Expert Systems With Applications 238 (2024) 121640

Contents lists available at ScienceDirect

Expert Systems With Applications

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

Aligning XAI explanations with software developers' expectations: A case study with code smell prioritization[☆]

Zijie Huang, Huiqun Yu [*], Guisheng Fan, Zhiqing Shao, Mingchen Li, Yuguo Liang

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

ARTICLE INFO

Keywords:
Code smell
Software quality assurance
Explainable artificial intelligence
Empirical software engineering

ABSTRACT

EXplainable Artificial Intelligence (XAI) aims at improving users' trust in black-boxed models by explaining their predictions. However, XAI techniques produced unreasonable explanations for software defect prediction since expected outputs (e.g., causes of bugs) were not captured by features used to build models. To set aside feature engineering limitations and evaluate whether XAI could adapt to developers, we exploit XAI for code smell prioritization (i.e., predicting criticalities of sub-optimal coding practices and design choices), whose features could capture developers' major expectations. We assess the gap between XAI explanations and developers' expectations in terms of (1) the *accuracy* of prediction, (2) the *coverage* of explanations on expectations, and (3) the *complexity* of explanations. We also narrow the gap by preserving the features related to developers' expectations as much as possible in feature selection. We find that XAI can explain smells with simpler causes in top 3 to 5 features. Complex smells can be explained in around 10 features, which need more expertise to interpret. Selecting features adapting to the developers' expectations improves *coverage* by 5% to 29%, with almost no negative impact on *accuracy* and *complexity*. Results also highlight the need of dividing coarse-grained prediction targets and developing fine-grained feature engineering.

1. Introduction

Software Quality Assurance (SQA) optimizes software development and maintenance by distributing limited quality assurance resources to the riskiest software modules. Machine learning approaches were proposed to support SQA decision-making, and they achieved ideal results in major tasks such as defect prediction and code smell identification. Compared with the significant progress in prediction, explaining why and how such predictions are made is still a less focused topic in academia but an important concern of practitioners (Jiarpakdee et al., 2022). In addition, it is the right of AI system users to know how predictors behave (Perera et al., 2019). Finally, transparency of models can improve users' trust (Papenmeier et al., 2022) on AI models, and thus their usage could be promoted.

Recent work reported eXplainable Artificial Intelligence (XAI) approaches could generate stable explanations in a typical SQA task, i.e., defect prediction (Rajbahadur et al., 2022). Although most developers regarded such explanations as helpful, they may still be impractical due to the misalignment with developers' expectations on XAI (see Section 3.1 for an example). The expectations include (1) allocating SQA resources, (2) understanding and explaining the cause related to defects, and (3) understanding why a prediction is made. Recent work of Aleithan (2021) revealed that XAI generated unreasonable explanations compared with the expected outputs, i.e., the root causes of bugs are missing for the up-mentioned expectation (2), and they abandoned the XAI-based explanations. From this example, we learned a lesson that assuring disruptiveness (Maltbie et al., 2021) and soundness of explanation is not enough to make it trustworthy, even if they could contribute significantly to expectations (1) and (3). If the explanation fails on expectation (2), the huge gap between XAI explanation and users' expectations will still result in mistrust and low acceptance of results. Nevertheless, we notice that the root causes of bugs are not captured by the features used to train the models, and thus XAI could never output an unknown factor. Consequently, under the condition of appropriate feature engineering, the potential of XAI is still unclear. This leads to our research goal of investigating how well can XAI

☆ 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).

* Corresponding author.
E-mail addresses: hzj@mail.ecust.edu.cn (Z. Huang), yhq@ecust.edu.cn (H. Yu), gsfan@ecust.edu.cn (G. Fan), zshao@ecust.edu.cn (Z. Shao), lmc@mail.ecust.edu.cn (M. Li), ygliang@mail.ecust.edu.cn (Y. Liang).

https://doi.org/10.1016/j.eswa.2023.121640
Received 24 May 2023; Received in revised form 2 August 2023; Accepted 13 September 2023
Available online 27 September 2023

techniques generate explanations that meet their expectations, and how can we improve them.

If expected explanations can be generated, instead of abandoning this line of research, we should develop more advanced feature extraction approaches specifically for XAI to generate comprehensive features that incorporate developers' expectations. However, the research goal can hardly be fulfilled in the context of defect prediction. First, present feature engineering approaches for defect prediction can barely capture the root causes of bugs. Second, we do not have a dataset including developers' perceptions of the root causes of defects or their expectations of explanations.

Apart from defect prediction, there exist various other SQA tasks using similar techniques, i.e., generating product and process metrics as features to build machine learning predictors. Code smells are sub-optimal code implementation and design choices that can hinder software maintainability and reliability in the long run. Practitioners need to focus on removing the worst code smells in advance using limited SQA resources. However, smell detection tools may produce an excessive number of results. Due to the high cost of manual inspection of all suspicious candidates, both rule and machine learning based code smell detectors were perceived as unhelpful by practitioners (Sae-Lim et al., 2018a). To discard trivial detection results, prior studies built machine learners capturing structural (e.g., coupling, cohesion, complexity) and contextual (e.g., error-proneness, change history, developer) information to rank the results. Pecorelli et al. (2020) further proposed a developer-driven approach to prioritize 4 code smells. Despite introducing new features, the study also collected comments and manual prioritizations from original developers. We think the study of Pecorelli et al. (2020) is more ideal as the context of assessing the capability of XAI for generating outputs that developers expected for the following 3 reasons. First, the reason why smells are prioritized by humans could be captured by the features used to build models. Second, the original developers provided detailed comments on why they prioritized the code. Third, prior work of Yedida and Menzies (2022) also used code smell as an alternative to defect prediction to validate their conclusion for SQA and software analytics because of the high relativeness and similarity in technical details and purposes of these two tasks.

We intend to investigate whether we can align XAI explanations with developers' expectations for code smell prioritization. First, we summarize and categorize the developers' comments to find out their major concerns toward code smells' criticality and determine their expected XAI output. Then, we build a prediction model and generate explanations using an XAI approach called SHapley Additive exPlanation (SHAP). Afterward, we assess whether inspecting a reasonable number of the most important features in XAI explanations could cover most developers' concerns. Furthermore, we also improve feature selection according to the developers' primary concerns to narrow the gap between XAI explanations and developers' expectations.

The major contributions of our work include:

- We summarize the concerns of developers related to their decision-making toward code smell criticality, including code design and implementation, code evolution, code functionality, and developer-related factors.
- To our knowledge, we propose the first work that quantifies the gap between XAI explanation and developers' expectations in code smell prioritization. The expectation could be huge even if all their concerns are captured by the features, e.g., more than 40% of the concerns of the developers do not appear in simple explanations.
- We discover that the gap could be narrowed to an acceptable extent by adapting to developers' when selecting features, i.e., preserving the features related to the major concerns of developers as much as possible.
- We conclude that if the gap is narrowed, inspecting the top 3 to 5 important features is sufficient to meet the developers' expectations in explaining issues with simpler causes such as *Spaghetti Code*, but the explanation may be less helpful for novice users in issues with complex or controversial causes such as *Shotgun Surgery*.
- We outline the challenges and opportunities of XAI for code smell prioritization and SQA in terms of feature engineering, problem definition, and XAI methodologies.

The rest of this paper is organized as follows. In Section 2 we summarize related work. Section 3 presents the background and an example that motivates this study. Section 4 introduces the dataset as well as our manual categorization of developers' comments. In Section 5 we propose research questions and describe the experimental settings. In Section 6 we present the results of the experiment, while Section 7 discusses the implications of our study. Section 8 overviews the threats to the validity and our effort to cope with them. Finally, Section 9 concludes the paper and describes future research. **The replication code[1] and online demo[2] are available as GitHub repositories.**

2. Related work

This section summarizes the studies related to code smell detection, prioritization, and XAI applications in SQA.

2.1. XAI research in SQA

In recent years, explainable artificial intelligence (XAI) has gained significant attention from the research community due to its potential to enhance transparency and trust in machine learning models. Since regulations and developers (Jiarpakdee et al., 2021; Perera et al., 2019) demand explanations from the predictions made by black-boxed models, XAI has become a major concern in SQA practice.

Not all models are eligible for generating XAI results due to data preprocessing and model performance issues, and generating appropriate models for XAI should meet certain standards. Tantithamthavorn et al. have conducted an empirical investigation into the impact of various, including class rebalancing (Tantithamthavorn et al., 2020), parameter tuning (Tantithamthavorn et al., 2019), and feature selection (Jiarpakdee et al., 2020), on the performance and interpretability of defect prediction models. Based on their findings and the relevant literature, it is recommended that practitioners adhere to certain empirical guidelines when constructing a reliable prediction model that provides clear explanations. Firstly, multicollinearity should be reduced through appropriate feature selection techniques. Secondly, the model's performance should exceed a certain threshold, such as an area under the receiver operating characteristic curve (AUC-ROC) of greater than 0.7, to ensure its reliability. Finally, data resampling should be avoided to minimize any potential biases and to ensure generalizability. By following these guidelines, practitioners can develop defect prediction models that are both accurate and interpretable, ultimately leading to better decision-making and increased trust in machine learning models.

After generating models, suitable XAI approaches should be applied to generate reliable explanations. The validity of explainable artificial intelligence (XAI) techniques has been explored in various studies. For instance, Rajbahadur et al. (2022) discovered that a high level of agreement between the feature importance generated by a Classifier Agnostic (CA) approach such as SHAP (Lundberg & Lee, 2017) and the Classifier Specific (CS) method can only be achieved if multicollinearity is eliminated. Conversely, Jiarpakdee et al. (2022) found that most developers found the feature importance generated by another CA approach (i.e., LIME Ribeiro et al., 2016) useful. Our research (Yang et al.,

[1] https://github.com/SORD-src/ESWA23.

[2] https://github.com/SORD-src/ESWA23_demo.

Table 1
An overview of code smell detection and prioritization researches and their used features (Y is for Yes, and N is for No)

Study	Task	Approach	Smells detected				Used metric/Feature types			
			Blob	Spaghetti Code	Complex Class	Shotgun Surgery	Structure	Compre-hension	Functionality	Evolution
Lanza et al. (2005)	Detect	Rule	Y	N	Y	Y	Y	Y	N	N
DECOR (Moha et al., 2010)	Detect	Rule	Y	Y	N	Y	Y	Y	Y(Class Names)	N
JDeodorant (Fokaefs et al., 2011)	Detect	Rule	Y	N	N	N	Y	N	N	N
Fontana et al. (2015)	Detect	Rule	Y	N	N	Y	Y	Y	N	N
HIST (Palomba et al., 2015)	Detect	MSR	Y	N	N	Y	N	N	N	Y
TACO (Palomba et al., 2016)	Detect	IR+Rule	Y	N	N	N	N	N	Y(Textual Similarity)	N
SMAD (Barbez et al., 2020)	Detect	ML (Ensemble)	Y	N	N	N	Y	Y	Y(DECOR)	Y(HIST)
Alazba and Aljamaan (2021)	Detect	ML	Y	N	N	N	Y	Y	N	N
Sharma et al. (2021)	Detect	DL	N	N	N	N	N	N	Y(Word Vectors)	N
Liu et al. (2021)	Detect	DL	Y	N	N	N	Y	Y	Y(Word Vectors)	N
Kovačević et al. (2022)	Detect	DL	Y	N	N	N	Y	Y	Y(Code Representation)	N
Sae-Lim et al. (2018a)	Prioritize	IR	Y	N	N	Y	N	N	Y(Textual Similarity)	Y
Guimarães et al. (2018)	Prioritize	Rule	Y	N	Y	Y	Y	N	N	Y
KBS (Fontana & Zanoni, 2017)	Prioritize	ML	Y	Y	Y	Y	Y	Y	N	N
MSR (Pecorelli et al., 2020)	Prioritize	ML	Y	Y	Y	Y	Y	Y	N	Y

2021) on Just-In-Time defect prediction models also revealed a high level of agreement among several established CA approaches, including LIME, SHAP, and BreakDown (Gosiewska & Biecek, 2019). However, there have been concerns regarding the validity of XAI outputs, such as the disagreement in feature, rank, and sign between LIME and SHAP explanations (Er et al., 2022). Gao et al. (2022) measured the effectiveness of defect predictors' local explanations by assessing their local faithfulness and explanation precision. They found such explanations are better than random guessing, and they are not ideal for deep learning based approaches. Recent studies in other domains of XAI have also suggested that the validity concerns regarding faithfulness (Papenmeier et al., 2022) and soundness (Kulesza et al., 2013) may not necessarily translate to meaningfulness in the eyes of users. Therefore, it is crucial to calculate feature importance using established CA approaches and to ensure that the results are sound and stable. However, generating reliable results is not enough for developers to accept and trust the model outputs. More research is needed to identify the factors that contribute to user acceptance and trust in XAI explanations.

Based on the transparency brought by XAI, apart from explaining models, XAI has more practical uses in SQA. In terms of actionability, Rajapaksha et al. (2022) 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 building more accurate models, dos Santos et al. (2020) used SHAP to explain defect predictors, and they achieved better performance with simpler models built with the most important features of SHAP explanations. Similarly, Zheng et al. (2022) exploited LIME to explain the built Just-In-Time defect prediction model, they picked the most important features and achieved 96% of the predictive model's original capacity at 45% of the original effort. Widyasari et al. (2022) proposed XAI4FL to enhance spectrum-based fault localization by inferring the importance of each program unit and their suspiciousness using SHAP and LIME.

2.2. Machine learning in code smell detection and prioritization

The term code smell was first empirically introduced by Fowler et al. (1999) to describe anti-patterns in software development. Most research on code smell detection is aimed at assessing such observations of smells on Java projects. We list some works of code smell detection and prioritization from established software engineering and machine learning venues in Table 1. The used approaches include rule (i.e., combinations of customizable metrics and thresholds), MSR (Mining Software Repository), IR (Information Retrieval), ML (Machine Learning), and DL (Deep Learning). To clarify, although DL is a kind of ML approach, it is separated from ML in this table because their feature extraction and model training processes could be extremely different. To compare the source of information they used for prediction, we also summarize whether 4 types of metrics and features are presented in these works. The detailed definition of metric types will be discussed in the upcoming sections (i.e., Section 4.3 and Table 4). Most code smell works are metric-based, for example, the structure (e.g., cohesion and coupling) and comprehension (e.g., complexity) features are mostly extracted from source code ASTs (Abstract Syntax Tree), and evolution features are calculated by mining historical information of software repositories. However, unlike the other 3 types of features, the functionality features could be generated from multiple sources of information. The functionality of code may either be judged by its class or method names, or be represented by word and code embeddings. Moreover, some work also exploits textual similarity approaches to locate code fragments with similar functionalities.

2.2.1. Considered features and aspects in code smell research

The most commonly considered aspect in code smell studies is code quality. 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, Fokaefs et al. (2011) developed JDeodorant aiming to detect and refactor code smells at the same time with the help of Integrated Development Environment (IDE) plugins. Based on these tools, Fontana et al. (2015) proposed various intensity indexes for code smells using combinations of code metric values. Furthermore, Fontana and Zanoni (2017) proposed the compared baseline study of Pecorelli et al. (2020) using pure code metrics as features to predict code smell severity, which will be described in Section 4.2.1.

Code change histories are considered in the detection and prioritization of certain code commit-sensitive smells, and such approaches were proposed to cope with the shortage of code metrics by ignoring the context of the examined code components. Palomba et al. (2013, 2015) 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*.

Semantic conceptions are also considered in smell detection. Palomba et al. (2016) designed an Information Retrieval based approach called TACO to capture textual smells, further research (Palomba et al., 2018) showed textual smells are significantly different from the structural ones.

Developers' perceptions are also considered since code smell presence may vary due to different developers' preferences. Guimarães et al. (2018) and Vidal et al. (2016) prioritized code smells and their agglomerations according to developers' preference (e.g., prefer to improve coupling or cohesion), their impact on software architecture, agglomeration size, and change histories. Sae-Lim et al. (2017a, 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, 2017b; Sae-Lim et al., 2018a) 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.

Pecorelli et al. (2020) combined most of the up-mentioned aspects, e.g., structural code metrics and process metrics of software systems to measure the code smell priority. More details of Pecorelli et al. (2020) will be introduced in Section 4.2.1.

2.2.2. Machine learning approaches for code smell studies

Ichtsis et al. (2022) discovered that the results of rule- and threshold-based detection tools have a low agreement, even for simple code smells. Since most of the up-mentioned approaches are threshold-dependent, researchers intend to explore if threshold-free machine learning could detect code smells.

To jointly consider the detection results of multiple detectors, Barbez et al. (2020) proposed a machine-learning based ensemble method to aggregate the results of various classical detectors such as HIST, DECOR, and JDeodorant. The model is evaluated on a mixture of existing manually detected datasets, and it outperformed other baselines using ensemble learning.

Another line of machine learning work used the QUALITUS dataset. A recent study by Alazba and Aljamaan (2021) suggested their approach stacking basic classifiers and 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 of 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. Recently, Lewowski and Madeyski (2022) also pointed out the reproducibility issues of code smell prediction studies such as unknown data sources, inexperienced annotators, and the lack of replication package.

