The Journal of Systems and Software 210 (2024) 111968

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

On the effectiveness of developer features in code smell prioritization: A replication study☆,☆☆

Zijie Huang, Huiqun Yu *, Guisheng Fan *, Zhiqing Shao, Ziyi Zhou, Mingchen Li

Department of Computer Science and Engineering, East China University of Science and Technology, Shanghai, 200237, China

ARTICLE INFO

Keywords:
Code smell prioritization
Feature selection
Explainable AI
Replication study
Empirical software engineering

ABSTRACT

Code smells are sub-optimal design and implementation choices that hinder software maintainability. Although significant progress has been achieved in code smell detection, numerous results are perceived as trivial by developers. In response, a code smell prioritization approach capturing developer features has been proposed by a prior study (MSR'20), and it outperformed a code metric baseline (KBS). The conclusion was validated on a dataset collected from original developers, which includes their comments on code smell priority. However, the low presence of developer aspects in the comments is inconsistent with the performance improvement after involving such features. To explain the inconsistency, we replicate the two studies by exploiting different feature selection methods and a model explanation technique called SHAP. Our major findings are: (i) Correlation-based Feature Selection should not be used as a default method since it could harm Krippendoff's Alpha by up to 72%, (ii) if better feature selection is applied, pure code metrics from KBS outperform the MSR features in 3 smells by up to 45% in Alpha, and (iii) the behavior of code metrics based models have more agreement with developers' comments. We suggest exploiting different feature selection methods and using code metrics to prioritize the 3 change-insensitive smells.

1. Introduction

Code quality is a major concern of software developers since they are closely related to software quality. However, to deliver software before deadlines, trade-offs are usually made between code quality and speed of delivery (Pecorelli et al., 2020). Code smells (*i.e.*, sub-optimal code implementation and design choices Fowler et al., 1999) are consequences of such trade-offs which can hinder software maintainability and reliability in the long run (Palomba et al., 2018b). Code smell detectors have been actively developed in the last 2 decades (Sobrinho et al., 2021), and they have achieved promising results detecting major code smells in various granularities (*e.g.*, statement Saboury et al., 2017, method, class, and package Moha et al., 2010) and types (*e.g.*, structural and textual Palomba et al., 2018c).

Practitioners need to focus on removing the worst code smells in advance (Pecorelli et al., 2020; Sae-Lim et al., 2017) since Software Quality Assurance (SQA) resources are limited. However, automatic detection tools of code smells may produce an excessive number of results. Manual inspection of every result would be time-consuming, but few of them are of high priority. Consequently, both rule and machine learning based automatic code smell detectors are perceived as unhelpful by practitioners (Pecorelli et al., 2020; Ferreira et al., 2021; Sae-Lim et al., 2018a). To improve the developers' acceptance of code smell detectors, prior studies focused on improving their adaptation to the developers' perceptions. They built machine learners capturing structural (Fontana and Zanoni, 2017; Pecorelli et al., 2020; Sae-Lim et al., 2018a) (*e.g.*, coupling, cohesion, complexity) and contextual (*e.g.*, error-proneness Sae-Lim et al., 2018a, change history, developer Pecorelli et al., 2020) information to rank the results.

The replicated MSR'20 paper (Pecorelli et al., 2020) presented major progress in code smell prioritization which proposed a developer-driven and machine learning based approach to rank 4 code smells including Blob, Complex Class, Spaghetti Code, and Shotgun Surgery. In terms of novelty, it introduced the concept of "developer-driven" which

☆ This work was partially supported by the National Natural Science Foundation of China (No. 62372174), the Natural Science Foundation of Shanghai, China (No. 21ZR1416300), the Capacity Building Project of Local Universities Science and Technology Commission of Shanghai Municipality, China (No. 22010504100), the Research Programme of National Engineering Laboratory for Big Data Distribution and Exchange Technologies, China, and the Shanghai Municipal Special Fund for Promoting High Quality Development, China (No. 2021-GYHLW-01007).

☆☆ Editor: W. Eric Wong.

* Corresponding author.
E-mail addresses: hzj@mail.ecust.edu.cn (Z. Huang), yhq@ecust.edu.cn (H. Yu).

[1] https://conf.researchr.org/home/msr-2020.

https://doi.org/10.1016/j.jss.2024.111968
Received 11 April 2023; Received in revised form 2 January 2024; Accepted 7 January 2024
Available online 8 January 2024

referred to (1) the features proposed are related to developers, *i.e.*, they include development process and developer experience aspects, (2) the dataset was collected from original developers with their comments included. In terms of paper influence, it was published in a premier venue for data science and machine learning in software engineering,[1] cited by 40 times according to Google Scholar, and is authored by the most influential scholars in code smell research (see node 9#, 14#, and 16# in Fig.12 in Sobrinho et al. (2021)).

The MSR paper used the model constructed using the feature generation method of the KBS paper (Fontana and Zanoni, 2017) as the baseline method. The KBS paper exploited multiple automatic static analysis tools to calculate software metrics such as cohesion, coupling, complexity, and size. The major conclusion of the MSR paper is that it suggested building prioritization models based on mixed features (*i.e.*, developer, process, and code metrics) instead of using pure code metrics from the compared KBS baseline.

However, after manually investigating the comments of developers available online,[2] we find that some developer-related factors are rarely presented in the comments. Unexpectedly, the MSR paper claimed involving features related to such factors improved model performance significantly. **Explaining such inconsistency is the motivation of our research.**

Thus, we replicate the MSR paper to discover the cause of the inconsistency between developer comments and performance improvement. We suspect inappropriate feature selection harmed the performance of the baselines, and led to the result that significant improvement is achieved by involving new features. To verify our assumption, first, we apply the state-of-the-art feature selection methods mentioned in prior work (Zhao et al., 2021; Jiarpakdee et al., 2018, 2020) on the MSR dataset and its KBS baseline to assess their ability to mitigate multicollinearity. Then, we use the selected features to build prediction models and to evaluate their impact on model performance based on additional performance metrics suitable for human rating assessment. Finally, we exploit XAI techniques to explain the best-performed model, and we manually assess the agreement between the model behaviors and the developers' comments.

The major findings in this replication study include:

(1) In terms of data preprocessing, inappropriate feature selection may lead to biased results. Correlation-based Feature Selection (CFS) should not be used as a default method, since it could negatively impact performance by up to 34% and 72% in AUC-ROC and Krippendorff's Alpha.

(2) In terms of model construction, pure code metrics could be better in the prioritization of 3 smells except for Shotgun Surgery, which indicates we may not need additional features to correctly prioritize change-insensitive code smells.

(3) In terms of performance evaluation, classification metrics may be biased for ordinal rating tasks, *i.e.*, such metrics ignore the distance between the predicted value and the actual label. Metrics such as Krippendorff's Alpha should be reported as well (Tian et al., 2016).

(4) In terms of model explainability, our manual verification shows the agreement between developers and the model built with pure code metric dataset is higher.

Replication Package. We provide an executable code capsule verified by CodeOcean[3] including the datasets and model training code with all parameters included for performance replication. The details are available in subsection A of the Appendix A. [We also provide an executable example[4] for demonstrating XAI for code smell prioritization.

The rest of this paper is organized as follows. In Section 2 we summarize related work. Section 3 presents the background and datasets, while Section 4 outlines the research questions and methodologies. In Section 5 we discuss the results, while Section 6 overviews the threats to the validity. Finally, Section 7 concludes the paper and describes future research.

2. Related work

This section describes studies related to code smell detection and prioritization, as well as XAI empirical studies in Software Quality Assurance (SQA).

2.1. Automatic Java code smell detection

The term code smell was coined by Fowler et al. (1999) to empirically describe anti-patterns in software development. Most research on code smell detection is aimed at quantifying empirical observations of smells on Java open-source projects.

Rule and Heuristic-Based Classical Approaches. Moha et al. (2010) proposed DECOR using Rule Cards (*i.e.*, a set of customizable rules with code metrics and thresholds presented in a file). Lanza et al. (2005) outlined detection approaches as well as thresholds statistically calculated in commercial systems. Furthermore, Tsantalis et al. (2018) developed JDeodorant aiming to detect and refactor code smells at the same time with the help of Integrated Development Environment (IDE) based plugins. Practically, some of the above-mentioned research outcomes have already been integrated into popular industrial tools such as SonarQube[5] and PMD.[6]

Change History Based. Such approaches were proposed to cope with the shortage of pure code metrics. Palomba et al. (2015, 2013) proposed HIST to capture historical changes in software systems, and they outperformed pure code analysis techniques in 5 code smells including Blob and Shotgun Surgery.

Textual Analysis Based. After HIST was proposed, the authors also designed a textual Information Retrieval (IR) approach called TACO (Palomba et al., 2016) to capture the conceptual smells, their research showed textual smells are significantly different from the structural ones (Palomba et al., 2018c).

Recently, Ichtsis et al. (2022) found that even for simple code smells, threshold-dependent methods may output unreliable results, and the agreement among tools is also low. Since most of the upmentioned approaches are threshold-dependent, researchers intend to explore if threshold-free machine learning could detect code smells.

Conventional Machine Learning Based. A recent study by Alazba and Aljamaan (2021) suggested machine learning could achieve nearly perfect performance. However, they used a dataset generation strategy criticized by Di Nucci et al. (2018). Azeem et al. (2019) and Di Nucci et al. (2018) found potential flaws in the dataset construction (*e.g.*, biased and impractical dataset, the lack of process metrics) and validation techniques (*e.g.*, the absence of metrics such as AUC-ROC which are insensitive to data distribution). In response, Jain and Saha (2021) used the modified dataset of Di Nucci et al. (2018) with hybrid feature selection, data balancing, and ensemble learning approaches which drastically improved performance.

Deep Learning Based. Deep learning methods were criticized by Fakhoury et al. (2018) since they were hard to interpret (*i.e.*, black-boxed), and achieved limited or no improvement while consuming a lot more computational resources. However, since they could save the effort of feature engineering, researchers were still actively improving them. Sharma et al. (2021) exploited rule-based detection tools to generate datasets for prediction, and represent code snippets using token-based approaches as inputs for deep learning and transfer learning. The reliability of the tool-generated results was validated by two

[2] https://figshare.com/s/94c699da52bb7b897074.
[3] https://doi.org/10.24433/CO.9879914.v2.
[4] https://github.com/escapar/JSS23_demo].
[5] https://sonarqube.org/.
[6] https://pmd.github.io/.

raters having a reasonable software engineering background. Liu et al. (2021, 2018) designed approaches for synthesizing code smell data (*e.g.*, randomly moving methods) based on real-world data for data augmentation, which achieved better performance in smell detection such as God Class, and they achieved better performance than classical baselines. However, synthesizing data for smells with more complex causes (*e.g.*, Shotgun Surgery) continues to be a challenge.

To conclude, detection results are upstream sources of the prioritization datasets, and related studies have achieved ideal performance. However, there still exist challenges regarding the practical usage and the validity of the results. We suggest manually verifying the result of automatic detection tools before they are used to train practical models.

2.2. Code smell prioritization

The studies that prioritized code smell detection results considered various aspects to foster effective refactoring.

Code Quality. Fontana et al. (2015) proposed various intensity indexes for code smells using combinations of code metric values. Furthermore, Fontana and Zanoni (2017) proposed the KBS baseline of our replicated paper using pure code metrics as features to predict code smell severity.

Task Relevance. Sae-Lim et al. (2017, 2018b) investigated the developers' perceptions toward code smell priority, and they revealed that task relevance and smell severity were the most important factors. Thus, they built prioritizing models (Sae-Lim et al., 2016; Sae-Lim et al., 2018a; Sae-Lim et al., 2017) capturing context-based information from Issue Tracking Systems (ITS) using IR-based textual similarity between issue reports and code components. Since their primary concern for prioritization was task relevance (*i.e.,* code related to more issue reports is more important), they validated the results using a strategy of evaluating recommending system (*e.g.*, measuring nDCG as task relevance) and manual verification.

