The Journal of Systems and Software 217 (2024) 112159

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

EvaluateXAI: A framework to evaluate the reliability and consistency of rule-based XAI techniques for software analytics tasks☆

Md. Abdul Awal *, Chanchal K. Roy

Department of Computer Science, University of Saskatchewan, 110 Science Pl, Saskatoon, S7N 5C9, SK, Canada

ARTICLE INFO

Dataset link: Replication-package

Keywords:
Software analytics
Machine learning
Reliability
Consistency
Explainability

ABSTRACT

The advancement of machine learning (ML) models has led to the development of ML-based approaches to improve numerous software engineering tasks in software maintenance and evolution. Nevertheless, research indicates that despite their potential successes, ML models may not be employed in real-world scenarios because they often remain a black box to practitioners, lacking explainability in their reasoning. Recently, various rule-based model-agnostic Explainable AI (XAI) techniques, such as PyExplainer and LIME, have been employed to explain the predictions of ML models in software analytics tasks. In this paper, we assess the ability of these techniques, particularly the state-of-the-art PyExplainer and LIME, to generate reliable and consistent explanations for ML models across various software analytics tasks, including Just-in-Time (JIT) defect prediction, clone detection, and the classification of useful code review comments. Our manual investigations find inconsistencies and anomalies in the explanations generated by these techniques. Therefore, we design a novel framework: Evaluation of Explainable AI (*EvaluateXAI*), along with granular-level evaluation metrics, to automatically assess the effectiveness of rule-based XAI techniques in generating reliable and consistent explanations for ML models in software analytics tasks. After conducting in-depth experiments involving seven state-of-the-art ML models trained on five datasets and six evaluation metrics, we find that none of the evaluation metrics reached 100%, indicating the unreliability of the explanations generated by XAI techniques. Additionally, PyExplainer and LIME failed to provide consistent explanations for 86.11% and 77.78% of the experimental combinations, respectively. Therefore, our experimental findings emphasize the necessity for further research in XAI to produce reliable and consistent explanations for ML models in software analytics tasks.

1. Introduction

In software maintenance and evolution, many software engineering tasks such as method name prediction (Alon et al., 2019, 2018), code summarization (Fernandes et al., 2018), code comment generation (Hu et al., 2018), code review (Bacchelli and Bird, 2013), code clone detection (Roy and Cordy, 2007; Nafi et al., 2019), and Just-in-Time (JIT) defect prediction (Catolino et al., 2019; Kamei et al., 2012; Yatish et al., 2019; Choi et al., 2022) have been widely applied to improve the software development and maintenance process. Thanks to the development of various machine learning (ML) models, researchers have proposed many ML-based approaches to facilitate those software engineering tasks in software maintenance and evolution. For example, in modern code review, it is unlikely to maintain the quality and perform exhaustive code review activities for all the incoming commits with limited Software Quality Assurance (SQA) resources (Pornprasit et al., 2021). Therefore, in this regard, the JIT defect prediction model can be used to predict defect-introducing commits before performing any commit operations. Thus, it assists software developers in prioritizing their limited SQA resources by focusing on the riskiest commits during code review (Catolino et al., 2019; Kamei et al., 2012). It also helps create proactive strategies for improving software quality to avoid past mistakes, which could result in software defects in future releases (McIntosh and Kamei, 2018).

Despite their potential usefulness in various software analytics tasks, ML models may not be employed in real-world scenarios because ML practitioners often need more understanding of why specific predictions are made (Jiarpakdee et al., 2020). For example, in the case of JIT defect prediction, they usually ask why a commit is being predicted as defect-introducing, as JIT defect models solely focus on prediction without explaining their decisions (Pornprasit et al., 2021;

☆ Editor: Dario Di Nucci.

* Corresponding author.
E-mail address: mda439@usask.ca (M.A. Awal).

https://doi.org/10.1016/j.jss.2024.112159
Received 30 November 2023; Received in revised form 25 June 2024; Accepted 16 July 2024
Available online 27 July 2024

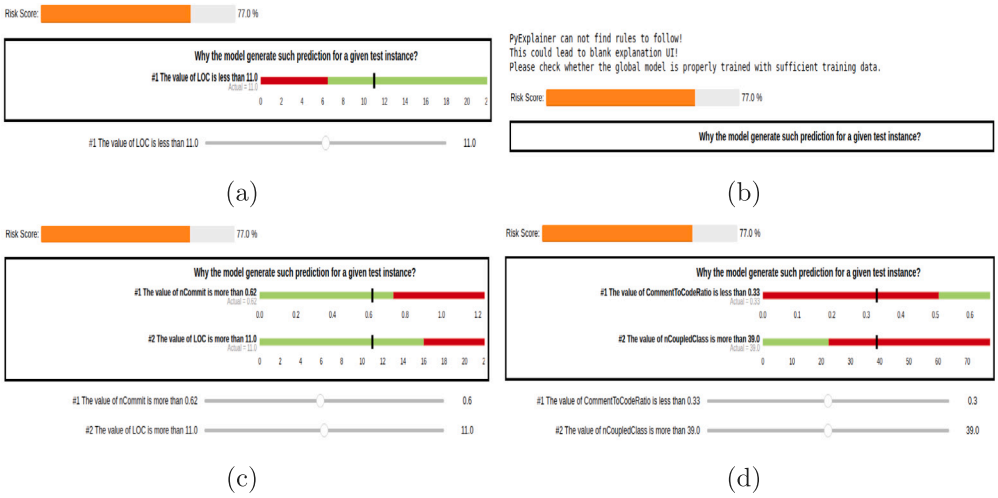


Fig. 1. Explanations generated by PyExplainer for a single instance reveal that the model predicts the instance as a defect-introducing commit with a 77% probability, as indicated by the Risk Score.

Jiarpakdee et al., 2020, 2021). Therefore, due to the lack of explainability to understand the reasoning behind the decisions of the JIT defect models, practitioners may encounter difficulties in enhancing software quality with limited SQA resources, potentially leading to less effective SQA practices. As a result, many XAI methods such as SHapley Additive exPlanation (SHAP) (Lundberg and Lee, 2017), Local Interpretable Model-agnostic Explanations (LIME) (Ribeiro et al., 2016), Anchors (Ribeiro et al., 2018), BreakDown (Gosiewska and Biecek, 2019), SQAPlanner (Rajapaksha et al., 2021), Integrated Gradient (IG) (Sundararajan et al., 2017), PyExplainer (Pornprasit et al., 2021) and so on have been developed to explain the outcome of the ML models.

The rule-based XAI methods have been used in software analytics very recently (Jiarpakdee et al., 2020; Rajbahadur et al., 2021; Roy et al., 2022). For example, the state-of-the-art model-agnostic[1] XAI technique called LIME (Ribeiro et al., 2016) was adopted in software analytics to explain the predictions of the JIT defect models at line-level and file-level (Jiarpakdee et al., 2020; Pornprasit and Tantithamthavorn, 2021). Research shows that the synthetic neighbourhood generation process significantly impacts how well LIME produces its explanations (Pornprasit et al., 2021). In addition, LIME fails to generate consistent explanations for the same instance if we apply it multiple times in a row (Jiarpakdee et al., 2020). Recently, a local rule-based model-agnostic XAI technique called PyExplainer was proposed to overcome the shortcomings of LIME in explaining the outcome of the JIT defect prediction models. However, our manual investigations find that they produce inconsistent explanations, as shown in Figs. 1 and 2 for the same instance if we rerun it. In addition, PyExplainer sometimes fails to generate explanations and shows a blank UI to its practitioners, as depicted in Fig. 1(b). Thus, the inconsistent explanations generated by XAI techniques may confuse and discourage practitioners from using XAI techniques in real-world applications (Roy et al., 2022).

Lundberg and Lee (2017) argued that instance explanation generation must remain consistent upon regeneration for the same instance. However, our manual investigations find that PyExplainer and LIME do not adhere to this statement when rerunning on the same instance. Moreover, they exhibit anomalous behaviours, as illustrated in Section 2, raising concerns about the reliability of the explanations they generate for ML models, such as JIT defect prediction models. Therefore, it is essential to systematically evaluate the reliability and consistency of the explanations produced by rule-based XAI techniques for various software analytics tasks, including JIT defect prediction, clone detection, and the classification of useful code review comments.

To the best of our knowledge, no framework or granular-level evaluation metrics are currently available for systematically assessing the ability of rule-based XAI techniques to generate reliable and consistent explanations for ML models in software analytics tasks. Therefore, in this paper, we introduce a novel framework: Evaluation of Explainable AI (*EvaluateXAI*), along with six granular-level evaluation metrics, to evaluate the reliability and consistency of the explanations generated by rule-based XAI techniques. Specifically, we use *EvaluateXAI* to assess PyExplainer and LIME in the context of seven machine learning models trained on five commonly used datasets in software analytics tasks. Thus, the key contributions of our study are listed below:

- We design a novel framework called *EvaluateXAI* to assess the reliability and consistency of rule-based XAI techniques in generating explanations for ML models in software analytics tasks.
- We adopt two metrics from Pornprasit et al. (2021) and develop four additional granular-level evaluation metrics within *EvaluateXAI*.
- We demonstrate that *EvaluateXAI* is applicable to any rule-based XAI techniques, enabling the evaluation of their reliability and consistency in generating explanations for ML models.
- Our code and the datasets used in this study are publicly available to advance the research in XAI. The replication package can be accessed through this link.[2]

2. Motivating examples

This study aims to evaluate the reliability and consistency of the explanations generated by rule-based XAI techniques by designing a novel framework and granular-level evaluation metrics. To achieve this, we consider state-of-the-art rule-based model-agnostic XAI techniques, such as PyExplainer and LIME, as candidates for designing the framework and evaluation metrics. We deliberately selected PyExplainer and LIME due to their extensive study in software analytics tasks, particularly for JIT defect predictions (Pornprasit et al., 2021; Jiarpakdee et al., 2020; Roy et al., 2022). Additionally, we emphasize JIT defect prediction as an illustrative example to demonstrate the challenges and complexities associated with applying rule-based XAI techniques to practical software analytics tasks.

[1] Model-agnostic refers to techniques that are universally applicable to ML models regardless of their specific type or architectural design.

