Int. j. inf. tecnol. (August 2022) 14(5):2701–2707
https://doi.org/10.1007/s41870-022-00943-8

ORIGINAL RESEARCH

SSHM: SMOTE-stacked hybrid model for improving severity classification of code smell

Jatin Nanda[2] · **Jitender Kumar Chhabra**[1]

Received: 11 January 2022 / Accepted: 13 April 2022 / Published online: 7 May 2022

Abstract Code Smells are structural characteristics of software that indicate design problems that lead to less maintainable code. It can be seen as symptom of underlying problems like defects and bugs. In the literature, many detection tools are available for code smell detection. However, due to lack of agreement, Machine learning is used for detecting code smell presence. Apart from the presence of code smell, its severity is also an important factor. In this paper, we analysed and corrected the datasets available in the literature to remove inconsistencies in the God class and Data class datasets. It resulted in better machine learning performance for severity classification of code smell. Thereafter, SSHM approach was proposed that employed SMOTE and Stacking in combination, for severity classification of four code smells namely God class, Data class, Feature envy, and Long method. Three performance parameters: accuracy, spearman's score and mean square error were used for evaluating performance of proposed approach. The proposed approach surpassed other literature study with peak accuracy improvement to 97–99% from 76 to 92% for various code smells.

Keywords Code smells detection · Severity classification · Machine learning · SMOTE · Stacking

✉ Jatin Nanda
jatinnanda0871@gmail.com

Jitender Kumar Chhabra
jitenderchhabra@gmail.com

[1] Computer Engineering Department, National Institute of Technology, Kurukshetra 136119, India

[2] Computer Engineering Department, National Institute of Technology, Kurukshetra, Kurukshetra, HR 136119, India

1 Introduction

Software maintenance and enhancement is a complex activity. A large amount of cost is consumed in it [1]. Various deadlines and constraints force developers to focus on functionality rather than design structures, leading to complex design and low-quality software [2]. One of the foremost indications of the presence of poorly designed software is represented by the presence of Code smells [3]. They are deviation in design characteristics from basic object-oriented principles like abstraction, modularity and modifiability. Most of the software developed initially have good software design, but design structures may be affected with subsequent updates [4]. Many smells are introduced during such updates and enhancements activities [5]. Many empirical research has been conducted to assess the impact of code smells on software quality and maintainability. Modules with code smells are found to be more prone to defects and changes [6]. Code smells have also been found in positive association of fault proneness [7]. These smells hamper the maintainability of the software developed [8]. Formal definitions [9], tools [10–12], and most research conducted are for binary classification. A class/method is classified as positive or negative based on its characteristics. An important notable aspect of code smell is its severity. A minor smell has a low impact on software quality, whereas a severe smell has a high impact [13].

Various machine learning (ML) based methodologies have been studied in the literature. Low ML performance for severity classification of class code smells motivated us to work in this area. The main contributions of this paper are:

- Removing inconsistencies from multinomial God and Data class datasets.
- Studying the performance of ML classifiers trained on corrected vs original datasets.
- Proposing a SMOTE-Stacked Hybrid Model (SSHM) further to improve the severity classification performance of four code smells: God class, Data class, Feature envy, and Long method.

2 Literature review

The literature review is divided into two sections: non-ML based studies for code smell identification and ML-based studies for code smell detection.

2.1 Non-ML studies for detection of code smell

Initially, Lanza et al. [9] proposed a rule-based detection approach providing rules using software metrics. Rules act as the threshold, and when a class/method are in accordance with the threshold, it is flagged positive for the smell; otherwise, not. Marinescu et al. [10] presented a detection tool, iPlasma. It could detect code smells in java and C++ language. Many large-scale projects were successfully modelled using this tool. Moha et al. [14] proposed DÉCOR, a "Rule card" concept, which acts as a sample template for Anti-pattern and Code smell. Average precision and recall during the evaluation were approximately 60% and 100%, respectively. Palomba et al. [15] proposed the HIST approach, which considers historical information from a version control system. Its accuracy increases as the number of versions available for analysis increases. Vidal et al. [16] proposed SpIRIT, a semi-automated process for prioritizing the code smell based on three criteria. They used two case studies to evaluate their approach and found that the developers considered prioritized code smell important. Fontana et al. [17] experimented with four code smell rule-based detectors and analyzed if these detectors were in agreement with each other. It was observed that most detectors did not agree with each other. Due to such a lack of consensus and subjectivity, various ML techniques were implemented in this field.