Deep learning methods were criticized since they were hard to interpret (i.e., black-boxed), and achieved limited or no improvement while consuming a lot more computational resources (Fakhoury et al., 2018). However, since they could save the effort of tedious feature engineering, researchers were still actively improving them. While the dataset used in other types of studies is not sufficient for training well-performed deep learners, Sharma et al. (2021) used an automatic code smell detector called Designite to generate a ground truth dataset for detection, and built a token-based code smell prediction model using deep learning and transfer learning. Liu et al. (2021, 2018) proposed deep learning approaches using CNN and RNN to detect 4 common smells including *Feature Envy*, *Long Method*, *Large Class*, and *Misplaced Class*. Meanwhile, they proposed a data generation strategy (e.g., randomly moving methods) based on real-world projects, which coped with the data-hungry problem of deep learners. However, the strategy is not feasible for generating data for more complex smells such as *Shotgun Surgery*. Kovačević et al. (2022) detected *Long Method* and *God Class* using pre-trained code embedding as features, and they outperformed metric-based machine learning predictors in terms of the performance in minority (smelly) classes. Their work is the first study to prioritize the smelly instances from the MLCQ dataset (Madeyski & Lewowski, 2020), which is collected from experienced third-party developers. Kovačević et al. (2022) also suggested inconsistencies and the lack of data annotation criteria in datasets should be addressed in further studies for better performance.

2.3. Code smell based defect prediction

The emergence of code smell indicates code maintenance degradation, which is closely related to code bug-proneness. Thus, code smell could be used for bug prediction. Taba et al. (2013) proposed antipattern(smell)-based metrics to predict bugs, and they found some antipatterns are related to higher bug density. Palomba et al. (2019) involved various code smell intensity indexes as predictors of bugs, integrated them into existing bug prediction models, and achieved ideal results. The authors also evaluated feature importance using Information Gain and they found code smell intensity is a significant contributing feature. Furthermore, through a systematic literature review, Piotrowski and Madeyski (2020) concluded that code smells and their intensities are good indicators of bugs, *God Class*, *God Method*, and *Message Chains* are useful smells for prediction. Recently, Sotto-Mayor et al. (2022) compared design code smells with traditional code smells in defect prediction, and they discovered that involving design smells extracted by Designite could boost model performance. Their study (Sotto-Mayor & Kalech, 2021) also evaluated code smell based defect prediction in the context of cross-project prediction, and using code smells alone performs better than the combination of other categories of features (e.g., smell and code metrics).

These studies reveal the potential of connecting code smell research with other SQA topics such as defect prediction. However, to achieve this goal, there still lacks research that explains how bugs are triggered by code smells from the perspective of XAI.

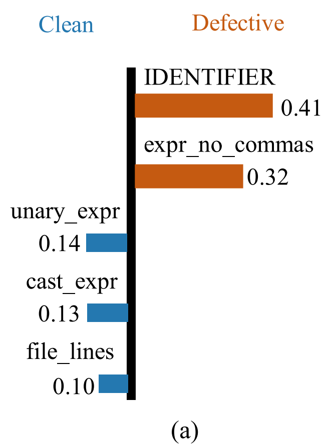
3. XAI explanations and developers' expectations

The main reason that practitioners want explanations from XAI is to (1) filter to a small set of causes to simplify their observation, and (2) generalize these observations into a conceptual model where they can predict and control future phenomena (Wang et al., 2019). Accordingly, we infer the main characteristic of code smell prioritizers should include (1) accurately prioritizing detection results according to developers' concerns to limit the number of files to inspect, and (2) explaining which factors are making a smell more critical or trivial. These advantages could enforce developers' trust in results, and save efforts on code comprehension. The first goal has been a major research direction for decades, and researchers achieved good results recently (Pecorelli et al., 2020). The second goal, however, is less frequently pursued and discussed. This section aims at clarifying the second goal.

3.1. An example of misalignment in explanation and expectation in defect prediction

Aleithan (2021) predicted a buggy instance of LINUX and explained the instance using CA approach. The prediction is correct and readable. The explanation is demonstrated in the inset (a) of Fig. 1. It mainly consists of the identifier and the syntax features of the expression. However, this example reveals a drawback of this approach. Further manual analysis shows that the bug is nothing about syntax or identifier, but actually related to an incorrect API usage, which is completely different from the explanation.

Based on the up-mentioned CA approach, Tantithamthavorn et al. (2021) further constructed an actionable advisor for refactoring buggy files. The output is demonstrated in inset (b) of Fig. 1. This advisor aimed at responding to the developers' expectations of "stop telling them what it is (why a prediction is made)" and "telling them what to do" to improve defect-prone classes. The advice is generated by focusing on the most important features in explanations. They provided the actual value of these features, and the ideal value of these features that may lead to a reduction of bug-proneness predicted by the classifier. However, this approach is also affected by the up-mentioned drawback,



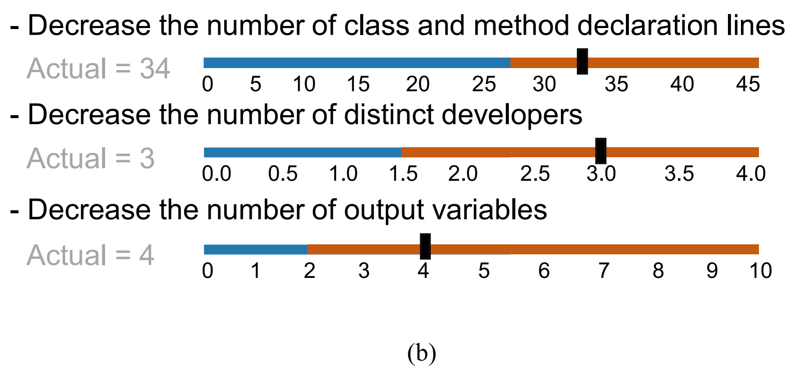


Fig. 1. Output of XAI approaches of defect prediction.

since information such as “API misuse” is not involved in the feature set.

Practically, we applaud the motivation of building actionable explanations. However, under the condition that the most relevant problems that trigger bugs are not captured by the features, we believe developers like (Aleithan, 2021) will be unlikely to trust this model. A similar point of view is proposed by Antinyan (2021) from Volvo Car Group. The author argued that irrespective of how good a predictor is, its usage should not be encouraged if it is misleading.

Recent advances in XAI made it possible to faithfully reflect how SQA model behaviors in most cases (Gao et al., 2022). However, the up-mentioned example showed that faithfulness (i.e., the ability to reflect the models’ behavior) (Papenmeier et al., 2022) and soundness (Kulesza et al., 2013) are not necessarily correlated with the users’ acceptance and trust of tools.

From our understanding, the cause of the problem is the explanations are not really expected by developers. Developers prefer the root causes to be presented in explanations, and thus they could make actionable decisions based on these explanations. If a model is built with features irrelevant to the root causes of defects, it may not be suitable for providing explanation to developers, no matter how well it performs. Currently, misalignment is unavoidable in general defect prediction since defects have complex and diverse causes, and present feature extraction and code comprehension methods can hardly capture them. However, for tasks like code smell prioritization, we can narrow our focus to the design, functional, and historical characteristics, i.e., the root causes of smells concerned by developers. We can thus investigate under the condition of effective feature engineering, whether XAI could explain SQA models. After closing the gap, we expect to see a deeper trust between users and XAI tools, as well as a wider adoption of XAI approaches in helping developers to find actionable solutions for SQA problems.

3.2. Outlining explanation and expectation

In the context of this paper, an explanation consists of (1) feature importance revealing the behavior of the model, and (2) the predicted criticality demonstrating the model decision. Explanations could be either factual or counterfactual. Factual explanations output why a prediction is made, and counterfactual ones output why a prediction is not made. We use factual explanation because (1) factual explanations are helpful for explaining correct predictions, and we are building a well-performed model (2) the appropriate way to exploit and understand counterfactual explanations is still unclear (Riveiro & Thill, 2021).

To compare the developers’ expectations with the generated explanations, we outline the users’ expectations as (1) whether the models behave as expected, which is described in developers’ comments on why they prioritized every smell, and (2) whether the models’ decisions are consistent with theirs. We use the developers’ comments as their expectations on XAI explanations, because (1) human-alike and human-friendly explanations are helpful in improving explainability and building trust between human and machine, especially for XAI (Ambsdorf et al., 2022), and people preferred the explanations consistent with their prior knowledge (Maltbie et al., 2021), (2) the original developers were acting as experienced independent prioritizers, and they were not influenced by bias such as backward reasoning (Wang et al., 2019), thus their comments are reliable for evaluation.

3.3. Motivating example

Fig. 2 depicts an example of XAI’s output of code smell prioritization and the process of matching the output with developers’ expected explanation. The example is extracted from an open-source project called Jackrabbit,[3] and the class is affected by the *Spaghetti Code* smell. Assuming that a developer has not yet inspected the smelly class, a waterfall plot generated by SHAP will be presented. The developer may check the explanation of prediction in the waterfall plot if he or she is not confident about the decision, or if he or she intends to save effort when inspecting the class. The red bar (ends with a right arrow) indicates the positive contribution of a feature to the predicted class, and the black bar (ends with a left arrow) represents the negative contribution. The length of the bar represents the extent of the contribution (longer is higher). To avoid backward reasoning, the model-predicted severity will be demonstrated later only if a preliminary human decision is made. Since the model uses multiple features for prediction, only top-K ($K = 10$) important features will be demonstrated to avoid information overload. For the example class in Fig. 2, the developer commented *the methods of this class are overly long and complex*, and labeled it as `MEDIUM` severity. In the scope of this paper, an explanation includes (1) the predicted severity, (2) the importance of top-K features that clarifies the reason that the prediction is made. We build an accurate prioritizer and exploit SHAP to generate local explanation for the `MEDIUM` prediction in the left part of the figure, and the features are ordered in descendant by importance. In the right part of the figure, we extract the aspects (codes) mentioned in the developer’s comment (i.e., size and complexity highlighted in yellow and green) to check if they appear in the top-K features (e.g., the features ringed by yellow and green squares). Finally, we generate a matrix to check if each aspect presents in comments and explanations. Further approaches to measuring the gap between explanations and expectations are assessed will be introduced in Section 5.2.

[3] https://github.com/apache/jackrabbit.

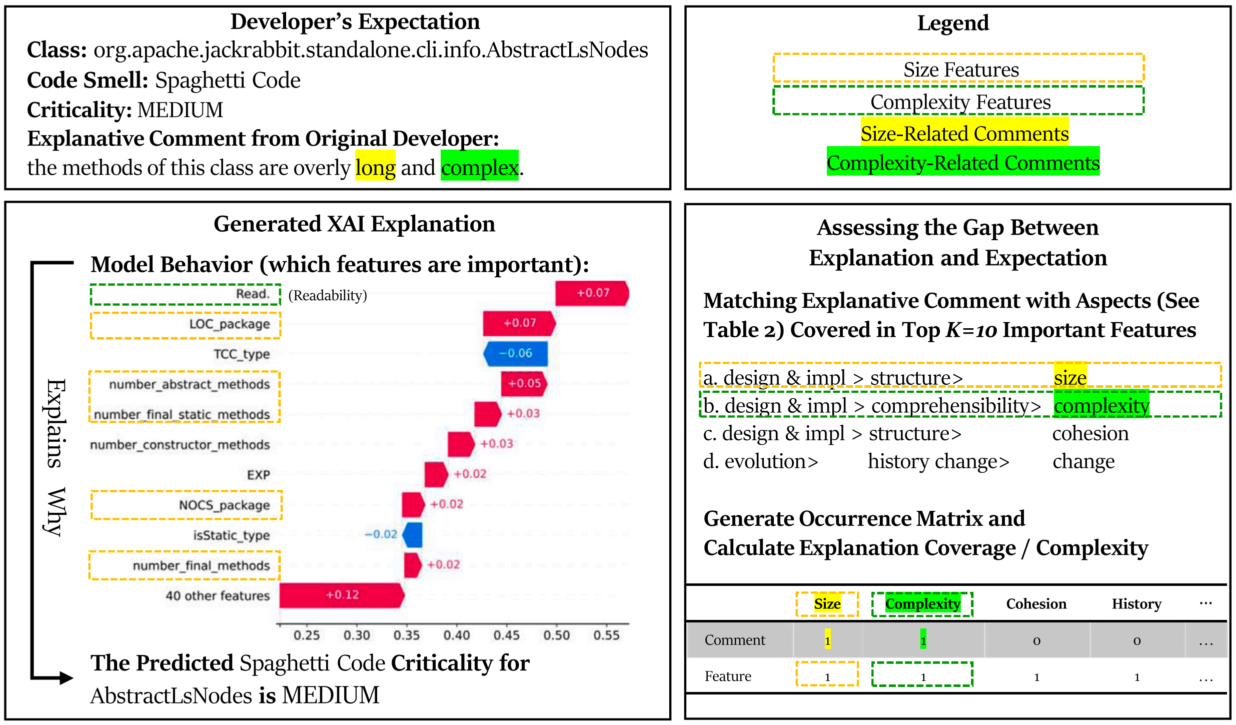

Developer's Expectation
Class: org.apache.jackrabbit.standalone.cli.info.AbstractLsNodes
Code Smell: Spaghetti Code
Criticality: MEDIUM
Explanative Comment from Original Developer:
the methods of this class are overly long and complex.
Legend
Size Features
Complexity Features
Size-Related Comments
Complexity-Related Comments
Generated XAI Explanation
Model Behavior (which features are important):
Explains Why
Read. (Readability) +0.07
LOC_package +0.07
TCC_type −0.06
number_abstract_methods +0.05
number_final_static_methods +0.03
number_constructor_methods +0.03
EXP +0.02
NOCS_package +0.02
isStatic_type −0.02
number_final_methods +0.02
40 other features +0.12
0.25 0.30 0.35 0.40 0.45 0.50 0.55
The Predicted Spaghetti Code Criticality for AbstractLsNodes is MEDIUM
Assessing the Gap Between Explanation and Expectation
Matching Explanative Comment with Aspects (See Table 2) Covered in Top K=10 Important Features
a. design & impl > structure> size
b. design & impl > comprehensibility> complexity
c. design & impl > structure> cohesion
d. evolution> history change> change
Generate Occurrence Matrix and Calculate Explanation Coverage / Complexity
Size Complexity Cohesion History ...
Comment 1 1 0 0 ...
Feature 1 1 1 1 ...

Fig. 2. An example of XAI's output and the process of matching it with developer's expectation.

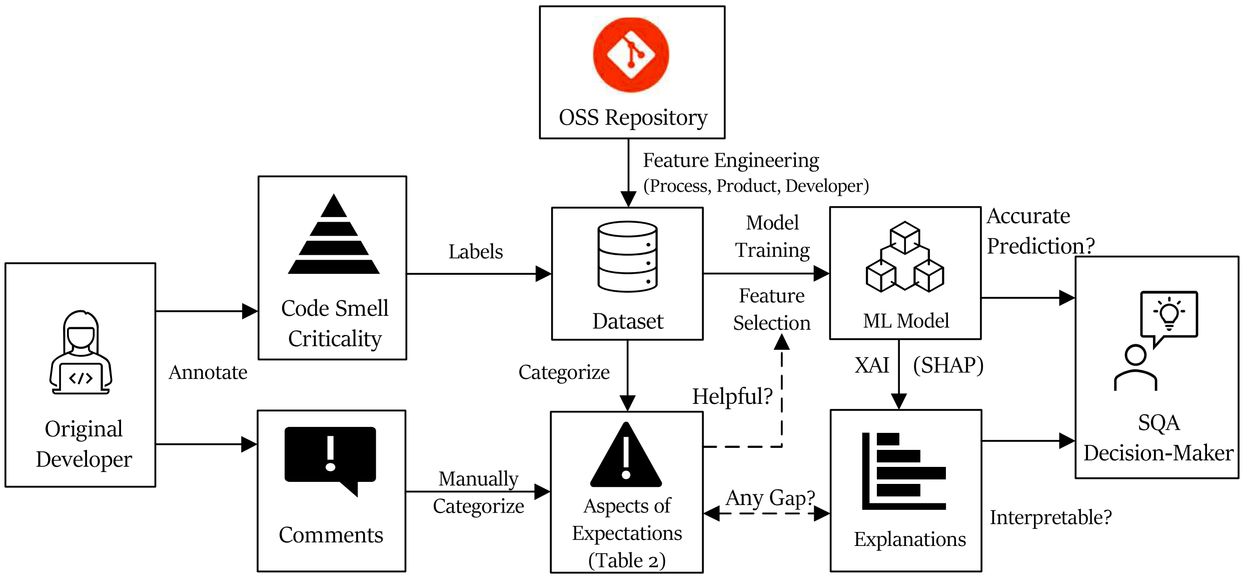


Fig. 3. Demonstration of data collection and experimental process.

4. Dataset construction and developer comment categorization

Fig. 3 demonstrates the data collection and experimental process. This section describes the dataset we used and our manual categorization of developers' comments (i.e., the left and the middle part of Fig. 3).

We conduct our study based on a recent work (Pecorelli et al., 2020) of code smell prioritization which proposed a developer-driven and machine learning based approach to rank 4 code smells. The dataset of Pecorelli et al. (2020) includes the ratings from original developers and their explanative comments about why they assign such criticality.