[2] Replication-package

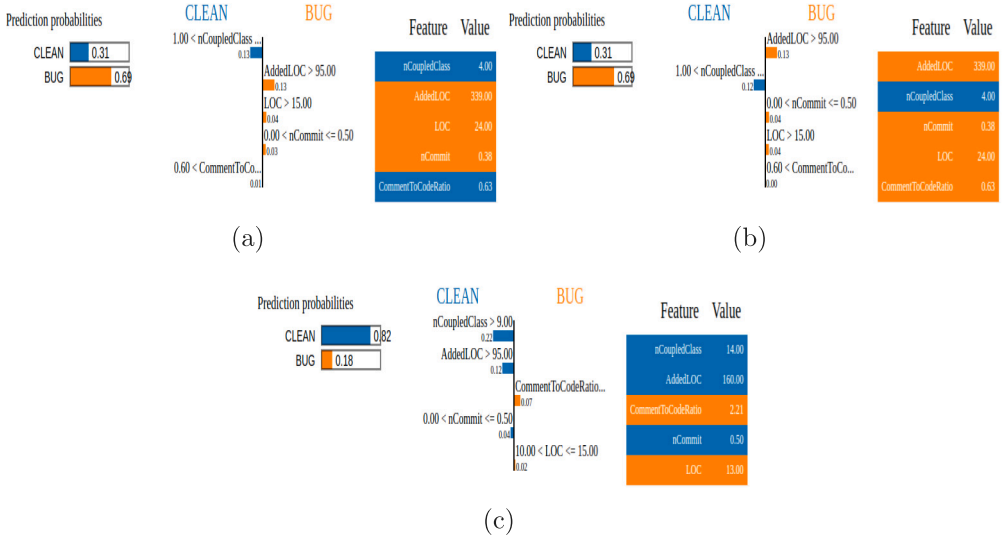


Fig. 2. Figs. 2(a) and 2(b) demonstrate that LIME produces different explanations when regenerated for the same instance. These two figures collectively visualize why the given instance is predicted as a buggy commit. Fig. 2(c) visually explains an instance predicted as a clean commit.

Table 1
An instance to be explained using PyExplainer and LIME.

nCommit	AddedLOC	nCoupledClass	LOC	CommentToCodeRatio
0.615385	246.0	39.0	11.0	0.33

PyExplainer was introduced to address the limitations of LIME in explaining the outcomes of JIT defect prediction models. However, while it has shown effectiveness in providing explanations for JIT defect prediction models, our manual investigations reveal inconsistencies and anomalies in its explanation generation. To illustrate the shortcomings of PyExplainer, let us consider a ML (e.g., Random Forest) model trained on JIT defect-introducing commit data. We follow the official "PART A - Quick Start" tutorial provided by the PyExplainer.[3] We want to know why PyExplainer says a commit introduces a bug before performing the commit operation and how consistent the generated explanations are by PyExplainer. It is important to note that to find the inconsistencies and anomalous behaviours of PyExplainer and LIME; we rerun them on the same instance 10 times in a row. Table 1 shows the instance on which we apply PyExplainer and LIME to generate explanations, and all the generated explanations are shown in Figs. 1, 2(a), and 2(b).

Lundberg and Lee (2017) argued that instance explanation generation must remain consistent upon regeneration for the same instance. However, Fig. 1 depicts that PyExplainer generates four different explanations for the same instance upon re-execution. Additionally, we observe that among the five features (as shown in Table 1), four of them (e.g., *LOC*, *nCommit*, *CommentToCodeRatio*, and *nCoupleedClass*) have been used to generate explanations for different executions. Furthermore, in Fig. 1(a), PyExplainer generates the explanation where the feature value *LOC* is less than 11.0. On the other hand, in Fig. 1(c), PyExplainer generates the explanation where the feature value *LOC* is more than 11.0. Finally, Fig. 1(b) represents the scenario where PyExplainer fails to generate an explanation. We observe a similar inconsistency for LIME as depicted in Figs. 2(a) and 2(b). These inconsistent explanations may confuse practitioners and discourage them from using PyExplainer and LIME to explain the outcomes of JIT defect prediction models (Roy et al., 2022).

Another essential aspect of PyExplainer is whether it generates quality explanations and whether we can rely on them. To illustrate this, let us consider a scenario where a ML model predicts the given instance (e.g., as shown in Table 1) as a buggy commit with a risk score of 77%, as shown in Fig. 1. The visual interpretation of PyExplainer (details in Section 3) illustrates that if we change the feature values in the *green* zone, the risk score must be decreased, and changing feature values in the *red* zone increases the risk score. Now, if we change the feature values in the *green* zone, the risk score decreases to 68% from 77%, considering Figs. 1(c) and 3(b). However, Fig. 4(b) shows that this is not always true. Fig. 1(a) shows that when the value of *LOC* is 11, the ML model predicts the given instance as defective commits with a risk score of 77%. Despite changing the feature value of *LOC* from 11.0 to 21.0 in the *green* zone, it remains the same instead of decreasing, as shown in Fig. 4(b).

Fig. 3(a) shows that changing feature values in the *red* zone increases the risk score from 77% (as shown in Fig. 1(c)) to 79%. However, Fig. 4(a) shows that this is not always true. For example, when considering Figs. 1(a) and 4(a), after changing the feature value of *LOC* from 11.0 to 1.0, the risk score decreased from 77% to 68% instead of increasing, demonstrating the anomalous behaviour of PyExplainer. Based on our manual investigations, it is evident that LIME produces more consistent explanations than PyExplainer, although it should be noted that the explanations are not entirely consistent. For example, LIME generates two different explanations, whereas PyExplainer generates four different explanations in 10 consecutive runs on the same instance. Therefore, these inconsistent and anomalous explanation generations prompt us to conduct in-depth investigations and design a novel framework with different evaluation metrics at a granular level to assess the reliability and consistency of the explanations generated by rule-based XAI techniques for software analytics tasks.

3. Background

We introduce *EvaluateXAI* and propose granular-level evaluation metrics using PyExplainer as a case study. Therefore, understanding the working principles and interpretation of visual explanations generated by PyExplainer is essential for comprehending *EvaluateXAI* and the granular-level metrics. Thus, this section provides a brief description of how PyExplainer works and how we interpret the explanations generated by PyExplainer for JIT defect prediction models, along with some basic terminologies related to the study.

Explainable Artificial Intelligence (XAI): It refers to a set of techniques, tools, and methods used in artificial intelligence (AI) and

[3] "PART A - Quick Start" Tutorial.

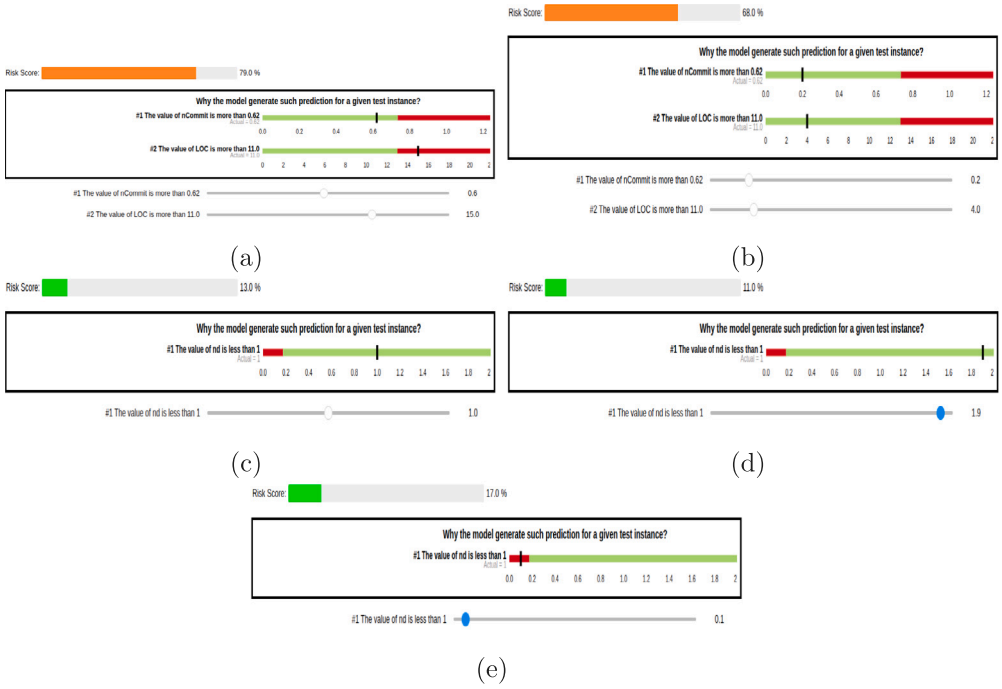


Fig. 3. Explanations generated by PyExplainer for a single instance reveal that the model predicts the instance as a defect-introducing commit with a 77% probability (as indicated by the Risk Score) as shown in Figs. 3(a) and 3(b). On the other hand, Figs. 3(c), 3(d), and 3(e) depict the generated explanations when the ML model predicts the given instance as a clean commit.

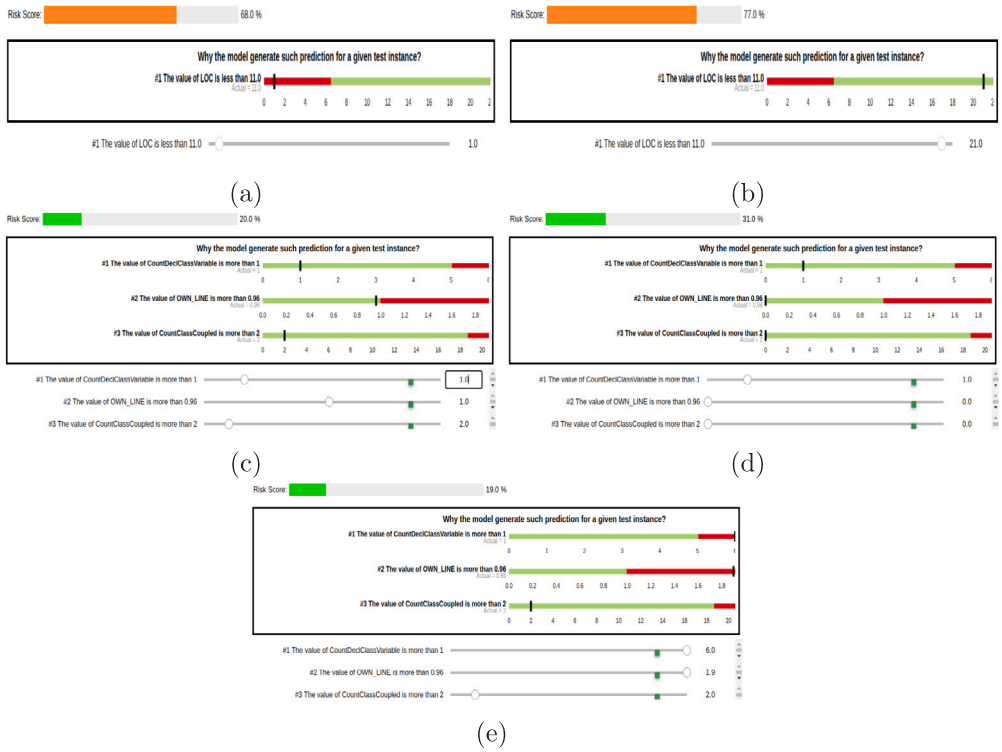


Fig. 4. While changing the feature values in the guided directions, PyExplainer fails to function correctly, which we refer to as the anomalous behaviour of PyExplainer.

machine learning (ML) to make the decisions and outputs of AI and ML systems understandable and interpretable by humans. XAI provides transparency and visibility into how AI or ML models make decisions.