Developer and Process. Our replicated study (Pecorelli et al., 2020) used structural code metrics and process metrics of software systems to measure the priority, *i.e.*, they focused on present and historical information of code rather than the relationships with other software artifacts. More details are introduced in Section 3. Vidal et al. prioritized code smells (Vidal et al., 2016; Guimarães et al., 2018) and their agglomerations (Vidal et al., 2019) according to developers' preference (*e.g.,* prefer to improve coupling or cohesion), their impact on software architecture, agglomeration size, and change histories.

Inter-Smell Relationship. Liu et al. (2012) arranged resolution sequences of code smells to save effort for refactoring. Palomba et al. (2017, 2018a) mined association rules among code smells to reveal their co-occurrence, and they suggested focusing on method smells that are likely to disappear along with the class-level ones. Inspired by their studies, we proposed a refactoring route generation approach (Wang et al., 2021) based on the association rules for eliminating Python code smells.

2.3. Application of XAI in SQA

In response to the developers' expectations and regulations (Perera et al., 2019; Jiarpakdee et al., 2021), decisions made by black-boxed models should be explained. To enhance model trustworthiness and explainability, XAI has become a major concern in SQA practice.

From our observation, the most trending model explanation technique in code smell related papers is a Classifier Agnostic (CA) global explanation approach called Information Gain (Pecorelli et al., 2020; Palomba et al., 2019; Catolino et al., 2020; Palomba and Tamburri, 2021; Huang et al., 2022; Palomba et al., 2021) measuring the reduction of uncertainty after involving a new feature. In other SQA tasks such as defect prediction, the application of XAI techniques is more diverse.

Table 1
Statistics of projects in the datasets.

Project	Number of commits	Number of developers	Number of classes	KLOC
Mahout	3054	55	813	204
Cassandra	2026	128	586	111
Lucene	3784	62	5506	142
Cayenne	3472	21	2854	542
Pig	2432	24	826	372
Jackrabbit	2924	22	872	527
Jena	1489	38	663	231
CDT	5961	31	1415	249
CXF	2276	21	655	106

Tantithamthavorn et al. investigated empirically the impact of class rebalancing (Tantithamthavorn et al., 2020), parameter tuning (Tantithamthavorn et al., 2019), and feature selection (Jiarpakdee et al., 2020) on the performance and explanations of defect prediction models. Furthermore, Rajapaksha et al. (2022) also proposed a planner for SQA using model explanation techniques in order to be responsive to the practitioners, *i.e.*, telling developers how to react to reduce the error-proneness of a class.

In terms of the validity of XAI techniques, Rajbahadur et al. (2021) confirmed a high agreement could be reached between the feature importance generated by a CA approach (*i.e.*, SHapley Additive exPlanation, also known as SHAP Lundberg and Lee, 2017) and the Classifier Specific (CS) ones only if multicollinearity is mitigated to a great extent. They also found CS feature importance is not stable and easily affected by feature interactions, however, CA feature importance is more robust. Thus, we use CA feature importance in this study. While Jiarpakdee et al. (2022) found the feature importance generated by another CA approach (*i.e.*, LIME Ribeiro et al., 2016) was perceived as useful by most developers. Our research (Yang et al., 2021) on the consistency of XAI outcomes of Just-In-Time defect prediction models also confirmed a high agreement among several established local explanation based CA approaches including LIME, SHAP, and BreakDown (Gosiewska and Biecek, 2019).

To generate reliable explanations, we should follow these empirical guidelines to build a prediction model: (1) multicollinearity should be mitigated using feature selection, (2) the model should perform well (*e.g.,* AUC-ROC > 0.7), (3) data resampling should be avoided, and (4) feature importance should be calculated by CA approaches.

3. Background and datasets

This section describes the dataset we used for replication as well as the background of our replicated study.

3.1. Dataset collection

The dataset contains a severity level of 4 code smells. All 4 smells are class-wide design problems related to coupling, cohesion, and complexity. They cause a high cognitive load for developers to comprehend, maintain, and refactor the code. The replicated paper (Pecorelli et al., 2020) chose these 4 smells because they were common and can be accurately assessed by developers with respect to their criticalities.

Blob (or God Class). Classes with low cohesion and do not follow the single responsibility principle. Blobs could be detected by cohesion code metrics such as LCOM5 (Lack of COhesion of Method) (Moha et al., 2010) and size metrics such as WMC (Weighted Method Count) (Palomba et al., 2019).

Complex Class. Classes with high complexity, *e.g.*, too many loops and conditional control statements. Complex Classes could be determined by complexity code metrics such as CYCLO (Brown et al., 1998) and code readability (Buse and Weimer, 2009).

Spaghetti Code. Classes do not follow Object-Oriented Programming (OOP) principles, *e.g.*, a container of long methods that do not interact with each other. Spaghetti Code could be detected by size metrics such as LOC (Line of Code) as well as the absence of inheritance (Moha et al., 2010).

Shotgun Surgery. Classes trigger small co-changes of other classes frequently. Shotgun Surgery refers to high coupling. However, it can be better detected by historical code change information (Pecorelli et al., 2020; Palomba et al., 2015, 2013) rather than code metrics. Shotgun Surgery is a change-sensitive smell because variations in code change metrics will greatly increase the severity of Shotgun Surgery. The severity of the other 3 smells investigated will not be directly influenced by frequent code changes.

The authors tracked the commits of 9 established projects of Apache and Eclipse open-source foundations in 6 months, and the information on the 6 projects is listed in Table 1. The authors used rule-based detectors to identify code smells daily, and they manually discarded false positives. Afterward, they sent emails to the original developers to collect their perceptions of the criticality of smells as soon as possible. The developers' perceived criticalities of the MSR paper originally range from 1 to 5. Since the margins of criticality levels {1, 2} and {4, 5} were not clear, the MSR paper (Pecorelli et al., 2020) merged the unclear criticalities to 3 new criticalities, *i.e.*, {`NON-SEVERE, MEDIUM, SEVERE`}. Thus, the prediction was performed over the merged criticalities. Finally, they received 1332 instances almost equally distributed among the 4 smells.

The authors also provided an online appendix[7] containing the original developers' comments. The online appendix contains short comments of original developers describing their attitudes toward the criticality of every code smell instance. However, we find 5 comments were missing from the relevant folder named after code smells, and thus they are discarded from the model explanation. Consequently, we are using 1327 samples in our replication study.

3.2. Feature generation

The MSR paper generated 20 features including product, process, developer, and code smell related features using automatic tools. The features are listed in the 3rd column of Table 2, and the detailed descriptions are available in Table 2 of the MSR paper (Pecorelli et al., 2020).

To compare the models built with the proposed features and the pure code metrics, the MSR paper used the KBS dataset as a baseline. They applied the dataset generation method of the original KBS paper to construct the baseline based on the 9 projects mentioned in the last section. The features are listed in the 4th column of Table 2. The features of the baseline were 61 pure code metrics in 3 granularities including class, package, and project. Method level metrics were discarded since method smells were not considered. The aspects of these features include size, complexity, coupling, inheritance, and encapsulation. Their descriptions are available in the online appendix of the KBS paper.[8]

3.3. Experimental settings

Validation Strategy. The MSR and the KBS dataset were all evaluated in the same context (*e.g.*, stratified 10-fold cross-validation Kohavi, 1995, no data balancing). The authors exploited cross-project prediction, *i.e.*, the dataset of each smell was constructed regardless of the project of the affected classes.

Feature Selection and Multicollinearity Mitigation. The authors applied CFS to mitigate multicollinearity which may hinder the interpretability of models. Note that collinearity (*i.e.*, correlation between pairs of features Jiarpakdee et al., 2020) and multicollinearity (*i.e.*, collinearity among multiple features Jiarpakdee et al., 2020) were not assessed and reported after CFS was exploited.

Prediction and Parameter Tuning. The authors tried multiple classifiers including Random Forest (RF), Logistic Regression, Vector Space Machine, Naive-Bayes, and Multilayer Perceptron with hyper-parameters configured by exploiting the Grid Search algorithm. RF was the best classifier.

Performance Assessment. The authors assessed the performance of the models using metrics for classifiers including AUC-ROC, MCC, and F-Measure. AUC-ROC and MCC are more reliable since they are not threshold-dependent and insensitive to imbalanced data (Tantithamthavorn et al., 2017; Yao and Shepperd, 2020). Meanwhile, it is a convention to report the F-Measure performance.

3.4. Developers' comments and our manual verification

We manually checked the developers' comments to verify if they reflect the following 4 aspects, ***i.e.*, development process (PROC), developer's perception (DEV), inner-class issues (INNER), and cross-class issues (CROSS)**. Their distributions are displayed in columns 1 to 4 in Table 3. The 5th column in Table 3 presents the distribution of all smell instances. The reason we involve PROC and DEV is that they are the major reasons that lead to performance improvement claimed by the replicated paper. The reason we consider INNER and CROSS aspects is that they are higher-level descriptions of common code quality characteristics that class-level code metrics measure (*e.g.*, coupling is a CROSS issue, while cohesion is an INNER issue). Specifically, we find that the word "complex" is ambiguous in developers' comments, which is not identical to the definition of the "complex" measured in complexity metrics. Although "complex" in the view of developers represents that they can hardly understand the code, however, they may either refer to complex control flow or complex interactions with other code components. The category of related comments is assigned according to their context (*e.g.*, whether coupling or cohesion is mentioned) and the smells they are referring to (*e.g.*, complex in Shotgun Surgery is related to the CROSS issue).

The manual identification is performed independently by the 1st author (Ph.D. student, Java developer with 5 years of experience) and the 5th author (Ph.D. student studying code summarization). Later, we involve the 6th author (Ph.D. student studying code generation) to validate the results together with the 2 authors. The initial agreement rate is 75.05%. Apart from validated misunderstanding and mislabeling, most of the inconsistencies appear in whether developers have expressed their own opinions toward the criticality, and whether the comments refer to "legacy code" expressed PROC factors. Finally, we reach an agreement that legacy code could be captured by process metrics.

To clarify, we are not using the grounded theory approach to define categories completely based on the developers' comments because the aim of this study is replication rather than a fine-grained analysis of developers' attitude toward code smells criticality, and the present category partitioning is closer to the design and the claimed improvement of the replicated paper.

3.5. The conclusions to replicate

Dramatically, process- and developer-related factors are almost never present in Spaghetti Code and Shotgun Surgery comments, which shapes our motivation to validate the controversial conclusions with respect to our observations. In the next sections, we replicate the following 3 conclusions of the MSR paper.

[7] https://figshare.com/s/94c699da52bb7b897074.

[8] https://essere.disco.unimib.it/wp-content/uploads/sites/71/2019/04/metric-definitions.pdf.

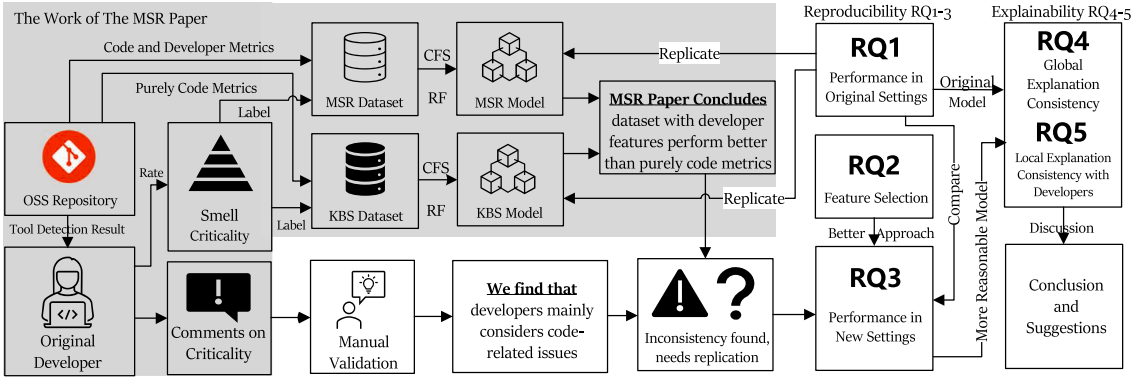

Fig. 1. Demonstration of experimental settings.

