major*, *minor*, and *none*. The *none* severity level is assigned to a code sample when the expert does not consider it as a code smell, i.e., a negative result. If however the sample is marked by any of the other severity levels then the sample should be considered as a positive result and thus as a code smell. In summary, the samples contain 984, 1057, 454, and 806 positive cases of blob, data class, feature envy, and long method, respectively. For negative results, 3092, 3012, 2883, and 2556 samples are available. The MLCQ dataset captures the contemporary understanding of professional developers towards code smells from 524 active Java open-source projects. This improves on other existing datasets that either rely on graduate and undergraduate students to collect and review software projects or use automatic code smell detectors tools that impose threshold values from legacy literature, to identify certain types of code smells. However, MLCQ is not ready to be used for our research as it does not provide any software metric of the code samples and software projects. Therefore, we expanded the dataset accordingly. Furthermore, since there are 14,853 reviews on 4,770 code samples, it is often more than one expert review on the same code samples. We thus needed to pre-process the dataset. Next, we describe this step.

Pre-processing and Code Smell Selection. Expert reviewers can disagree on the interpretation of a code smell on a given code sample, in particular to the severity levels assigned to it. To combine the multiple reviews on a code sample to a single result, we need to ensure the validity of the results. In other words, the combined result must stay positive when the majority of the reviewers did not evaluate the severity level of the code sample as *none*. Likewise, the combined results must be negative if the majority assigned *none* to the sample. Thus, we mapped the severity level of a review to a corresponding numerical severity score ($critical = 3$, $major = 2$, $minor = 1$, $none = 0$), and calculated the average severity score for each code sample. The result of the last step can be considered as the *average review score*. If the experts agree on the definition of a code smell but they have different opinions for the severity level, the approach still can identify which sample is a certain type of code smells and which is not.

Table 1 presents the distribution of the number of reviews together with the average of the average review score and standard deviation calculated for each group of reviews separately. Surprisingly, for blob and data class, the review results have no significant difference. However, when considering long method and feature envy, we noticed a considerable disagreement on the reviews. For blob, the variation is mostly within one category, either in *critical* to *major* (occurring in one sample with 6 reviews) or in *minor* to *none* (as we see in the cases of 3 to 5 reviews per sample). A similar situation happens when we consider the data class. In case of feature envy, however, when there are four

Table 1. Distribution of the number of reviews (first column). Per code smell, we show for each encountered number of reviewers, the average of the average review score (ARS) that was granted, as well as the average standard deviation (calculated across standard deviations per code sample).

Rev.	Blob		Data class		Feature envy		Long method	
	No. of Samp.	Average ARS ± Std	No. of Samp.	Average ARS ± Std	No. of Samp.	Average ARS ± Std	No. of Samp.	Average ARS ± Std
1	1562	0.00 ± 0.00	1566	0.00 ± 0.00	1954	0.00 ± 0.00	1967	0.00 ± 0.02
2	147	0.88 ± 0.12	147	0.49 ± 0.00	82	0.20 ± 0.16	162	0.68 ± 1.16
3	409	0.50 ± 0.70	411	0.66 ± 0.69	303	0.43 ± 0.73	302	0.00 ± 0.31
4	198	0.66 ± 0.84	196	0.79 ± 0.79	70	0.73 ± 0.97	73	0.96 ± 1.96
5	39	0.73 ± 0.88	39	0.87 ± 0.88	6	0.53 ± 0.87	7	0.97 ± 2.20
6	1	2.33 ± 0.52	1	0.00 ± 0.00	0	N/A	0	N/A

reviews for a sample, the combined severity score has a variation from *none* to *major*, which indicates a diversity of reviewers' opinions. This is not an incident, as the numbers in the table refers to 70 samples. The situation for the long method is even worse. There are 73 code samples with four reviews, leading to an average of the average review score of 0.96 with a standard deviation of 1.96. There are 7 code samples with 5 reviewers, having an average of the average review score of 0.97 and standard deviation of 2.20. The high average standard deviation in these two cases reveal a more spread out disagreement among the human experts. Therefore, we decided to omit feature envy and long method from our experiments.

Selecting Code Smell Detectors. Among all the popular tools in the literature [7,19,24], we selected PMD [22] because it is an active source code analyzer that can automatically analyze a Java project to identify our targeted code smells and also long method, by using metrics and threshold values. By exploring PMD's documentation, relevant sets of Java rules, and PMD source code, we found that the latest version of PMD (6.35.0) available at the time of writing this paper still adopts metrics and threshold values from [15]. Therefore, we decided to use PMD as a representative of metric-based code smell detectors based on threshold values from legacy literature to describe the characteristics of a class containing a particular code smell.