Instance: In machine learning, an 'instance' typically refers to a single occurrence or example of data used in training, testing, or prediction. An instance is a specific set of input features that the machine learning algorithm processes to make predictions or learn patterns. For example, in the context of just-in-time (JIT) defect prediction, an instance would be a unit of code or a software component, and the features associated with that instance would include various metrics or characteristics related to the code, such as lines of code, complexity, historical defect information, and so on.

Risk Score: The risk score defines the probability that an instance is predicted as a predefined class. To clarify, let us consider a scenario where a ML model was trained on the JIT defect prediction dataset, such as cross-project mobile apps. Fig. 1(a) shows that the model predicts the selected instance as a defect-introducing commit with 77% probability. Therefore, we can say that the risk score for that instance is 77%. In this paper, we use the terms 'prediction probability for positive class' and 'risk score' interchangeably, denoting the same thing.

Simulated Instance: It represents the instance we generate by changing the feature values in the guided direction as per the PyExplainer explanation. Section 4.3 provides a detailed overview of the simulated instance generation process.

PyExplainer Overview: PyExplainer is a model-agnostic method that uses local rules to explain predictions made by JIT defect models. It consists of four steps: in step 1, PyExplainer constructs synthetic neighbours around the instance to be explained using crossover and mutation approaches. In step 2, PyExplainer obtains the predictions of the synthetic neighbours from the global model to mimic the global model's behaviour. In step 3, PyExplainer creates a local model based on the logistic regression technique called RuleFit. RuleFit aggregates ensembles trees and linear models to interpret the model while making sense of the logical reasons learned from the rule features. Finally, in step 4, PyExplainer generates an explanation from the local rule-based model and visualizes the explanation as shown in Fig. 1.

Interpret PyExplainer Visual Explanation: The PyExplainer tool represents rule-based explanations through bullet plot visualization and written descriptions. For example, the visual illustration shown in Fig. 1(c) tells us three key pieces of information: (1) the rules that describe why a commit is predicted as defect-introducing; (2) the vertical black bars that denote the actual feature values; and (3) the risk score is associated with a set of feature values that fall within a specific range. In addition, the feature values that fall within a risky range are represented by *red* shades, while *green* shades denote the non-risky range of feature values. Fig. 1(c) shows two textual descriptions: (1) "*the value of nCommit is more than 0.62*", and (2) "*the value of LOC is more than 11.0*". Finally, the risk score is 77% which denotes that the instance is predicted as a defect-introducing commit with 77% probability by the JIT defect model.

Ideally, if we increase the value of both features in the direction of the *red* zone, the risk score must be increased, as shown in Fig. 3(a). On the other hand, if we decrease the value of both features in the direction of the *green* zone, the risk score must be reduced, as shown in Fig. 3(b). Thus, we can change the feature values to alter the prediction of buggy commits to clean commits and vice-versa. In addition, Fig. 1(c) demonstrates that the features *nCommit* and *LOC* influence the JIT defect model to predict the given instance as a buggy commit.

Fig. 3(c) describes that the selected instance (e.g., an instance of the cross-project mobile apps dataset) is predicted as clean commits with 13% probability. Ideally, if we change the feature value of *nd* (number of modified directories) in the *green* zone, the probability score must be decreased, as shown in Fig. 3(d). But, on the other hand, if we change the feature value of *nd* in the *red* zone, then the probability score must be increased, as shown in Fig. 3(e). Finally, we observe that any changes in the feature values affect the prediction of the JIT defect model. Thus, PyExplainer generates visual explanations to explain the predictions of the JIT defect model.

4. Framework design

In this section, we discuss the key components of our proposed framework called *EvaluateXAI*, which is designed to evaluate the reliability and consistency of the explanations generated by rule-based XAI techniques through an extensive study. The framework consists of four key elements, each denoted as a step, as illustrated in Fig. 5. Below, we provide a detailed description of each of these steps.

4.1. Dataset selection and pre-processing

The dataset selection process is crucial because the quality and suitability of the dataset directly impact the validity and generalizability of ML models' results. Additionally, dataset pre-processing is critical in the data analysis and ML pipeline. Therefore, in this study, we carefully select datasets and pre-process them in the following way:

4.1.1. Dataset selection

This study evaluates the reliability and consistency of the explanations generated by XAI techniques for ML models trained on tabular data for various software analytics tasks, such as defect prediction, cross-language clone detection (CLCDSA), and code review comment classification. We utilize five different publicly available datasets: three JIT defect prediction datasets, one clone detection dataset, and one code review comment classification dataset from related studies. For JIT defect prediction, we employ three datasets: a dataset from cross-project mobile apps (Catolino et al., 2019) and Java project datasets containing data from 10 different Java projects (Kamei et al., 2012). In this category, we utilize a carefully selected cross-project mobile apps dataset with 14 features and approximately 30,000 data points. Various Java projects contribute datasets with 65 distinct features, comprising about 25,000 data points. Additionally, we select the Postgres dataset, containing 14 features and approximately 20,000 data points, from a pool of six subject systems for Desktop applications, as introduced by Yatish et al. (2019).

The code review comment dataset (Rahman et al., 2017), featuring 15 distinct attributes and consisting of 1100 data points, aids in identifying valuable comments. We also extend our analysis to include the cross-language clone (CLCDSA) dataset by Nafi et al. (2019). The CLCDSA dataset, designed for cross-language clone detection, comprises 18 distinctive features and roughly 30,000 data points. By leveraging these curated datasets, each with its unique attributes, our study offers comprehensive insights into the reliability and consistency of XAI techniques when generating explanations for ML models across various software analytics tasks.

4.1.2. Feature engineering

In our methodology, we employ feature engineering strategies to enhance the performance of our chosen ML models. We draw inspiration from prior research endeavours to ascertain the significance of specific attributes. For instance, Catolino et al. (2019) employed the *InfoGain* method for feature selection, identifying 6 relevant features out of the original 14. These selected attributes are then integrated into our model for training. Similarly, Nafi et al. (2019) conducted feature selection within their code fragments, resulting in the retention of 9 features from an initial pool of 24. As a result, each clone pair in their dataset featured 18 feature values and a single class label, forming the basis for our ML model training.

When dealing with the Java project dataset, we confront the issue of collinearity among its features. To overcome this challenge, we employ the *AutoSpearman* technique, a method previously utilized in research endeavours by Yatish et al. (2019), Pornprasit et al. (2021), Catolino et al. (2019), and Roy et al. (2022). This technique enables us to identify and retain 27 relevant features from the original set of 65 in our experiment on the Java project dataset. In examining the Postgres dataset, we also encounter the challenge of multicollinearity. To effectively address this issue, we implement the same feature selection and normalization methodologies outlined in the study by Kamei et al. (2012). This approach ensures that our model is well-equipped to handle the complexities inherent in the dataset while preserving the integrity of the selected features.

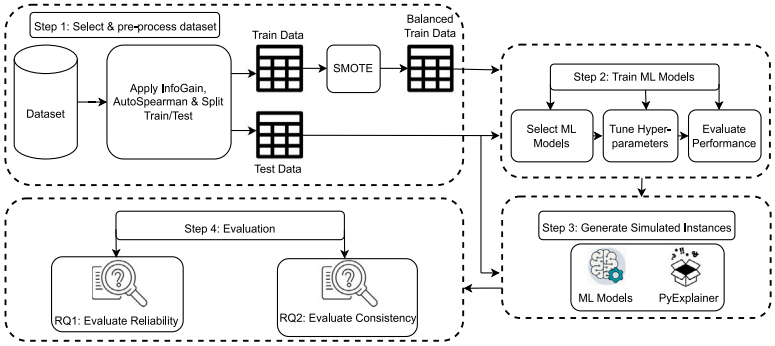


Fig. 5. Our proposed framework for evaluating the reliability and consistency of rule-based XAI techniques. Step 1 includes the dataset selection and pre-processing methods. Step 2 involves the training of ML models, while Step 3 involves the generation of simulated instances. Step 4 investigates the effectiveness of PyExplainer and LIME in generating reliable and consistent explanations for ML models in software analytics tasks.

4.1.3. Dataset balancing

The task of JIT defect prediction suffers from an imbalanced binary classification problem, where one class (e.g., clean commit) has a disproportionately large number of examples compared to the other class (e.g., bug-introducing commit). To address this issue, we apply the Synthetic Minority Over-sampling Technique (SMOTE) (Chawla et al., 2002), which has been used in previous studies (Catolino et al., 2019; Roy et al., 2022; Pornprasit et al., 2021) on the same dataset. Similarly, we use the same technique to address the imbalanced dataset problem in the cross-language clone dataset. It is important to note that we exclusively apply SMOTE to the training dataset, leaving the test dataset unaltered (Jiarpakdee et al., 2020).

4.2. Machine learning model selection and training

ML model selection and training are critical to developing a successful ML project. These steps involve choosing an appropriate algorithm or model architecture for the specific task and training the selected model on the pre-processed dataset. Additionally, finding the best combination of hyperparameter values that optimize the model's performance ensures that the model generalizes well to new, unseen data and avoids overfitting the training data.