2.2 ML for detection of code smell

Maiga et al. [18] proposed the SVMDetect approach based on Support Vector Machine classifier to identify four anti-patterns in three open-source java applications. Barbez et al. [19] proposed CAME, which considered the historical evolution metrics and structural aspects and assessed smell presence. These metrics were then fed to the CNN classifier. CAME was better at identifying code smell when compared to various other approaches. Fontana et al. [20] experimented with a large set of machine learning techniques on four Code smells binary datasets. The accuracy varied from 96 to 99%. Nucci et al. [21] experimented with highly imbalanced datasets with more than one smell to present a more realistic picture. It was found that classification performance reported in the literature cannot be achieved with the modified datasets. Guggulothu et al. [22] corrected datasets in literature to remove disparity among instances. They experimented with three multi-label classifiers and achieved an accuracy of $\sim$ 95%. Barbez et al. [23] proposed SMAD, an ensemble approach by taking three different detection tools and merging their output as a single vector. This vector was used as input for Multi-Layered Perceptron. This technique gave better results than the individual tools that were clubbed together. Liu et al. [24] studied code smell prediction using deep learning. They experimented with four code smells and validated on multiple open-source projects. Their approach performed better than other approaches. Alazba et al. [25] investigated the application of the stacking ensemble and feature selection in code smell detection. 14 classifiers were applied individually and then stacked together. Three different stacking ensembles were used for experiments. Logistic regression and SVM as meta-classifiers resulted in better code smell detection. Kaur and Kaur [26] experimented with bagging and random forest ensemble learning and three feature selection techniques in various combinations. Random forest was the best performing classifier, and BFS was the best among feature selection methods. Pecorelli et al. [27] conducted an empirical comparison between six different data balancing techniques for code smell classification. It was observed that data balancing did not significantly increase classifiers accuracy.

Fontana et al. [13] presented a multinomial severity classification approach based on ML techniques for four code smells. They found that the severity classification of method level code smell could be done accurately, whereas there is further scope of improvement for the class level smell. Gupta et al. [28] developed an automated hybrid approach for assigning the severity of code smell. They used the CART model for the severity assignment based on the metrics distribution of positive instances. Zhang et al. [29] investigated six code smells and presented the order of refactoring based on the association of faults with code smells.

Our proposed work extends the study of Fontana et al. [13] (from here on 'reference study'). Class-level smells classification performance was lower than method-level smells ($\sim$ 75% and $\sim$ 90%, respectively) in their study. Our work aims to resolve this disparity and provide a

Table 1 Comparison of the proposed approach with various severity classification and refactoring prioritization studies

	Fontana et al. [13]	Gupta et al. [28]	Vidal et al. [16]	Proposed approach
Basis of severity assignment	Score based on negative impact on software quality as assessed by expert	Severity based on software metrics distribution of smelly instances	Refactoring order based on software developer's point of view and other factors	Score based on negative impact on software quality as assessed by expert
Issues addressed	Supervised traditional ML classification of various code smell based on their severity levels assigned by human expert	Detection of code smell in Kotlin and refactoring ordering of positive instances based on software metrics distribution	Prioritizing code smells using many software histories related details and information obtained after interviews with developers	Increasing classification performance of ML classifiers for various code smells using proposed SSHM approach
Issues not addressed	Disparity in classification performance of class and method level smell was not addressed	Developer/expert view was not taken into account for assigning severity index	Applied on a minimal set of software. A more extensive evaluation is pending	Historic details were not taken into account for severity classification
Methodology	Prepared multinomial severity dataset from 76 software and applied various traditional ML classifiers to it	Prepared Binary dataset from 30 software and applied CART model on it for smell identification and severity indexing	Three criteria i.e. History of updation, relevance of smell and modifiability scenarios, were used for the ordering of refactoring opportunities	Corrected dataset in the literature. Then proposed SSHM approach was applied to improve severity classification performance