Table 2
Features, examples of developer comments, and their reflected aspects.

Aspect	Comment example	MSR features	KBS features
Inner-class (INNER)	This is a complex class with too many methods. Methods are poorly cohesive	LCOM5, C3, RFC, WMC, Read.	LCOM5_type, TCC_type, RFC_type, NMO_type, WMC_type, WOC_type, LOC_*, num*, NO* (* is a wildcard for any characters)
Cross-class (CROSS)	High coupling, the code has became complex to be understood.	CBO, MPC	CBO_type, ATFD_type, CFNAMM_type, FANOUT_type, DIT_type
Developer's perception (DEV)	The original developers are not anymore contributing so it is hard to refactor it.	OWN, EXP, NCOM	
Development process (PROC)	The class is rarely modified, this is not in our list of things to do.	AVG_CS, NC, NF, CE, NR, NCOM, EXP, DSC, OWN, Pers	

Table 3
Aspects of contents presented in the comments of developers.

	PROC	DEV	INNER	CROSS	Number of instances
Blob	66	175	206	2	341
Complex Class	37	164	304	3	349
Spaghetti Code	0	5	307	2	311
Shotgun Surgery	4	3	125	225	326
Total	107	347	942	232	1327
Proportion%	8.06	26.15	70.99	17.26	100

C1. The prioritization model using the MSR dataset (mixed features) outperformed the one using the KBS dataset (pure code metrics) by up to 17% of AUC-ROC in all 4 code smells. The MSR model was significantly superior in prioritizing 3 code smells other than Spaghetti Code.

C2. Shotgun Surgery is the most challenging smell to prioritize.

C3. The developers' perceived criticalities could be better explained using mixed features rather than using pure code metrics.

4. Research questions and methodologies

The experimental process is depicted in Fig. 1. The part with gray background represents the work of the MSR paper, while the white parts are experiments conducted in our study.

The ***goal*** of our study is to validate whether pure code metrics (KBS) are inferior compared to mixed features (MSR) if better experimental settings (*e.g.*, feature selection) are applied, with the ***purpose*** of validating whether some conventional processes of building code smell related machine learning models are reliable. Our ***perspective*** is of both researchers and practitioners: our replication could help the former to adopt more reasonable features to build models, while we can also provide explanations for every instance of the predicted classes for the latter. To these ends, we propose the following 5 research questions.

4.1. Research questions

RQ1: Can we replicate the model performance using MSR and KBS datasets in the context of the MSR paper?

Motivation. This RQ aims to replicate the MSR paper to present the basic performance of the models built with the MSR and the KBS datasets.

Approach. We use the experimental settings of the MSR paper and exploit the implementation of Scikit-Learn (Pedregosa et al., 2011) for building classifiers, *i.e.,* we exploit cross-project prediction, apply feature selection, and perform grid search over parameters of multiple evaluated classifiers. Specifically, we perform CFS independently for every code smell. Moreover, we report the performance before and after feature selection. We also involve 2 more common performance metrics related to regression and rating recommended in related empirical study (Tian et al., 2016), *i.e.*, Krippendorff's Alpha (Krippendorff, 1970), and Cohen's Kappa (Cohen, 1960), and they will be described in Section 4.2 in detail. We exclude Precision and Recall to avoid reporting an excessive number of metrics since classification metrics could summarize them. We also exploit the Wilcoxon Ranksum Test and Cliff's Delta to compare the prediction results in 10 folds using CFS and without any feature selection, to verify if there exists a significant difference with non-negligible effect size.

RQ2: Is there a better choice as a default option for feature selection that guarantees stable performance?

Motivation. The MSR paper used CFS as the feature selection method by default since CFS is known for its stable performance of correlation removal. However, a valid default option should guarantee

good or best performance in most cases. We think the inconsistency between our observation on comments and the original conclusions (*e.g.*, ***C1*** and ***C3***) may be caused by feature selection. Since CFS was reported unideal in a recent defect prediction empirical study (Jiarpakdee et al., 2020), we intend to investigate whether CFS is the most stable method in terms of both multicollinearity mitigation and the impact on model performance for code smell prioritization.

Approach. We exploit the feature selection methods applied in Zhao et al. (2021) and AutoSpearman (Jiarpakdee et al., 2020, 2018). These methods will be described in Section 4.3. Then, we measure the proportion of the feature pairs with collinearity presented (Spearman's $\rho > 0.7$ Jiarpakdee et al., 2020) and any feature with multicollinearity presented (VIF > 5 Jiarpakdee et al., 2020). Finally, the results of multicollinearity mitigation and the impact on performance are tested by SK-ESD (Scott-Knott Effect Size Difference) (Tantithamthavorn et al., 2017; Tantithamthavorn et al., 2019), which groups and ranks the values of performance metrics. The above-mentioned statistical approaches will be described in Section 4.4.

> **RQ3:** Can we achieve better performance through better feature selection which mitigates collinearity and multicollinearity?

Motivation. This RQ is designed to validate ***C1*** and ***C2*** in our context. Since RQ2 reveals the overall performance of feature selection methods, we evaluate their actual impact on model performance with respect to RQ1.

Approach. We pick the best-performed feature selection algorithm for every code smell and present the result and the variations in terms of model performance. We then exploit the Wilcoxon Ranksum Test and Cliff's Delta to compare the prediction results in 10 folds using the best-performed feature selection and without any feature selection.

> **RQ4:** Based on features selected in RQ3, can we generate global feature importance similar to the MSR paper?

Motivation. This RQ is designed to validate the dependability of ***C3*** in terms of global explanation. Since multicollinearity is mitigated, we could exploit XAI technique to validate the consistency of our conclusion with ***C3***.

Approach. First, we generate global feature importance, *i.e.*, the mean of absolute values of SHAP feature importance (which will be introduced in Section 4.5). Afterward, we compare our conclusion with ***C3***.

> **RQ5:** Based on features selected in RQ3, which dataset could better reflect the opinions of the original developers toward code smell criticality?

Motivation. This RQ is designed to validate the dependability of ***C3*** in terms of local explanation. Since SHAP could generate local explanations for every predicted instance, we intend to check the agreement of the model behavior with the developers' perceptions to reveal which dataset is more ideal. We use the opinions of developers to judge whether applying certain features is feasible because a recent study Aleithan (2021) found that developers will not be likely to trust a model's explanation if it is completely unexpected or misses some key preferences they are expecting, which is in line with other XAI studies saying people preferred the explanations consistent with their prior knowledge Maltbie et al. (2021). We also believe that following the decision of the original developer would be helpful in making a more reliable prioritization closer to the ground truth.

Approach. We classify the features into the 4 categories presented in Table 2. Then, we check the consistency of the categories of the top-ranked features and the categories of the developers' perception. In terms of consistency, we refer to (1) **perfect match**, *i.e.*, the top-ranked features that SHAP derive are in exactly the same categories with respect to the developers' comments covered categories, (2) **partial match**, *i.e.*, we measure the proportion of the matched categories and do not restrict all categories to be the same for each predicted Java class, and (3) **Kappa and Alpha agreement** treating the model and the original developer as two individual raters, and data showing whether their explanations appear in categories are used as input. For each category, the rating is 1 if an explanation falls in a category, and 0 otherwise. Note that we only check if any categorized factor was presented instead of identifying its positive or negative impact on criticality because some comments were ambiguous. **The features' categories and a detailed example of agreement calculation are available in the** Appendix B.

4.2. Performance evaluation metrics

AUC-ROC, MCC, and F-Measure are common performance metrics for classification tasks, where larger values indicate better performance. AUC-ROC ranges from 0 to 1, where a value of 0.50 indicates a model's performance is equivalent to random guessing, and a value greater than 0.70 indicates good performance (Jiarpakdee et al., 2020). F-Measure ranges from 0 to 1, where a value greater than 0.70 indicates ideal performance. MCC ranges from −1 to 1, and based on the comparison between F-Measure and MCC values (see Figure 2 in Yao and Shepperd (2020)), a value greater than 0.50 for MCC indicates good performance. In terms of the definitions of the metrics, AUC-ROC is calculated as the area under the TPR–FPR curve. The equations of TPR, FPR, MCC, and F-Measure are listed in (1) to (4).

$$TPR = \frac{TP}{TP + FN}, \tag{1}$$

$$FPR = \frac{TN}{FP + TN}, \tag{2}$$

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}, \tag{3}$$

$$F - Measure = \frac{2 \times TP}{2 \times TP + FP + FN} \tag{4}$$

where *TP* is for true positive (positive sample predicted as positive), *FN* is for false negative (positive sample falsely predicted as negative), *TN* is for true negative, and *FP* is for false positive.

Krippendorff's Alpha measures inter-raters' agreement (higher is better), which is different from classification metrics since it takes the distance between the predicted and real criticalities into account (Tian et al., 2016). We involve them as a related study in bug report prioritizing (Tian et al., 2016) suggested. Alpha > 0.66 indicates a reasonable agreement. The equation for calculating Alpha is listed in (5).

$$\alpha = 1 - \frac{D_o}{D_e} \tag{5}$$

D_o is the observed disagreement between code smell criticality assigned by the original developers, and D_e is the disagreement expected when the rating of code smells can be attributed to chance rather than due to the inherent property of the code smells themselves, their calculation is listed in (6) and (7).

$$D_o = \frac{1}{n} \sum_c \sum_k o_{ck}\, {}_{metric}\delta^2_{ck} \tag{6}$$

$$D_e = \frac{1}{n(n-1)} \sum_c \sum_k n_c \cdot n_k\, {}_{metric}\, \delta^2_{ck} \tag{7}$$

where o_{ck}, n_c, n_k and n refer to the frequencies of values in the coincidence matrices, and ${}_{metric}\delta^2_{ck}$ refers to any difference function that is appropriate to the *metric* (i.e., levels of measurement for comparison) of the given data. We use the ordinal metric[9] ${}_{ordinal}\ \delta^2_{ck}$ to calculate

[9] https://repository.upenn.edu/asc_papers/43.

Table 4
Feature selection methods applied.

Family	Type	Name or Strategy	Abbreviation	Thresholds
Filter-based ranking	Statistics-based	Chi-Square	ChiSq	Select {15, 45, 75}%
		Correlation	Corr	Select {15, 45, 75}%
	Probability-based	Probabilistic Significance	Sig	Select {15, 45, 75}%
		Information Gain	IG	Select {15, 45, 75}%
		Gain Ratio	Gain	Select {15, 45, 75}%
		Symmetrical Uncertainty	Symm	Select {15, 45, 75}%
	Instance-based	ReliefF	Relief	Select {15, 45, 75}%
	Classifier-based	One-Rule	OneR	Select {15, 45, 75}%
		SVM	SVM	Select {15, 45, 75}%
Filter-based subset	Correlation-based	Best First(BF), GreedyStepwise(GS)	CFS	
	Consistency-based	BF, GS	Consist	
Wrapper-based subset (Wrap)	Nearest Neighbor	BF, GS	KNN	K = 1
	Logistic Regression	BF, GS	Log	
	Naive Bayes	BF, GS	NB	
	Repeated Incremental Pruning	BF, GS	JRip	
Hybrid	VIF- and Correlation-based	AutoSpearman	AutoSpearman	VIF = 5, $\rho = 0.7$
None			None	

inter-annotator agreement, since the criticalities could be transferred to ordinal values (*e.g.*, {0, 1, 2} for {NON-SEVERE, MEDIUM, SEVERE}). α is ranged in 0 and 1. $\alpha = 1$ indicates perfect agreement between developer and model. When $\alpha = 0$ the agreement is no better than random guessing.

The calculation of Kappa is listed in Eq. (9),

$$\kappa = \frac{p_o - p_e}{1 - p_e} \tag{8}$$

where p_o is observed agreement between two raters, calculated as the proportion of cases where both raters agree on the same severity, and p_e is expected agreement between two raters by chance, calculated as the product of the proportion of cases where each rater assigned a particular severity. Kappa values greater than 0.60 indicate reliable ratings (Amidei et al., 2019).