4.2.1. Machine learning models

This study evaluates the reliability and consistency of rule-based XAI techniques for generating explanations for classical ML models. Therefore, we target classical ML models extensively used in different software analytics tasks and XAI research. For example, in software defect prediction, Logistic Regression and Random Forest are two of the most commonly utilized classification techniques (Shihab, 2012; Hall et al., 2011; Nadim and Roy, 2023; Nadim et al., 2022). Additionally, although PyExplainer is a model-agnostic XAI technique, its current implementation only supports Sklearn's supervised classification models.[4] Therefore, we consider this criterion when selecting classical ML models. Thus, for our experiments, we select three classical ML models (Logistic Regression (LR), Multi-Layer Perceptron (MLP), and Decision Tree (DT)) and four ensemble methods (Gradient Boosting Classifier (GBC), Random Forest (RF), AdaBoost (ADA), and Bagging (BAG)). These models have been used in previous software analytics tasks and XAI studies (Catolino et al., 2019; Kamei et al., 2012; Pornprasit et al., 2021; Roy et al., 2022; Huang et al., 2024; Brughmans et al., 2024; Begum et al., 2023; Mehta and Patnaik, 2021; Sharma et al., 2023; Singh and Chhabra, 2024; Malhotra and Meena, 2023; Ali et al., 2023; Kumar and Chaturvedi, 2023; Ali et al., 2024; Zhang et al., 2023; Abdou and Darwish, 2024; Rao et al., 2023; Dewangan et al., 2023; Shrimankar et al., 2022; Saha and Chatterjee, 2022; Wang et al., 2023; Xiao et al., 2004; Feng et al., 2024); hence, we select all of them.

[4] https://pyexplainer.readthedocs.io/en/latest/usage.html.

Table 2
Optimized hyperparameter combinations selected from different search techniques.

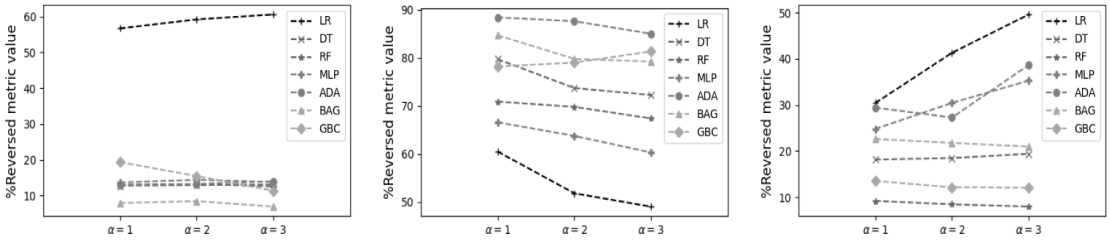
Search technique	AUC value	Parameter value
Grid search	**0.69**	C: 2.78, dual: True, max iter: 130, penalty: 'l2', solver: 'liblinear'
Random search	0.66	solver: 'lbfgs', penalty: 'l2', max_iter: 110, dual: False, C: 166.81
Bayesian optimization	0.67	solver: 'lbfgs', penalty: 'l2', max_iter: 130, dual: False, C: 1291.55

4.2.2. Hyperparameters tuning

The main objective of hyperparameter tuning is to enhance the model's performance on a validation dataset, ensuring its ability to effectively generalize to new and unseen data while avoiding overfitting the training data. By exploring various combinations of hyperparameters, tuning methods aim to identify the most suitable set of values that yield the highest model performance. In our study, we follow commonly used hyperparameter settings to train ML models (Catolino et al., 2019; Kamei et al., 2012; Pornprasit et al., 2021; Roy et al., 2022; Tantithamthavorn et al., 2016a,b, 2018; Yang and Shami, 2020). To tune the hyperparameters, we apply grid search (Liashchynskyi and Liashchynskyi, 2019), random search (Bergstra and Bengio, 2012), and Bayesian optimization (Wu et al., 2019). After obtaining the results from each technique, we compare the AUC values to identify the best-performing hyperparameter combinations. For example, Table 2 presents the three sets of hyperparameters after tuning the LR model on the cross-project mobile app dataset. This table shows that the random search provides us with the best hyperparameter combinations regarding the AUC value. This ensures that our final selection leverages the strengths of each optimization method, leading to the most effective hyperparameter tuning. Thus, we select the best hyperparameter combinations from the three search techniques. The thorough optimization process is fundamental to our ML model training, ensuring that our models are precisely tuned for optimal performance.

4.2.3. Evaluate model performance

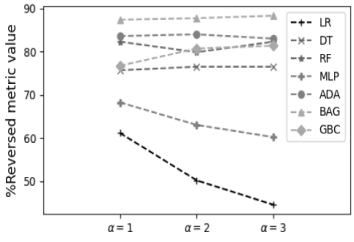
Validation of ML models is a crucial step during model development, aimed at evaluating the model's capability to perform well on new and unseen data. Therefore, to ensure that PyExplainer and LIME provide explanations based on well-trained models, we evaluate the performance of the studied classification techniques. Since PyExplainer often takes excessive time to generate an explanation for a single instance, we split the dataset into training (90%) and testing (10%) to speed up the overall process. To validate ML models using the 90% training data, we apply a 10-fold cross-validation technique like other defect prediction studies (D'Ambros et al., 2010; Wahono, 2015).



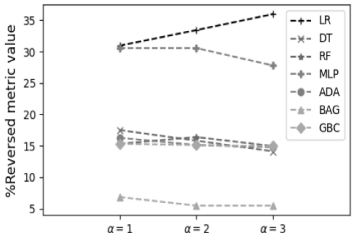
(a) Cross-project (correct predictions)

(b) Cross-project (wrong predictions)

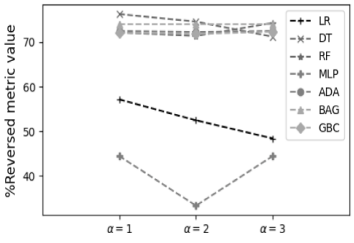
(c) Java (correct predictions)



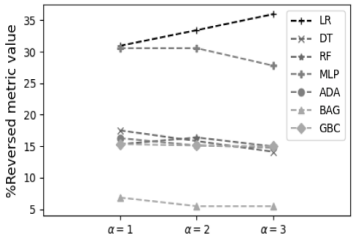
(d) Java (wrong predictions)



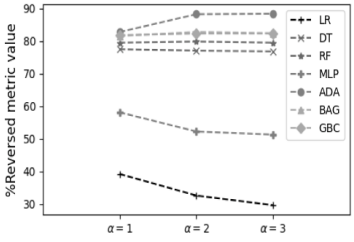
(e) Postgres (correct predictions)



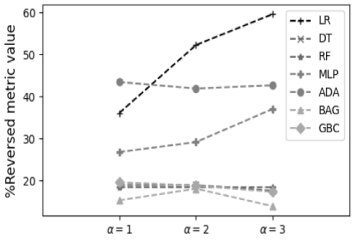
(f) Postgres (wrong predictions)



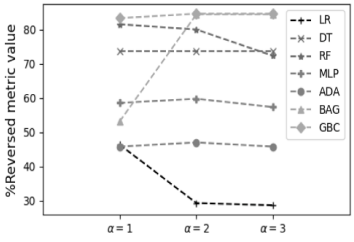
(g) CLCDSA (correct predictions)



(h) CLCDSA (wrong predictions)



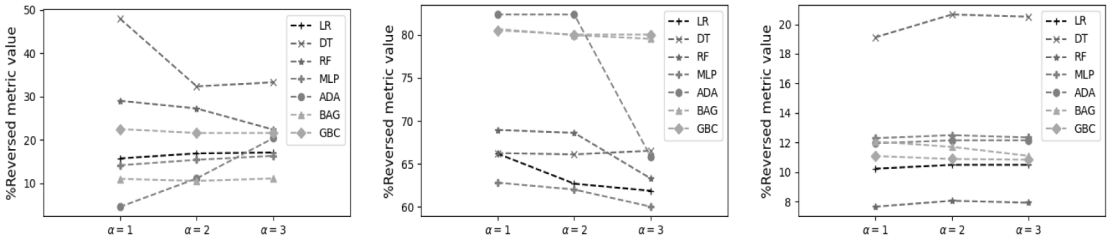
(i) Code review (correct predictions)



(j) Code review (wrong predictions)

Fig. 6. The characteristics of *%Reversed* metric values with different α when we apply PyExplainer to the simulated instances of the selected datasets. For example, Fig. 6(a) represents the variation in *%Reversed* metric values for different α when ML models make correct predictions.

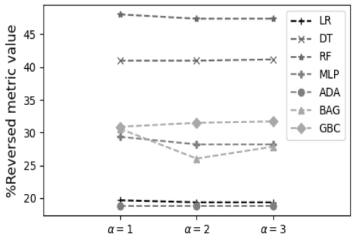
We assess the model's performance based on accuracy and F1-score. Additionally, we employ the Area Under the Receiver Operating Characteristic Curve (AUC-ROC) to assess the discriminatory power of our models, following the recommendation of recent research (Catolino et al., 2019; Jiarpakdee et al., 2020; Lyu et al., 2021; Ghotra et al., 2015; Lessmann et al., 2008; Rahman and Devanbu, 2013), and the 0.75 test-AUC value is the minimum threshold for ensuring the reliability of explanations (Roy et al., 2022; Lyu et al., 2021). The AUC curve measures the true negative rate (coverage of the negative class) on the x-axis and the true positive rate (coverage of the positive class) on the y-axis. As a threshold-independent performance measure, the AUC evaluates the models' ability to discriminate between positive (e.g., defective commits) and negative (e.g., clean commits) instances. The AUC values range from 0 (worst) to 0.5 (no better than random guessing) up to 1 (best) (Hanley and McNeil, 1982). Thus, the AUC-ROC is a comprehensive evaluation technique for binary classification models, gauging their effectiveness in distinguishing between positive and negative instances.