We use this dataset because (1) it is collected from original developers, since identifying code smell requires developers' experience on target projects (de Mello et al., 2022), it is more reliable than the other datasets (Fontana & Zanoni, 2017; Madeyski & Lewowski, 2020) collected from third-party developers and researchers, (2) it includes detailed comments on rated criticality for further analysis, and to our knowledge, such feedback is not collected in other prioritization datasets, and (3) it has significant academical impact, e.g., cited 44 times in 3 years according to Google Scholar,[4] produced by the most influential scholars in code smell research (Sobrinho et al., 2021), and published in a major venue (MSR'20) focusing on mining software repository.

[4] Data retrieved in July 16, 2023.

Table 2
Keywords used to generate class functionality features.

Feature name	Description	Keywords
is_controller	Whether the class is a controller class (Moha et al., 2010).	manage, process, control, ctrl, command, cmd, process, proc, ui, drive, system, subsystem, parser, service
is_procedural	Whether the class contain procedural (Moha et al., 2010) instructions.	make, create, factory, exec, compute, display, view, calculate, batch, thread, cluster
is_test	Whether the class is a test class or it is designed to facilitate testing.	test, junit
is_util	Whether the class is a utility (Palomba et al., 2014) class.	util, helper
is_external	Whether the class is extrinsic (Rodríguez-Pérez et al., 2022) or copied from other systems.	org.tartarus.snowball.ext, org.apache.tools.bzip2r

4.1. Dependent variables: The criticality of 4 code smells

The 4 smells concerned 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. According to Pecorelli et al. (2020), they were common and can be accurately assessed by developers with respect to their criticalities.

Blob (or God Class) refers to 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 refers to 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 & Weimer, 2010).

Spaghetti Code refers to classes that 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 refers to classes that frequently trigger co-changes of other classes. *Shotgun Surgery* is caused by high coupling. However, it can be better detected by historical change information (Palomba et al., 2015) rather than code metrics.

The developers' perceived criticalities of the MSR paper (Pecorelli et al., 2020) originally ranged from 1 to 5. Since the margins of criticality levels $\{1, 2\}$ and $\{4, 5\}$ were not clear, the MSR paper merged the unclear criticalities to 3 new criticalities, i.e., {`NON-SEVERE`, `MEDIUM`, `SEVERE`}. Thus, the prediction was performed over the merged criticalities, and the dependent variables are the 3-leveled criticalities of the up-mentioned 4 smells.

4.2. Independent variables: Datasets and extension

To ensure better coverage of developers' concerns, we merge the 2 datasets (i.e., the MSR dataset and the dataset generated by the tools employed by the KBS paper). These datasets capture characteristics in structural and development process aspects of smelly classes to generate independent variables for prediction. The complete list of used features is available in the Appendix (Table 19).

4.2.1. The MSR and KBS code smell prioritization datasets

The authors of the MSR paper tracked commits of 9 established projects of Apache and Eclipse open-source foundations in 6 months. They 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. Finally, they received 1332 instances almost equally distributed among the 4 smells. They also provided an online appendix[5] containing the original developers' comments. To perform prediction, the authors generated 20 dependent variables including product, process, developer-oriented, and code smell related features. However, their replication package only includes a subset of the features, and 6 features (i.e., `EXP`, `OWN`, `NR`, `CE`, `Intensity`, and `Refactorable`) are missing from the public version of the dataset. We sent e-mails to all authors of Pecorelli et al. (2020) on March 10, 2022 and asked for a complete copy of dataset as well as the commit id of the measured code components, but we did not receive any response. Moreover, we cannot replicate the 6 features because the commit ids are not available in the dataset, and thus we can hardly locate the exact commit they used to generate features. Nevertheless, the goal of our study is not to replicate their work, and the missing features are trivial according to the MSR paper. Thus, ignoring the missing features will not greatly impact our conclusion.

To compare the models built with the proposed features and the pure code metrics, the MSR paper used the KBS methodology as a baseline. They applied the dataset generation method of the original KBS paper to construct the baseline based on the 9 projects used in the MSR paper. 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 assessed by these features include size, complexity, coupling, inheritance, and encapsulation.

4.2.2. The extension of class functionality features

Based on class names, we extend 5 features concerning the type and context of the measured classes including `is_test`, `is_util`, `is_controller`, `is_procedural`, and `is_external`. The motivation for involving these contextual features is driven by the developers' comments in this dataset (e.g., some developers commented *it is just a test, it is good to have large tests* and assigned `NON-SEVERE` priority to *Blob* tests, and some developers ignored *Spaghetti Code* smells of utility classes). Thus, we think these features should also be included for prediction. They are generated by checking if lower-cased class names contain keywords in Table 2. The keywords of the first 4 features are extracted from (1) a state-of-the-art code smell detection tool called DECOR (Moha et al., 2010), (2) relevant words in the class names of the dataset. The keywords of `is_external` are extracted from package names of the smelly classes because some projects directly copy third-party classes into their projects (which is not a good practice), and developers perceived code smells in such classes as irrelevant.

4.3. Developers' comments and our manual categorization

The online appendix[6] of the MSR paper contains comments from 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 model explanation.

We follow these steps to summarize and categorize the comments of developers:

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

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

Table 3
The mentioned concerns in developers' comments.

Core Category	Sub-category	Code	Keyword Examples	Corresponding Features
design & impl. (88.51%)	**structure (31.47%)**	**cohesion (3.72%)**	**cohesion, related, coherence**	LCOM5, C3, TCC
		coupling (11.17%)	**coupling, external, request**	CBO, MPC, ATFD, CFNAMM, FANOUT, RFC
		size (17.56%)	**long, large, huge, blob, loc**	WMC*, num*, NO*, LOC*, NMO
	comprehensibility (36.89%)	**complexity (34.76%)**	**complex, simplify, confuse**	WMC*,Read.,WOC,DIT,NIM
		documentation (2.75%)	(lacks) documentation, messaging	
	refactorable (21.60%)	chance (18.04%)	refactor, improve, restruct	N/A (not captured)
		approach (3.96%)	reduce (complexity), divide	
	redundancy	dead code	dead	
		code clone	duplicate, repetitive	
	general design (10.11%)	design (8.98%)	pattern, structure, terrible, suck	N/A (too generalized)
		purpose (1.13%)	goal, responsibility, objective	
	impl. (14.72%)	impl. (14.72%)	call, loop, parameter	
functionality (11.25%)	**testing (5.58%)**	**testing (5.58%)**	**test**	is_test
	production (5.66%)	**production (5.66%)**	**factory, service, batch, util(ity)**	is_util,is_controller, is_procedural,is_external
evolution (16.18%)	**history change (11.25%)**	**change (10.28%)**	**change, modify, maintain**	AVG_CS,NC,NF,NCOM, DSC,Persistence
		bug-proneness (1.54%)	bug, defect, flaw	
	current impact (2.67%)	importance (2.18%)	priority, importance, critical	N/A (not captured)
		code deprecation	legacy, old, deprecated	
		risk	risk(y to refactor)	
	future delivery (5.50%)	time constraint (5.50%)	deliver (pressure), asap, postpone	
developer	community	discussion	discuss, agreement	
		organization	organization	
	individual	expertise	expert, newcomer	

1. We perform a fine-grained manual analysis in word granularity. First, we apply tokenization, stemming, lemmatization, and stop word removal. Then, we discard non-smell related words, and identified 279 keywords out of 875 words. The examples of keywords are available in our online appendix.
2. Inspired by open coding, the first author manually aggregates the words into 24 codes according to the technical, development process, and social aspects they expressed.
3. Following the principles of axial coding, we assign the codes of words to the sentences they belong to, and we summarize 13 sub-categories from the codes.
4. We invite 5 master and Ph.D. students to check if each comment belongs to each code since the meanings of sentences may not be identifiable by inspecting only words, each volunteer is responsible for 4 to 5 codes, as a result, we corrected 151 comments. We adjust the sub-categories if they conflict or overlap with each other.
5. We perform selective coding and identify 4 core concepts of these comments.
6. Finally, we assign the identified code to features in the datasets according to feature definitions.

The results are presented in Table 3. We also demonstrate their rate of occurrence in parentheses if the rate is greater than 1%. The rate is calculated by the number of comments that express an aspect divided by the number of all comments. There exist numerous size features in the KBS dataset, thus we use a wildcard(*) to represent all of them. We merge design and implementation because they are used interchangeably by developers since design could be reflected in implementation. The bolded codes are the major aspects we consider (also called "Concerned Aspects" in Fig. 3) in further RQs for explanation. The protocol of involving aspects for explanation is that they should be (1) major concerns of developers, i.e., core category mentioned in more than 10% comments, (2) covered by the features, e.g., we do not involve categories without features covered because XAI cannot generate explanations that do not exist in the feature set, (3) precise and measurable, we exclude sub-categories such as "general design" because they are either too general or can be simulated by the features in other categories.

Based on the up-mentioned protocols, we discarded 319 instances out of 1332 to ensure all aspects concerned by developers are reflected in the samples for further explanation. The details are listed in Table 4. To assess whether adapting the developers' concerns to the model generation process would help XAI in the upcoming sections, we also need to locate the most significant concerns of developers. The highlighted cells in the table represent dominant concerns of developers for each smell. For *Spaghetti Code*, *Shotgun Surgery*, and *Complex Class*, it is easy to identify the dominant ones since the trivial ones are mentioned in less than 50% samples than the dominant aspects. However, we find that the aspects reflected are more diverse for *Blob*, and we believe it is closely related to its definition, i.e., implementing multiple responsibilities. Moreover, *Blob* explanations are more likely to be discarded since developers make general and unclear comments on the reason that they assign criticalities. Thus we highlight the cells with more than 10% samples covered specifically for this smell.

5. Experimental design

The aim of our study is to evaluate whether and to what extent the behaviors of well-performed code smell prioritizers could be explained to meet the expectations of developers. The purpose of our study is helping both researchers and developers by generating human-alike and human-friendly explanations for black-box models to save efforts of SQA decision-making. To these ends, we propose the following 3 research questions.

RQ1: *Can we build an accurate and reasonable prediction model for XAI explanation?*

The motivation for proposing this RQ is to clarify the basic experimental settings according to prior guidelines of XAI for SQA (mentioned in Section 2.1). Meanwhile, we also intend to ensure the model performs well and reasonably by diagnosing its behavior and inspecting its performance metrics.

RQ2: *Is there a gap between XAI explanations and developers' expectations?*

Table 4
Distributions of concerned aspects in filtered samples.

	Coupling	Change	Cohesion	Size	Complexity	Production	Testing	Number of samples
Spaghetti Code	1%	0%	3%	63%	47%	2%	0%	249
Shotgun Surgery	63%	2%	12%	4%	24%	3%	0%	295
Complex Class	2%	14%	0%	6%	81%	13%	10%	291
Blob	2%	33%	4%	24%	15%	13%	20%	178
Total Mentions	16%	9%	4%	27%	47%	7%	6%	1013

Although (Jiarpakdee et al., 2022) suggested a well-performed SQA model should be built specifically for XAI explanation, practitioners found they may not behave like humans. Thus, there may be a gap between XAI explanations and developers' expectations. In this RQ, we evaluate how large the gap is in terms of the presence of developers' concerns and the cognitive load for developers to interpret the explanation.

RQ3: *Can we narrow the gap in RQ2 by considering the developers' expectations while building predictors?*

Based on the results of RQ2, we intend to narrow the gap by improving the automatic feature selection method by considering the developers' concerns summarized in Table 3. We assume that the prior knowledge of developers' concerns might be helpful for aligning XAI explanations with developers' expectations.

5.1. RQ1: Building models for explanation

5.1.1. Model generation and validation

In terms of model validation, we use Leave-One-Out Cross Validation (LOOCV) to assess the performance of the model and generate an explanation for every instance because it is more reliable (Tantithamthavorn et al., 2017) than the Stratified 10-fold Cross-Validation strategy used by the MSR paper. For the rest of the model validation settings, we follow the conventional settings in the MSR paper, i.e., no data balancing (since no class should be balanced compared with other classes) and cross-project prediction (i.e., the dataset of each smell was constructed regardless of the project of the affected classes).

We select a subset of features for XAI by mitigating collinearity (i.e., the correlation between pairs of features) and multicollinearity (i.e., collinearity among multiple features), since the interpretability of XAI methodologies may be harmed due to the interchangeability of these correlated features. We do not use CFS as the MSR paper did since it is designed to trade off between mitigating correlation and preserving the performance by assessing the merits of removing features. The tradeoff may fail, and it may greatly impact performance. In contrast, we exploit an automatic approach called AutoSpearman (Jiarpakdee et al., 2020) which mitigates collinearity and multicollinearity separately in 2 phases while keeping most features as possible. AutoSpearman requires 2 thresholds to determine to what extent multicollinearity should be mitigated, and we use the conventional values validated by empirical study in SQA ($\rho = 0.7$, $VIF = 5$) (Jiarpakdee et al., 2020). Feature selection is performed independently for datasets of each code smell.

In terms of classifier selection, we apply the scikit-learn package from Python to train machine learners using multiple classifiers, including K-Nearest Neighbors, Random Forest, Support Vector Machine, Multilayer Perceptron, Adaboost, Naive-Bayes, Logistic Regression, and Linear Regression. To avoid an excessively large search space, we only tune their most sensitive parameters (Jiarpakdee et al., 2020; Yu et al., 2019). The detail of tuned parameters is available in our online appendix.[7]

We assess the performance of the models using metrics for classifiers including AUC-ROC (ranged from 0 to 1), MCC (ranged from −1 to 1), and F-Measure (ranged from 0 to 1). AUC-ROC and MCC are more reliable since they are not threshold-dependent and insensitive to imbalanced data (Tantithamthavorn et al., 2017; Yao & Shepperd, 2020). AUC-ROC > 0.7 (Jiarpakdee et al., 2022) is considered an indicator of good performance, and models with MCC > 0.5 could be considered as state-of-the-art (according to Yao and Shepperd (2020)) when F-Measure ≥ 0.7). Meanwhile, it is a convention to report the F-Measure performance. We do not include Precision and Recall to avoid reporting an excessive number of metrics since classification metrics could summarize them. 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.

Moreover, we involve another performance metric called Krippendorff's Alpha (Krippendorff, 1970). Alpha measures inter-raters' agreement (higher is better), and it is different from classification metrics since it takes the distance between the predicted and real criticalities into consideration (Tian et al., 2016). We involve it as a related study in bug report prioritizing (Tian et al., 2016) suggested Alpha could handle uncertainty in human rating better than classification metrics. (Krippendorff, 1970) suggested agreement with $\alpha \geq 0.8$ is reliable, α between 0.667 and 0.8 should be considered, and agreement with α < 0.667 is not reliable. XAI results should at least not be unreliable, and they should be helpful for the decision-making process, and thus we choose $\alpha \geq 0.667$ as a signal of reasonable agreement. The equation for calculating α 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* refers to any metric or level of measurement,

[7] https://github.com/SORD-src/ESWA23/blob/main/parameters.png.

we use the ordinal metric (Krippendorff, 2011) to calculate 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.

5.1.2. Model diagnosing and explanation

We exploit an XAI approach called SHAP to generate global feature importance based on its local explanations and inspect model behavior to ensure the model does not generally make decisions based on unacceptable reasons or obvious bias. If so, we try to remove the anomaly behavior of the model.

SHAP measures the contribution of a feature value to the difference between the actual local prediction and the global mean prediction to distribute the credit for a classifier's output among its features (Rajbahadur et al., 2022) using the game-theory based Shapley values (Lundberg & 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. (8):

$$g(z) = \phi_0 + \sum_{i=1}^{M} \phi_i z_i, \tag{8}$$

where $z \in \{0,1\}^M$ is the coalition vector (also known as simplified features), 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 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. We use the Python implementation of SHAP in our study.

5.2. RQ2: Measuring the gap

Based on the filtered instances mentioned in Table 4, we generate SHAP explanations of the most accurate models for every code smell instance.

Apart from $accuracy$ metrics, we also assess the gap between XAI explanations and developers' expectations in terms of (1) the $coverage$ of explanations on expectations, i.e., to what extent developers' expectations present in the comments, and (2) the $complexity$ of explanations, i.e., whether the generated explanation is too complex for understanding, because it is impractical to let practitioners with limited cognitive ability to interpret very complex explanations for every smell. A narrower gap could be characterized by higher $coverage$ and lower $complexity$, which means the explanation can meet the expectation of developers to a greater extent with less redundant information.

$Coverage$ is calculated by the $Recall$ of features' codes that appear in codes reflected in developers' comments (see Section 3.3 for an example). Higher $Recall$ means the contents expected by the developers have more presence in XAI explanations.

$Complexity$ is measured in two aspects. (1) K is the number of most important features demonstrated to users to avoid information overload, which should be limited to the greatest extent. However, there is a tradeoff between K and $coverage$, since displaying too little information will lead to unsound results (Kulesza et al., 2013). (2) $Entropy$ of feature importance matrix (Maltbie et al., 2021) is also measured to complement with K. K is not enough to completely reflect the cognitive complexity of interpreting the explanations. Additional information displayed in larger K could be less important and informative. The $entropy$ of explanations may not increase consistently as K grows.