hybrid model (SSHM) for code smell classification by combining Stacking and SMOTE to improve classification performance further. To the best of our knowledge, SMOTE and Stacking have not been applied in combination in code smell severity classification. A comparison of our proposed approach with various code smell severity classification and refactoring prioritization approaches are presented in Table 1.

3 Dataset

Four datasets have been taken from the reference study. God class (GC) and Data class (DC) are for class level smells, while Feature envy (FE) and Long method (LM) are for method level smells. Dataset was created using 76 Software from Qualitas corpus [30] release 20120401r. There were four levels used to represent four levels of severity, where 1 represents 'No smell' and 4 means 'Severe smell'. Each dataset consists of 420 instances. There were 63 features in class smell datasets and 84 features in method smell datasets.

3.1 Dataset correction

All four multinomial datasets were inspected for errors. Binary datasets of the same code smell and identical instances presented by the same researchers in their previous study [20] were also used for comparison. A significant number of inconsistencies between binary and multinomial datasets of GC and DC were identified during analysis. These discrepancies were rectified, and the remaining instances were evaluated for any misclassification and severity readjustment (from 2/3/4 to 2/3/4). Many advisors were employed during the correction phase, namely iPlasma [10], JSpIRIT [11] and PMD[1] for GC and iPlasma tool for DC. The final decision and severity label was assigned using expert opinion. Table 2 summarises the different corrections made for various reasons.

The composition of multinomial GC and DC datasets for severity labels 1,2,3,4 after corrections were 281, 9, 34, 96 and 274, 32, 77, 37, respectively. Numerous conflicting instances accounted for more than 75% of the correction in both datasets. After correcting datasets, an experiment was conducted to evaluate changes in the classifier's performance.

3.2 Experiment

This experiment applied identical pre-processing from the reference study to evaluate the extent of performance change in ML classifiers with corrected datasets. Jrip model was experimented with in Weka 3.8.5, and the rest in Python 3.8.8 using the scikit-learn library. A computer system with 8 GB RAM, Intel i5-9300H and windows 10 was used for this experiment. Hyperparameters were set manually based on the best setting achieved.

3.3 Performance evaluation with improved datasets

Five simple ML classifiers and their corresponding five Adaboost boosted methods were evaluated, namely

[1] https://github.com/pmd/pmd.

Table 2 Details of corrected instances

Reason	God Class	Data Class
A. Conflict between dataset	128	129
B. Misclassified as negative instance	005	012
C. Misclassified as positive instance	006	006
D. Severity reclassification	018	020
E. Non-conflicting changes (B + C + D)	029	038
Total (A + E)	157	167

Table 3 Average performance of classifiers trained on the corrected and original dataset

Performance metric	God Class		Data Class	
	OD	CD	OD	CD
Accuracy	0.67	0.88	0.68	0.82
Spearman's score	0.77	0.94	0.79	0.83
MSE	0.76	0.21	0.68	0.40

CD = Corrected Dataset, OD = Original Dataset

Decision Tree-Pruned (DT), Random Forest (RF), Libsvm-C-SVM-RBF (SVM), JRIP, Naïve Bayes (NB), B-Decision tree-Pruned (B-DT), B-Random Forest (B-RF), B-Libsvm-C-SVM-RBF (B-SVM), B-JRIP and B-Naïve Bayes (B-NB). Three performance criteria, namely Accuracy, Spearman's score, and mean square error (MSE) of the reference study were used for evaluation.

3.3.1 Results

The performance results of ML classifiers trained on the original dataset were directly taken from the reference study. SVM-RBF's performance was poor in it. As a result, the best SVM and B-SVM results were selected for comparison. Table 3 summarises the average performance of classifiers trained on corrected and original datasets.