4.3. Multicollinearity mitigation and feature selection methods

Multicollinearity among features will possibly lead the machine learner not to distinguish which of them should be considered when predicting the dependent variable (Palomba and Tamburri, 2021), and cause performance decline. Moreover, the explainability of these models will be harmed (Jiarpakdee et al., 2022) since the importance of correlated features cannot be distinguished, and the output will be inconsistent. To solve it, feature selection techniques are introduced. However, feature selection methods may still discard important features and keep redundant features. Thus, appropriate feature selection should be applied to maximize performance and minimize multicollinearity.

Existing feature selection methods can be classified into three families, *i.e.,* filter-based ranking methods, filter-based subset methods, and wrapper- based subset methods (Chen et al., 2021). Filter-based ranking techniques select features by estimating their capacity for contributing to the fitness of the classification model. Filter-based subset techniques select features that collectively have good predictive capability. Practically, they generate feature subsets with lower inner correlation and higher correlation with classes to predict. The aforementioned methods are called "filter-based" because they use the selected statistical metrics to filter out irrelevant attributes. Unlike the filter-based approaches, wrapper-based techniques use a learning algorithm to evaluate the importance of a feature in a predictive model. They generate predictors built with subsets in advance to find the subset that achieves the best performance. The main difference between filter-based and wrapper-based feature selection techniques is that the former mainly uses statistics to measure the importance of each feature towards the class labels while the latter uses a predetermined classification model and a performance metric to measure the importance of a feature subset (Zhao et al., 2021). Apart from these 3 families, there also exist approaches using a mixture of these techniques (*e.g.,* AutoSpearman Jiarpakdee et al., 2020) to select features.

There exist numerous methods in each family. Since we are unable to reproduce and apply all of them, we utilize several representative types (Xu et al., 2016) of approaches available in WEKA (Hall et al., 2009). These approaches are commonly used in software quality assurance research (Zhao et al., 2021; Chen et al., 2021; Xu et al., 2016). The overview of the used feature selection methods is available in Table 4.

Filter-based feature ranking techniques sort features based on feature importance, and thus they require a threshold of the number of features to select. Zhao et al. (2021) suggested selecting 15% of features as a threshold. To avoid a significant decline in performance, we also test 45% and 75%. We use 75% as a threshold because it represents the third quartile, and 45% is the average value of the other 2 thresholds which is also close to the second quartile. We are not able to perform a grid search in the threshold since it will result in an excessively large search space, and thus we use these 3 representative values.

For AutoSpearman, we use the open-source Python implementation provided by the original authors.[10] The algorithm consists of two stages. In the first stage, Spearman rank correlation coefficients are calculated to reduce collinearity between feature pairs; in the second stage, the variance inflation factor (VIF) is calculated to reduce multicollinearity. While maximizing collinearity reduction, AutoSpearman retains as many features as possible, thereby reducing its impact on performance.

4.4. Statistical methods

Spearman rank correlation analyzes the correlation between two sets of ordered data with unknown probability distributions and outputs its correlation coefficient ρ. The correlation coefficient ρ evaluates the correlation between two statistical variables using a monotonic function, and its range is [−1, 1]. The positive or negative value of ρ corresponds to a positive or negative correlation, and the magnitude of ρ corresponds to the degree of correlation. When $\rho > 0.7$, the variables have a strong correlation.

Wilcoxon Ranksum Test could analyze the significance of the difference in the distributions of pairs of data. We use it since it is non-parametric data and does not assume that data is drawn from a

[10] https://github.com/awsm-research/PyExplainer/blob/master/pyexplainer/pyexplainer_pyexplainer.py.

Table 5
Prioritization performance using all features and features selected by CFS.

Blob					Complex Class				
Metric	CFS		No selection		Metric	CFS		No selection	
	MSR-CFS	KBS-CFS	MSR	KBS		MSR-CFS	KBS-CFS	MSR	KBS
Alpha	**0.75**	0.19	0.80	**0.91**	Alpha	0.44	**0.83**	0.46	**0.83**
Kappa	**0.73**	0.28	0.81	**0.96**	Kappa	0.63	**0.82**	0.65	**0.83**
AUC-ROC	**0.87**	0.64	0.91	**0.98**	AUC-ROC	0.83	**0.91**	0.84	**0.92**
F-Measure	**0.85**	0.58	0.89	**0.98**	F-Measure	0.77	**0.89**	0.78	**0.90**
MCC	**0.74**	0.29	0.81	**0.96**	MCC	0.64	**0.83**	0.66	**0.84**
Winner	MSR-CFS		KBS		**Winner**	KBS & KBS-CFS			
Spaghetti Code					Shotgun Surgery				
Metric	CFS		No selection		Metric	CFS		No selection	
	MSR-CFS	KBS-CFS	MSR	KBS		MSR-CFS	KBS-CFS	MSR	KBS
Alpha	0.49	0.30	0.61	**0.80**	Alpha	**0.52**	0.10	**0.55**	0.01
Kappa	0.41	0.49	0.52	**0.70**	Kappa	**0.52**	−0.01	**0.58**	−0.04
AUC-ROC	0.70	0.74	0.76	0.83	AUC-ROC	**0.75**	0.49	**0.79**	0.48
F-Measure	0.65	0.67	0.71	**0.81**	F-Measure	**0.69**	0.35	**0.73**	0.34
MCC	0.41	0.50	0.53	**0.72**	MCC	**0.52**	−0.01	**0.59**	−0.04
Winner	Tie		KBS		**Winner**	MSR & MSR-CFS			

normal distribution (Palomba et al., 2018c). Wilcoxon Ranksum Test produces a p-value, and we use p-value < 0.05 as an indicator of statistical significance. To ensure the effect size of the statistical result is significant, we also calculate Cliff's Delta (*i.e.*, the extent of the difference) for each pair of values. The effect size is negligible if $|\delta| < 0.147$, small if $0.147 \leq |\delta| < 0.33$, medium if $0.33 \leq |\delta| < 0.474$, and large if $|\delta| \geq 0.474$. We mark the result as non-negligible only if Wilcoxon $p < 0.05$ and the effect size is non-negligible ($|\delta| \geq 0.147$).

VIF (Fox and Monette, 1992) constructs a linear regression model to measure to what extent a feature can be predicted by other features, and then reduces the multicollinearity between a single feature and other features. VIF > 5 indicates high multicollinearity.

SK-ESD (Tantithamthavorn et al., 2019; Tantithamthavorn et al., 2017) is an improvement on the Scott-Knott algorithm. The Scott-Knott algorithm uses hierarchical clustering to compare and distinguish multiple groups of data with normal distribution, thereby ranking performance metrics and other data. Within a ranking, there may be multiple groups of performance data without significant statistical differences. The SK-ESD algorithm adds a correction function for non-normally distributed data and removes statistically insignificant groups based on effect size, and thus expanding the applicability of the original algorithm. In this paper, we use the R implementation of SK-ESD provided by the original authors.

4.5. SHAP feature importance

SHAP measures the contribution of a feature value to the difference between the actual local prediction and the global mean prediction (Lundberg et al., 2020) to distribute the credit for a classifier's output among its features (Rajbahadur et al., 2021) using the game-theory based Shapley values (Lundberg and Lee, 2017). For each instance in the training set, SHAP transforms the features of the instance into a space of simplified binary features as input. Afterward, SHAP builds the model g for explanation defined as a linear function of binary values, more specifically in Eq. (9):

$$g(\boldsymbol{z}) = \phi_0 + \sum_{i=1}^{M} \phi_i z_i, \tag{9}$$

where $\boldsymbol{z} \in \{0,1\}^M$ is the coalition vector (also known as simplified features Lundberg and Lee, 2017), and M is the maximum size of the coalition vector (*i.e.*, the number of simplified features). Specifically, z_i is the ith binary value in $\boldsymbol{z}$, where $z_i = 1$ means the corresponding feature is included in the coalition, and $z_i = 0$ indicates the feature is absent from the coalition. ϕ_0 is the average prediction value of the model, and ϕ_i is the Shapley value of the ith feature. Larger positive ϕ_i indicates a greater impact of the ith feature on the positive prediction result of the model. Note that $|\phi_i|$ is SHAP feature importance score that is guaranteed in theory to be locally, consistently, and additively accurate for each data point (Rajbahadur et al., 2021). We use the Python implementation of SHAP (Lundberg et al., 2020) in our study.

5. Result and discussion

In this section, we demonstrate the results of our experiment and answer the proposed research questions. Meanwhile, we describe our conclusions, findings, and implications.

5.1. RQ1: Performance of the replicated original models

Table 5 shows the performance using all features and after exploiting CFS. Better performance with a statistically significant improvement and non-negligible effect size is bold. We can find that KBS is superior to MSR in Blob, Complex Class, and Spaghetti Code prioritization when CFS is not applied, and the statistical result shows. almost all performance metrics are significantly different in distribution and with large effect size, except for 1 performance metric (AUC-ROC for KBS with no feature selection)[11]

CFS greatly impacts the performance of the model built with KBS features which leads to a decrease of AUC-ROC by up to 34%, and a decrease of Alpha by up to 72% (see the underlined performance in Table 5). In statistical tests, we find that the negative impact is significant in terms of the KBS Spaghetti Code and Blob performance. The effect size is large in all 5 performance metrics.[12] Such a great extent of negative impact on performance varies the conclusion. We find that the MSR dataset processed by CFS derives bad performance in terms of the raters' agreement metrics in Spaghetti Code and Shotgun Surgery, *i.e.*, although the classification performance seems ideal, Alpha, and Kappa indicate no agreement is achieved. Table 6 shows the multicollinearity of the CFS processed datasets, *i.e.*, it is not mitigated as suggested (Jiarpakdee et al., 2020).

Although the performance is not the same as the MSR paper, we can almost replicate ***C1*** and ***C2***. However, we fail to replicate the result of Complex Class prioritization. We think the cause might be the difference in CFS implementation or application since it does not

[11] Detailed experimental results are available in /results/sig_cfs_rq1.csv of https://doi.org/10.24433/CO.9879914.v2.

[12] Detailed experimental results are available in /results/sig_dataset_rq1.csv of https://doi.org/10.24433/CO.9879914.v2.

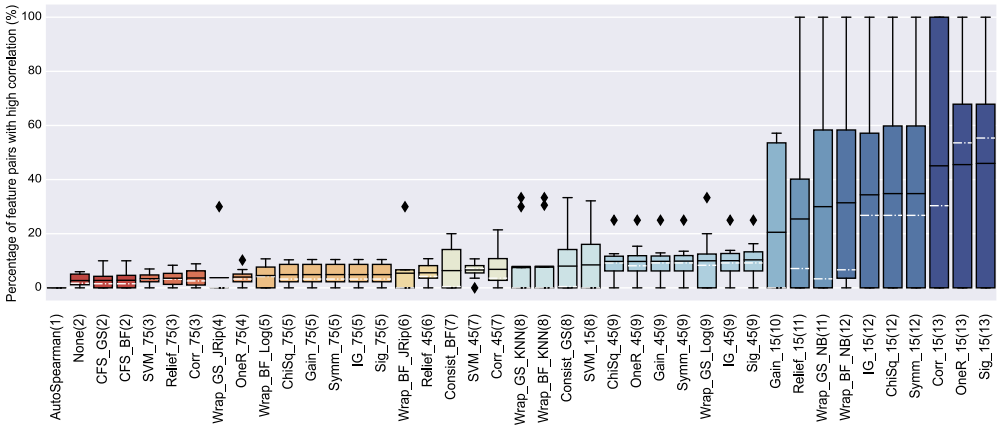

Fig. 2. Proportion of correlated pairs of features after feature selection. The names of feature selection approaches consist of `Algorithm Abbreviation_Strategy(GS/BF)_Ranking-based Thresholds(15/45/75)`. Specifically, wrapper-based approaches start with `Wrap_` to tell apart from filter-based approaches.