At present, there lack of empirical evidence for determining an ideal level of $complexity$ and $coverage$. Thus, to examine whether $complexity$ and $coverage$ of XAI explanations are reasonable, we demonstrate line plots of these metrics in inspecting K features ranging from 2 to 15 separately. However, in terms of $complexity$, empirical suggestions are made for ideal K values in related work, i.e., $K = 3$ was evaluated in related work in defect prediction (Rajbahadur et al., 2022), and experts thought inspecting top-$K = 10$ features would be helpful for explanation (Maltbie et al., 2021). We specifically focus on the two representative K values, and we also test K ranged from 3 to 10 would be more reasonable. In terms of $coverage$, we use $Recall > 0.7$ as an optimal standard since related work in defect prediction also uses this threshold to determine state-of-the-art models (Jiarpakdee et al., 2022).

5.3. RQ3: Improving feature selection to narrow the gap between XAI and developers

In this RQ, we intend to narrow the gap in explanations between XAI and developers by improving the $coverage$ in RQ2, and in the meantime, assure $complexity$ is acceptable by novice or expert users. Our assumption is that adapting to the developers' perceptions should be helpful to accomplish this goal. Compared with the present feature engineering and complex black-boxed models, we think that the feature selection process is not carefully designed with respect to the context of SQA. In the scope of this paper, the context to focus on is developers' concerns about code smell prioritization. Without this focus, features that developers are concerned about will be treated like others in feature selection when compared mathematically with correlation values, and will likely be discarded. Thus, we modify the state-of-the-art feature selection algorithm AutoSpearman (Jiarpakdee et al., 2020) to consider the priority of features according to developers' concerns in the granularity of categories. Our approach is also applicable to other feature selection algorithms like CFS. Since they are less competitive in correlation mitigation and preserving performance (Jiarpakdee et al., 2020), we are not using them as the basic approaches to improve.

5.3.1. The original AutoSpearman feature selection

The original AutoSpearman is superior to other classical alternatives such as CFS (Correlation-based Feature Selection) in terms of the consistency of selected feature subsets and the mitigation of multicollinearity. Such advantages are proved beneficial to XAI (Jiarpakdee et al., 2020). Moreover, although it is not designed for improving or preserving performance, since it is preserving features as much as possible, it has less impact on performance. AutoSpearman is performed in 2 phases, it mitigates collinearity according to Spearman's rank correlation value (ρ) in the first phase and multicollinearity according to VIF value in the second phase. AutoSpearman removes features repetitively in each phase until all remaining features are deriving VIF and ρ values lower than 2 given thresholds (conventionally, 5 for VIF and 0.7 for ρ).

5.3.2. Enhancing AutoSpearman with feature priority awareness

We intend to preserve the features that meet developers' expectations according to their priorities. Table 5 lists the priorities of different categories. The priorities are assigned according to Table 4, i.e., for dominant aspects (highlighted cells in Table 4), we assign a priority according to the number of samples mentioned, and we assign the positive infinity value to the priority of more trivial aspects (unhighlighted cells in Table 4).

We enhance the adaptation of AutoSpearman to our context while preserving its original advantages. In Algorithms 1 and 2, we propose 2 modified versions of the original AutoSpearman called AS_Reductive

Table 5
The priorities of features in different categories (generated according to Table 4).

Smell	Category	Priority
Blob	Change	1
Blob	Size	2
Blob	Testing	3
Blob	Complexity	4
Blob	Production	5
Blob	Coupling, Cohesion	+∞
Shotgun Surgery	Coupling	1
Shotgun Surgery	Complexity	2
Shotgun Surgery	Change, Cohesion, Size, Production, Testing	+∞
Complex Class	Complexity	1
Complex Class	Coupling, Change, Cohesion, Size, Production, Testing	+∞
Spaghetti Code	Complexity	1
Spaghetti Code	Size	2
Spaghetti Code	Coupling, Change, Cohesion, Production, Testing	+∞

and AS_Additive. The source code[8] of the algorithms is available in our replication package.

Algorithm 1: AS_Reductive

```
Input: $M$ is a set of candidate metrics for selection
Output: $M'$ is a subset of $M$ with multicollinearity mitigated
1  $M' = M$
2  $S = Spearman(M, M)$
3  $C_S = \{c(m_i, m_j) \in S | abs(c(m_i, m_j)) \geq 0.7\}$
       // 0.7 is a conventional Spearman's rank test threshold indicating high correlation
4  $C_S = sort(C_S)$
   /* Phase 1, Mitigating Collinearity */
5  for $c(m_i, m_j)$ in $C_S$ do
6      if $priority(m_i)$ != $priority(m_j)$ then
7          $selected.metric = min(priority(m_i), priority(m_j))$
               // Keeping the metric with higher priority
8      else
9          $selected.metric = min(mean(abs(Spearman(m_i, M - \{m_i, m_j\}))), mean(abs(Spearman(m_j, M - \{m_i, m_j\}))))$
           // Keeping the metric with a lower correlation if the two metrics share the same priority
10     $removed.metric = \{(m_i, m_j)\} - selected.metric$
11     $C_S = \{c(m_i, m_j\} \in C_S | m_i \neq removed.metric \wedge m_j \neq removed.metric)$
12     $M' = M' - removed.metric$
   /* Phase 2, Mitigating Multi-Collinearity */
13 repeat
14     $V = VIF(M')$
15     $C_V = \{v(m_i) \in V | v(m_i) \geq 5\}$    // 5 is a conventional threshold of VIF score indicating high multi-collinearity
16     $removed.metric = \{m_i | v(m_i) \in C_V \cup v(m_i) = max(priority(C_V))$
         // Discard metric with the lowest priority
17     $M' = M' - removed.metric$
18 until $|C_V| = 0$
19 return $M'$
```

First, we introduce AS_Reductive. The algorithm is described in Algorithm 1, where S is a set of Spearman coefficients for each pair of metrics, C_S is a set of Spearman coefficients that are above a threshold value. It is named reductive since the major process of the algorithm is to discard features from a full candidate metric set M. In the original and this modification of AutoSpearman, there exist multiple candidates to remove in each loop (e.g., we should remove either m_i or m_j in the correlated pair (m_i, m_j) of line 5 in Algorithm 1). The original AutoSpearman directly removes the metric having more correlation with other metrics (i.e., line 9 in Algorithm 1). This functionality overlaps with the second phase since multicollinearity also measures the correlation between one feature and the other remaining features. AS_Reductive first compares the priority of the candidate metrics to remove (i.e., the $priority()$ function in line 6), and then removes the one with lower priority (i.e., higher priority value). The $priority()$ function extracts the priority value of any feature according to Tables 3 and 5, for example, the priority of the LOC metric of *Blob* prioritization could be determined by (1) locating the category of LOC in Table 3, which is "size" in the case, and (2) extracting the priority of the category (i.e., "size") in Table 5, in this case, the priority is 2. If the priorities are consistent, the algorithm falls back to the original AutoSpearman solution in line 9. Similarly, the algorithm may identify multiple candidates in the second phase, and we discard the one with higher priority (see line 16) in advance.

Algorithm 2: AS_Additive

```
Input: $M$ is a set of candidate metrics for selection
Output: $M'$ is a subset of $M$ with multicollinearity mitigated
1  $M_{hp} = \{m \in M | priority(m) < +\infty\}$    // Filtered metrics with higher priority
2  $M_{lp} = M - M_{hp}$    // Candidate metrics with lower priority
3  $M' = AutoSpearman(M_{hp})$    // Keep metrics with higher priority as much as possible
4  for $candidate.metric$ in $M_{lp}$ do
5      if $max(abs(Spearman(candidate.metric, M'))) < 0.7$ and $VIF(M' + candidate.metric) < 5$ then
6          $M' = M' + candidate.metric$    // Add metric if it does not introduce high multi-collinearity
7  return $M'$
```

Then, we introduce AS_Additive listed in Algorithm 2. This variation is designed to preserve most features with higher priority (i.e., with non-infinite priority value, M_{hp} in Algorithm 2) in advance, then adding features with lower priority (M_{lp} in Algorithm 2) as much as possible. Thus, it first generates a subset of features with high priority using AutoSpearman. Then, it iterates through the low priority metrics in line 4 to check if adding each of them would cause a significant increase in collinearity and multicollinearity in line 5. If introducing any candidate metric with lower priority does not cause a violation of threshold, we add it to the result set in line 6.

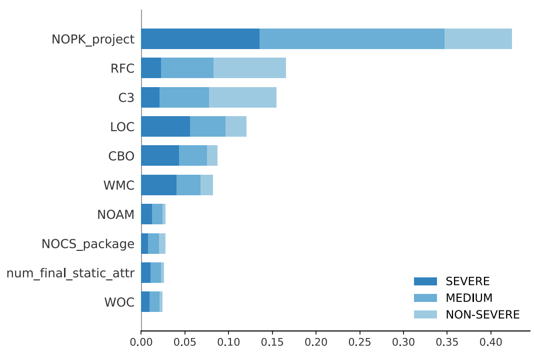
5.3.3. Statistical tests on the significance of improvement

We apply statistical measures, *i.e.*, Wilcoxon Ranksum Test ($\alpha = 0.05$, $p < \alpha$ for statically significant) to analyze the significance of the difference in the distribution of the performance metrics between the original AutoSpearman and each of our two variations. Wilcoxon Ranksum Test is applicable for data with nonparametric distributions. Meanwhile, we calculate Cliff's Delta (δ) to measure the effect size (*i.e.*, the extent of the difference) for each pair of feature values that lead positive and negative outcomes which is negligible when $|\delta| < 0.147$, small when $0.147 \leq |\delta| < 0.33$, medium when $0.33 \leq |\delta| < 0.474$, and large otherwise.

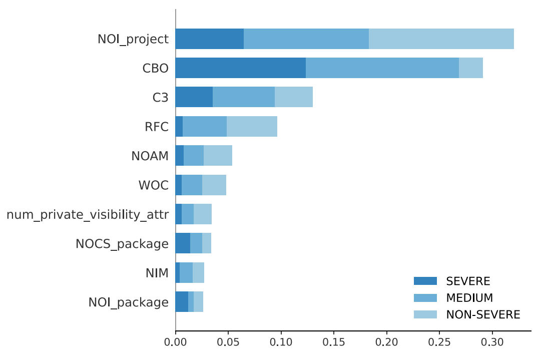
We apply statistical measures in both $complexity$ and $coverage$. The null hypothesis is that there exists no significant difference in data distribution of $complexity$ or $coverage$ between AutoSpearman and each of our two variations.

Since we intend to improve $coverage$, we hope that the null hypothesis could be rejected for $coverage$. The effect size should be

[8] https://github.com/SORD-src/ESWA23/blob/main/AutoSpearmanPriority.py.



(a) *Blob*



(b) *Complex Class*

Fig. 4. Top-10 feature importance with project metric included.

Table 6
Distribution of *Complex Class* and *Blob* priorities in different projects.

	Complex Class			Blob		
	NON-SEVERE	MEDIUM	SEVERE	NON-SEVERE	MEDIUM	SEVERE
Cassandra	0	**14**	0	7	0	**20**
Cayenne	0	0	**7**	0	0	0
CXF	12	0	20	0	0	**24**
Jena	0	**32**	0	0	**43**	14
Solr-Lucene	0	0	**21**	2	**85**	0
CDT	64	**143**	9	0	11	10
Jackrabbit	14	1	10	0	0	0
Mahout	0	0	**2**	0	0	0

non-negligible. Moreover, the value of performance metrics should increase.

For *complexity*, if the null hypothesis is rejected, *complexity* is not different from the comparator. Under the condition of *coverage* improvement, it is acceptable. If the null hypothesis is accepted, and the effect size is non-negligible, the *entropy* values should be decreasing as a sign of a simpler explanation.

These measures could not be used for *accuracy* values since our prediction is a cross-project prediction, and thus it is a directly comparable absolute value instead of a set of data.

6. Case study results

6.1. RQ1: Model diagnosis and performance

First, we train Random Forest models to diagnose them using SHAP to generate global feature importance, i.e., the mean absolute value of local SHAP feature importance. We find that `NOPK_project` and `NOI_project` (Number Of PacKages/Interfaces of a project) are ranked top for *Blob* and *Complex Class* in Fig. 4. We think it is not reasonable that a project-level metric could effectively predict whether a class is smelly to such a great extent, and developers have never mentioned their concern in the granularity of project.

To investigate why the behavior is unreasonable, we summarize the distribution of *Complex Class* and *Blob* priorities in different projects in Table 6. Apparently, certain priorities in some projects are dominant (e.g., `MEDIUM` priority in Cassandra), and some projects contain only one priority. However, project metrics remain the same within a project, and it will leak project information (e.g., Cassandra instances may be more likely to be predicted as `MEDIUM` priority since their `NOPK_project` and `NOI_project` features remain the same in the training set and test set). Consequently, the predictor may predict based on leaked project characteristics instead of code structure. Darker cells in Table 6 indicate potential bias may be introduced by project metrics due to the distribution of samples. To assess if the bias is a consequence of value distribution, we also replace all unique values of `NOPK_project` and `NOI_project` with randomly generated values, and its feature importance ranks remain the same (details available in the online appendix[9]). This indicates that the uniqueness of these values is more meaningful than the structural information they are representing. To clarify, `NOPK_project` and `NOI_project` are the only project level metrics in the selected feature set for *Blob* and *Complex Class*, and other project level metrics are discarded by AutoSpearman, because project metrics highly correlate with each other.

To ensure a more reasonable model behavior, we discard all project metrics from the features. Afterward, we train multiple classifiers to find the best-performed one. Table 7 demonstrates the best performance that each classifier could achieve. AdaBoost and Random Forest are well-performed classifiers. Compared with AdaBoost, Random Forest performs better in most cases. Thus, we use Random Forest as the classifier in the later experiments. Compared with the performance using all features, the performance of *Blob* and *Complex Class* in Random Forest declines by 7%, 6% for AUC-ROC, and 8%, 23% in terms of Alpha. However, the generated Random Forest models are still well-performed according to most performance metrics in Table 7, since AUC-ROC > 0.7 and MCC > 0.5 is considered good performance. However, the Alpha performance of *Shotgun Surgery* and *Spaghetti Code* models reveal that their agreement with developers is questionable, i.e., they are good classifiers rather than good prioritizers.

Response to RQ1

After diagnosing model behavior, we find that all project metrics should be discarded. Afterward, we can still build well-performed Random Forest models for all smells with AUC-ROC ranging from 0.72 to 0.91. However, the Alpha metric shows *Spaghetti Code* and *Shotgun Surgery* models have a low agreement with developers as prioritizers.

6.2. RQ2: The coverage and complexity of XAI's explanations on baseline approach

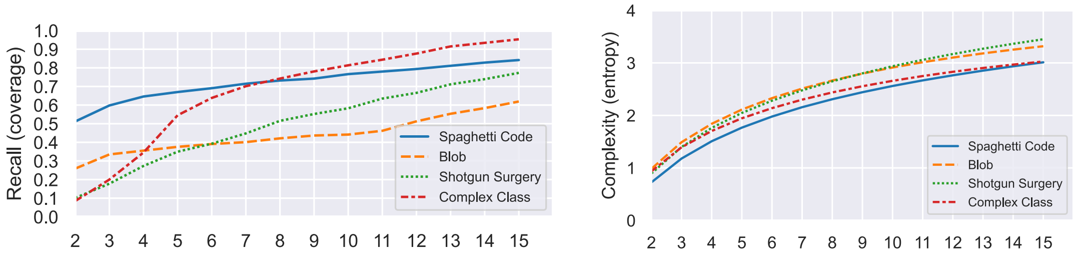
Based on the models built in RQ1 (with project metrics excluded), we exploit SHAP to generate feature importance explanations and measure their *coverage* using different *K* values. The results are demonstrated in Fig. 5.

In terms of *coverage*, a simple explanation inspecting only top-3 features (Rajbahadur et al., 2022) will unlikely be useful for all smells, because more than 40% concerned aspects are not covered. *Complex*

[9] https://github.com/SORD-src/ESWA23.

Table 7
Model performance and hyper-parameter value.