PMD detects blobs by using the following metrics: weighted method count (WMC), access to foreign data (AFTD), and tight class cohesion (TCC) [22]. PMD detects data classes by using the following metrics: weight of class (WOC), number of public attributes (NOPA), number of accessor methods (NOAM), and WMC are employed to identify a sign of encapsulation violation, poor data behavior proximity, and strong coupling [22]. For long method, PMD uses 100 lines of code as a default threshold value to indicate excessive method length [22]. We could not find out on what study this value is based on. However, to the best of our knowledge, there seems not to be a common agreement on threshold values for long methods. According to the MLCQ dataset, the average length of code samples identified as a long method is 20.7 lines, a threshold much lower than the one used in PMD.

Table 2. Performance of PMD methods against code smells determined by human experts. The assessment of the human expert was labelled as code smell if the average review was higher than 0.75.

Code smells	TP	TN	FP	FN	Precision	Recall	F1-score
Blob	111	1822	207	186	0.349	0.374	0.361
Data class	80	1987	46	217	0.635	0.269	0.378
Long method	80	1830	326	166	0.197	0.325	0.245

As a preliminary experiment, we deploy the results from PMD on the classes evaluated in the MLCQ dataset. As such, we compared the detection results against the average review score we calculated from the MLCQ dataset. Because there are a few archives of Java projects in the MLCQ datasets that are either corrupted or no longer exist, we could only analyze and compare 518 projects in total. These can be considered as a baseline of the contemporary understanding of professional developers. Table 2 presents the comparison results in terms of the number of true positive (TP), true negative (TN), false positive (FP), false negative (FN), as well as precision, recall, and F1-Score.

From the comparison results, we can see that the metric-based approach is far from being accurate when considering human experts' perceptions of code smells. The only exception is perhaps the precision of data class (with a poor recall), but we will see later that even in this case, the machine learning approach will perform better. Note that the precision scores are the lowest for long method, indicating the discrepancy between the threshold value set by PMD and the much lower perceived average value calculated from the expert reviews.

Collecting the Software Metrics. In order to deploy machine learning models to detect code smells, we need to extract metrics from each code file. PMD is a metric-based code smell detector, and the API allows us to extract the metrics it calculates, which turns out to be a good starting point. When we employed PMD to analyze the 518 Java open-source projects, PMD presented the code smell identification results with the corresponding metrics. However, PMD does not provide any metric information for the healthy classes and methods as well as their locations in the project. In other words, PMD only provides positive cases with a set of metrics and identified locations. To obtain the negative cases, we customized PMD to present relevant metric information for every path PMD traversed. For instance, when the customized PMD is run and the *GodClass* rule (for detecting the code smell blob) for a Java program is called, WMC, AFTD, and TCC are calculated for the examining class. Note that there are cases when TCC cannot be calculated, e.g., there is no violation of a certain rule. In which case, PMD presents *NaN* as the metric value.

Additionally, we employed the Understand tool by SciTools [26] version 6.0 (build 1055) to analyzed the 518 Java projects. Understand is static code analysis and code visualization tool that can analyze 60 metrics for Java, project structure, and relationships among files, classes, functions, and variables. The

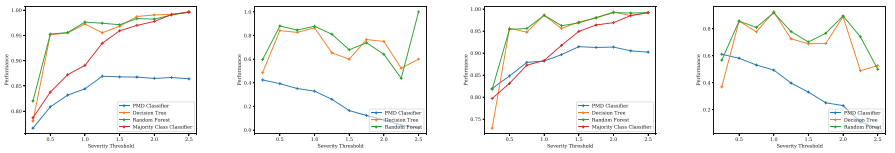
metrics calculated by Understand also cover the CK metrics suite [4]. As a result, we constructed another set of data containing a wide variety of metrics and a very detailed program structure.

4 Empirical Study Definition and Evaluation

Generally, these machine learning models are induced based on a dataset $\mathcal{D} = \{(x_j, y_j) \mid j = 1, \ldots, n\}$ with n datapoints to map an input x to output $f(x)$, which closely represents y. In this setting, x_i is typically the vector of numerical features $\mathcal{F}(\mathcal{C})$ from some code block $\mathcal{C}$. To deploy a machine learning classifier on the task of predicting whether a certain piece of code is considered code smell, we need to have a notion of ground truth (labels) and some features that describe labeled pieces of code. The human expert described per class that was inspected whether it is considered to be a certain type of code smell or not, representing our n data points, and for each of these we now have label y_j (human expert assessment). The main challenge is representing a piece of code $\mathcal{C}$ as feature vector $\mathcal{F}(\mathcal{C})$. For this, we will use two sources of features: features extracted by PMD and features extracted by the Understand tool.