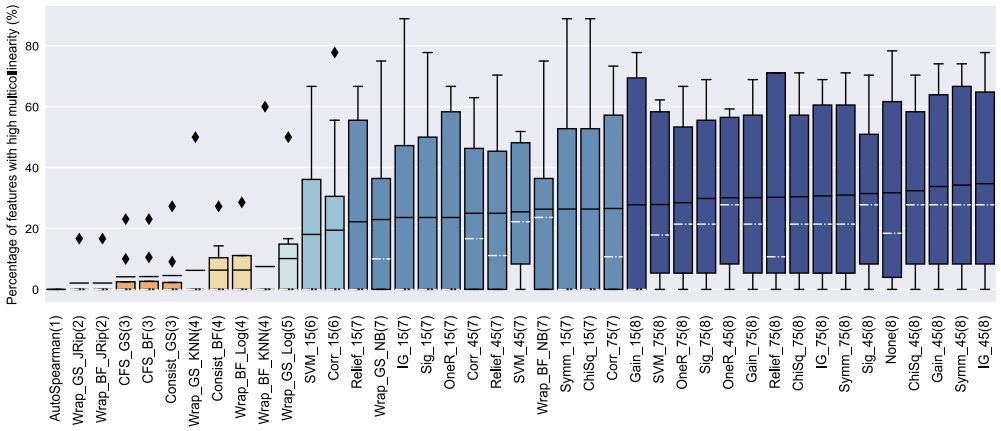

Fig. 3. Proportion of features having multicollinearity after feature selection. The naming conventions are the same as in Fig. 2.

negatively impact the performance of the KBS dataset in our case, *i.e.*, in our experiment, CFS does not exclude some vital features. We also find that some inappropriate experimental settings may lead to biased conclusions, (i) using CFS as a feature selection method may lead to a significant performance decline in KBS datasets, (ii) using only classification metrics in rating tasks may cover up the actual distance of the predicted results towards the desirable results of prioritization models, and (iii) multicollinearity is not mitigated as suggested, which may hinder the reliability of XAI conclusions.

Finding 1. ***C1*** and ***C2*** are mostly replicable in the original context of the MSR paper. However, using CFS as a feature selection method, using only classification metrics, and multicollinearity appearance may lead to biased conclusions. Thus, we revise these experimental settings in the next RQs for a fair replication.

5.2. RQ2: The capabilities of feature selection methods

Figs. 2–3 demonstrate feature selection's capability of mitigating collinearity and multicollinearity by measuring the proportion of features affected by these issues in all predictions performed in KBS and MSR. Their SK-ESD rankings are also presented in the labels of the X-axis. Mean values are presented in solid lines, while medians are in dashed lines. Autospearman and CFS are the first- and second-best choices in terms of collinearity mitigation. Meanwhile, the rankings are inconsistent for other methods.

Table 6
Multicollinearity presence in the CFS processed dataset.

Code Smell	Collinearity (ρ)%		Multicollinearity (VIF)%	
	KBS	MSR	KBS	MSR
Blob	0.00	0.00	0.00	0.00
Complex Class	**4.58**	**3.03**	**10.52**	**23.08**
Spaghetti Code	0.00	0.00	0.00	**10.00**
Shotgun Surgery	0.00	**4.76**	0.00	0.00

Fig. 4 shows the negative impact of feature selection on model performance. Most feature selection methods negatively impact model performance (positive values), while some (*e.g.*, Wrap-BF-KNN) may improve performance (negative values). Although we cannot determine the best method for feature selection, we can conclude that Autospearman mitigates multicollinearity while preserving much performance.

Finding 2. AutoSpearman is a better default method than CFS. However, it does not guarantee the best performance. We suggest exploiting different feature selection methods instead of using default methods in smell prioritization.

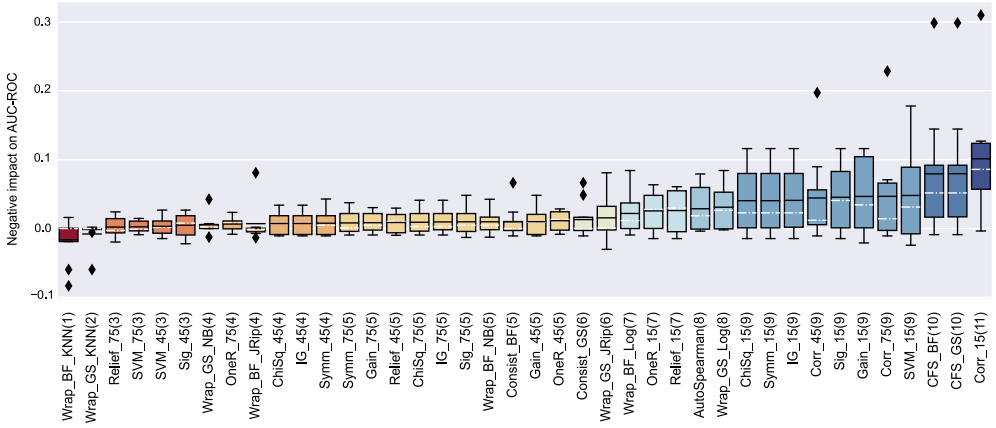

Fig. 4. Feature selections' negative impact on AUC-ROC (negative values indicate positive impact). The naming conventions are the same as in Fig. 2.

Table 7
The impact of better feature selection on performance (Using datasets with all features as a comparator).

Blob					Complex Class				
Metric	Wrap-GS-JRip		GS-Consist		Metric	Wrap-BF-KNN		Wrap-GS-Jrip	
	Variation	MSR-FS	Variation	KBS-FS		Variation	MSR-FS	Variation	KBS-FS
Alpha	−0.01	0.79	+0.01	**0.90**	Alpha	−0.05	0.41	+0.03	**0.86**
Kappa	–	0.81	−0.04	**0.92**	Kappa	−0.02	0.63	–	**0.83**
AUC-ROC	−0.01	0.90	+0.06	**0.96**	AUC-ROC	−0.01	0.83	−0.01	**0.91**
F-Measure	–	0.89	−0.03	**0.95**	F-Measure	–	0.78	–	**0.90**
MCC	–	0.81	−0.04	**0.92**	MCC	−0.01	0.65	–	**0.84**
Winner	KBS-FS				**Winner**	KBS-FS			
Spaghetti Code					Shotgun Surgery				
Metric	Wrap-GS-KNN		Wrap-GS-JRip		Metric	AutoSpearman		Wrap-BF-KNN	
	Variation	MSR-FS	Variation	KBS-FS		Variation	MSR-FS	Variation	KBS-FS
Alpha	+0.04	0.65	−0.03	**0.77**	Alpha	+0.01	**0.56**	+0.15	0.16
Kappa	+0.15	0.67	−0.03	0.67	Kappa	+0.01	**0.59**	+0.26	0.22
AUC-ROC	+0.07	0.83	−0.01	0.82	AUC-ROC	–	**0.79**	+0.13	0.61
F-Measure	+0.09	0.80	−0.02	0.79	F-Measure	–	**0.73**	+0.15	0.49
MCC	+0.15	0.68	−0.04	0.68	MCC	+0.01	**0.60**	−0.37	0.22
Winner	KBS-FS[a]				**Winner**	MSR-FS			

[a] Winning in *Alpha* and *Alpha* > 0.66 (Tian et al., 2016).

5.3. RQ3: Performance after better feature selection

Table 7 demonstrates the impact of feature selection on performance (using results without feature selection as baselines). Better performance with a statistically significant improvement and non-negligible effect size is bold. To ensure simplicity and readability, we only mention the best-performed feature selection approaches for each smell using each dataset.

After better feature selection, the model built with the KBS dataset is superior to MSR approaches in Blob, Complex Class, and Spaghetti Code, which is the same as the conclusion using all features in RQ1. The effect size of performance advantage including Blob, Shotgun Surgery, and Complex Class is large. In terms of Spaghetti Code, although 4 out of 5 differences in performance metrics have negligible effect size, the KBS dataset is significantly superior in the Alpha metric.[13] The advantage of using pure code metrics is up to 8% and 45% in AUC-ROC and Alpha respectively (see the underlined performance in Table 7).

We also check that compared with the model performance using the raw dataset without any feature selection, whether the improvement or decline of performance brought by the feature selection approaches is significant. The statistical result shows that better feature selection is significantly superior in 2 out of 8 cases (i.e., KBS Shotgun Surgery and MSR Spaghetti Code). For the rest of the 6 cases, compared with the original dataset without any processing, we find no significant performance variation. Thus, we find that feature selection need not negatively impact performance, meanwhile, it may even improve it.[14]

Although wrapper-based approaches are not consistently mitigating multicollinearity in RQ2, they perform well in the prioritization of specific smells because they aim at selecting the feature subsets yielding the best prediction performance. Since they are selecting a small number of features, the chance of selecting features with high multicollinearity is reduced, and thus they perform well in this RQ.

Moreover, Shotgun Surgery prioritization is failed according to the metrics such as Alpha. **Thus, in our context, *C2* can be reproduced, but *C1* cannot be replicated.**

> **Finding 3.** The pure code metric based model performs better than the model proposed by the replicated paper after applying better feature selection in the prioritization of 3 out of 4 code smells.

[13] Detailed experimental results are available in /results/sig_dataset_rq3.csv of https://doi.org/10.24433/CO.9879914.v2.

[14] Detailed experimental results are available in /results/sig_fs_rq3.csv of https://doi.org/10.24433/CO.9879914.v2.

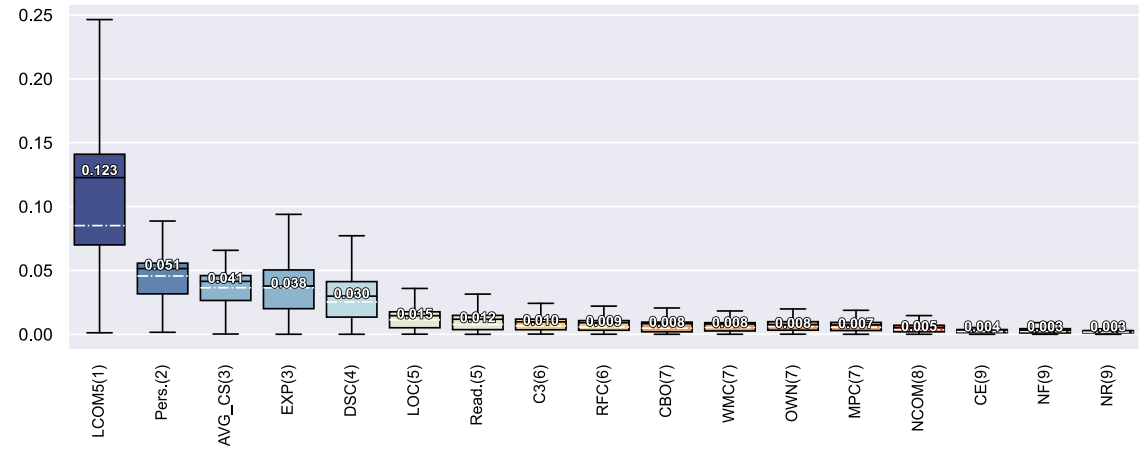

Fig. 5. SHAP Feature importance of Shotgun Surgery prioritization (using the MSR dataset). Descriptions of the abbreviations are available in Pecorelli et al. (2020).

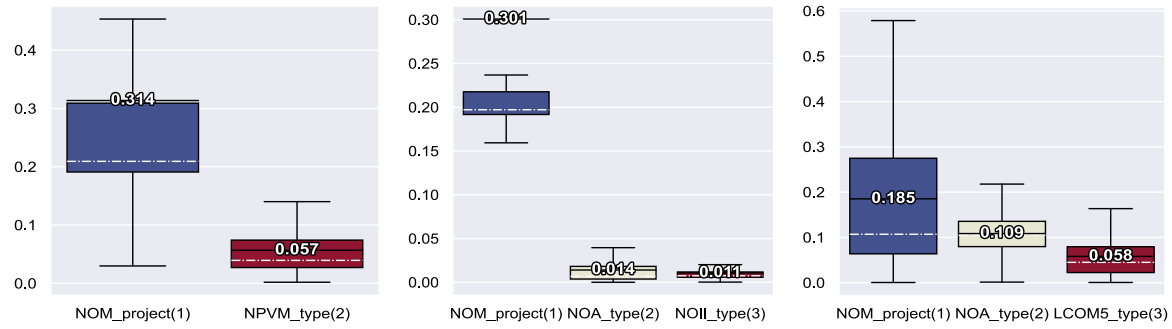

Fig. 6. SHAP Feature importance of the 3 change-insensitive code smells' prioritization (the KBS dataset). (a) Blob, (b) Complex Class, (c) Spaghetti Code.