Classifier	Smell	AUC-ROC	MCC	F-Measure	Alpha	Hyperparameter
Random Forest	Blob	**0.91**	**0.82**	**0.89**	0.80	n_estimators = 140
AdaBoost	Blob	0.87	0.78	0.87	**0.82**	n_estimators = 200
Logistic Regression	Blob	0.83	0.66	0.80	0.64	tolerance = 0.1
K-Nearest Neighbors	Blob	0.81	0.61	0.77	0.64	n_neighbors = 5
Multilayer Perceptron	Blob	0.78	0.57	0.75	0.53	alpha = 0.1
Naive Bayes	Blob	0.75	0.49	0.69	0.57	var_smoothing = 0.001
Support Vector Machine	Blob	0.68	0.47	0.78	0.49	C = 0.1
Linear Regression	Blob	0.58	0.20	0.48	0.37	normalize = True
Random Forest	Complex Class	**0.88**	**0.77**	**0.86**	**0.67**	n_estimators = 100
AdaBoost	Complex Class	0.87	0.74	0.84	0.61	n_estimators = 150
Logistic Regression	Complex Class	0.83	0.65	0.79	0.58	tolerance = 0.1
Multilayer Perceptron	Complex Class	0.81	0.63	0.78	0.60	alpha = 0.000F1
K-Nearest Neighbors	Complex Class	0.75	0.51	0.71	0.53	n_neighbors = 7
Naive Bayes	Complex Class	0.74	0.48	0.68	0.37	var_smoothing = 1.00E−07
Linear Regression	Complex Class	0.52	0.08	0.39	0.24	normalize = True
Support Vector Machine	Complex Class	0.50	0.00	0.71	0.00	C = 0.1
Random Forest	Shotgun Surgery	0.72	**0.50**	**0.66**	**0.46**	n_estimators = 100
AdaBoost	Shotgun Surgery	**0.73**	0.46	0.65	0.25	n_estimators = 20
Naive Bayes	Shotgun Surgery	0.72	0.44	0.64	0.45	var_smoothing = 1.00E−05
Multilayer Perceptron	Shotgun Surgery	0.67	0.34	0.58	0.32	alpha = 0.01
K-Nearest Neighbors	Shotgun Surgery	0.64	0.29	0.53	0.17	n_neighbors = 7
Logistic Regression	Shotgun Surgery	0.61	0.24	0.51	0.34	tolerance = 0.01
Linear Regression	Shotgun Surgery	0.52	0.06	0.36	0.13	normalize = True
Support Vector Machine	Shotgun Surgery	0.50	0.00	0.63	0.00	C = 0.1
Random Forest	Spaghetti Code	**0.75**	**0.54**	**0.70**	0.48	n_estimators = 20
AdaBoost	Spaghetti Code	0.73	0.47	0.68	**0.49**	n_estimators = 80
Logistic Regression	Spaghetti Code	0.68	0.37	0.61	0.35	tolerance = 0.01
Naive Bayes	Spaghetti Code	0.67	0.43	0.59	0.30	var_smoothing = 1.00E−09
Multilayer Perceptron	Spaghetti Code	0.66	0.31	0.58	0.24	alpha = 0.0001
K-Nearest Neighbors	Spaghetti Code	0.65	0.31	0.57	0.23	n_neighbors = 7
Linear Regression	Spaghetti Code	0.59	0.22	0.45	0.36	normalize = True
Support Vector Machine	Spaghetti Code	0.55	0.24	0.59	0.11	C = 0.1



(a) *coverage* (b) *complexity*

Fig. 5. The *coverage* and *complexity* of XAI's explanations using different K values.

Class and *Spaghetti Code* prioritizers could cover most developers' concerns (*Recall* > 0.7) when inspecting more than 7 features. As for the complex explanation (e.g., inspecting 10 features), the coverage increases gradually for *Shotgun Surgery*, but still fails to achieve ideal performance. Inspecting 3 and 10 features does not make much difference in terms of *coverage* of the *Blob* model, and about 60% of concerns are not covered for *Blob* in complex explanations using top-10 features. Moreover, the *coverage* of developers' comments do not necessarily in line with the performance of models, which reveals the fact that well-performed models may not behave like humans.

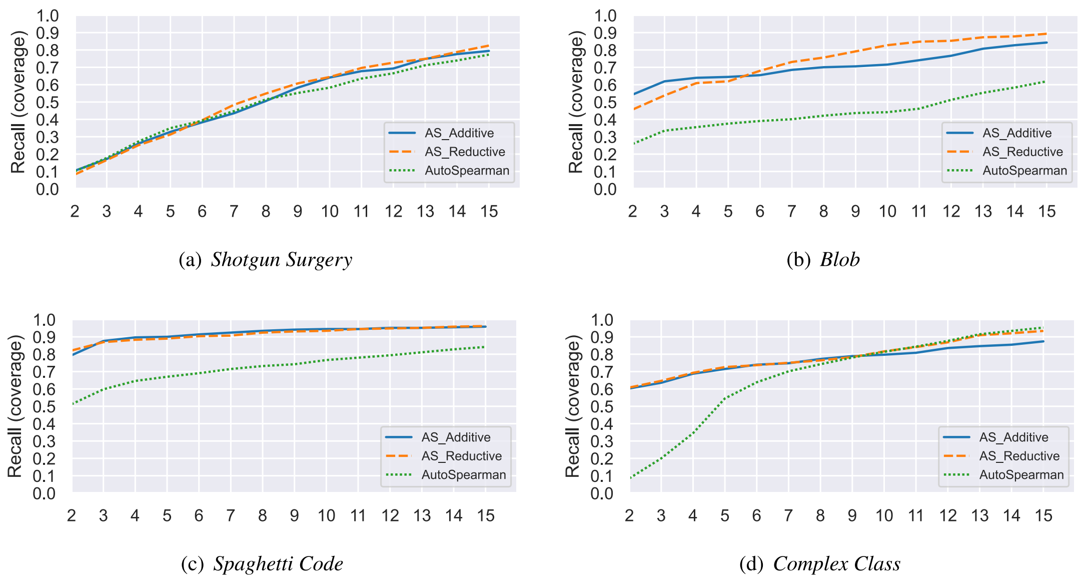
In terms of *complexity*, the *entropy* increases as more features are inspected, but it does not increase at a consistent speed as the feature number grows. This reveals the less important features that appeared later are less informative. We can also find that smells that derive much better *coverage* (e.g., *Complex Class*) are less complex in inset (b) of Fig. 5, indicating that their criticalities are easier to explain. However, compared with the *coverage* data, the *complexity* among explanations of different models does not differ to a great extent.

Response to RQ2

Using the model built in RQ1, no smell can be well explained with high *coverage* (*Recall* > 0.7) of developers' concerns using simple explanations. *Complex Class* and *Spaghetti Code* can be explained using more complex explanations that experts may tolerate, while most instances of *Blob* and *Shotgun Surgery* are not likely to be explained in top-10 features, which reveals an obvious gap of explanations between XAI and developers.

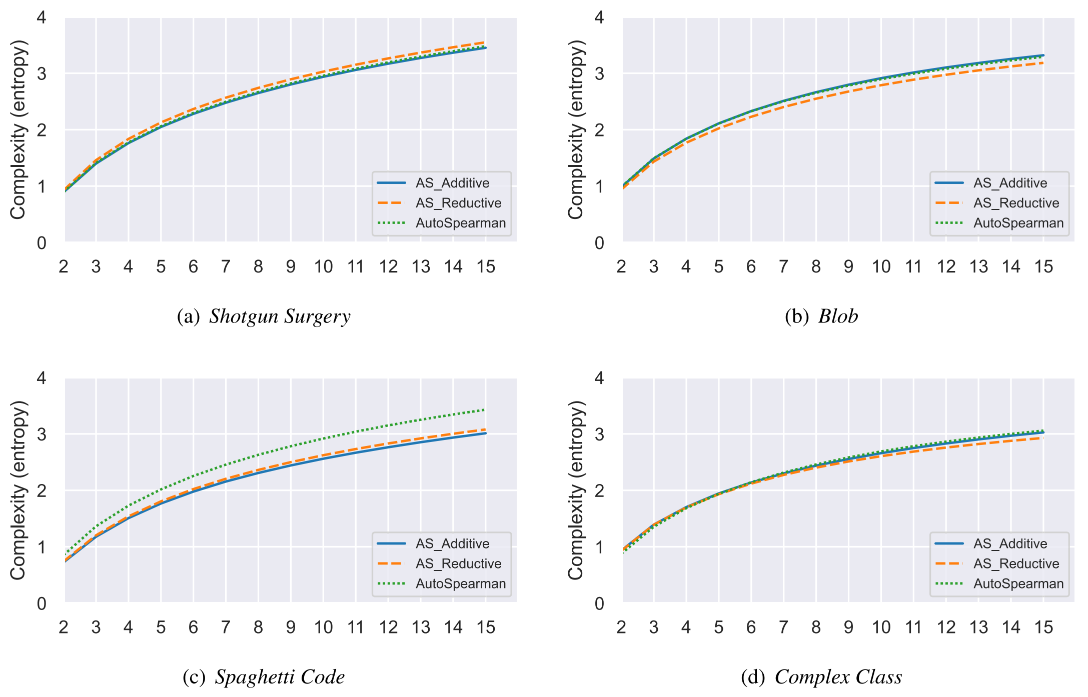
6.3. RQ3: XAI explanations after feature selection considering developers' expectation

Fig. 6 depicts the *coverage* of XAI's explanations using different modifications of AutoSpearman as feature selection methods, while Fig. 7 demonstrates the *complexity* they derive. Since Fig. 7 indicates all methods derive similar *complexity*, we focus on Fig. 6 to determine which methods are superior. We find that the modified versions of



(a) *Shotgun Surgery* (b) *Blob* (c) *Spaghetti Code* (d) *Complex Class*

Fig. 6. *Coverage* of different feature selection methods.



(a) *Shotgun Surgery* (b) *Blob* (c) *Spaghetti Code* (d) *Complex Class*

Fig. 7. *Complexity* of different feature selection methods.

AutoSpearman achieve better *coverage* for *Blob* and *Spaghetti Code*. In terms of *Complex Class*, compared with the original version, the modified versions improve the *coverage* of *Complex Class* in K ranging from 2 to 9, which is within the range of more acceptable *complexity*. In terms of *Shotgun Surgery*, the difference between 3 methodologies is not significant in K ranging from 2 to 6, but for larger K values that derive more reasonable *coverage* (i.e., > 0.7), the advantage of the modified feature selection is observable.

The hypothesis testing result from Table 8 is in line with our observation. The *complexity* statistical results show no significance in most cases 3 smells other than *Spaghetti Code*, and for *Spaghetti Code*, the difference is caused by the reduction of *complexity*, and thus our approach does not increase or even reduces *complexity*. The *coverage* statistical results are significant in 3 smells other than *Shotgun Surgery*. For *Shotgun Surgery*, although there are improvements in absolute values, they are not statistically significant.

In terms of *accuracy*, Table 9 shows the impact of feature selection to model performance. The new feature selection methods impose slight (no more than 1%) performance impact to *Blob*, *Complex Class*, and *Shotgun Surgery* prediction. However, it greatly increases the *Spaghetti Code* AUC-ROC performance by 12%, because the original AutoSpearman is discarding important size metrics, which will be introduced later in this section. It is worth noticing that the *Spaghetti Code* model built in this section is well-performed with the standard (> 0.66) of the Alpha metric. In conclusion, the modified feature selection methods are not imposing a significant negative impact on *accuracy*.

To make a practical analysis, we intend to further explain for typical K values, to what extent the improvement is. To choose a reasonable K for a detailed comparison, we use the smallest K value ranging

Table 8
Hypothesis testing result for *complexity* and *coverage*.

Smell	Approach	Agreement (*coverage*)				Entropy (*complexity*)			
		p	$p < 0.05$	δ	Effect size	p	$p < 0.05$	δ	Effect size
Blob	AS_Additive	0.005	True	1.000	Large	0.059	False	0.060	Negligible
Blob	AS_Reductive	0.005	True	0.980	Large	0.005	True	−0.120	Negligible
Spaghetti Code	AS_Additive	0.005	True	1.000	Large	0.005	True	−0.280	Small
Spaghetti Code	AS_Reductive	0.005	True	1.000	Large	0.005	True	−0.240	Small
Complex Class	AS_Additive	0.028	True	0.260	Small	0.799	False	−0.020	Negligible
Complex Class	AS_Reductive	0.009	True	0.320	Small	0.114	False	−0.060	Negligible
Shotgun Surgery	AS_Additive	0.959	False	0.010	Negligible	0.005	True	−0.100	Negligible
Shotgun Surgery	AS_Reductive	0.221	False	0.060	Negligible	0.005	True	0.100	Negligible

Table 9
The impact of feature selection to model performance.

Smell	AUC-ROC	MCC	F-Measure	Alpha	*n*_estimators	Feature Selection
Spaghetti Code	+0.12	+0.22	+0.15	+0.22	80	AS_Additive
Shotgun Surgery	−0.01	−0.01	−0.01	0.00	200	AS_Reductive
Complex Class	+0.01	0.00	0.00	0.00	120	AS_Reductive
Blob	−0.01	−0.01	0.00	0.00	30	AS_Reductive

Table 10
Settings used for comparison.

Smell	Inspected Features	*Recall*	*Entropy*	*Recall* Improvement	*Entropy* Difference
Spaghetti Code	3	0.88	1.18	+0.29	+0.18
Shotgun Surgery	11	0.70	3.15	+0.07	−0.07
Complex Class	5	0.73	1.93	+0.29	+0.01
Blob	10	0.72	2.91	+0.28	+0.04

Table 11
Coverage improvement in different categories.

	Coupling	Change	Cohesion	Size	Complexity	Production	Testing
Spaghetti Code	0.33	0.00	−0.71	+0.05	+0.66	0.00	0.00
Shotgun Surgery	+0.09	0.00	+0.20	0.00	−0.07	+0.13	+1.00
Complex Class	0.00	0.00	0.00	0.00	+0.31	+0.02	+0.30
Blob	0.00	+0.98	−0.14	0.00	−0.04	+0.21	+0.44

from 3 to 10 that can derive *coverage* ≥ 0.7. Table 10 lists the setting used for comparison. Compared with the original version, the modified AutoSpearman achieved an improvement of *coverage* by 7% to 29%, with negligible changes in *complexity* for *Shotgun Surgery*, *Blob*, and *Complex Class*, as well as lower *complexity* for *Spaghetti Code*.

Based on the settings in Table 10, we list the difference of *coverage* in the concerned categories in Table 11. Darker cells are major concerns of developers (same as Table 4). For *Complex Class* and *Spaghetti Code*, the *coverage* of major concerns significantly increases. For *Blob*, although the *coverage* of the complexity category decreases slightly by 4%, the *coverage* of the other 3 major concerns of change, production, and testing increases by 98%, 21%, and 44% respectively. The cost of such improvement is that some aspects less concerned by developers (e.g., cohesion) are less covered. We think it is an acceptable tradeoff because practitioners prefer explanations within the scope of their prior knowledge (Maltbie et al., 2021) and meet their expectations (Kocielnik et al., 2019). They tend to ignore information that is inconsistent with their prior beliefs (Riveiro & Thill, 2021). Meanwhile, it does not increase the *complexity* of the explanation outputs. We will further discuss the cost of covering such rare concerns in Section 7.2. For *Shotgun Surgery*, its improvement in *coverage* is less significant since the major concerns of complexity decline by 7% while the *coverage* of another major concern (i.e., coupling) increases by 9%.

Furthermore, in Tables 12–15, we demonstrate the most frequently appeared K features in the top-K important features of XAI explanations. The Proportion column demonstrates the proportion of a feature that appears in the top K important features of an instance with respect to all instances. The bolded metrics are significant differences that lead to improvement.

For *Spaghetti Code* in Table 12, a cohesion metric is replaced with a complexity metric. In terms of *Complex Class* in Table 13, a cohesion metric is replaced with a metric specified in complexity and size, which is more adaptive to the definition of this smell. Similarly, for *Blob* explanations in Table 14, a cohesion feature is replaced with a process metric. These new features are almost equally important compared with the original ones, and they can cover the major concerns of developers in Table 11, which results in the improvement of *coverage* while not hindering *accuracy* and *complexity*. In Table 15, a new coupling feature (i.e., `CBO`) is introduced to replace another coupling feature (i.e., `CF-NAMM`), which improves the *coverage* of coupling concern. A size metric is replacing a complexity metric, and thus results in the decline of the *coverage* of complexity concern. Consequently, the improvement of *coverage* for *Shotgun Surgery* is not as significant as the other 3 smells.

Response to RQ3

The modified AutoSpearman feature selection methods could generate feature subsets more adaptive to the developers' expectations. The *coverage* could be increased by up to 29%, with little or no negative impact on *complexity* and *accuracy*, and thus all smells could be explained in a simple manner (e.g., 3 and 5 features for *Spaghetti Code* and *Complex Class*) and a more complex manner preferred by experts (e.g., about 10 features for *Blob* and *Shotgun Surgery*).

7. Discussions and implications

In this section, we discuss some open questions and present the implications related to XAI for SQA.

Table 12
Top-3 frequently appeared features in top-3 important features of XAI for *Spaghetti Code* prioritization.

Rank	AutoSpearman			AS_Additive		
	Metric	Category	Proportion	Metric	Category	Proportion
1	**C3**	**Cohesion**	85%	**Readability**	**Complexity**	81%
2	NOM_package	Size	29%	NOM_package	Size	76%
3	num_abstract_methods	Size	44%	NOCS_package	Size	16%

Table 13
Top-5 frequently appeared features in top-5 important features of XAI for *Complex Class* prioritization.