All classifiers performed significantly better when trained with corrected datasets. It supports our claim that discrepancies hampered the classifier's learning. Detailed results of this experiment, corrected datasets and instance-wise evaluation of multinomial datasets of GC and DC can be accessed from here https://drive.google.com/drive/folders/16BqUdNlKNgdM_qrrJqGWKdQ_NfEdRPVD?usp=sharing. The performance of classifiers on class level smell is now comparable to method level smells ($\sim$ 90% peak accuracy). However, we see further scope of improvement as specific concerns such as imbalanced datasets, non-utilization of heterogeneous ensemble, etc., are still present.

4 Proposed approach: SMOTE-stacked hybrid model (SSHM)

In this section, our proposed model of SSHM is presented for better severity classification of four Code smells, namely GC, DC, FE and LM. SSHM is a hybrid approach formed by combining SMOTE with Stacking. SMOTE is employed to handle class imbalance problems. As all the datasets under consideration are imbalanced, balancing them may yield better performance. Stacking is an ensemble method for combining similar or different classifiers, potentially improving performance of ML models. Stacking provides exceptional customization ability, which is missed in most ensemble techniques. The only concern for Stacking is that it requires a larger dataset for adequate learning as learning is divided into two parts. First, base classifiers learn, followed by meta-classifiers. To prevent overfitting, instances of both learning parts need to be mutually exclusive. As a result, each classifier is given fewer instances to train on. In an imbalanced environment, SMOTE will increase samples available for training. Consequently, we anticipate combining SMOTE with Stacking will significantly improve performance in imbalanced environment. The proposed approach can be defined in the following steps:

Step 1: Select the Feature set
Step 2: Balance all four classes using SMOTE (label 1/2/3/4)
Step 3: Split the balanced dataset into training and testing data
Step 4: Create a Stack of base classifiers
Step 5: Select the meta-classifier and its parameters
Step 6: Train the Stacked ensemble and record the test results.

4.1 Experiment

Three experiments were conducted: SMOTE, Stacking and SSHM. In homogeneous Stacking, five instances of same classifiers were employed as base classifiers with significantly different hyperparameters. Each classifier was used only once in heterogeneous Stacking. All stacking models used Logistic regression as a meta-classifier. ML classifier's parameters were set manually based on the best setting achieved. fivefold cross-validation with ten repetitions was performed, and average performance was considered. Out of all features, some prominent features chosen for various code smells are as follows:

GC: AMWNAMM, LCOM5, LOCNAMM, LOCS_Package, NOMNAMM, NOI_Package, TCC, Is_Static_Type,