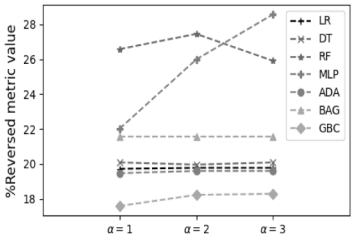
(a) Cross-project (correct predictions)

(b) Cross-project (wrong predictions)

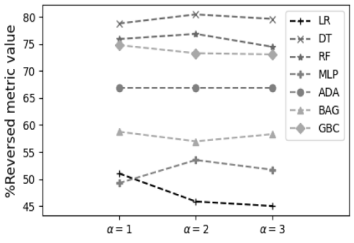
(c) Java (correct predictions)



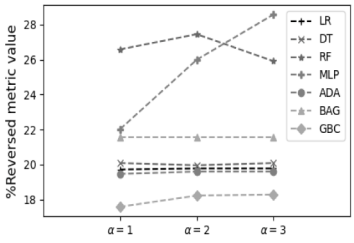
(d) Java (wrong predictions)



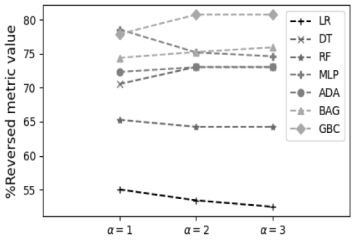
(e) Postgres (correct predictions)



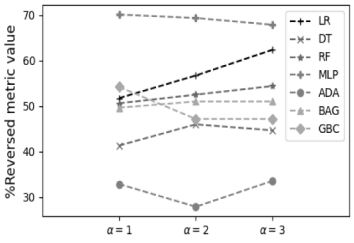
(f) Postgres (wrong predictions)



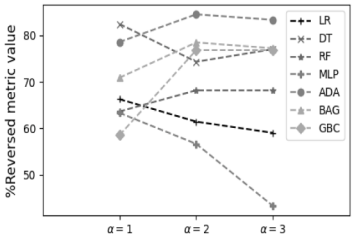
(g) CLCDSA (correct predictions)



(h) CLCDSA (wrong predictions)



(i) Code review (correct predictions)

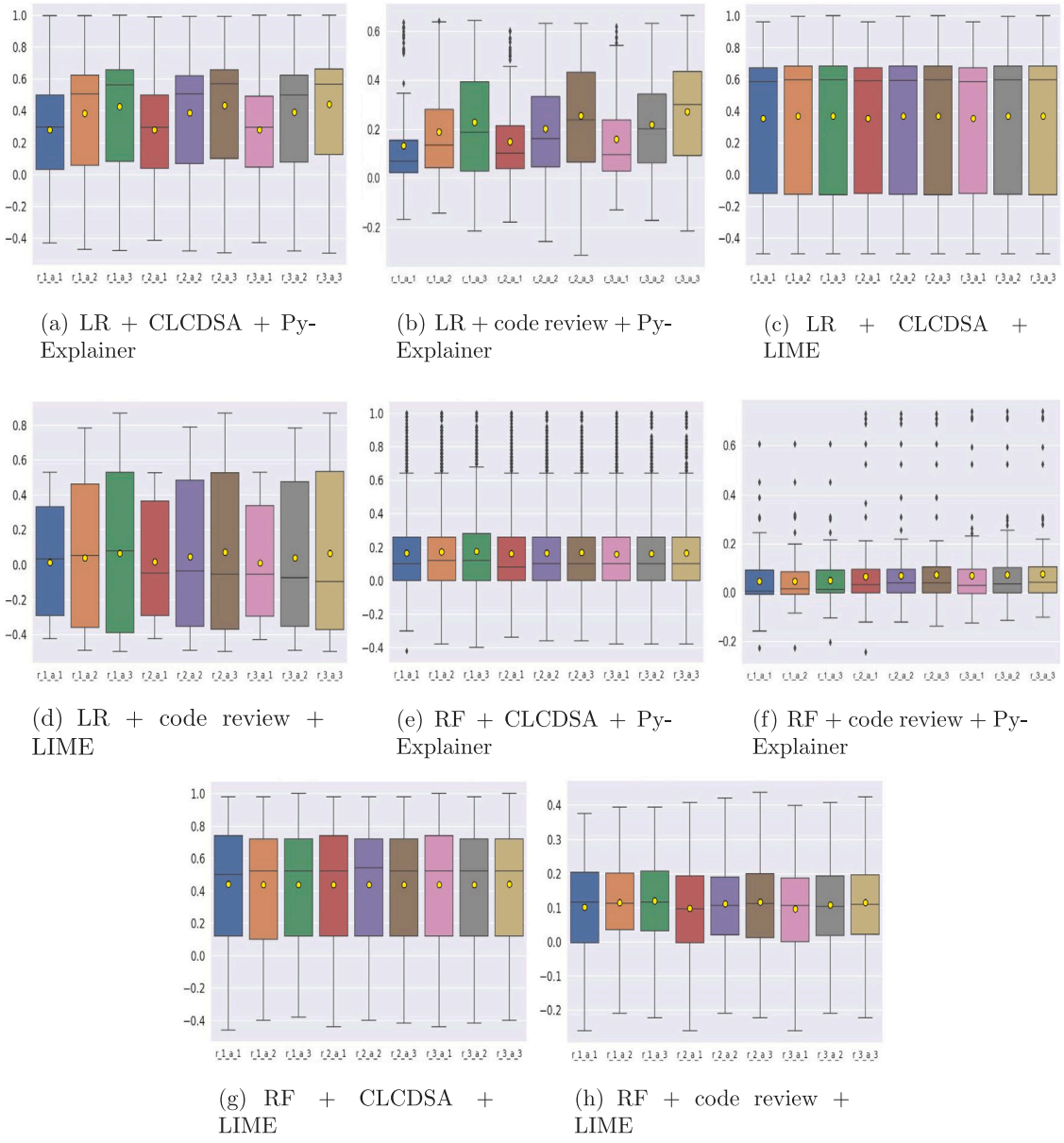


(j) Code review (wrong predictions)

Fig. 7. The characteristics of *%Reversed* metric values with different α when we apply LIME to the simulated instances of the selected datasets. For example, Fig. 6(a) represents the variation in *%Reversed* metric values for different α when ML models make correct predictions.

4.3. Simulated instance generation

We want to evaluate the reliability and consistency of the explanations generated by rule-based XAI techniques (e.g., PyExplainer and LIME) based on the simulated instances (e.g., a single commit) using *EvaluateXAI* and granular-level evaluation metrics described in 4.4. Pornprasit et al. (2021) changed the feature values by one standard deviation (STD) from the threshold values found in the explanation. For example, we want to alter the prediction of defect-introducing commit to clean commit, as shown in Fig. 1(a). From training data, we calculate the STD value of each feature, and let us assume the STD value of the feature *LOC* is 1.9. From Fig. 1(a), we also see the threshold value of *LOC* is approximately 6.35. As we want to change the prediction from defect to clean, we must add the STD value to the threshold value to increase the value in the *green* zone. Therefore, the simulated feature value is set to 1.90+6.35 = 8.25. In the case of Fig. 1(c), we must subtract one STD value from the threshold value of the *nCommit* to move in the *green* zone.



(a) LR + CLCDSA + Py-Explainer

(b) LR + code review + Py-Explainer

(c) LR + CLCDSA + LIME

(d) LR + code review + LIME

(e) RF + CLCDSA + Py-Explainer

(f) RF + code review + Py-Explainer

(g) RF + CLCDSA + LIME

(h) RF + code review + LIME

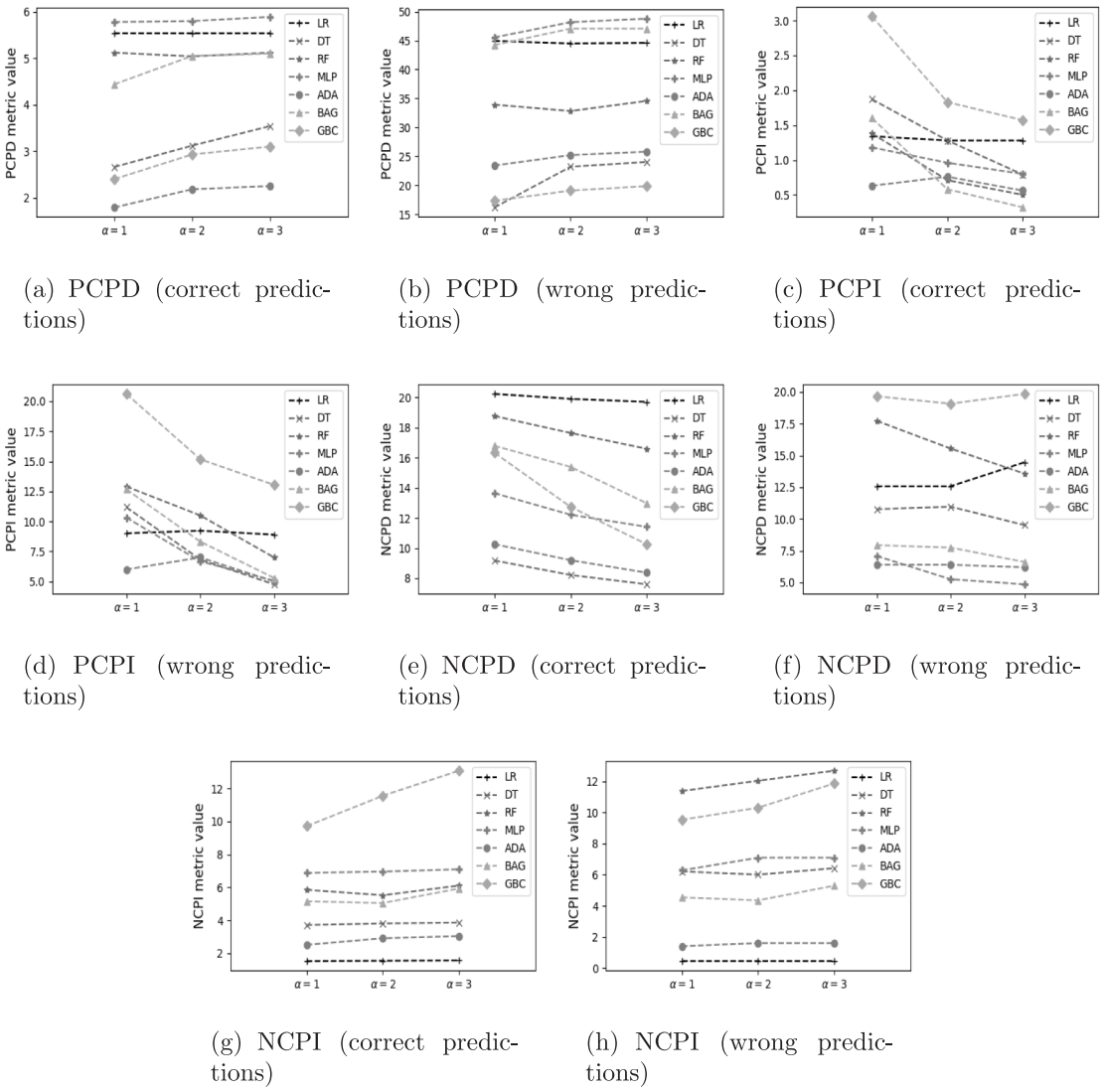
Fig. 8. Distribution of the probability difference between the original and the simulated instances. Figs. 8(a) and 8(b) show the probability difference distribution for the LR model trained on CLCDSA and code review datasets when we use PyExplainer for explanation generation. Figs. 8(c) and 8(d) show the probability difference distribution for the LR model trained on the CLCDSA and code review datasets when we use LIME for explanation generation. Figs. 8(e) and 8(f) show the probability difference distribution for the RF model trained on CLCDSA and code review datasets when we use PyExplainer for explanation generation. Figs. 8(g) and 8(h) show the probability difference distribution for the RF model trained on the CLCDSA and code review datasets when we use LIME for explanation generation. Here, r_i_a_j denotes the ith execution with $\alpha = j$, and LR + CLCDSA + PyExplainer denotes that PyExplainer is applied to explain the outcome of the LR model trained on the CLCDSA dataset.