Rank	AutoSpearman			AS_Reductive		
	Metric	Category	Proportion	Metric	Category	Proportion
1	CBO	Coupling	98%	CBO	Coupling	96%
2	RFC	Coupling	92%	RFC	Coupling	96%
3	**C3**	**Cohesion**	87%	**WMC**	**Complexity, Size**	92%
4	NOAM	Size	60%	NOAM	Size	79%
5	WOC	Size	45%	RFC	WOC	48%

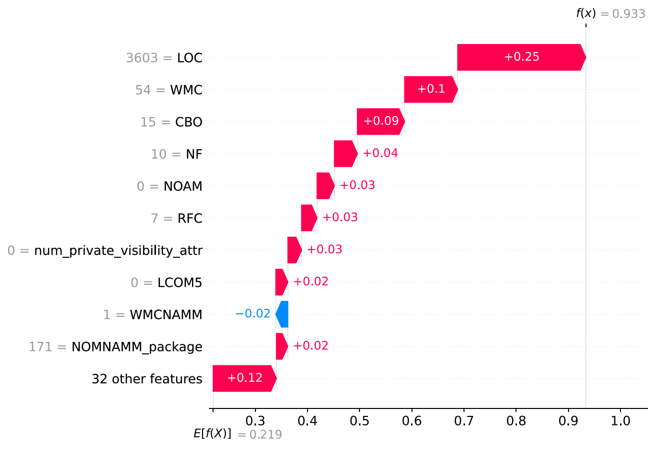


Fig. 8. XAI of *Blob* prioritization of IDLLexer.

7.1. Explaining the overgeneralized size metrics

Code size patterns are widely considered in SQA. In the scope of defect prediction, dos Santos et al. (2020) even found using only the LOC metric can achieve very ideal defect prediction performance. Recent work (Antinyan, 2021) questioned whether it is practical to include size metrics such as lines-of-code (LOC) in defect prediction models. They argued LOC is misleading, since LOC cannot serve for any quality improvement activity. Size metrics like LOC are usually simple and important because they provide high level abstraction which could boost up the performance of predictors. However, they raise the concern that the overgeneralization of size metrics is hindering model interpretability.

In terms of code smell prioritization, involving LOC is more reasonable in motivation since lengthy code is presented in the definition of certain smells such as *Spaghetti Code*. However, we also discover some worrying trends caused by the overgeneralized size metrics. For example, the *Blob* class IDLLexer[10] is perceived as smelly because it has *too many responsibilities*. However, from Fig. 8, we can see that the predictors are making decisions mainly based on size metrics, instead of cohesion metrics expected by developers. Such predictors and their behaviors are less comprehensible, thus they may be considered not trustworthy by practitioners. For example, it could be simply tricked by adding meaningless code components. If size metrics are included in explanation, it remains an open question to interpret what they represent.

7.2. Meeting functionality concerns of developers

Fig. 9 depicts XAI explanations of a *Complex Class* called CommandFactory,[11] the developer commented *the factory classes are all hard to understand because they loop over multiple elements and this may create confusion … Nevertheless, they would not require urgent changes*, and assigned a MEDIUM criticality. The inset (a) demonstrates why the model makes a MEDIUM prediction, and the inset (b) reveals why it does not make a NON-SEVERE prediction. From the rows of is_procedural in the 2 insets, we can find that a factory class (captured by the is_procedural feature) is less likely to have higher criticality. We believe such an explanation is practical and useful since the original developer thought the functionality and characteristic of factory class is an important factor that lowers the criticality of *Complex Class*.

However, although the functionality aspect revealed in Fig. 9 is important, it is rare and poorly covered. Table 16 demonstrates the *coverage* of different aspects, the aspects are ranked in ascendant according to the proportion of mentions in all samples, and the gray cells are dominant aspects (major expectations of developers). We can find that the functionality aspects (i.e., testing and production) are less covered by XAI explanations, revealing that XAI has less potential in capturing such rare but important concerns.

Boosting up the numbers of involved features could effectively raise the *coverage* on them. For example, in the context of Table 16, *Spaghetti Code* explanations are performed using only 3 features, and they almost ignore the aspects other than the dominant aspects. Boosting up the inspected feature numbers to 24 could ensure all 2 functionality aspects are covered at the cost of a dramatic increase of *complexity* (from 1.18 to 3.46). There exists a trade-off between *coverage* and *complexity*, and we think the cost of covering these aspects is too high. However, the trade-off should be customizable in practice, especially for expert users, it may be feasible to demonstrate more detailed explanations for them. However, instead of tuning explanations, we believe it is more practical to introduce customizable filters for developers to discard the classes that they are unwilling to maintain.

7.3. Debiasing and adapting to practitioners' beliefs

During manual verification, we realize that some developers have a bias toward code maintenance and refactoring (e.g., maintainability is worth improving, tests are important, and so on). For example, the developer of SolrTestCaseJ4[12] thought *Blob* tests are good practice (*it is a test, it is good to have large tests*), and the developers of multiple tests thought they should not be maintained unless they malfunction (*it is a test, so no need to really think about its complexity*, *the tests work well*, etc.). Meanwhile, others (e.g., a Lucene developer) believe *tests are important and especially this class has a key role in Lucene. So, it needs to be properly refactored.* Although an agreement has not been reached in the

[10] The full class name is org.apache.cxf.tools.corba.processors.idl.IDLLexer.

[11] The full class name is org.eclipse.cdt.debug.mi.core.command.CommandFactory.

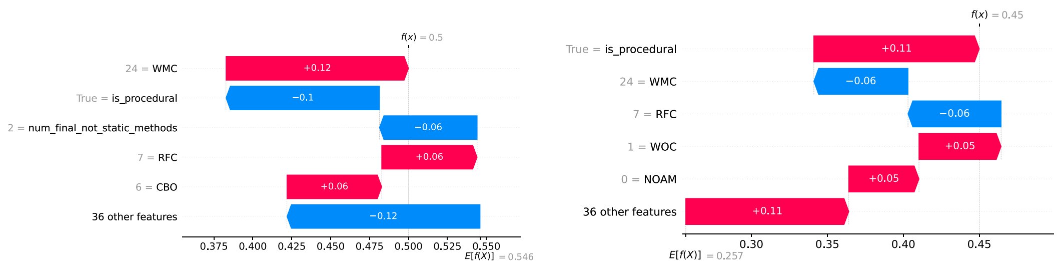
[12] The full class name is org.apache.solr.SolrTestCaseJ4.

Table 14
Top-10 frequently appeared features in top-10 important features of XAI for *Blob* prioritization.

Rank	AutoSpearman			AS_Reductive		
	Metric	Category	Proportion	Metric	Category	Proportion
1	RFC	Coupling	99%	**NF**	**Change**	99%
2	WMC	Complexity, Size	98%	RFC	Coupling	99%
3	CBO	Coupling	93%	LOC	Size	93%
4	**C3**	**Cohesion**	85%	CBO	Coupling	85%
5	LOC	Size	74%	WMC	Complexity, Size	74%
6	num_final_static_attr	Size	70%	NOAM	Size	70%
7	NOAM	Size	58%	NOCS_package	Size	58%
8	NOCS_package	Size	54%	WOC	Complexity	54%
9	TCC	Cohesion	62%	TCC	Cohesion	51%
10	WOC	Complexity	58%	num_final_static_attr	Size	39%

Table 15
Top-11 frequently appeared features in top-11 important features of XAI for *Shotgun Surgery* prioritization.

Rank	AutoSpearman			AS_Reductive		
	Metric	Category	Proportion	Metric	Category	Proportion
1	AVGCS	Change	100%	AVGCS	Change	99%
2	is_test	Testing	86%	is_test	Testing	98%
3	persistence	Change	75%	persistence	Change	76%
4	DSC	Change	63%	TCC	Cohesion	63%
5	NOCS_package	Size	54%	DSC	Change	58%
6	TCC	Cohesion	51%	LOCNAMM	Size	37%
7	num_constructor_NotDC	Size	37%	MPC	Coupling	36%
8	LOCNAMM	Size	35%	NOCS_package	Size	34%
9	MPC	Coupling	34%	CBO	Coupling	34%
10	CFNAMM	Coupling	32%	num_standard_design_methods	Size	34%
11	WOC	Complexity	30%	num_constructor_DC	Size	29%



(a) XAI of `MEDIUM` *Complex Class* Prediction (b) XAI of `NON-SEVERE` *Complex Class* Prediction

Fig. 9. The XAI results of `CommandFactory` prioritization.

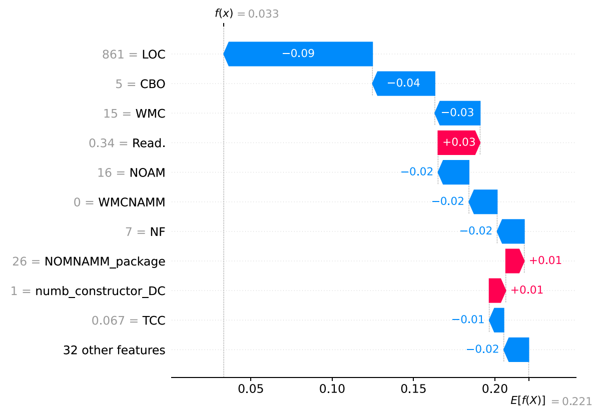
Table 16
The *coverage* of different aspects.

	Complex Class	Shotgun Surgery	Spaghetti Code	Blob
Cohesion	100%	83%	0%	86%
Testing	3%	100%	0%	61%
Production	26%	13%	0%	25%
Change	0%	100%	N/A (not mentioned)	100%
Coupling	100%	73%	0%	100%
Size	100%	100%	99%	100%
Complexity	98%	51%	81%	96%

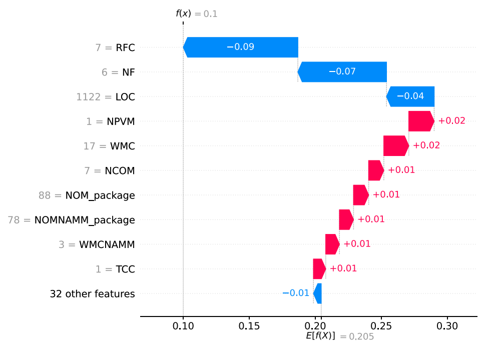
industry, in empirical studies of academia, it is generally accepted that test quality is worth improving (Wu et al., 2022). It is a huge challenge to inform XAI users about some information they do not consider useful but actually important (Dam et al., 2018).

Sometimes, developers do not know how to interpret code smells with complex causes. For example, the majority of *Blob* instances are discarded due to the overgeneralized comments, because they cannot clearly identify the technical debt in the code. Meanwhile, many developers considered the reasons more related to other smells while rating *Shotgun Surgery* criticalities (e.g., *class does too much*, *the code is complex*, *documentation is lacking and it makes hard to understand some methods*, and so on). Moreover, developers tend to focus on code structure issues rather than other factors. For example, a major symptom of *Shotgun Surgery* is that it triggers changes. However, change factors are not major concerns of developers, and they account for only 2% in *Shotgun Surgery* comments according to Table 4.

On the one hand, we may need an additional set of features to capture the preferences of individual developers from a personalized perception. For example, for developers who only perform refactoring on defective classes, recommending defect-prone classes may be more useful. On the other hand, although the adaptation to the developers' perception is vital, the reaction toward their opinions should be reconsidered. We may need to guide them if they could not really interpret the problems, which calls for the disruptiveness of XAI explanations. In terms of disruptiveness, although we cannot reach the original developers and ask whether the explanations provide additional insights, it is very likely that the ignored change aspects of XAI explanations of *Shotgun Surgery* will be helpful for them to make SQA decisions. We believe that under the condition that XAI explanations are perceived as trustworthy by developers, further study should be conducted on whether XAI could help and inspire developers to make actionable decisions.



(a) `SEVERE` *Blob* Prediction of `FILTERFROMFILE`



(b) `NON-SEVERE` *Blob* Prediction of `L3`

Fig. 10. Examples of contrastive explanations for wrong predictions.

7.4. Providing contrastive (why-not) explanations for wrong predictions

Molnar (2022) suggested human-friendly explanations should be "contrastive" (Lipton, 1990), i.e., XAI should be able to answer why a prediction is made instead of another one. Although we cannot evaluate the original developers' opinions toward contrastive explanations, we can inspect the cases in which models make predictions that are different from the developers' annotations. Thus, we can discuss whether reasonable contrastive explanations could be generated.

For a *Blob* class called `FILTERFROMFILE`,[13] the developer commented *Too many LOC, it should be refactored*, and annotated it with a `SEVERE` criticality. In contrast, the model makes a `MEDIUM` prediction. Instead of checking why the model makes its prediction, the developer may ask why the prediction is not `SEVERE`. The inset (a) of Fig. 10 depicts the feature importance of why the prediction is not `SEVERE`. Although `LOC` is also an important feature for the model to make its decision, the actual `LOC` value is causing the model to make a lower priority prediction. After checking the average values of `LOC` for code components in different criticalities, we find that the average `LOC` values for the `SEVERE`, `MEDIUM`, `NON-SEVERE` criticalities for *Blob* are 2537, 1084, and 1036 respectively. Thus, we infer that the `LOC` value smaller than 1000 is not considered a characteristic of `SEVERE` smell by the classifier.

For another *Blob* class called `L3`,[14] the developer commented *the modifications to this class are generally simple, that is why we do not refactor it*, and annotated it with a `NON-SEVERE` criticality. Instead, the model makes a `MEDIUM` prediction. The contrastive explanation in inset (b) of Fig. 10 suggests that the number of fixes (`NF`) is one of the major reasons that the prediction is not `NON-SEVERE`, which indicates that fixes are performed frequently in this class. After checking the commit logs, we find that the modifications are logically simple modifications but may cost a lot of effort. For example, to fix a bug[15] related to this class concerning Map-Reduce query inconsistency between 2 systems, it requires 187 additions and 195 deletions among 13 classes in the same package. Thus, we believe this contrastive explanation may encourage the developer to reconsider the condition of the class.

Although the up-mentioned cases show contrastive explanations are informative, we cannot conclude that they are always useful. Meanwhile, prior work (Riveiro & Thill, 2021) also discovered that for wrong predictions, providing only contrastive explanations is not enough for improving end-users trust in the models. However, their experiment is performed in a relatively simple task, i.e., classification of texts, which is different from our work. We believe how and to what extent contrastive explanations can contribute to narrowing the gap between XAI and developers would be worth exploring in future work.

7.5. Using rule-based XAI approaches to prioritize code smells

The decision-making process of rule-based approaches is transparent to developers by nature. Compared with black-boxed learning-based prediction models such as Random Forest, it has huge advantages in interpretability.

For *Complex Class* and *Spaghetti Code*, using only 3 to 5 features will be able to generate an explanation that meets developers' expectations to a great extent. Thus, we are curious about whether simpler rule-based methods may also work for such issues. We assess correlations of every feature with respect to the criticality, and we find that only 1 feature (i.e., `NF`) has a high correlation (> 0.7) with the criticality of the *Spaghetti Code* smell, and no feature has such high correlation with the criticality of *Complex Class*. Thus, we intend to explore whether highly correlated values would be helpful for rule-based XAI to better fit the predicted class.

We exploit a rule visualizing XAI technique called RuleMatrix (Ming et al., 2019) to generate explanations from models. RuleMatrix is a matrix-based visualization of generated rules from the black-box model to help users navigate and verify the rules and the model, and it achieved high fidelity and evidence on the tested datasets. We use the same dataset in our previous experiments to build an MLP (Multi-Layer Perceptron) model (as suggested in the demo and the paper), and explain the model using RuleMatrix. However, the approach derives low fidelity (i.e., the approximation of a given model) and low evidence (i.e., the accuracy of the generated rules). For example, it achieves 38% accuracy (ACC) for *Complex Class*, and 61% ACC for *Spaghetti Code*. More details are included in the online appendix.[16] Thus, from our preliminary result, we conclude that rule-based approaches such as RuleMatrix are not practical, even for simple SQA tasks.

[13] The full class name is `org.apache.pig.test.utils.FILTERFROMFILE`.
[14] The full class name is `org.apache.pig.test.pigmix.mapreduce.L3`.
[15] https://issues.apache.org/jira/browse/PIG-3915.
[16] https://github.com/SORD-src/ESWA23.

7.6. The opportunities and challenges of XAI for SQA

XAI could be used to explain how SQA models behave and diagnose them. From the results of our study and the findings of prior SQA studies (Jiarpakdee et al., 2022, 2021; Rajbahadur et al., 2022), we believe XAI approaches can reveal model behavior while enabling AI and data experts in software analytics to understand why the model makes predictions technically. Through diagnosing the models, they may tune the models to obtain more reasonable behaviors (e.g., the process of discarding project level features in RQ1). Model diagnosing could be a major purpose of XAI to reveal the technical details and explain why the model performs well (or badly).

XAI explanations could be aligned to developers' expectations. From the results in our work, we believe XAI results for well-performed models could assist code smell prioritization for developers, especially for issues with simpler causes. However, for complex smells, interpreting the results may be challenging for novice users.

XAI has the potential to discover ignored factors related to SQA problems. For example, change history for *Shotgun Surgery* prioritization in our work is rarely mentioned by developers, but they are actually important factors to prioritize this smell. Furthermore, researchers (Jiarpakdee et al., 2022) still hoped XAI could discover new findings in SQA and build empirical theories. We believe the major challenge of this goal is that the explanations from developers and XAI may be seriously misaligned (e.g., in terms of complexity, concerning issues, the granularity of measurement, and so on), and thus developers may perceive them as untrustworthy or less useful. Moreover, if relevant or determining factors of an SQA problem cannot be captured, we can hardly build any reliable empirical theories. At present, we believe investigating the developers' expectations of SQA models and capturing these factors is the most urgent task.