5.4. RQ4: Global feature importance consistency

Figs. 5–6 depict the feature importance of the best-performed models in RQ3. NPVM refers to Number of Public Visible Methods, NOII refers to Number Of Interfaces Implemented, NOM and NOA refer to Number Of Methods and Attributes.

We can find that the size of a project is a very important feature since the prediction is performed in a cross-project scenario. The criticality of Blob, Complex Class, and Spaghetti Code can be determined by class size related metrics, while cohesion metric (LCOM5) is also important for Spaghetti Code prioritization. In terms of Shotgun Surgery prioritization, the lack of cohesion is the most determining aspect, and the features in rank 2 and 3 are all process metrics. Since Shotgun Surgery is a change-sensitive code smell related to coupling, and coupling problems co-occur with cohesion problems (Palomba et al., 2018a), we think this result is reasonable.

The feature importance in our context agrees with the MSR paper in Blob and Complex Class in terms of class size and cohesion. However, all process metrics presented in their paper are missing in our context since the KBS dataset does not involve process metrics.

Since the 3 smells (Blob, Complex Class, and Spaghetti Code) are not closely related to changes, we think it is better to construct simpler models (Menzies and Shepperd, 2019; dos Santos et al., 2020; Fakhoury et al., 2018) if pure code metrics are good enough for prioritization.

Finding 4. We reach an agreement with the replicated paper in terms of feature importance of the class size and the cohesion ones. However, the high (*i.e.*, in top-3) importance of the process and developer features can only be replicated in Shotgun Surgery prioritization.

5.5. RQ5: Local explanation agreement with practitioners

Table 8 demonstrates the agreements between explanations generated by the best-performed models (in Table 7) based on the KBS and the MSR datasets. Better agreement is bold.

The explanations using pure code metrics have more agreement on developers' comments since development process information captured by process metrics rarely appear in comments (*e.g.*, it is the least appeared factor in Table 3, accounting for only 8.06% occurrence among all comments), and they always appear in the explanation for the MSR dataset based model. The difference is lower in partial match tests since they measure categories independently, and the agreement rate also improves even if human and model outputs are not strictly consistent. The Kappa and Alpha metric results show that although our approach significantly improves agreement over the baseline approach, the agreement between the model and developers is still not ideal for smells other than Spaghetti Code, which reveals more effort is needed to build models for explanation.

In our context, *C3* cannot be replicated in both global (RQ4) and local explanations (RQ5).

Finding 5. The model based on pure code metric could better reflect the developers' opinions. We suggest using purely code metrics to prioritize code smells insensitive to code change since the built models are simpler and have advantages in model performance and explainability.

Table 8
Agreement between model explanation and comments.

Code Smell	Perfect match%		Partial match%		Kappa		Alpha	
	KBS	MSR	KBS	MSR	KBS	MSR	KBS	MSR
Blob	**39.59**	4.40	**72.29**	51.98	**0.33**	0.04	**0.33**	0.01
Complex Class	**47.28**	2.29	**82.16**	62.54	**0.60**	0.25	**0.59**	0.24
Spaghetti Code	**97.75**	0.96	**99.12**	50.56	**0.98**	−0.31	**0.98**	−0.31
Shotgun Surgery	**10.12**	1.53	**73.31**	40.80	**0.44**	−0.11	**0.42**	−0.21

5.6. Discussion

We attempted to contact the authors of the MSR paper via email to confirm the validity of our experimental settings and request further commit hash details of the dataset in order to extend the feature set for further exploration. However, we did not receive their response. Since the dataset used in this paper did not introduce additional features, and the performance was validated by more performance metrics, we believe the conclusions of this paper are more reasonable. However, there are still some experimental results and open issues that are worth discussing. In this section, we present these issues and other notable findings.

The Trade-off between Multicollinearity Mitigation and Performance. Although multicollinearity presence will harm model explainability and performance, discarding relevant features will not always lead to better performance. The reason is that the discarded features still contribute extra predictive power to the model since they capture useful information. In our paper, most feature selection approaches impose a negative impact on model performance, which leads to the discussion of how can we trade-off between two goals, i.e., (1) correlation mitigation and (2) performance preserving. On the one hand, some approaches are designed to automatically perform the trade-off, e.g., CFS is a feature selection method designed to maximize the merit of accomplishing two goals as much as possible. However, in our study, such a tradeoff is found unstable, and it may fail in both two goals. For example, the AUC performance declines by 34% in Blob prediction using KBS features. Meanwhile, highly correlated features are still present. We think such an unstable and uncontrollable tradeoff is unacceptable for the sake of scientific rigor and practicability. On the other hand, some approaches do not internally impose any tradeoff between these two goals, but they can still accomplish them. Considering the second goal in advance, most approaches aim at maximizing performance by selecting the features related to the dependent variables (e.g., wrapper-based approaches). They cannot always guarantee the mitigation of multicollinearity, and some results could be inappropriate for constructing explainable models for practitioners. Considering the first goal in advance, AutoSpearman aims to accomplish it by design. Although AutoSpearman lacks the consideration for the second goal mathematically, our result and another empirical study (Jiarpakdee et al., 2020) still show it outperforms the commonly-used CFS in both two goals. Such an improvement in the second goal is achieved by preserving as many features as possible. Thus, it has more chance of keeping useful features, while guaranteeing the mitigation of multicollinearity. Indeed, the impact on performance may be larger than the methods aiming specifically at the second goal. However, models built using AutoSpearman could be used for both prediction and explanation. Furthermore, AutoSpearman may be improved by considering feature importance, e.g., choosing a feature with a higher correlation to the dependent variable to discard in its second phase algorithm, to improve its impact on performance. Thus, depending on the purpose of the prediction, we conclude using AutoSpearman as a feature selection approach is a generic choice to tradeoff between the two goals. However, the tradeoff made by a single algorithm has its limitations, and we suggest testing various approaches to maximize performance in two goals.

Data Balancing. Although the 1327 samples in the original data of the MSR study are distributed relatively evenly among the four types of code smells (as shown in the last column of Table 3), the distribution of samples with each severity level is unbalanced. However, this paper follows the experimental setting of the MSR paper and does not use any data balancing approach since (1) data balancing may not be fair and reasonable according to the MSR paper (Pecorelli et al., 2020), *i.e.*, "no class needs to be balanced relative to any other class", (2) it may impose unexpected impact on the reliability of model interpretation results, *e.g.*, recent work on defect prediction (Tantithamthavorn et al., 2020) found that the model interpretation method exhibits abnormal behavior when applying the class balancing algorithm. Therefore, we do not use data balancing algorithms to process any data.

Project Level Metrics. Developers may be confused about metrics such as the number of methods within a project which they are working on. However, the prediction is performed in a cross-project scenario, and these metrics are actually "simulating (Pecorelli et al., 2020)" the overall situation of a project. Moreover, there are other features available using the full dataset of models, which may be important in other prediction scenarios. Thus, we think an evaluation of within-project prediction is urgent and necessary.

Rule-based Prioritization. Since we are using only 2 to 3 features to prioritize certain smells, rule-based methods may also work. We assess correlations of every feature with respect to the criticality, and we find that only NOM_project has a high correlation of 0.71 with the criticality of Blob. Thus, we believe a rule- and threshold-based approach will not be likely to recover the models' behavior. Moreover, a recent defect prediction study (dos Santos et al., 2020) also reported a similar conclusion, *i.e.*, using less than 4 features can lead to better performance, sometimes using 1 feature is good enough for defect prediction. Nevertheless, we still believe using techniques such as LoRMIkA (Rajapaksha et al., 2020) to generate rules from models may be worth trying.

The Reliability of Developers' Annotations. During manual verification, we also realize that some developers have biased opinions on basic facts (*e.g.,* maintainability is worth improving, tests are important, and so on), and sometimes they do not know how to interpret code smells with complex causes (*e.g.,* Shotgun Surgery), which may explain why MSR features work better but still have less agreement with their perceptions in Shotgun Surgery. Although the adaptation to the developers' perception is vital, the reaction towards their opinions should be reconsidered as Shrikanth et al. (2021) suggested. For example, we may need an additional set of features to capture the preferences of individual developers from a personalized perception (Huang et al., 2022; Eken and Tosun, 2021). We may need to guide them rather than adapt to their perspectives if they could not really interpret the problems. Since tests and utility classes can be detected by rule-based approaches (Huang et al., 2019), a customizable filter for the developers who perceive such classes as trivial may be better. Moreover, for the developers who only perform refactoring on defective classes, recommending ITS-related results (Sae-Lim et al., 2018a) may be more useful.

6. Threats to validity

We share most of the threats to validity with the replicated MSR paper (Pecorelli et al., 2020). In this section, we list other notable issues.

6.1. Conclusion validity

Conclusion validity is related to treatment and outcome.

The first threat is the used classifier. RF is used since it is an ideal classifier for tabular data in software engineering (Azeem et al., 2019; Tantithamthavorn et al., 2019) and it performs well in the replicated paper (Pecorelli et al., 2020). Since it is insensitive to parameters (Tantithamthavorn et al., 2019), we only tune the most important one (Jiarpakdee et al., 2018), *i.e.,* n-estimators, in the range of 10 to 300 with a step of 10, to avoid an excessive large search space.

A notable threat is that feature selection is a rapidly developing research topic, and there exist numerous methods to select optimal features. Thus, we are not able to exhaust all possible feature selection methods. To cope with it practically, we select representative types of feature selection methods used in related software quality assurance works and implemented in WEKA, a state-of-the-art machine learning library used by numerous software engineering researchers. Although this limitation does not affect our conclusion that there exist better feature selection methods compared with CFS, the improvement brought by better feature selection may be underestimated.

Another threat may be the reliability of SHAP. First, we followed the guidelines (see Section 2.3) to generate reliable explanations. Second, we also test its agreement in top-1 and top-3 (if applicable) feature importance (Rajbahadur et al., 2021) with Information Gain, *i.e.*, a conventionally applied algorithm in code smell researches to generate only global feature importance. We find that the top-1 and top-3 ranked features are consistent for the 2 algorithms in global feature importance, which reflects the reliability of the results.

6.2. Internal validity

Internal validity concerns the variables that could affect the outcome. As is discussed in Section 5.6, the reliability of subjective aspects are unavoidable threats, *e.g.*, our identification of developers' comments may also include bias. We involve experienced developers and researchers and a process to minimize the disagreement among them. The annotated data set is also available online (see the Appendix A).

6.3. External validity

External validity is about the generalizability of the results. Cross-project predictions are designed to handle cold start problems, which is practical in real-world cases. For example, heuristic-based code smell detection tool (e.g., P3C[15]) is used by the Alibaba Group to assure code quality. If more advanced machine learning based approaches are applied instead, they will not perform normally like P3C over new projects, i.e., they will be very likely to meet cold start problems in new projects. To solve it, they could use the data source of other projects in the same corporation or organization as a source to train a cross-project code smell prediction or prioritization model. However, there are other scenarios to cover, *e.g.,* studies in within-project scenarios should be conducted to replicate the results. Since validated large-scale within-project code smell dataset is hard to collect, this threat is unavoidable at present.