Table 4 Performance of classifiers on different approaches for four code smell

Code Smell and classifiers	Accuracy				Spearman's score (ρ)				MSE			
	F	S	ST	SSHM	F	S	ST	SSHM	F	S	ST	SSHM
God class												
DT	0.76	0.92	0.91	0.94	0.85	0.96	0.96	0.97	0.46	0.10	0.14	0.08
RF	0.75	0.93	0.93	0.97	0.83	0.97	0.98	0.98	0.53	0.07	0.08	0.04
SVM	0.60	0.90	0.91	0.93	0.67	0.94	0.95	0.96	1.05	0.13	0.14	0.10
NB	0.55	0.76	0.87	0.85	0.69	0.88	0.92	0.91	1.16	0.31	0.36	0.29
DNSR	–	–	0.92	0.95	–	–	0.97	0.98	–	–	0.09	0.05
B-DT	0.75	0.96	0.91	**0.98**	0.84	0.98	0.95	**0.99**	0.48	0.04	0.14	**0.02**
B-RF	0.74	0.96	0.92	**0.98**	0.85	0.98	0.96	**0.99**	0.43	0.04	0.11	**0.02**
B-SVM	0.58	0.89	0.92	0.93	0.64	0.95	0.95	0.94	1.11	0.13	0.15	0.24
B-NB	0.55	0.82	0.86	0.86	0.69	0.90	0.89	0.91	1.16	0.23	0.39	0.23
B-DNSR	–	–	0.91	**0.98**	–	–	0.95	**0.99**	–	–	0.19	**0.02**
Data class												
DT	0.76	0.89	0.85	0.92	0.86	0.93	0.88	0.94	0.43	0.17	0.28	0.14
RF	0.77	0.89	0.88	0.90	0.85	0.94	0.89	0.94	0.43	0.14	0.24	0.14
SVM	0.63	0.86	0.85	0.86	0.71	0.92	0.87	0.91	0.89	0.21	0.29	0.23
NB	0.55	0.69	0.75	0.71	0.71	0.76	0.59	0.77	1.11	0.59	0.94	0.57
DNSR	–	–	0.92	0.93	–	–	0.95	0.96	–	–	0.12	0.10
B-DT	0.73	0.96	0.86	**0.97**	0.84	0.98	0.90	0.97	0.47	0.05	0.25	0.07
B-RF	0.76	0.95	0.88	0.96	0.86	0.96	0.90	**0.98**	0.39	0.09	0.24	**0.05**
B-SVM	0.64	0.86	0.83	0.91	0.73	0.92	0.85	0.93	0.78	0.18	0.42	0.18
B-NB	0.55	0.83	0.83	0.86	0.71	0.90	0.84	0.89	1.11	0.25	0.39	0.31
B-DNSR	–	–	0.89	0.96	–	–	0.92	0.97	–	–	0.20	0.06
Feature envy												
DT	0.93	0.94	0.90	0.95	0.93	0.97	0.91	0.97	0.14	0.08	0.18	0.08
RF	0.91	0.95	0.91	0.96	0.93	0.97	0.92	0.98	0.15	0.07	0.15	0.05
SVM	0.69	0.97	0.90	0.97	0.66	0.97	0.91	0.98	0.50	0.27	0.20	0.05
NB	0.76	0.84	0.86	0.85	0.75	0.91	0.84	0.92	0.55	0.24	0.33	0.20
DNSR	–	–	0.92	**0.98**	–	–	0.94	0.98	–	–	0.13	0.04
B-DT	0.91	0.97	0.93	**0.98**	0.92	0.98	0.93	0.98	0.17	0.05	0.15	0.04
B-RF	0.91	0.97	0.92	**0.98**	0.92	0.98	0.93	**0.99**	0.15	0.05	0.12	0.03
B-SVM	0.83	0.97	0.90	**0.98**	0.81	0.97	0.93	**0.99**	0.50	0.27	0.14	0.03
B-NB	0.77	0.94	0.91	0.96	0.76	0.97	0.91	0.98	0.48	0.09	0.16	0.06
B-DNSR	–	–	0.91	**0.98**	–	–	0.94	**0.99**	–	–	0.12	**0.02**
Long method												
DT	0.88	0.91	0.92	0.95	0.95	0.96	0.98	0.98	0.18	0.10	0.08	0.05
RF	0.91	0.98	0.96	0.98	0.96	0.99	0.99	**0.99**	0.13	0.03	0.04	0.02
SVM	0.86	0.98	0.98	0.98	0.89	0.98	0.98	**0.99**	0.29	0.04	0.04	0.03
NB	0.79	0.81	0.89	0.89	0.80	0.88	0.89	0.94	0.57	0.30	0.29	0.18
DNSR	–	–	0.94	**0.99**	–	–	0.97	**0.99**	–	–	0.08	0.02
B-DT	0.91	0.98	0.94	**0.99**	0.96	0.98	0.98	**0.99**	0.14	0.04	0.08	0.02
B-RF	0.92	0.98	0.96	0.98	0.96	0.99	0.99	**0.99**	0.12	0.02	0.04	0.02
B-SVM	0.88	0.98	0.98	0.98	0.91	0.98	0.98	0.98	0.25	0.04	0.04	0.04
B-NB	0.85	0.93	0.93	0.96	0.88	0.96	0.96	0.98	0.35	0.10	0.11	0.05
B-DNSR	–	–	**0.95**	**0.99**	–	–	**0.99**	**0.99**	–	–	**0.05**	**0.01**