Based on the up-mentioned discussions, we suggest several future research directions of XAI for SQA. Since "bugs are not the same" (Catolino et al., 2019), i.e., the root cause, impact, and representation of bugs in terms of code characteristics and project information are not similar. Such differences do not only exist in bugs, but also in other SQA issues such as code smell, e.g., characteristics and causes of smells differ in different domains or contexts of applications. While studies have already been conducted on the root causes of different kinds of SQA issues, little is known about how to capture the root cause of them through feature engineering. We think our present goal of XAI for SQA should shift from explaining general defects, and it should be set in the context of generating reliable explanations for SQA tasks in narrowed scope or divided subtasks (e.g., specific issues in a given context) through (1) more detailed empirical studies on bugs with different origins or in a different domain, and (2) more advanced technologies in feature engineering or automatic feature extraction approaches.

8. Threats to validity

In this section, we clarify the threats to the validity of our work. Construct validity refers to the relationship between theory and observation. Conclusion validity is related to treatment and outcome. Internal validity concerns the variables that could affect the outcome. External validity is about the generalizability of results.

8.1. Construct validity

The reliability of SHAP may be a threat to construct validity. CA approaches were proved reliable in related study (Jiarpakdee et al., 2022), and we followed the guidelines to generate reliable explanations.

Table 17
Agreement between SHAP and Information Gain.

	Blob	Complex Class	Spaghetti Code	Shotgun Surgery
Top-3 features	75%	75%	60%	75%
Top-10 features	71%	67%	63%	63%

However, the up-mentioned studies did not test Information Gain, i.e., an algorithm conventionally applied in code smell research to generate only global feature importance. We also evaluate its agreement (Rajbahadur et al., 2022) in top-3 and top-10 feature importance with SHAP. From the results in Table 17, we find that the two approaches may not agree with each other to a great extent (i.e., the agreement could be lower than 70%). However, we believe these two feature importance measures are focusing on different aspects. Information Gain is calculated based on data rather than models, and it measures the features' capability of eliminating the uncertainty of the dataset. Instead, SHAP measures directly the behavior of models over local instances, and we believe it is more appropriate in our paper since we need to specifically explain why the model makes the prediction for every instance.

Meanwhile, the lack of theory to measure the outcome of XAI for SQA may be a problem. First, in terms of the concerned aspects of developers summarized and categorized in Table 2, they may be biased due to a limited number of samples. However, since both structural and contextual concerns are covered in our manual assessment, we believe to a great extent, our summarization can match with the observation of practitioners. Second, measuring the gap in the granularity of the concerned aspects may not be ideal for practitioners, e.g., although a practitioner may focus on history changes, they may not focus on the number of fixes (`NF`) in history. `NF` is an important feature that improves the performance of *Blob* explanation in Table 14. Such an explanation may not be regarded as precise. According to the observation in Pecorelli et al. (2020), features may "simulate" others to some extent, and we think simulating behaviors of features in the same category can be tolerated because they are closely related. In practice, we may provide an explanation for each feature and explain the effect of such "simulation", or we should present the features of the same categories discarded by AutoSpearman due to high correlation. The design of the interactive interface is within the scope of HCI research rather than this study. Third, there might be other aspects to measure the gap between explanations of XAI and practitioners, and in some cases, such differences in explanations may become a good feature providing extra insights (i.e., disruptiveness) for practitioners. However, we believe practitioners are willing to see their concerns in the explanations in the first place, and it proves the classifier is trustworthy. Otherwise, the developers will not be able to confirm whether the "insights" are actually random errors generated by chance.

8.2. Conclusion validity

Since the MSR dataset is not fully available, and we cannot recover the missing process and product metrics (see Section 4.2.1 for details), which may cause an overestimation of the gap since some aspects may be covered by the missing features. However, such features are proved trivial in the MSR paper, and thus they are less likely to appear in the top important features. Moreover, except for the `Refactorable` feature, the missing features can be classified into the major aspects that are already covered by other features. Currently, the models we build are already state-of-the-art according to performance metrics. Thus, we think they are not likely to affect the conclusion to a great extent.

Specifically, we need to explain the threat of missing the `Refactorable` feature since it may fit the sub-category with the same name in Table 3. The authors used a code smell detector called JDeodorant (Fokaefs et al., 2011) to generate this feature since it is a detector that also provides refactoring suggestions. However, we think the developers' concerns in the refactorable sub-category can hardly be covered by the feature, which only focuses on the technical aspects measurable by the structure and complexity metrics. The barriers to refactoring are the feasibility to refactor and the limited resources for refactoring rather than the ability of detection tools. When developers in this dataset mention they cannot refactor, they usually mean by (1) refactoring should be postponed because of delivery deadlines or development plans (e.g., *I would postpone the refactoring till it is stable*), and (2) they cannot understand the code or architecture (e.g., *this is naturally complex, I would not know how to refactor it*). To capture the characteristic of the first issue, we should capture more development process, community, and developer features. For the second issue, such difficulties could be reflected by complexity metrics. As a result, we think the threat of missing the `Refactorable` feature is not significant.

8.3. Internal validity

The reliability of subjective aspects is an unavoidable threat to our work. First, the annotations and comments from the original developers may contain errors or biases, e.g., the biases we discussed in Section 7.2. In terms of errors, we find that a class from Apache Pig named `TezDagBuilder`[17] is mistakenly considered as a test class (the developer commented *since this is a test, it is ok for us to have it a bit larger than other classes*), and such error should impact the validity of our work. However, through careful manual identification, we find no more obvious errors in the dataset. Second, our identification of developers' comments may also include bias. We involve experienced developers and researchers to address it, and we also provide a replication package of our study for the research community to replicate our results in other contexts.

8.4. External validity

A notable generalizability issue is that this study is conducted in the context of cross-project prediction, which is designed to cope with cold start problems. Apparently, there are other scenarios to consider, 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 in the context of code smell at present. The reason that we use the MSR dataset rather than other ones is also described in Section 4.

Another generalizability issue is to what extent the findings in this case study can be transferred to other notable SQA analytics tasks. In the scope of machine learning approaches relies on feature engineering, we believe generalizability is determined by (1) to what extent the major concerns of practitioners about these problems are reflected in the features or data used to make predictions, and (2) how precise and narrowed the prediction target is. If the features do not include such concerns, the gap will be much larger, and it can hardly be closed. If the prediction target is overgeneralized, the explanations will be unlikely to be informative and helpful, which is also discussed in Section 7.6.

In the scope of deep learning approaches that do not rely on manual feature engineering to such a great extent, our conclusion is not applicable since these methods react in a different manner. Although it is possible to visualize co-reacting neurons to observe how neural networks act internally (Dam et al., 2018), and their behavior may be similar to developers in tasks such as code summarization (Richter et al., 2023), explaining the actual aim of them acting as a whole is still challenging (Umer et al., 2020). For example, we can hardly conclude what useful information the models can learn in each propagation. Moreover, the existing dataset for code smell prioritization is not ideal for such approaches. The scale of the dataset is limited and deep learning approaches require an excessively large number of samples for training (i.e., data-hungry Munappy et al., 2019).

Moreover, recent advances in Large Language Model (LLM) also reveal its potential to comprehend and interpret code. Notable models such as GPT-4 and Vicuna could assess code quality and explain their decisions in detailed natural language. We attempted to perform a systematic evaluation using LLM, and we also tested the validity of GPT-4 and Vicuna-13b in code smell prioritization tasks. However, we find that although LLMs could explain their judgment, these approaches cannot produce consistent results, even within the same session. Consequently, although their output is more interpretable, they are not comprehensive. Thus, we believe they are not ready for such tasks and require fine-tuning. Afterward, they may have the potential of achieving better XAI for SQA.

9. Conclusion

We conducted a case study of code smell prioritization to investigate and narrow the gap in explanations between XAI and developers. First, we inspected developers' comments on code smell criticalities from a dataset annotated by original developers, and summarized their major concerns. Then, we measured the gap by checking if inspecting a reasonable number of top important features in XAI explanations could cover the major concerns of original developers. Finally, we modified the AutoSpearman feature selection approach for XAI to preserve the features that could reflect developers' major concerns as much as possible and narrowed the gap.

Result showed the gap between XAI explanations and developers' expectations is obvious for code smell prioritization, even if the most concerned aspects of developers could be covered by fine-grained features. Fortunately, the gap could be narrowed by adapting to the developers' concerns in feature selection, and the explanations could achieve reasonable $coverage$ ($> 70\%$) of developers' concerns with acceptable $complexity$. As a result, for smells with simpler causes, a simple explanation (e.g., inspecting top-3 or top-5 features) is enough. However, explaining smells with complex or controversial causes requires more expertise to inspect around 10 features.

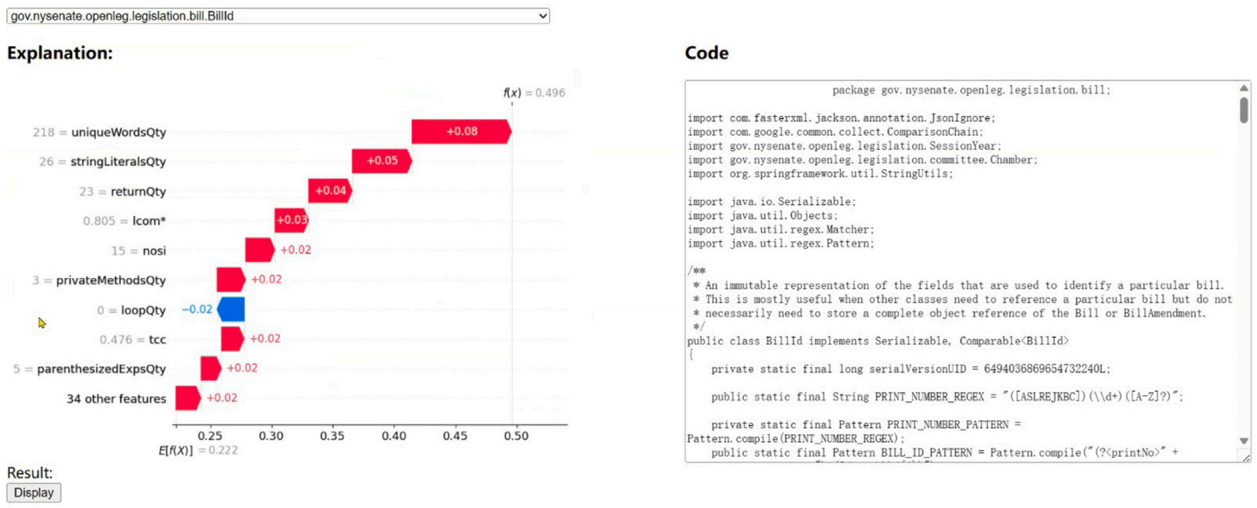
Based on case studies, we also inferred that current XAI for SQA practice is ideal for diagnosing models, but it is not ready for explaining general SQA issues with complex or diverse causes (e.g., general software defects), no matter the causes could be captured by more advanced feature engineering approaches or not. Moreover, we thought narrowing the prediction target to more specific problems (e.g., a certain type of bug or smell) in a given context (e.g., a specific type of software) would be more reasonable and practical. Meanwhile, we also called for more empirical studies for specific problems in different contexts to provide more evidence for building adaptive XAI models.

Future work includes (1) collecting commented single project datasets in industrial settings with clear guidelines of criticality anno-

[17] The full class name is `org.apache.pig.backend.hadoop.executionengine.tez.TezDagBuilder`.

Table 18
Overview of the analyzed projects.

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



```
                      package gov.nysenate.openleg.legislation.bill;

import com.fasterxml.jackson.annotation.JsonIgnore;
import com.google.common.collect.ComparisonChain;
import gov.nysenate.openleg.legislation.SessionYear;
import gov.nysenate.openleg.legislation.committee.Chamber;
import org.springframework.util.StringUtils;

import java.io.Serializable;
import java.util.Objects;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * An immutable representation of the fields that are used to identify a particular bill.
 * This is mostly useful when other classes need to reference a particular bill but do not
 * necessarily need to store a complete object reference of the Bill or BillAmendment.
 */
public class BillId implements Serializable, Comparable<BillId>
{
    private static final long serialVersionUID = 6494036869654732240L;

    public static final String PRINT_NUMBER_REGEX = "([ASLREJKBC])(\\d+)([A-Z]?)";

    private static final Pattern PRINT_NUMBER_PATTERN =
Pattern.compile(PRINT_NUMBER_REGEX);
    public static final Pattern BILL_ID_PATTERN = Pattern.compile("(?<printNo>" +
```

Fig. 11. A screenshot of the demo tool.

tation, (2) developing an IDE-based plugin for code smell prioritization and refactoring, and (3) validating whether IDE-based XAI techniques could help developers recognize severe smells in real-world scenarios.

CRediT authorship contribution statement

Zijie Huang: Conceptualization of this study, Methodology, Software. **Huiqun Yu:** Supervision. **Guisheng Fan:** Writing – review & editing. **Zhiqing Shao:** Funding, Review & editing. **Mingchen Li:** Data curation, Visualization. **Yuguo Liang:** Visualization, Investigation, Original draft.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Guisheng Fan reports financial support was provided by the 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. Guisheng Fan reports was provided by the Research Programme of National Engineering Laboratory for Big Data Distribution and Exchange Technologies. Huiqun Yu reports was provided by the Shanghai Municipal Special Fund for Promoting High Quality Development.

Data availability

Data released on GitHub.

Appendix A. Features and projects used for prediction

The evaluated projects are listed in Table 18. The features used for prediction are available in Table 19.

Appendix B. Online appendix and replication package

The online appendix and the replication package[18] of this paper are available as a GitHub repository. It contains the procedural data and result of manual assessment, the code for generating experimental data, model performance, the code of modified AutoSpearman, and various additional tests (e.g., the agreement between SHAP and Information Gain, the result of RuleMatrix application, and the correlation between features and prediction targets).

Appendix C. Demo tool

The demo tool.[19] of this paper is also available as a GitHub repository. A screenshot is available in Fig. 11 Practitioners could input their GitHub repository using Java code, and the tool will output predictions and explanations for every class. The tool calculates code and process metrics and makes predictions in real time.

[18] https://github.com/SORD-src/ESWA23.
[19] https://github.com/SORD-src/ESWA23_demo.

Table 19
Features and their definitions.

Feature name	Category	Description
ATFD	Coupling	The number of unrelated attributes (foreign data) accessed directly or by invoking accessor methods.
AVG_CS (Avg-commit-size)	Change	The average number of classes that co-changed in commits involving a class.
C3	Cohesion	The average cosine similarity computed among all method pairs of a class.
CBO	Coupling	The number of classes to which it is coupled.
CFNAMM	Coupling	The number of called not accessor or mutator methods declared in unrelated classes respect to the measured one.
DIT	Complexity	The maximum length from the class node to the root of its inheritance hierarchy tree.
DSC	Change	The average number of distinct subsystems in which the committers of a class made changes.
FANOUT	Coupling	The number of called classes.
is_controller	Production	Whether the class is a controller class.
is_external	Production	Whether the class is extrinsic or copied from other systems.
is_procedural	Production	Whether the class contain procedural instructions.
is_test	Testing	Whether the class is a test class or it is designed to facilitate testing.
is_util	Production	Whether the class is a utility class.
is_static	N/A	Whether the class is a static class.
LCOM5	Cohesion	Lack of cohesion in methods.
LOC	Size	The number of lines of code of a class, including blank lines and comments.
LOC_package	Size	The number of lines of code of a package.
LOCNAMM	Size	The number of lines of code of a class excluding accessor and mutator methods and corresponding comments.
MPC	Coupling	Number of method calls made by a class to external classes of the system.
NC (number-changes)	Change	Number of commits in the change history of the system involving a class.
NCOM (number-committers)	Change	Number of distinct developers who performed commits on a class in the change history of the system.
NF (number-fixes)	Change	Number of bug fixing activities performed on a class in the change history of the system.
NIM	Complexity	The number of inherited methods.
NMO	Size	The number of methods that have been overridden.
NOA	Size	The number of attributes of a class.
NOAM	Size	The number of accessor (getter and setter) methods of a class.
NOC	Size	The number of children counts the immediate subclasses subordinated to a class in the class hierarchy.
NOCS	Size	The number of nested classes of a class.
NOCS_package	Size	The number of classes of a package.
NOI_package	Size	The number of interfaces declared in a package.
NOII	Size	The number of implemented interfaces by a class.
NOM	Size	The number of methods defined locally in a class.
NOM_package	Size	The number of methods defined locally in a package.
NOMNAMM	Size	The number of methods defined locally in a class, excluding accessor or mutator methods.
NOMNAMM_package	Size	The number of methods defined locally in a package, excluding accessor or mutator methods.
NOPA	Size	The number of public attributes of a class.
num_abstract_methods	Size	The number of abstract methods by a class.
num_constructor_DC	Size	The number of constructor methods by a class.
num_constructor_NotDC	Size	The number of non-constructor methods by a class.
num_final_attr	Size	The number of final attributes by a class.
num_final_methods	Size	The number of final methods by a class.
num_final_not_static_attr	Size	The number of final and non-static attributes by a class.
num_final_not_static_methods	Size	The number of final and non-static methods by a class.
num_final_static_attr	Size	The number of final and static attributes by a class.
num_final_static_methods	Size	The number of final and static methods by a class.
num_not_abstract_not_final_methods	Size	The number of non-abstract and non-final methods by a class.
num_not_final_not_static_attr	Size	The number of non-final and non-static attributes by a class.
num_not_final_not_static_methods	Size	The number of non-final and non-static methods by a class.
num_not_final_static_methods	Size	The number of non-final and static methods by a class.
num_package_visibility_attr	Size	The number of attributes with default visibility by a class.
num_package_visibility_methods	Size	The number of methods with default visibility by a class.
num_private_visibility_attr	Size	The number of private attributes by a class.
num_private_visibility_methods	Size	The number of private methods by a class.
num_protected_visibility_attr	Size	The number of protected attributes by a class.
num_protected_visibility_methods	Size	The number of protected methods by a class.
num_public_visibility_methods	Size	The number of public methods by a class.
num_standard_design_methods	Size	The number of methods intended for duplication or repetitive manufacture.
num_static_attr	Size	The number of static attributes by a class.
num_static_methods	Size	The number of static methods by a class.
num_static_not_final_attr	Size	The number of static and non-final attributes by a class.
persistence	Change	Number of subsequent major/minor releases in which a certain smell affects a class.
Read. (Readability)	Complexity	Measure of source code readability based on 25 features, see Buse and Weimer (2010) for computational details.
RFC	Coupling	Response for a class, the number of methods that can be invoked in response to a message to an object of the class.
TCC	Cohesion	Tight class cohesion.
WMC	Size, Complexity	The sum of e Cyclomatic Complexity (CYCLO) of the methods that are defined in the class.
WMCNAMM	Size, Complexity	The WMC metrics calculated over non-accessor or mutator methods.
WOC	Complexity	The number of "functional" public methods divided by the total number of public members.