There exist other notable datasets in code smell prioritization, *e.g.*, the dataset in Sae-Lim et al. (2018a), the dataset of the KBS baseline (Fontana and Zanoni, 2017), and the MLCQ industrial dataset (Madeyski and Lewowski, 2023). We are not involving these datasets because they do not contain detailed comments from developers. The first two datasets are relatively small and collected from researchers and students, and the third dataset is annotated by third-party developers which may have no experience in maintaining the annotated code, and thus (Kovačević et al., 2022) found that the annotation agreement in the MLCQ dataset is not agreeable. Since the MSR dataset is a result of tracing established open source projects in 6 months, and the feedback is collected from original developers, we believe it is of higher quality, and such long-term data collection on a larger scale should be encouraged to address data validity issues in code smell studies.

[15] https://github.com/alibaba/p3c.

7. Conclusion

We replicated the MSR paper (Pecorelli et al., 2020) to validate its 3 major conclusions by performing better feature selection over the MSR and the KBS datasets. We found that in our context, only 1 conclusion (*i.e.*, ***C2***) is replicable, and using CFS as a default feature selection technique is the cause of such inconsistency.

Since our approach improved the overall performance of the prioritization models, for researchers and practitioners, our suggestions are: (1) exploiting different feature selection methods other than CFS to mitigate multicollinearity before prediction, (2) using pure code metrics to construct Blob, Complex Class, and Spaghetti Code prioritization models, (3) assessing also regression and inter-rater agreement metrics for performance evaluation, and (4) be careful with the developers' unreliable ratings and comments on code smell criticality.

Future work includes: (1) developing a rule-based prioritization heuristic according to XAI outcomes, (2) extending the scope to within-project prediction for more code smells, and (3) validating whether the XAI techniques could help developers recognize severe code smells in real-world scenarios.

CRediT authorship contribution statement

Zijie Huang: Conceptualization, Methodology, Writing – original draft, Visualization. **Huiqun Yu:** Validation, Writing – review & editing, Funding acquisition. **Guisheng Fan:** Validation, Writing – review & editing. **Zhiqing Shao:** Validation, Writing – review & editing. **Ziyi Zhou:** Validation, Writing – review & editing. **Mingchen Li:** Validation, Writing – review & editing.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Huiqun Yu reports financial support was provided by National Natural Science Foundation of China. Huiqun Yu reports financial support was provided by Natural Science Foundation of Shanghai. Huiqun Yu reports financial support was provided by the Capacity Building Project of Local Universities Science and Technology Commission of Shanghai Municipality. Huiqun Yu reports financial support was provided by the Research Programme of National Engineering Laboratory for Big Data Distribution and Exchange Technologies. Huiqun Yu reports was provided by Shanghai Municipal Special Fund for Promoting High Quality Development. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data shared in the appendix.

Appendix A. The performance replication CodeOcean capsule

The code for prediction and the dataset are available at this CodeOcean URL: https://doi.org/10.24433/CO.9879914.v2

Table 9
An instance extracted from the rated comments.

Smell	Dataset	Class	Comments	PROC_ Rated	DEV_ Rated	INNER_ Rated	CROSS_ Rated	PROC_ Model	DEV_ Model	INNER_ Model	CROSS_ Model
Complex Class	KBS	Abstract JCR2SPI Test	As long as the test finds bugs, it is ok to have complexity.	0 (not present)	1 (present)	1	0	0	0	1	0

A.1. Usage

CodeOcean is a replication platform recommended by many established open science venues in software engineering, *e.g.*, the Artifact Evaluation Track of the ICSME conference,[16] the Original Software Publication of the Science of Computer Programming journal, and so on.

After accessing the URL using a modern browser (*e.g.*, Google Chrome), user could see an interface of a CodeOcean capsule containing executable code, used data, parameters, and experimental results of model performance. The results are verified by third parties (CodeOcean) for reproducibility and consistency. The capsule already contains configured execution environment, and thus the code could also be re-executed by simply pressing the "Reproducible Run" button on the upper right corner, the execution time is about 45 s, and the result will be consistent.

Free registration may be required for program execution. If registration is not applicable, it is also possible to download it and execute it offline.

A.2. The location of code and data files

A brief performance result is located in /results/performance.csv. The detailed performance of each fold of prediction is located in /results/performance_folds.csv. The results of the significance test in RQ1 and RQ3 are in /results/sig_dataset_rq1.csv, /results/sig_cfs_rq1.csv, /results/sig_fs_rq3.csv, and /results/sig_dataset_rq3.csv. You may look into /code/stats.py for further explanations of the statistical results. The used data can be found in the /data directory. The prediction code is located in /code/main.py, and the statistical code is in /code/stats.py. The console output is available in /code/performance_execution_output.txt, and it will be consistent with the console output online. Note that to ensure a fast online replication, the best hyperparameters found by grid search are hard-coded in the get_settings() function of main.py.

Apart from the code artifacts, this capsule also contains some in-process data.

/code/comments_rated.csv contains the manual annotated dataset as well as the output of the XAI approach. These data are used in agreement calculation in RQ4-5 (see the next section for more details).

/code/metrics_categories.txt contains the metrics used in each aspect in RQ5.

Appendix B. An example of agreement calculation

Table 9 shows an instance (a row) of model and human rater explanation extracted from comments_rated.csv. The prediction is made by the model built upon the KBS dataset to prioritize a smell called Complex Class presented in org.apache.jackrabbit.jcr2spi.AbstractJCR2SPITest, and the priority is correctly predicted. The number 0 indicates a certain aspect does not exist in this comment, while 1 indicates an aspect exists in this comment.

[16] https://conf.researchr.org/track/icsme-2023/icsme-2023-artifact-evaluation-track-and-rose-festival.

B.1. Developer's perception (columns in red named after {Category Name}_Rated)

We think the developers expressed this class is affected by inter-class complexity, which fits the definition of the category called "INNER" in our paper. Meanwhile, the authors also think complexity in tests does not matter, and thus we think this expresses his/her subjective point of view towards test quality, so it falls into the "DEV" category.

B.2. Model explanation (columns in blue named after {Category Name}_Model)

According to SHAP feature importance, NOM_project is the most important feature that leads to the final prediction. According to metrics_categories.txt in our replication package, NOM_project is categorized as a "INNER" feature. Meanwhile, "DEV" and "PROC" features do not exist in the KBS dataset.

B.3. Calculating agreement

B.3.1. Partial match

The model and the original developer agree on the PROC, INNER, and CROSS categories, but they derive different explanations in the DEV category. Thus, the agreement in terms of partial match is 3/4 = 75% for this instance.

B.3.2. Perfect match

Apparently, the developer and the model's explanations are not the same in all 4 classes since they derive different explanations in the DEV class. Thus, no perfect match is presented in this instance. Perfect match is determined only if the developer and the model's explanations agree on all 4 classes.

References

Alazba, A., Aljamaan, H., 2021. Code smell detection using feature selection and stacking ensemble: An empirical investigation. Inf. Softw. Technol. 138, 106648.

Aleithan, R., 2021. Explainable just-in-time bug prediction: Are we there yet? In: Proc. IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). pp. 129–131.

Amidei, J., Piwek, P., Willis, A., 2019. Agreement is overrated: A plea for correlation to assess human evaluation reliability. In: 12th International Conference on Natural Language (INLG). pp. 344–354.

Azeem, M.I., Palomba, F., Shi, L., Wang, Q., 2019. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. Inf. Softw. Technol. 108, 115–138.

Brown, W.H., Malveau, R.C., McCormick, H.W.S., Mowbray, T.J., 1998. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons, Inc., Boston, MA.

Buse, R.P., Weimer, W.R., 2009. Learning a metric for code readability. IEEE Trans. Softw. Eng. 36 (4), 546–558.

Catolino, G., Palomba, F., Fontana, F.A., Lucia, A.D., Zaidman, A., Ferrucci, F., 2020. Improving change prediction models with code smell-related information. Empir. Softw. Eng. 25 (1), 49–95.

Chen, X., Yuan, Z., Cui, Z., Zhang, D., Ju, X., 2021. Empirical studies on the impact of filter-based ranking feature selection on security vulnerability prediction. IET Softw. 15 (1), 75–89.

Cohen, J., 1960. A coefficient of agreement for nominal scales. Educ. Psychol. Meas. 20 (1), 37–46.

Di Nucci, D., Palomba, F., Tamburri, D.A., Serebrenik, A., De Lucia, A., 2018. Detecting code smells using machine learning techniques: Are we there yet? In: IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 612–621.