PyExplainer always creates two types of rules while generating explanations for a given instance: (1) the *less than* rule, which denotes that the value of a particular feature is always less than a threshold value (e.g., **The value of LOC is less than 11.0**, as shown in 1(a)) impacting the model's prediction, and (2) the *more than* rule, which denotes that the value of a particular feature is always greater than a threshold value (e.g., **The value of nCommit is more than 0.62**, as shown in 1(c)) impacting the model's prediction. Considering Figs. 1(a) and 1(c), we observe two rules and if we want to change the prediction of the JIT defect models from defect to clean: (1) for *less than* rule, we add one STD value with the threshold to generate simulated instances; (2) in contrary, for *more than* rule, we subtract one STD value from the threshold to generate simulated instances. Furthermore, if we want to change the prediction of the JIT defect model from clean to defect, we have to do the exact opposite, as described above. Thus, we generate simulated instances for our experiments. We define the generation of a single simulated instance by the following equation:

$$sim(x') = \forall f \in F, |x[f] \pm (\alpha * std)| \quad (1)$$

where x denotes a single instance of a dataset X, x' represents the generated simulated instance, and f is a single feature selected from the set of features F. In Eq. (1), α acts as the controlling parameter, determining the amount of standard deviation to be added or subtracted from each feature value (e.g., the threshold obtained from explanation) of the original instance in generating a simulated instance. Since LIME provides explanations for tabular data in a rule-based format similar to PyExplainer, we can generate simulated instances for LIME using the process outlined above. Below, we describe the detailed process of generating simulated instances for LIME.

The visual explanation provided by PyExplainer illustrates that altering feature values within the *red* zone increases the probability



(a) PCPD (correct predictions) (b) PCPD (wrong predictions) (c) PCPI (correct predictions)

(d) PCPI (wrong predictions) (e) NCPD (correct predictions) (f) NCPD (wrong predictions)

(g) NCPI (correct predictions) (h) NCPI (wrong predictions)

Fig. 9. The values of granular-level evaluation metrics when PyExplainer is applied to the simulated instances of the cross-project mobile apps dataset.

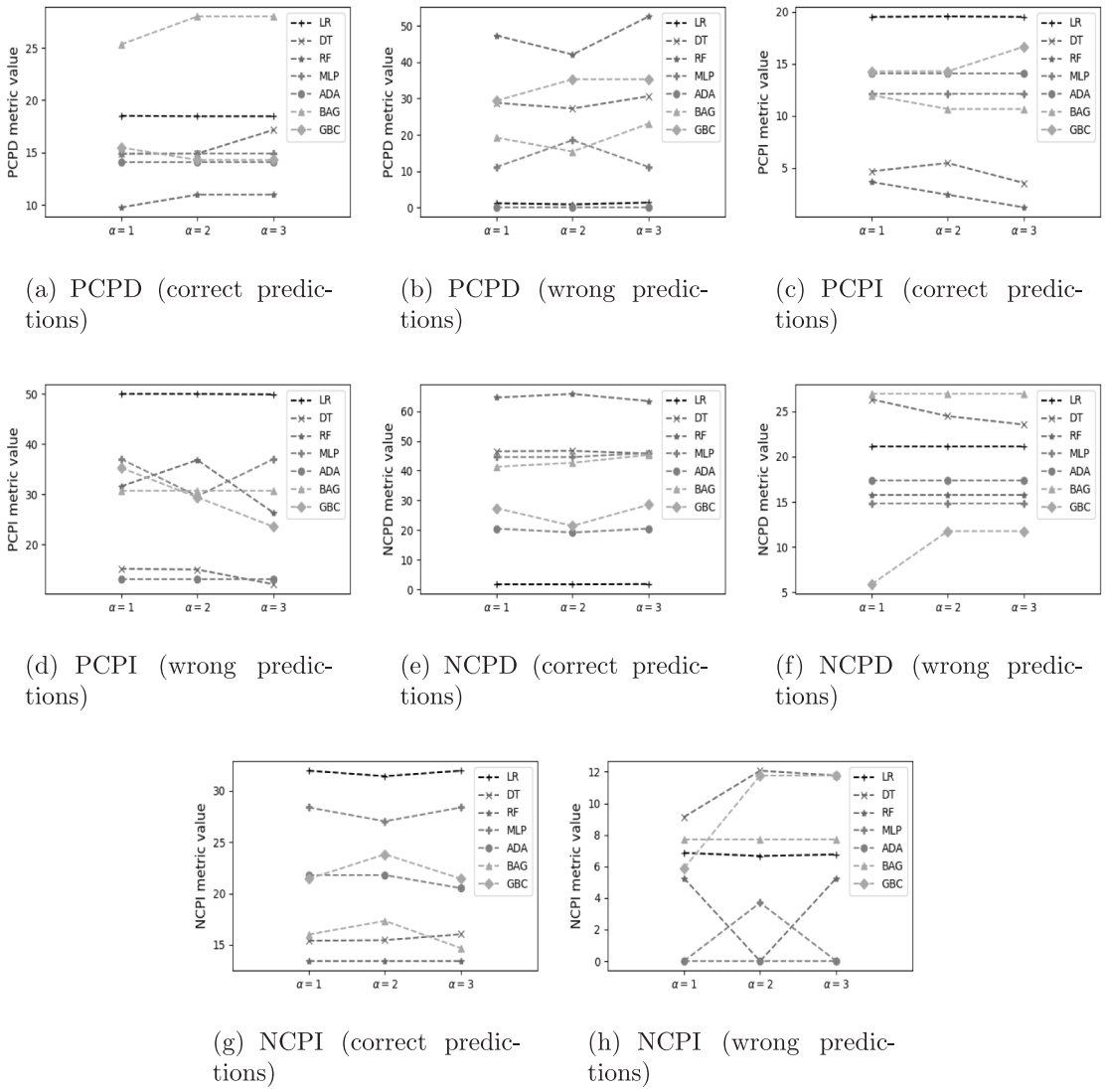
(e.g., risk score) of the given instance being predicted as a positive class (e.g., bug-inducing commit, cross-language clone, and useful code review comments). Conversely, adjusting feature values within the *green* zone reduces the probability. In Fig. 1(a), it is demonstrated that the given instance is predicted as a buggy commit with a probability of 77%, and the associated rule is "the value of LOC is less than 11.0 ($LOC < 11.0$)". Thus, in alignment with the visual explanation, adhering to this rule and decreasing the value of *LOC* in the guided direction (e.g., satisfying the rule $LOC < 11.0$) increases the probability, and vice versa.

A similar procedure can be applied to generate simulated instances using LIME. Fig. 2(a) indicates that the instance is predicted as a buggy commit with a probability of 69%. In this context, the *orange* colour represents the *red* zone. Therefore, adjusting feature values within the *orange* zone to satisfy the rules increases the probability of the given instance being predicted as a positive class (e.g., a buggy commit).

Fig. 2(a) illustrates the influence of three features on the ML model's prediction of the instance as a buggy commit. For example, the actual value of the *AddedLOC* feature is 339.00. In the generated rule (e.g., $AddedLOC > 95.00$), any *AddedLOC* value surpassing 95.00 (by altering feature values within the *orange* zone in accordance with the rules, referred to as the guided direction) raises the probability of the instance being predicted as a positive class. Conversely, modifying feature values counter to the guided direction diminishes the probability of the given instance being predicted as a positive class. For instance, decreasing the feature value (e.g., $AddedLOC < 95.00$) results in a reduced probability of the instance being predicted as a positive class. Thus, this process allows us to generate a simulated instance for any given instance that is predicted as a positive class (e.g., a buggy commit).

We follow similar steps as illustrated above to generate simulated instances for the negative class (e.g., clean commit). Fig. 2(c) indicates that the given instance is predicted as a clean commit with a probability of 82%. In this case, the *blue* colour corresponds to the *green* zone. Therefore, adjusting feature values within the *blue* zone to adhere to the rules increases the probability of the given instance being predicted as a negative class (e.g., a clean commit).

Fig. 2(c) illustrates the influence of three features on the ML model's prediction of the instance as a clean commit. For instance, the actual value of the *nCoupledClass* feature is 14.00. In the generated rule (e.g., $nCoupledClass > 9.00$), any *AddedLOC* value exceeding 9.00 (by altering feature values within the *green* zone in accordance with the rules, referred to as the guided direction) increases the probability of the instance being predicted as a negative class. Conversely, modifying feature values counter to the guided direction decreases the probability of the given instance being predicted as a negative class. For instance,



(a) PCPD (correct predictions)

(b) PCPD (wrong predictions)

(c) PCPI (correct predictions)

(d) PCPI (wrong predictions)

(e) NCPD (correct predictions)

(f) NCPD (wrong predictions)

(g) NCPI (correct predictions)

(h) NCPI (wrong predictions)

Fig. 10. The values of granular-level evaluation metrics when LIME is applied to the simulated instances of the cross-project mobile apps dataset.

reducing the feature value (e.g., $nCoupledClass < 9.00$) leads to a decreased probability of the instance being predicted as a negative class. Thus, this process enables us to generate a simulated instance for any given instance predicted as a negative class (e.g., a clean commit).

4.4. Evaluation

We aim to evaluate the reliability and consistency of the explanations generated by rule-based XAI techniques. In this section, we introduce a set of evaluation metrics at different granularity levels to assess the reliability and consistency of these explanations. Since PyExplainer was designed to address the shortcomings of LIME in explanation generation, we use PyExplainer as a case study to develop the evaluation metrics. We adopt two evaluation metrics from Pornprasit et al. (2021) for reliability and consistency assessment and design four additional granular-level metrics, as detailed below.

Positive class: We consider defective commits, cloned-pair, and useful comments in code review as the positive class.