Although both Understand and the human experts report a fully-qualified Java name (which can also be at a subclass in a specific file), PMD uses a different convention. Although PMD is also capable of reporting at (sub-)class level, it does not report the fully-qualified Java name, meaning that ambiguities can arise with duplicated class names, when automatically making the mapping with the class labels. We solve this by predicting code smells at the file level, rather than at the class level. As such, we have more observations in our dataset than actual observations from the human expert dataset. For each class in which a code smell was detected, we now need to study all subclasses as well. If in either of these a code smell was detected, we consider this file a positive case.

We are confronted with the following design choice. The human experts have graded the code smell severity with four levels, i.e., *none*, *minor*, *major* and *critical*. Additionally, since some code pieces were judged by multiple reviewers, we have a broad range of severity levels. If we were to employ a binary classifier, we have to decide from which severity level we consider a piece of code a positive class (code smell). As such, we have to determine a *severity threshold*. To avoid subjectivity, we run the experiment with various ranges of severity. The categorical assessments of the human experts are averaged as described in Sect. 3, such that severity levels around 0 correspond to a negative class, severity levels around 1 correspond to minor code smell, severity levels around 2 correspond to major code smell, and severity levels around 3 correspond to critical code smell. As such, the assumption is that when increasing the severity threshold, detecting the code smell should become easier for the machine learning approach. We ran the experiment several times, with each severity threshold ranging from 0.25 until 2.50 with intervals of 0.25.



(a) Acc. Blob (b) Prec. Blob (c) Acc. Data class (d) Prec. Data class

Fig. 1. Results of various machine learning classifiers on prediction whether a certain file has a code smell of the indicated type, where Acc. is accuracy and Prec. is precision.

5 Results and Discussion

We compared the performance of the machine learning approach in identifying the code smell based on the ground truth decided by human experts. Although we could use any model, we show decision tree [23] and random forest [3], as these both have a good trade-off between performance and interpretability. Both are used as implemented in Scikit-learn [21]. Additionally, we also show the majority class classifier. As most important baseline, we employ the PMD classifier. Indeed, the PMD classifier can also identify code smell based on its own decision rules, and this can be evaluated against the ground truth set by human experts. Note that the PMD classifier is a set of static decision rules, whereas the machine learning models learn these patterns based on the data. Per figure, we show a different performance measure: accuracy and precision. For the majority class classifier we only show accuracy, as it fails to identify any positive class.

Figure 1a and b show the results for blob. The x-axis shows the severity threshold at which a certain experiment was run, and the y-axis shows the performance of the given experiment. As can be seen, the machine learning approaches outperform both baselines in terms of accuracy and precision. Also in terms of recall, the machine learning models are better than the PMD classifier for most of the severity thresholds (figures omitted). It seems that the random forest has a slight edge over the decision tree classifier, and also focuses slightly more on precision.

For data class, the results in Fig. 1c and d confirm that the machine learning techniques have superior accuracy and precision. Altogether, the results seem to indicate that the machine learning techniques are capable of better identifying blob and data class than the static PMD rules.

6 Conclusion and Future Directions

This research intends to mimic contemporary developer's perception of code smell to machine learning and support automated analysis. More specifically, our first research question was 'Can we mimic a developer's perception of a code smell?'

To this aim, we investigated which data we could leverage for building a machine learning classifier. A recent and reliable dataset containing code smells

and developers' perceptions of the design flaws are crucial for this. MLCQ contains four types of code smell, due to constraints with other tools we could use two (data class and blob) for this research. This provides for a wide number of classes information whether a developer considers it a code smell or not. As such, we can employ a binary classifier. We enriched this dataset by automatically extracted metrics from two common tools, i.e., Understand and PMD. These features enable us to train machine learning models on the data and make the machine learning model detect code smells. The machine learning models were able to outperform a majority class baseline on all settings.

The second research question was 'How does machine learning perform when comparing to existing tools?' We compared this machine learning model to PMD, a static metric-based code smell detection tool. We employed both the random forest and decision tree classifier, in settings that had to classify code smells from various severity levels. We measured both accuracy and precision. The results indicate that the machine learning-based models outperform the metric-based tool for both code smells.

Finally, we also make the dataset derived from MLCQ and developed in this research publicly available on OpenML [29]. The dataset would elevate and support advanced studies in the research areas.

Acknowledgements. We would like to thank Khon Kaen University for awarding KKU Outbound Visiting Scholarship to the first author.