Eken, B., Tosun, A., 2021. Investigating the performance of personalized models for software defect prediction. J. Syst. Softw. 181, 111038.
Fakhoury, S., Arnaoudova, V., Noiseux, C., Khomh, F., Antoniol, G., 2018. Keep it simple: Is deep learning good for linguistic smell detection? In: 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 602–611.
Ferreira, M.M., da Silva Bigonha, M.A., Ferreira, K.A.M., 2021. On the gap between software maintenance theory and practitioners' approaches. In: 8th IEEE/ACM International Workshop on Software Engineering Research and Industrial Practice (SER&IP@ICSE). pp. 41–48.
Fontana, F.A., Ferme, V., Zanoni, M., Roveda, R., 2015. Towards a prioritization of code debt: A code smell intensity index. In: IEEE 7th International Workshop on Managing Technical Debt (MTD). pp. 16–24.
Fontana, F.A., Zanoni, M., 2017. Code smell severity classification using machine learning techniques. Knowl.-Based Syst. 128, 43–58.
Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA.
Fox, J., Monette, G., 1992. Generalized collinearity diagnostics. J. Amer. Statist. Assoc. 87 (417), 178–183.
Gosiewska, A., Biecek, P., 2019. iBreakDown: Uncertainty of model explanations for non-additive predictive models. arXiv:1903.11420. URL: http://arxiv.org/abs/1903.11420.
Guimarães, E.T., Vidal, S.A., Garcia, A.F., Pace, J.A.D., Marcos, C.A., 2018. Exploring architecture blueprints for prioritizing critical code anomalies: Experiences and tool support. Softw. - Pract. Exp. 48 (5), 1077–1106.
Hall, M.A., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H., 2009. The WEKA data mining software: an update. SIGKDD Explor. 11 (1), 10–18.
Huang, Z., Chen, J., Gao, J., 2019. The smell of blood: Evaluating anemia and bloodshot symptoms in web applications. In: 31st International Conference on Software Engineering and Knowledge Engineering (SEKE). pp. 141–186.
Huang, Z., Shao, Z., Fan, G., Yu, H., Yang, X., Yang, K., 2022. Community smell occurrence prediction on multi-granularity by developer-oriented features and process metrics. J. Comput. Sci. Tech. 37 (1), 182–206.
Ichtsis, A., Mittas, N., Ampatzoglou, A., Chatzigeorgiou, A., 2022. Merging smell detectors: Evidence on the agreement of multiple tools. In: 5th IEEE/ACM International Conference on Technical Debt (TechDebt). pp. 61–65.
Jain, S., Saha, A., 2021. Improving performance with hybrid feature selection and ensemble machine learning techniques for code smell detection. Sci. Comput. Program. 212, 102713.
Jiarpakdee, J., Tantithamthavorn, C., Dam, H.K., Grundy, J., 2022. An empirical study of model-agnostic techniques for defect prediction models. IEEE Trans. Softw. Eng. 48 (1), 166–185.
Jiarpakdee, J., Tantithamthavorn, C.K., Grundy, J., 2021. Practitioners' perceptions of the goals and visual explanations of defect prediction models. In: IEEE/ACM 18th International Conference on Mining Software Repositories (MSR). pp. 432–443.
Jiarpakdee, J., Tantithamthavorn, C., Treude, C., 2018. AutoSpearman: Automatically mitigating correlated software metrics for interpreting defect models. In: IEEE 34th International Conference on Software Maintenance and Evolution (ICSME). pp. 92–103.
Jiarpakdee, J., Tantithamthavorn, C., Treude, C., 2020. The impact of automated feature selection techniques on the interpretation of defect models. Empir. Softw. Eng. 25 (5), 3590–3638.
Kohavi, R., 1995. A study of cross-validation and bootstrap for accuracy estimation and model selection. In: 14th International Joint Conference on Artificial Intelligence (IJCAI). pp. 1137–1145.
Kovačević, A., Slivka, J., Vidaković, D., Grujić, K.-G., Luburić, N., Prokić, S., Sladić, G., 2022. Automatic detection of long method and god class code smells through neural source code embeddings. Expert Syst. Appl. 204, 117607.
Krippendorff, K., 1970. Estimating the reliability, systematic error and random error of interval data. Educ. Psychol. Meas. 30 (1), 61–70.
Lanza, M., Marinescu, R., Ducasse, S., 2005. Object-Oriented Metrics in Practice. Springer-Verlag, Berlin, Heidelberg.
Liu, H., Jin, J., Xu, Z., Bu, Y., Zou, Y., Zhang, L., 2021. Deep learning based code smell detection. IEEE Trans. Softw. Eng. 47 (9), 1811–1837.
Liu, H., Ma, Z., Shao, W., Niu, Z., 2012. Schedule of bad smell detection and resolution: A new way to save effort. IEEE Trans. Softw. Eng. 38 (1), 220–235.
Liu, H., Xu, Z., Zou, Y., 2018. Deep learning based feature envy detection. In: 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 385–396.
Lundberg, S.M., Erion, G., Chen, H., DeGrave, A., Prutkin, J.M., Nair, B., Katz, R., Himmelfarb, J., Bansal, N., Lee, S.-I., 2020. From local explanations to global understanding with explainable AI for trees. Nat. Mach. Intell. 2 (1), 56–67.
Lundberg, S.M., Lee, S.-I., 2017. A unified approach to interpreting model predictions. In: 31st International Conference on Neural Information Processing Systems (NIPS). pp. 4768–4777.
Madeyski, L., Lewowski, T., 2023. Detecting code smells using industry-relevant data. Inf. Softw. Technol. 155, 107112.
Maltbie, N., Niu, N., Van Doren, M., Johnson, R., 2021. XAI tools in the public sector: A case study on predicting combined sewer overflows. In: Proc. 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). pp. 1032–1044.
Menzies, T., Shepperd, M., 2019. "Bad smells" in software analytics papers. Inf. Softw. Technol. 112, 35–47.
Moha, N., Gueheneuc, Y.-G., Duchien, L., Le Meur, A.-F., 2010. DECOR: A method for the specification and detection of code and design smells. IEEE Trans. Softw. Eng. 36 (1), 20–36.
Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., De Lucia, A., 2018a. A large-scale empirical study on the lifecycle of code smell co-occurrences. Inf. Softw. Technol. 99, 1–10.
Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., De Lucia, A., 2018b. On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation. Empir. Softw. Eng. 23 (3), 1188–1221.
Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., Poshyvanyk, D., 2013. Detecting bad smells in source code using change history information. In: 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 268–278.
Palomba, F., Bavota, G., Penta, M.D., Oliveto, R., Poshyvanyk, D., De Lucia, A., 2015. Mining version histories for detecting code smells. IEEE Trans. Softw. Eng. 41 (5), 462–489.
Palomba, F., Oliveto, R., De Lucia, A., 2017. Investigating code smell co-occurrences using association rule learning: A replicated study. In: IEEE 1st Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE). pp. 8–13.
Palomba, F., Panichella, A., De Lucia, A., Oliveto, R., Zaidman, A., 2016. A textual-based technique for smell detection. In: IEEE 24th International Conference on Program Comprehension (ICPC). pp. 1–10.
Palomba, F., Panichella, A., Zaidman, A., Oliveto, R., Lucia, A.D., 2018c. The scent of a smell: An extensive comparison between textual and structural smells. IEEE Trans. Softw. Eng. 44 (10), 977–1000.
Palomba, F., Tamburri, D.A., 2021. Predicting the emergence of community smells using socio-technical metrics: A machine-learning approach. J. Syst. Softw. 171, 110847.
Palomba, F., Tamburri, D.A., Fontana, F.A., Oliveto, R., Zaidman, A., Serebrenik, A., 2021. Beyond technical aspects: How do community smells influence the intensity of code smells? IEEE Trans. Softw. Eng. 47 (1), 108–129.
Palomba, F., Zanoni, M., Fontana, F.A., De Lucia, A., Oliveto, R., 2019. Toward a smell-aware bug prediction model. IEEE Trans. Softw. Eng. 45 (2), 194–218.
Pecorelli, F., Palomba, F., Khomh, F., De Lucia, A., 2020. Developer-driven code smell prioritization. In: IEEE/ACM 17th International Conference on Mining Software Repositories (MSR). pp. 220–231.
Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al., 2011. Scikit-Learn: Machine learning in python. J. Mach. Learn. Res. 12 (85), 2825–2830.
Perera, H., Hussain, W., Mougouei, D., Shams, R.A., Nurwidyantoro, A., Whittle, J., 2019. Towards integrating human values into software: Mapping principles and rights of GDPR to values. In: IEEE 27th International Requirements Engineering Conference (RE). pp. 404–409.
Rajapaksha, D., Bergmeir, C., Buntine, W., 2020. LoRMIkA: Local rule-based model interpretability with K-optimal associations. Inform. Sci. 540, 221–241.
Rajapaksha, D., Tantithamthavorn, C., Bergmeir, C., Buntine, W., Jiarpakdee, J., Grundy, J., 2022. SQAPlanner: Generating data-informed software quality improvement plans. IEEE Trans. Softw. Eng. 48 (8), 2814–2835.
Rajbahadur, G.K., Wang, S., Ansaldi, G., Kamei, Y., Hassan, A.E., 2021. The impact of feature importance methods on the interpretation of defect classifiers. IEEE Trans. Softw. Eng. 48 (7), 2245–2261.
Ribeiro, M.T., Singh, S., Guestrin, C., 2016. "Why should I trust you?": Explaining the predictions of any classifier. In: 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD). pp. 1135–1144.
Saboury, A., Musavi, P., Khomh, F., Antoniol, G., 2017. An empirical study of code smells in JavaScript projects. In: IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 294–305.
Sae-Lim, N., Hayashi, S., Saeki, M., 2016. Context-based code smells prioritization for prefactoring. In: IEEE 24th International Conference on Program Comprehension (ICPC). pp. 1–10.
Sae-Lim, N., Hayashi, S., Saeki, M., 2017. How do developers select and prioritize code smells? A preliminary study. In: IEEE 33rd International Conference on Software Maintenance and Evolution (ICSME). pp. 484–488.
Sae-Lim, N., Hayashi, S., Saeki, M., 2017. Revisiting context-based code smells prioritization: On supporting referred context. In: XP2017 Scientific Workshops. pp. 1–5.
Sae-Lim, N., Hayashi, S., Saeki, M., 2018a. Context-based approach to prioritize code smells for prefactoring. J. Soft.: Evol. Process 30 (6), e1886.
Sae-Lim, N., Hayashi, S., Saeki, M., 2018b. An investigative study on how developers filter and prioritize code smells. IEICE Trans. Inf. Syst. 101-D (7), 1733–1742.
dos Santos, G.E., Figueiredo, E., Veloso, A., Viggiato, M., Ziviani, N., 2020. Understanding machine learning software defect predictions. Autom. Softw. Eng. 27 (3), 369–392.
Sharma, T., Efstathiou, V., Louridas, P., Spinellis, D., 2021. Code smell detection by deep direct-learning and transfer-learning. J. Syst. Softw. 176, 110936.
Shrikanth, N.C., Nichols, W., Fahid, F.M., Menzies, T., 2021. Assessing practitioner beliefs about software engineering. Empir. Softw. Eng. 26 (4), 73.

Sobrinho, E.V.d.P., De Lucia, A., Maia, M.d.A., 2021. A systematic literature review on bad smells–5 w's: Which, when, what, who, where. IEEE Trans. Softw. Eng. 47 (1), 17–66.
Tantithamthavorn, C., Hassan, A.E., Matsumoto, K., 2020. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. IEEE Trans. Softw. Eng. 46 (11), 1200–1219.
Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K., 2017. An empirical comparison of model validation techniques for defect prediction models. IEEE Trans. Softw. Eng. 43 (1), 1–18.
Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K., 2019. The impact of automated parameter optimization on defect prediction models. IEEE Trans. Softw. Eng. 45 (7), 683–711.
Tian, Y., Ali, N., Lo, D., Hassan, A.E., 2016. On the unreliability of bug severity data. Empir. Softw. Eng. 21 (6), 2298–2323.
Tsantalis, N., Chaikalis, T., Chatzigeorgiou, A., 2018. Ten years of JDeodorant: Lessons learned from the hunt for smells. In: IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 4–14.
Vidal, S.A., Marcos, C.A., Pace, J.A.D., 2016. An approach to prioritize code smells for refactoring. Autom. Softw. Eng. 23 (3), 501–532.
Vidal, S.A., Oizumi, W.N., Garcia, A., Diaz-Pace, J.A., Marcos, C., 2019. Ranking architecturally critical agglomerations of code smells. Sci. Comput. Program. 182, 64–85.
Wang, G., Chen, J., Gao, J., Huang, Z., 2021. Python code smell refactoring route generation based on association rule and correlation. Int. J. Softw. Eng. Knowl. Eng. 31 (09), 1329–1347.
Xu, Z., Liu, J., Yang, Z., An, G., Jia, X., 2016. The impact of feature selection on defect prediction performance: An empirical comparison. In: IEEE 27th International Symposium on Software Reliability Engineering (ISSRE). pp. 309–320.
Yang, X., Yu, H., Fan, G., Huang, Z., Yang, K., Zhou, Z., 2021. An empirical study of model-agnostic interpretation technique for just-in-time software defect prediction. In: 17th EAI International Conference on Collaborative Computing (CollaborateCom). pp. 420–438.
Yao, J., Shepperd, M., 2020. Assessing software defection prediction performance: Why using the matthews correlation coefficient matters. In: 24th International Conference on the Evaluation and Assessment in Software Engineering (EASE). pp. 120–129.
Zhao, K., Xu, Z., Yan, M., Zhang, T., Yang, D., Li, W., 2021. A comprehensive investigation of the impact of feature selection techniques on crashing fault residence prediction models. Inf. Softw. Technol. 139, 106652.

Zijie Huang received his B.S. and M.E. degrees from Shanghai Normal University in 2016 and 2020, both in computer science. He is currently pursuing his Ph.D. degree in computer science with the East China University of Science and Technology. His research interests include code smell, software quality assurance, and empirical software engineering.

Huiqun Yu received the M.Sc. and Ph.D. degrees in Computer Science from the East China University of Science and Technology and Shanghai Jiaotong University, in 1992 and 1995, respectively. He is presently a professor of the Department of Computer Science and Engineering, East China University of Science and Technology. His main research interests include software engineering, trustworthy computing and big data intelligence.

Guisheng Fan received the B.S. degree in computer science from the Anhui University of Technology in 2003, and the M.S. and Ph.D. degrees in computer science from the East China University of Science and Technology in 2006 and 2009, respectively, where he is currently working in with the Department of Computer Science and Engineering. His research interests include formal methods for complex software system, service oriented computing, and techniques for analysis of software architecture.

Zhi-Qing Shao received his B.S. degree in mathematical logic from Nanjing University, Nanjing, in 1986, M.S. degree in pure mathematics from the Institute of Software, Chinese Academy of Sciences, Beijing, in 1989, and Ph.D. degree in computer software from Shanghai Jiao Tong University, Shanghai, in 1998. He is currently a professor at East China University of Science and Technology, Shanghai. His research interests include software engineering and network computing.

Ziyi Zhou got his Ph.D. in Computer Science and Technology from East China University of Science and Technology in 2023. He is currently a Postdoctoral Fellow at the Institute of Natural Sciences of Shanghai Jiao Tong University. His research interests include program comprehension, natural language processing, and deep learning for software engineering.

Mingchen Li received his BS degree in Applied Mathematics from the Science college of East China University of Science and Technology in 2020. He is presently a Ph.D. student of the Department of Computer Science and Engineering, East China University of Science and Technology. His main research interests include software engineering and deep learning.