F Fontana et al. [13], *S* SMOTE, *ST* Stack, *SSHM* proposed SMOTE stack hybrid model

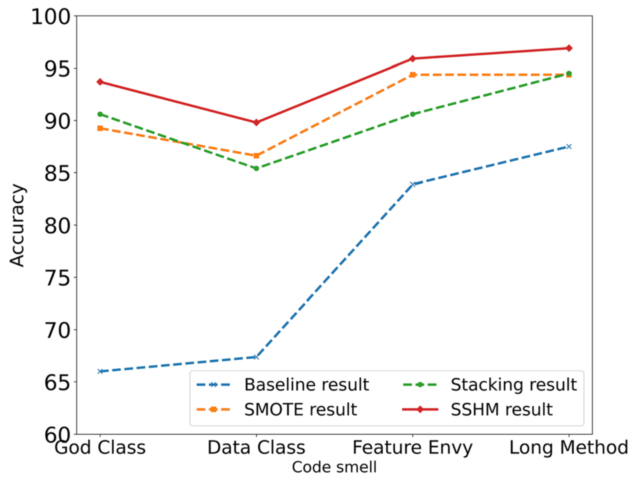

Fig. 1 Accuracy comparisons of proposed SSHM approach with baseline study, stacking and SMOTE on different code smells

number_constructor_DefaultConstructor_methods, number_constructor_NotDefaultConstructor_methods
DC: AMWNAMM, LCOM5, LOCNAMM, CFNAMM, NOMNAMM, ATFD, NIM, NMO, NOC, NOPA, WMCNAMM, CBO, RFC
FE: LCOM5, FDP, NOA, CFNAMM, FANOUT, ATFD, LAA, CLNAMM, TCC, CDISP, ATLD, CBO NOLV
LM: AMWNAMM, CYCLO, LOCNAMM, CFNAMM, LOC, MAXNESTING, LAA, CLNAMM, NOAV, Is_Static_Type, ATLD, AMW, NOLV.

4.2 Evaluation

The corrected datasets (GC and DC) and original datasets (FE and LM) were used for evaluation, and three performance metrics were employed: Accuracy, Spearman's score, and MSE. Four classifiers and their Adaboost boosted versions, namely DT, RF, SVM, NB, B-DT, B-RF, B-SVM, B-NB and one heterogeneous stacking ensemble (applicable for Stacking and SSHM only) and its Adaboost version DNSR (DT, NB, SVM and RF) and B-DNSR (B-DT, B-NB, B-SVM and B-RF) were evaluated in this experiment. The study conducted by Fontana et al. [13] is used as the baseline for the performance evaluation of our proposed approach. The baseline performance values have been directly taken from their study. In addition, the performance of SMOTE and Stacking in isolation have also been used for evaluation.

4.3 Results and discussion

Table 4 shows the comparative results of baseline studies and the proposed approach. The best results for each parameter are indicated with bold letters.

The average accuracy rate of classifiers on all four smells for the baseline, SMOTE, Stack and SSHM approach were 76.19, 91.13, 90.32 and 94.07, respectively. It provides a broad overview of how the proposed hybrid approach performed in comparison to other approaches. The heterogeneous SSHM classifier's performance (B-DNSR) was best among all classifiers with accuracy, spearman's score and MSE of 98%, 0.99 and 0.02, respectively. Figure 1 shows the accuracy comparison of the baseline study, SMOTE, Stacking, and SSHM approaches for different code smells. It is visible that the application of SMOTE and Stacking approach, when applied individually, improves the efficiency noticeably. However, by using them in combination as proposed in our approach, significantly better results were observed.

For all four code smells, it can be seen that the SSHM approach performs better than any other technique. These results suggested that the proposed SSHM approach provides superior severity classification. Applied with the appropriate classifiers, the proposed approach can give near-perfect ($\sim$ 98%) severity classification accuracy for all four code smells.