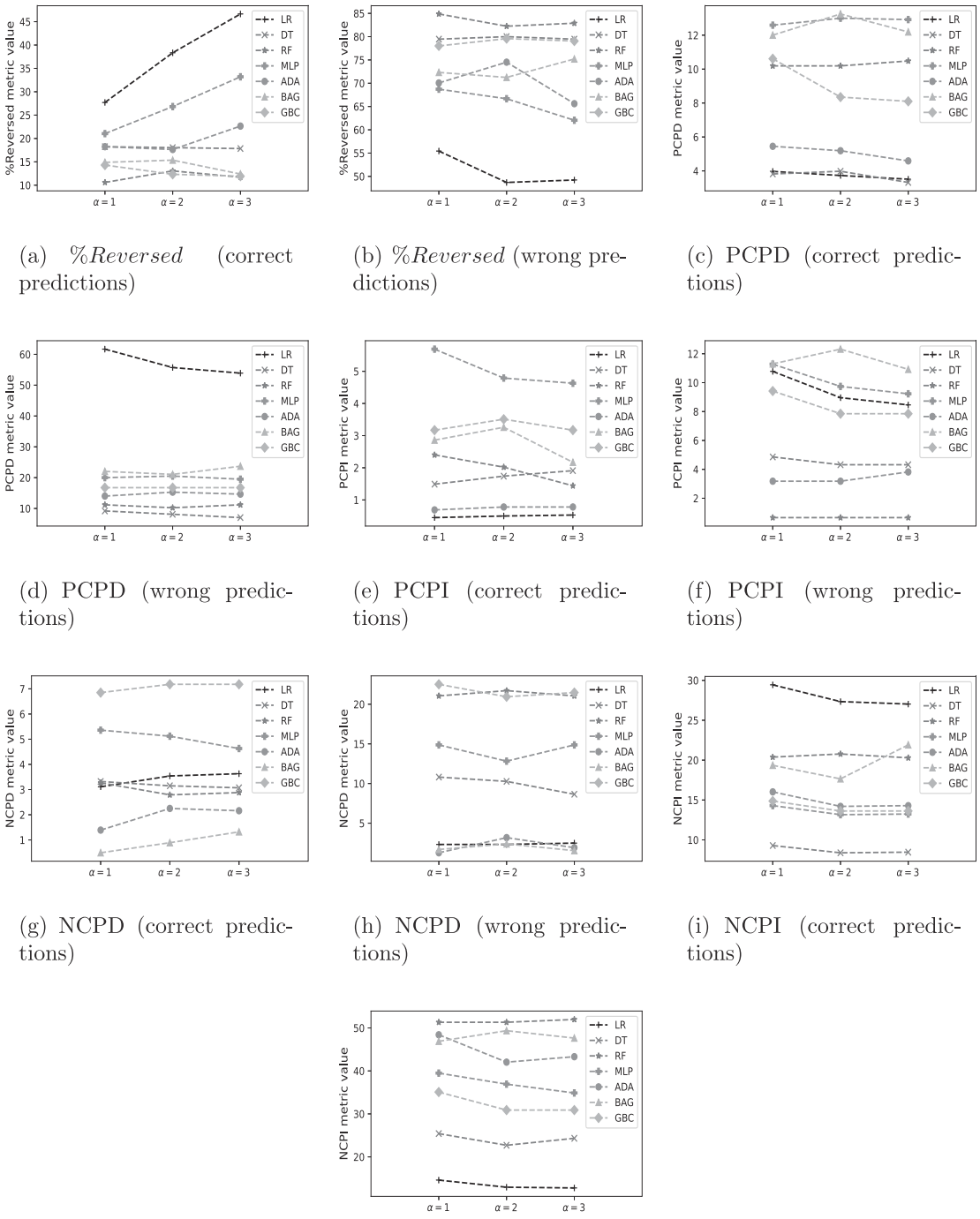
Negative class: We consider clean commit, not clone-pair, and not useful comments in code review as negative class.

%Reversed: $\%Reversed$ assesses the reliability of the explanations generated by PyExplainer. Ideally, the simulated instances generated based on the explanations should cause the model's predictions to flip on those instances. We generate simulated instances as depicted in Fig. 3 and described in Section 4.3. Subsequently, we calculate the proportion of simulated instances that successfully flip the predictions of the ML models. A higher $\%Reversed$ value indicates the effectiveness of PyExplainer in generating reliable explanations for ML models. The $\%Reversed$ metric is defined as:

$$\%Reversed = \frac{|\{x | x \in X \wedge M(x') \neq M(x)\}|}{|X|} \quad (2)$$

where X is a dataset, $x \epsilon X$ is the original instance, x' is the generated simulated instance, and M is the ML model. We separately calculate the $\%Reversed$ metric value for correct and wrong predictions. For instance, consider an ML model trained on a dataset containing 1000 samples with a reported accuracy of 90%. We then determine, out of the 900 correct predictions, how many instances experience a prediction flip. Similarly, we assess how many samples undergo prediction reversal among the 100 wrong predictions. As a result, the combined sum of $\%Reversed$ metrics for correct and wrong predictions may not equate to 100%.

%Prob_diff: $\%Prob\_diff$ denotes the difference between the probability of the original and the simulated instances. Ideally, the probability difference between them should be as high as possible. A higher $\%Prob\_diff$ value demonstrates PyExplainer's effectiveness in producing reliable explanations for ML models. The $\%Prob\_diff$ is defined as:



(a) *%Reversed* (correct predictions)

(b) *%Reversed* (wrong predictions)

(c) PCPD (correct predictions)

(d) PCPD (wrong predictions)

(e) PCPI (correct predictions)

(f) PCPI (wrong predictions)

(g) NCPD (correct predictions)

(h) NCPD (wrong predictions)

(i) NCPI (correct predictions)

(j) NCPI (wrong predictions)

Fig. 11. The characteristics of different metric values with different α when we apply PyExplainer to the simulated instances of the Java project dataset. For example, Fig. 11(a) represents the variation in *%Reversed* metric values for different α when ML models make correct predictions.

$$\%Prob\_diff = \forall x \in X, |P(M(x)) - P(M(x'))| \quad (3)$$

In Eq. (3), $P(M(x))$ denotes the prediction probability of the original instance, $P(M(x'))$ denotes the prediction probability of the simulated instance, and M denotes the ML model. For the positive class, we calculate *%Prob_diff* by subtracting the simulated instance's prediction probability from the original instance's prediction probability. Therefore, in the ideal case, the *%Prob_diff* value must never be negative for each test instance of the positive class. Similarly, for the negative class, we calculate *%Prob_diff* by subtracting the original instance's prediction probability from the simulated instance's prediction probability. Again, the *%Prob_diff* value must never be negative for each test instance of the negative class.

In addition to *%Reversed* and *%Prob_diff* metrics, we design the following four granular-level evaluation metrics:

Positive Class Probability Decrease (PCPD): If we change the feature values of an instance of the positive class in the *green* zone, the

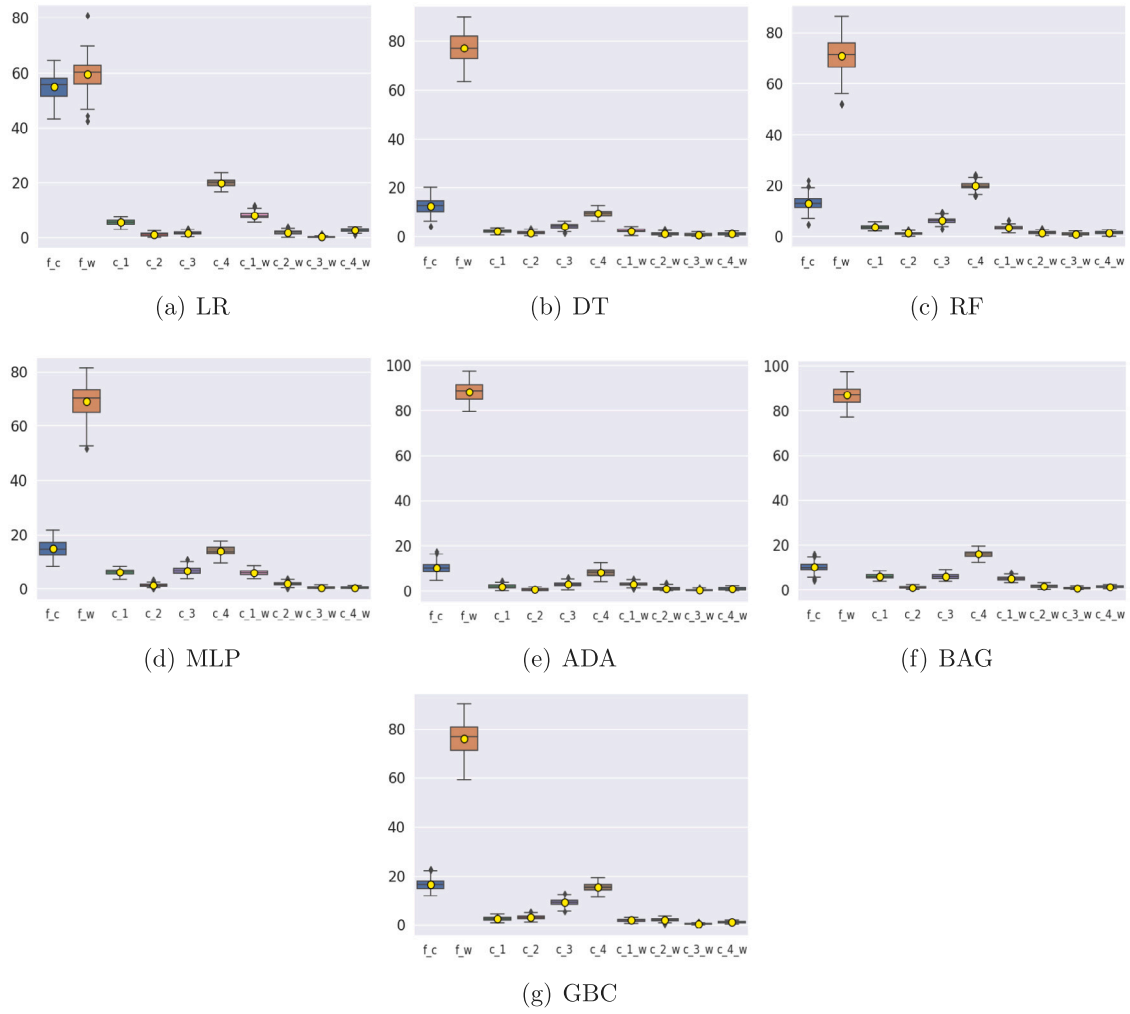


Fig. 12. The distribution of *%Reversed*, PCPD, PCPI, NCPD, and NCPI metrics, encompassing both correct and wrong predictions using PyExplainer on the cross-project mobile apps dataset. Here, *f_c* and *f_w* represent the *%Reversed* metric for correct and wrong predictions, respectively, while c_1, c_2, c_3, c_4 and $c\_1\_w, c\_2\_w, c\_3\_w, c\_4\_w$ PCPD, PCPI, NCPD, NCPI evaluation metrics for correct and wrong predictions, respectively.