References

1. Amorim, L., Costa, E., Antunes, N., et al.: Experience report: evaluating the effectiveness of decision trees for detecting code smells. In: 2015 IEEE 26th International Symposium on ISSRE, pp. 261–269. IEEE (2015)
2. Azeem, M.I., Palomba, F., Shi, L., et al.: Machine learning techniques for code smell detection: a systematic literature review and meta-analysis. Inf. Softw. Technol. **108**, 115–138 (2019)
3. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001)
4. Chidamber, S.R., Kemerer, C.F.: Towards a metrics suite for object oriented design. In: Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications, pp. 197–211 (1991)
5. Di Nucci, D., Palomba, F., Tamburri, D.A., et al.: Detecting code smells using machine learning techniques: are we there yet? In: 2018 IEEE 25th International Conference on SANER, pp. 612–621. IEEE (2018)
6. Elkhail, A.A., Cerny, T.: On relating code smells to security vulnerabilities. In: 2019 IEEE 5th International Conference on BigDataSecurity, IEEE International Conference on HPSC, and IEEE International Conference on IDS, pp. 7–12. IEEE (2019)
7. Fernandes, E., Oliveira, J., Vale, G., et al.: A review-based comparative study of bad smell detection tools. In: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, pp. 1–12 (2016)
8. Arcelli Fontana, F., Mäntylä, M.V., Zanoni, M., et al.: Comparing and experimenting machine learning techniques for code smell detection. Empir. Softw. Eng. **21**, 1143–1191 (2016). https://doi.org/10.1007/s10664-015-9378-4

9. Fontana, F.A., Zanoni, M.: Code smell severity classification using machine learning techniques. Knowl. Based Syst. **128**, 43–58 (2017)
10. Fowler, M., Beck, K., Brant, W., et al.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co. Inc., Boston, USA (1999)
11. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional (2018)
12. Kamaraj, N., Ramani, A.: Search-based software engineering approach for detecting code-smells with development of unified model for test prioritization strategies. Int. J. Appl. Eng. Res. **14**(7), 1599–1603 (2019)
13. Kaur, A., Jain, S., Goel, S., et al.: A review on machine-learning based code smell detection techniques in object-oriented software system(s). Recent Adv. Electr. Electron. Eng. **2021**(14), 290–303 (2021)
14. Kreimer, J.: Adaptive detection of design flaws. Electron. Notes Theoret. Comput. Sci. **141**(4), 117–136 (2005)
15. Lanza, M., Marinescu, R.: Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-39538-5
16. Liu, H., Jin, J., Xu, Z., et al.: Deep learning based code smell detection. IEEE Trans. Softw. Eng. **47**, 1811–1837 (2019)
17. Madeyski, L., Lewowski, T.: MLCQ: industry-relevant code smell data set. In: Proceedings of the Evaluation and Assessment in Software Engineering, pp. 342–347 (2020)
18. Martin, R.C.: Clean Code: A Handbook of Agile Software Craftsmanship, 1st edn. Prentice Hall, USA (2008)
19. Paiva, T., Damasceno, A., Figueiredo, E., Sant'Anna, C.: On the evaluation of code smells and detection tools. J. Softw. Eng. Res. Develop. **5**(1), 1–28 (2017). https://doi.org/10.1186/s40411-017-0041-1
20. Pecorelli, F., Palomba, F., Khomh, F., et al.: Developer-driven code smell prioritization. In: Proceedings of the the 17th International Conference on MSR, pp. 220–231 (2020)
21. Pedregosa, F., Varoquaux, G., Gramfort, A., et al.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
22. PMD: an extensible cross-language static code analyzer. https://pmd.github.io. Accessed 31 May 2021
23. Quinlan, J.R.: Learning decision tree classifiers. ACM Comput. Surv. (CSUR) **28**(1), 71–72 (1996)
24. Rasool, G., Arshad, Z.: A review of code smell mining techniques. J. Softw. Evol. Process **27**(11), 867–895 (2015)
25. Santos, J.A.M., Rocha-Junior, J.B., Prates, L.C.L., et al.: A systematic review on the code smell effect. J. Syst. Softw. **144**, 450–477 (2018)
26. Understand by SciTools. https://www.scitools.com/. Accessed 31 May 2021
27. Sharma, T., Efstathiou, V., Louridas, P., et al.: Code smell detection by deep direct-learning and transfer-learning. J. Syst. Softw. **176**, 110936 (2021)
28. Sirikul, K., Soomlek, C.: Automated detection of code smells caused by null checking conditions in Java programs. In: 2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE), pp. 1–7. IEEE (2016)
29. Vanschoren, J., Van Rijn, J.N., Bischl, B., Torgo, L.: OpenML: networked science in machine learning. ACM SIGKDD Expl. Newsl. **15**(2), 49–60 (2014)
30. Yamashita, A., Moonen, L.: To what extent can maintenance problems be predicted by code smell detection? An empirical study. Inf. Softw. Technol. **55**(12), 2223–2242 (2013)