References

Alazba, A., & Aljamaan, H. (2021). Code smell detection using feature selection and stacking ensemble: An empirical investigation. *Information and Software Technology*, *138*, Article 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).

Ambsdorf, J., Munir, A., Wei, Y., Degkwitz, K., Harms, H. M., Stannek, S., Ahrens, K., Becker, D., Strahl, E., Weber, T., & Wermter, S. (2022). Explain yourself! effects of explanations in human-robot interaction. In *Proc. 31st IEEE international conference on robot and human interactive communication (RO-MAN)* (pp. 393–400).

Antinyan, V. (2021). Hypnotized by lines of code. *Computer*, *54*(1), 42–48.

Azeem, M. I., Palomba, F., Shi, L., & Wang, Q. (2019). Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information* *and Software Technology*, *108*, 115–138.

Barbez, A., Khomh, F., & Guéhéneuc, Y.-G. (2020). A machine-learning based ensemble method for anti-patterns detection. *Journal of Systems and Software*, *161*, Article 110486.

Brown, W. H., Malveau, R. C., McCormick, H. W. S., & Mowbray, T. J. (1998). *AntiPatterns: Refactoring software, architectures, and projects in crisis*. Boston, MA: John Wiley & Sons, Inc..

Buse, R. P., & Weimer, W. R. (2010). Learning a metric for code readability. *IEEE Transactions on Software Engineering*, *36*(4), 546–558.

Catolino, G., Palomba, F., Zaidman, A., & Ferrucci, F. (2019). Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software*, *152*, 165–181.

Dam, H. K., Tran, T., & Ghose, A. (2018). Explainable software analytics. In *Proc. 40th international conference on software engineering: new ideas and emerging results (ICSE-NIER)* (pp. 53–56).

de Mello, R., Oliveira, R., Uchôa, A., Oizumi, W., Garcia, A., Fonseca, B., & de Mello, F. (2022). Recommendations for developers identifying code smells. *IEEE Software*, 2–10.

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).

dos Santos, G. E., Figueiredo, E., Veloso, A., Viggiato, M., & Ziviani, N. (2020). Understanding machine learning software defect predictions. *Automated Software Engineering*, *27*(3), 369–392.

Er, L., Laberge, G., Roy, B., Khomh, F., Nikanjam, A., & Mondal, S. (2022). Why don't XAI techniques agree? Characterizing the disagreements between post-hoc explanations of defect predictions. In *Proc. 38th IEEE international conference on software maintenance and evolution (ICSME)* (pp. 444–448).

Fakhoury, S., Arnaoudova, V., Noiseux, C., Khomh, F., & Antoniol, G. (2018). Keep it simple: Is deep learning good for linguistic smell detection? In *Proc. 25th international conference on software analysis, evolution and reengineering (SANER)* (pp. 602–611).

Fokaefs, M., Tsantalis, N., Stroulia, E., & Chatzigeorgiou, A. (2011). Jdeodorant: Identification and application of extract class refactorings. In *Proc. of the 33rd international conference on software engineering (ICSE)* (pp. 1037–1039).

Fontana, F. A., Ferme, V., Zanoni, M., & Roveda, R. (2015). Towards a prioritization of code debt: A code smell intensity index. In *Proc. 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. *Knowledge-Based Systems*, *128*, 43–58.

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the design of existing code*. Boston, MA: Addison-Wesley.

Gao, Y., Zhu, Y., & Yu, Q. (2022). Evaluating the effectiveness of local explanation methods on source code-based defect prediction models. In *IEEE/ACM 19th international conference on mining software repositories (MSR)* (pp. 640–645). IEEE.

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. *Software - Practice and Experience*, *48*(5), 1077–1106.

Ichtsis, A., Mittas, N., Ampatzoglou, A., & Chatzigeorgiou, A. (2022). Merging smell detectors: Evidence on the agreement of multiple tools. In *Proc. 5th 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. *Science of Computer Programming*, *212*, Article 102713.

Jiarpakdee, J., Tantithamthavorn, C., Dam, H. K., & Grundy, J. (2022). An empirical study of model-agnostic techniques for defect prediction models. *IEEE Transactions on Software Engineering*, *48*(1), 166–185.

Jiarpakdee, J., Tantithamthavorn, C., & Grundy, J. (2021). Practitioners' perceptions of the goals and visual explanations of defect prediction models. In *Proc. IEEE/ACM 18th international conference on mining software repositories (MSR)* (pp. 432–443).

Jiarpakdee, J., Tantithamthavorn, C., & Treude, C. (2020). The impact of automated feature selection techniques on the interpretation of defect models. *Empirical Software Engineering*, *25*(5), 3590–3638.

Kocielnik, R., Amershi, S., & Bennett, P. N. (2019). Will you accept an imperfect ai? Exploring designs for adjusting end-user expectations of AI systems. In *Proc. 2019 CHI conference on human factors in computing systems (CHI)* (pp. 1–14).

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 Systems with Applications*, *204*, Article 117607.

Krippendorff, K. (1970). Estimating the reliability, systematic error and random error of interval data. *Educational and Psychological Measurement*, *30*(1), 61–70.

Krippendorff, K. (2011). Computing Krippendorff's alpha-reliability. URL: https://repository.upenn.edu/asc_papers/43.

Kulesza, T., Stumpf, S., Burnett, M., Yang, S., Kwan, I., & Wong, W.-K. (2013). Too much, too little, or just right? Ways explanations impact end users' mental models. In *Proc. 2013 IEEE symposium on visual languages and human centric computing (VLHCC)* (pp. 3–10).

Lanza, M., Marinescu, R., & Ducasse, S. (2005). *Object-oriented metrics in practice*. Springer Science & Business Media.

Lewowski, T., & Madeyski, L. (2022). How far are we from reproducible research on code smell detection? A systematic literature review. *Information and Software Technology*, *144*, Article 106783.

Lipton, P. (1990). Contrastive explanation. *Royal Institute of Philosophy Supplements*, *27*, 247–266.

Liu, H., Jin, J., Xu, Z., Bu, Y., Zou, Y., & Zhang, L. (2021). Deep learning based code smell detection. *IEEE Transactions on Software Engineering*, *47*(9), 1811–1837.

Liu, H., Xu, Z., & Zou, Y. (2018). Deep learning based feature envy detection. In *Proc. 33rd IEEE/ACM international conference on automated software engineering (ASE)* (pp. 385–396).

Lundberg, S. M., & Lee, S.-I. (2017). A unified approach to interpreting model predictions. In *Proc. 31st international conference on neural information processing systems (NIPS)* (pp. 4768–4777).

Madeyski, L., & Lewowski, T. (2020). MLCQ: Industry-relevant code smell data set. In *Proc. 24th international conference on evaluation and assessment in software engineering (EASE)* (pp. 342–347).

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).

Ming, Y., Qu, H., & Bertini, E. (2019). RuleMatrix: Visualizing and understanding classifiers with rules. *IEEE Transactions on Visualization and Computer Graphics*, *25*(1), 342–352.

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 Transactions on Software Engineering*, *36*(1), 20–36.

Molnar, C. (2022). *Interpretable machine learning: A guide for making black box models explainable* (2nd ed.). URL: https://christophm.github.io/interpretable-ml-book/explanation.html#good-explanation.

Munappy, A., Bosch, J., Olsson, H. H., Arpteg, A., & Brinne, B. (2019). Data management challenges for deep learning. In *Proc. 45th euromicro conference on software engineering and advanced applications (SEAA)* (pp. 140–147).

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., & De Lucia, A. (2014). Do they really smell bad? A study on developers' perception of bad code smells. In *Proc. 30th IEEE international conference on software maintenance and evolution (ICSME)* (pp. 101–110).

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 *Proc. 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 Transactions on Software Engineering*, *41*(5), 462–489.

Palomba, F., Panichella, A., De Lucia, A., Oliveto, R., & Zaidman, A. (2016). A textual-based technique for smell detection. In *Proc. IEEE 24th international conference on program comprehension (ICPC)* (pp. 1–10).

Palomba, F., Panichella, A., Zaidman, A., Oliveto, R., & Lucia, A. D. (2018). The scent of a smell: An extensive comparison between textual and structural smells. *IEEE Transactions on Software Engineering*, *44*(10), 977–1000.

Palomba, F., Zanoni, M., Fontana, F. A., De Lucia, A., & Oliveto, R. (2019). Toward a smell-aware bug prediction model. *IEEE Transactions on Software Engineering*, *45*(2), 194–218.

Papenmeier, A., Kern, D., Englebienne, G., & Seifert, C. (2022). It's complicated: The relationship between user trust, model accuracy and explanations in AI. *ACM Transactions on Computer-Human Interaction*, *29*(4), Article 35.

Pecorelli, F., Palomba, F., Khomh, F., & De Lucia, A. (2020). Developer-driven code smell prioritization. In *Proc. IEEE/ACM 17th international conference on mining software repositories (MSR)* (pp. 220–231).

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 *Proc. IEEE 27th international requirements engineering conference (RE)* (pp. 404–409).

Piotrowski, P., & Madeyski, L. (2020). In A. Poniszewska-Marańda, N. Kryvinska, S. Jarząbek, & L. Madeyski (Eds.), *Software defect prediction using bad code smells: A systematic literature review. Data-centric business and applications: Towards software development (Volume 4)* (pp. 77–99). Cham: Springer International Publishing.

Rajapaksha, D., Tantithamthavorn, C., Jiarpakdee, J., Bergmeir, C., Grundy, J., & Buntine, W. (2022). Sqaplanner: Generating data-informed software quality improvement plans. *IEEE Transactions on Software Engineering*, *48*(8), 2814–2835.

Rajbahadur, G. K., Wang, S., Oliva, G. A., Kamei, Y., & Hassan, A. E. (2022). The impact of feature importance methods on the interpretation of defect classifiers. *IEEE Transactions on Software Engineering*, *48*(7), 2245–2261.

Ribeiro, M. T., Singh, S., & Guestrin, C. (2016). "Why should I trust you?": Explaining the predictions of any classifier. In *Proc. 22nd ACM SIGKDD international conference on knowledge discovery and data mining (KDD)* (pp. 1135–1144).

Richter, C., Haltermann, J., Jakobs, M.-C., Pauck, F., Schott, S., & Wehrheim, H. (2023). Are neural bug detectors comparable to software developers on variable misuse bugs? In *Proc. 37th IEEE/ACM international conference on automated software engineering (ASE)*. Article 9.

Riveiro, M., & Thill, S. (2021). "That's (not) the output I expected!" on the role of end user expectations in creating explanations of AI systems. *Artificial Intelligence*, *298*, Article 103507.

Rodríguez-Pérez, G., Nagappan, M., & Robles, G. (2022). Watch out for extrinsic bugs! a case study of their impact in just-in-time bug prediction models on the OpenStack project. *IEEE Transactions on Software Engineering*, *48*(4), 1400–1416.

Sae-Lim, N., Hayashi, S., & Saeki, M. (2016). Context-based code smells prioritization for prefactoring. In *Proc. IEEE 24th international conference on program comprehension (ICPC)* (pp. 1–10).

Sae-Lim, N., Hayashi, S., & Saeki, M. (2017a). How do developers select and prioritize code smells? A preliminary study. In *Proc. IEEE 33rd international conference on software maintenance and evolution (ICSME)* (pp. 484–488).

Sae-Lim, N., Hayashi, S., & Saeki, M. (2017b). Revisiting context-based code smells prioritization: On supporting referred context. In *Proc. XP2017 scientific workshops* (pp. 1–5).

Sae-Lim, N., Hayashi, S., & Saeki, M. (2018a). Context-based approach to prioritize code smells for prefactoring. *Journal of Software: Evolution and Process*, *30*(6), Article e1886.

Sae-Lim, N., Hayashi, S., & Saeki, M. (2018b). An investigative study on how developers filter and prioritize code smells. *IEICE Transactions on Information and Systems*, *101-D*(7), 1733–1742.

Sharma, T., Efstathiou, V., Louridas, P., & Spinellis, D. (2021). Code smell detection by deep direct-learning and transfer-learning. *Journal of Systems and Software*, *176*, Article 110936.

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 Transactions on Software Engineering*, *47*(1), 17–66.

Sotto-Mayor, B., Elmishali, A., Kalech, M., & Abreu, R. (2022). Exploring design smells for smell-based defect prediction. *Engineering Applications of Artificial Intelligence*, *115*, Article 105240.

Sotto-Mayor, B., & Kalech, M. (2021). Cross-project smell-based defect prediction. *Soft Computing*, *25*(22), 14171–14181.

Taba, S. E. S., Khomh, F., Zou, Y., Hassan, A. E., & Nagappan, M. (2013). Predicting bugs using antipatterns. In *2013 IEEE international conference on software maintenance (ICSM)* (pp. 270–279).

Tantithamthavorn, C., Hassan, A. E., & Matsumoto, K. (2020). The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering*, *46*(11), 1200–1219.

Tantithamthavorn, C., Jiarpakdee, J., & Grundy, J. (2021). Actionable analytics: Stop telling me what it is; please tell me what to do. *IEEE Software*, *38*(4), 115–120.

Tantithamthavorn, C., McIntosh, S., Hassan, A. E., & Matsumoto, K. (2017). An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, *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 Transactions on Software Engineering*, *45*(7), 683–711.

Tian, Y., Ali, N., Lo, D., & Hassan, A. E. (2016). On the unreliability of bug severity data. *Empirical Software Engineering*, *21*(6), 2298–2323.

Umer, Q., Liu, H., & Illahi, I. (2020). CNN-based automatic prioritization of bug reports. *IEEE Transactions on Reliability*, *69*(4), 1341–1354.

Vidal, S. A., Marcos, C. A., & Pace, J. A. D. (2016). An approach to prioritize code smells for refactoring. *Automated Software Engineering*, *23*(3), 501–532.

Wang, D., Yang, Q., Abdul, A., & Lim, B. Y. (2019). Designing theory-driven user-centric explainable AI. In *Proc. 2019 CHI conference on human factors in computing systems (CHI)* (pp. 1–15).

Widyasari, R., Prana, G. A. A., Haryono, S. A., Tian, Y., Zachiary, H. N., & Lo, D. (2022). XAI4fl: Enhancing spectrum-based fault localization with explainable artificial intelligence. In *Proc. IEEE/ACM 30th international conference on program comprehension (ICPC)* (pp. 499–510).

Wu, H., Yin, R., Gao, J., Huang, Z., & Huang, H. (2022). To what extent can code quality be improved by eliminating test smells? In *Proc. 2nd international conference on code quality (ICCQ)* (pp. 19–26).

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 *Proc. 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 *Proc. 24th international conference on the evaluation and assessment in software engineering (EASE)* (pp. 120–129).

Yedida, R., & Menzies, T. (2022). How to improve deep learning for software analytics (a case study with code smell detection). In *Proc. IEEE/ACM 19th international conference on mining software repositories (MSR)* (pp. 156–166).

Yu, X., Bennin, K. E., Liu, J., Keung, J. W., Yin, X., & Xu, Z. (2019). An empirical study of learning to rank techniques for effort-aware defect prediction. In *Proc. IEEE 26th international conference on software analysis, evolution and reengineering (SANER)* (pp. 298–309).

Zheng, W., Shen, T., Chen, X., & Deng, P. (2022). Interpretability application of the just-in-time software defect prediction model. *Journal of Systems and Software*, *188*, Article 111245.