5 Conclusion

We began by addressing the disparity between classifiers peak accuracy of class ($\sim$ 75%) and method ($\sim$ 90%) level smell in literature. It was achieved by identifying various inconsistencies in class level datasets. After removing inconsistencies, ten ML classifiers were trained over the corrected datasets using similar experiment. The performance of these classifiers was then compared to the previous study. The results showed an increase in peak performance from approx. 75 to 90%, thereby removing performance disparity between class and method level smells.

However, the potential for further improvement motivated us to propose the SMOTE-Stacked hybrid model (SSHM) approach for the severity classification of four code smells. Our proposed approach was evaluated using ten ML classifiers. The SSHM approach was found to outperform the baseline approach. Compared to the baseline technique with a peak classification accuracy of 76–92%, the proposed SSHM approach had a peak classification accuracy of 97–99% for various code smells.

Acknowledgements We appreciate authors [13, 20] for making their datasets publicly available on https://essere.disco.unimib.it/machine-learning-for-code-smell-detection/ that provided us the means to conduct this study.

Declarations

Conflict of interest All authors certify that they have no affiliations with or involvement in any organization or entity with any financial or non-financial interest in the subject matter or materials discussed in this manuscript.

References

1. Gupta V, Chhabra JK (2012) Package level cohesion measurement in object-oriented software. J Braz Comput Soc 18:251–266. https://doi.org/10.1007/s13173-011-0052-4
2. Prajapati A, Chhabra JK (2020) Information-theoretic remodularization of object-oriented software systems. Inf Syst Front 22:863–880. https://doi.org/10.1007/s10796-019-09897-y
3. Fowler M (2018) Refactoring: improving the design of existing code. Addison-Wesley, Reading
4. Puchala SPR, Chhabra JK, Rathee A (2022) Ensemble clustering based approach for software architecture recovery. Int J Inf Technol. https://doi.org/10.1007/s41870-021-00846-0
5. Tufano M, Palomba F, Bavota G, Oliveto R, Di Penta M, Lucia AD, Poshyvanyk D (2015) When and why your code starts to smell bad. In: IEEE/ACM international conference on software engineering, IEEE, vol 37, pp 403–414.https://doi.org/10.1109/icse.2015.59
6. Zazworka N, Shaw MA, Shull F, Seaman C (2011) Investigating the impact of design debt on software quality. In: Proceedings of workshop on managing technical debt. ACM, pp 17–23. https://doi.org/10.1145/1985362.1985366
7. Shatnawi R, Li W (2006) An investigation of bad smells in object-oriented design. In: International conference on information technology: new generations, vol 3, pp 161–165. https://doi.org/10.1109/ITNG.2006.31
8. Yamashita A, Moonen L (2012) Do code smells reflect important maintainability aspects?. In: Proc. IEEE Int. Conf. Softw. Maintenance (ICSM), vol 28, pp 306–315. https://doi.org/10.1109/ICSM.2012.6405287
9. Lanza M, Marinescu R (2007) Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems. Springer Science and Business Media. https://doi.org/10.1007/3-540-39538-5
10. Marinescu C, Marinescu R, Mihancea P, Ratiu D, Wettel R (2005) iPlasma: an integrated platform for quality assessment of object-oriented design. ICSM 21:77–80
11. Vidal S, Vazquez H, Diaz-Pace JA, Marcos C, Garcia A, Oizumi W (2015) JSpIRIT: a flexible tool for the analysis of code smells. In: International conference of the Chilean Computer Science Society, vol 34, pp 1–6. https://doi.org/10.1109/SCCC.2015.7416572
12. Nongpong K (2012) Integrating 'Code Smells' detection with refactoring tool support. University of Wisconsin Milwaukee, Theses and Dissertations 13. http://dc.uwm.edu/etd/13
13. Fontana FA, Zanoni M (2017) Code smell severity classification using machine learning techniques. Knowl-Based Syst 128:43–58. https://doi.org/10.1016/j.knosys.2017.04.014
14. Moha N, Gueheneuc Y, Duchien L, Meur AL (2010) DECOR: a method for the specification and detection of code and design smells. IEEE Trans Softw Eng 36:20–36. https://doi.org/10.1109/TSE.2009.50
15. Palomba F, Bavota G, Penta MD, Oliverto R, Poshyvanyk D, Lucia AD (2015) Mining version histories for detecting code smells. IEEE Trans Softw Eng 41:462–489. https://doi.org/10.1109/TSE.2014.2372760
16. Vidal SA, Marcos C, Díaz-Pace JA (2016) An approach to prioritize Code smells for refactoring. Autom Softw Eng 23:501–532. https://doi.org/10.1007/s10515-014-0175-x
17. Fontana FA, Braione P, Zanoni M (2012) Automatic detection of bad smells in code: an experimental assessment. J Object Technol 11:1–38. https://doi.org/10.5381/jot.2012.11.2.a5
18. Maiga A, Ali N, Bhattacharya N, Sabané A, Guéhéneuc YG, Antoniol G, Aïmeur E (2012) Support vector machines for anti-pattern detection. In: Proceedings of IEEE/ACM international conference on automated software engineering, vol 27, pp 278–281. https://doi.org/10.1145/2351676.2351723
19. Barbez A, Khomh F, Guéhéneuc Y (2019) Deep learning anti-patterns from code metrics history. In: IEEE international conference on software maintenance and evolution (ICSME), vol 35, pp. 114–124. https://doi.org/10.1109/ICSME.2019.00021
20. Fontana FA, Mäntylä MV, Zanoni M, Marino A (2016) Comparing and experimenting machine learning techniques for code smell detection. Empir Softw Eng 21:1143–1191. https://doi.org/10.1007/s10664-015-9378-4
21. Nucci DD, Palomba F, Tamburri DA, Serebrenik A, Lucia AD (2018) Detecting code smells using machine learning techniques: are we there yet? In: IEEE international conference on software analysis, evolution and reengineering (SANER), vol 25, pp 612–621. https://doi.org/10.1109/SANER.2018.8330266
22. Guggulothu T, Moiz SA (2020) Code smell detection using multi-label classification approach. Softw Qual J 28:1063–1086. https://doi.org/10.1007/s11219-020-09498-y
23. Barbez A, Khomh F, Guéhéneuc YG (2020) A machine-learning based ensemble method for anti-patterns detection. J Syst Softw 161:110486. https://doi.org/10.1016/j.jss.2019.110486
24. Liu H, Jin J, Xu Z, Zou Y, Bu Y, Zhang L (2021) Deep learning based code smell detection. IEEE Trans Softw Eng 47:1811–1837. https://doi.org/10.1109/TSE.2019.2936376
25. Alazba A, Aljamaan H (2021) Code smell detection using feature selection and stacking ensemble: an empirical investigation. Inf Softw Technol 138:106648. https://doi.org/10.1016/j.infsof.2021.106648
26. Kaur I, Kaur A (2021) A novel four-way approach designed with ensemble feature selection for code smell detection. IEEE Access 9:8695–8707. https://doi.org/10.1109/ACCESS.2021.3049823
27. Pecorelli F, Nucci DD, Roover CD, Lucia AD (2020) A large empirical assessment of the role of data balancing in machine-learning-based code smell detection. J Syst Softw 169:110693. https://doi.org/10.1016/j.jss.2020.110693
28. Gupta A, Chauhan NK (2021) A severity-based classification assessment of code smells in Kotlin and Java application. Arab J Sci Eng. https://doi.org/10.1007/s13369-021-06077-6
29. Zhang M, Baddoo N, Wernick P, Hall T (2011) Prioritizing refactoring using code bad smells. In: IEEE international conference on software testing, verification and validation workshops, vol 4. https://doi.org/10.1109/icstw.2011.69
30. Tempero E, Anslow C, Dietrich J, Han T, Li J, Lumpe M, Melton H, Noble J (2010) The qualitas corpus: a curated collection of java code for empirical studies. Asia Pac Softw Eng Conf 17:336–345. https://doi.org/10.1109/APSEC.2010.46