risk score must decrease. For instance, Fig. 1(c) shows that when the actual values of $nCommit = 0.6$ and $LOC = 11$, the ML model predicts the given instance as defective commit with a risk score of 77%. Now, after changing the feature values to $nCommit = 0.2$ and $LOC = 4$ in the *green* zone, as demonstrated in Fig. 3(b), the risk score decreases to 68%. However, Fig. 4(b) illustrates that this is not always the case. For example, Fig. 1(a) shows that when the value of $LOC = 11$, the ML model predicts the given instance as defective commits with a risk score of 77%. Despite changing the feature value of *LOC* from 11.0 to 21.0, it remains the same instead of decreasing, as shown in Fig. 4(b). To detect this anomalous behaviour, we define *PCPD* as follows:

$$PCPD = \frac{\sum_{x,x' \epsilon X_p} 1 : if\, R(M(x)) > R(M(x'))}{|X_p|} \quad (4)$$

Positive Class Probability Increase (PCPI): If we change the feature values of an instance of the positive class in the *red* zone, the risk score must increase. For instance, Fig. 1(c) shows that when the actual values of $nCommit = 0.6$ and $LOC = 11$, the ML model predicts the given instance as defective commit with a risk score of 77%. Now, after changing the feature value to $LOC = 15$ in the *red* zone, as shown in Fig. 3(a), the risk score increases to 79%. However, Fig. 4(a) shows this is not always true. For example, considering Figs. 1(a) and 4(a), after changing the feature values of *LOC* from 11.0 to 1.0, the risk score decreased from 77% to 68% instead of increasing. To detect this anomalous behaviour, we define *PCPI* as follows:

$$PCPI = \frac{\sum_{x,x' \epsilon X_p} 1 : if\, R(M(x')) > R(M(x))}{|X_p|} \quad (5)$$

Negative Class Probability Decrease (NCPD): Keeping similarity with the *PCPD*, if we change the feature values of an instance of the negative class in the *green* zone, the risk score must decrease. For instance, Fig. 3(c) shows that when the actual value of *nd* is 1, the ML model predicts the given instance as a clean commit with a risk score of 13%. Now, after changing the feature values to $nd = 1.9$ in the *green* zone, the risk score decreases to 11%, as shown in Fig. 3(d). However, Fig. 4(d) shows this is not always true. For example, considering Figs. 4(c) and 4(d), after changing the feature values of $OWN\_LINE = 0$ and $CountClassCoupled = 0$, the risk score increases to 31% from 20% instead of decreasing. To detect this anomalous behaviour, we define *NCPD* as follows:

$$NCPD = \frac{\sum_{x,x' \epsilon X_c} 1 : if\, R(M(x)) > R(M(x'))}{|X_c|} \quad (6)$$

Negative Class Probability Increase (NCPI): Keeping similarity with the *PCPI*, if we change the feature values of an instance of the negative class in the *red* zone, the risk score must increase. For instance, Fig. 3(c) shows that when the actual value of *nd* is 1, the ML model predicts the given instance as a clean commit with a risk score of 13%. Now, after changing the feature value to $nd = 0.1$ in the *red* zone, the

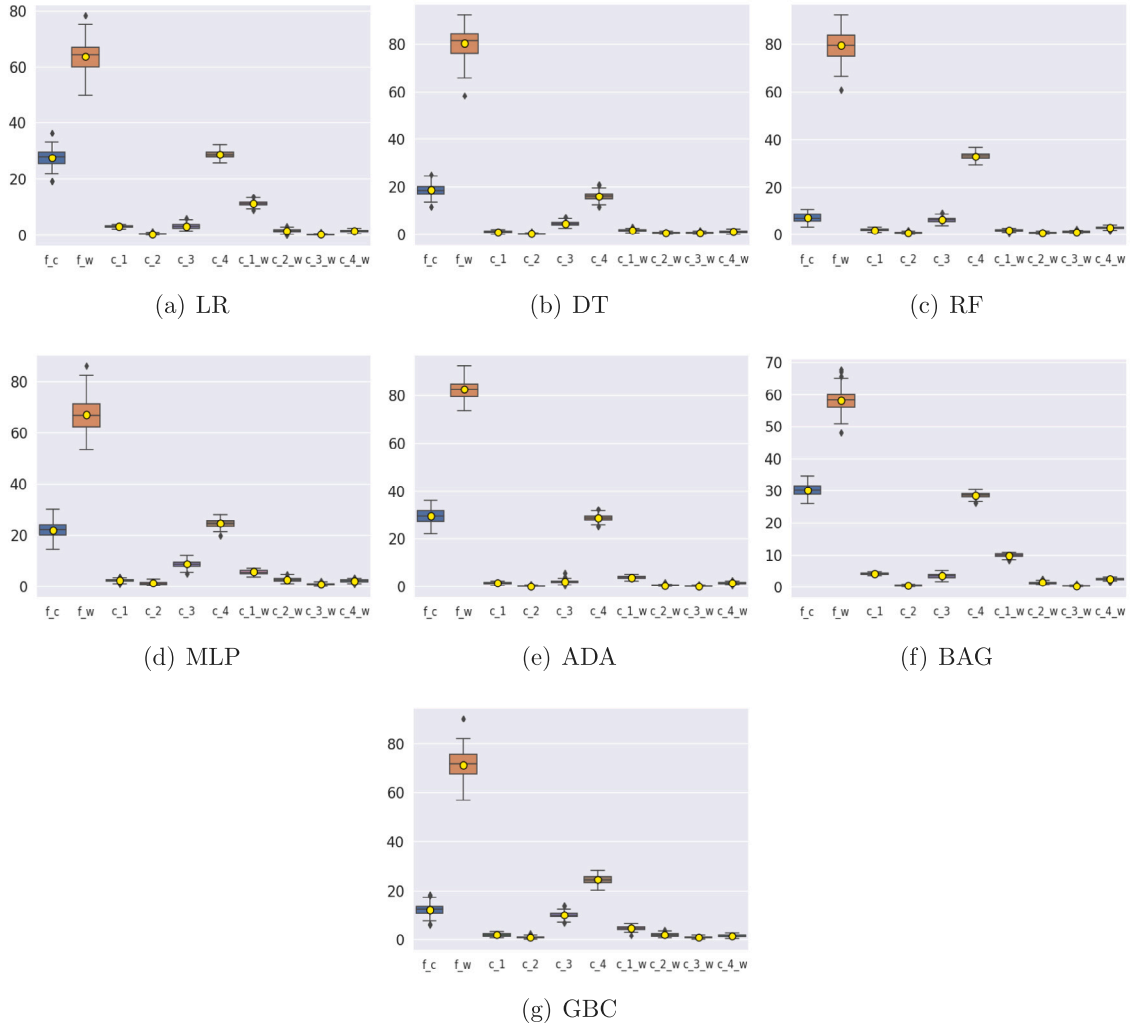


Fig. 13. The distribution of *%Reversed*, PCPD, PCPI, NCPD, and NCPI metrics, encompassing both correct and wrong predictions using PyExplainer on the Java project dataset. Here, f_c and f_w represent the *%Reversed* metric for correct and wrong predictions, respectively, while c_1, c_2, c_3, c_4 and $c\_1\_w, c\_2\_w, c\_3\_w, c\_4\_w$ PCPD, PCPI, NCPD, NCPI evaluation metrics for correct and wrong predictions, respectively.

risk score increases to 17%, as shown in Fig. 3(e). However, Fig. 4(e) shows that this is not always true. For example, considering Figs. 4(c) and 4(e), after changing the feature values of OWN_LINE to 1.9 and $CountDeclClassVariable$ to 6.0, the risk score decreases to 19% from 20% instead of increasing. To detect this anomalous behaviour, we define *NCPI* as follows:

$$NCPI = \frac{\sum_{x,x' \epsilon X_c} 1 : if\, R(M(x')) > R(M(x))}{|X_c|} \quad (7)$$

In Eqs. (4) and (5), X_p is the subset of the dataset X containing only the positive class instances, in Eqs. (6) and (7), X_c is the subset of the dataset X containing only the negative class instances, $R(M(x))$ denotes the risk score of the original instance, $R(M(x'))$ denotes the risk score of the simulated instance.

We evaluate the reliability and consistency of the explanations generated by PyExplainer and LIME against all four granular-level evaluation metrics for each test instance. For example, for each instance, we increment the value of *PCPD* by one when PyExplainer or LIME satisfies that specific metric. We apply the same rule to all other granular-level evaluation metrics. Ideally, we anticipate an increase in the value of each metric for every test instance. For instance, a *PCPD* value approaching 100% indicates the effectiveness of XAI techniques (e.g., PyExplainer, LIME) in producing reliable explanations for ML models and vice-versa. Finally, the increasing controlling parameter α indicates that a higher STD value will be added or subtracted from the original feature values during the generation of a simulated instance. Therefore, as the value of α increases, the values of each evaluation metric also increase, demonstrating the effectiveness of XAI techniques in generating reliable explanations.

Lundberg and Lee (2017) stated that instance explanation generation must remain the same upon regeneration for the same instance. Therefore, we evaluate the consistency of XAI techniques in generating explanations by running them on the same instance multiple times using *EvaluateXAI*, as well as considering the various evaluation metrics described above. Any discrepancies among the multiple runs indicate the inconsistency of XAI techniques in explanation generation.

5. Experimental results

In our study, we utilize scikit-learn,[5] LIME,[6] and PyExplainer libraries[7] to conduct our experiments. We consider five different datasets: three for JIT defect prediction, one for clone detection, and one for code review comment classification and seven ML models previously used in other studies, resulting in a total of $5 \times 7 =$

[5] https://scikit-learn.org/stable/.
[6] https://github.com/marcotcr/lime.
[7] https://pyexplainer.readthedocs.io/en/latest/index.html.

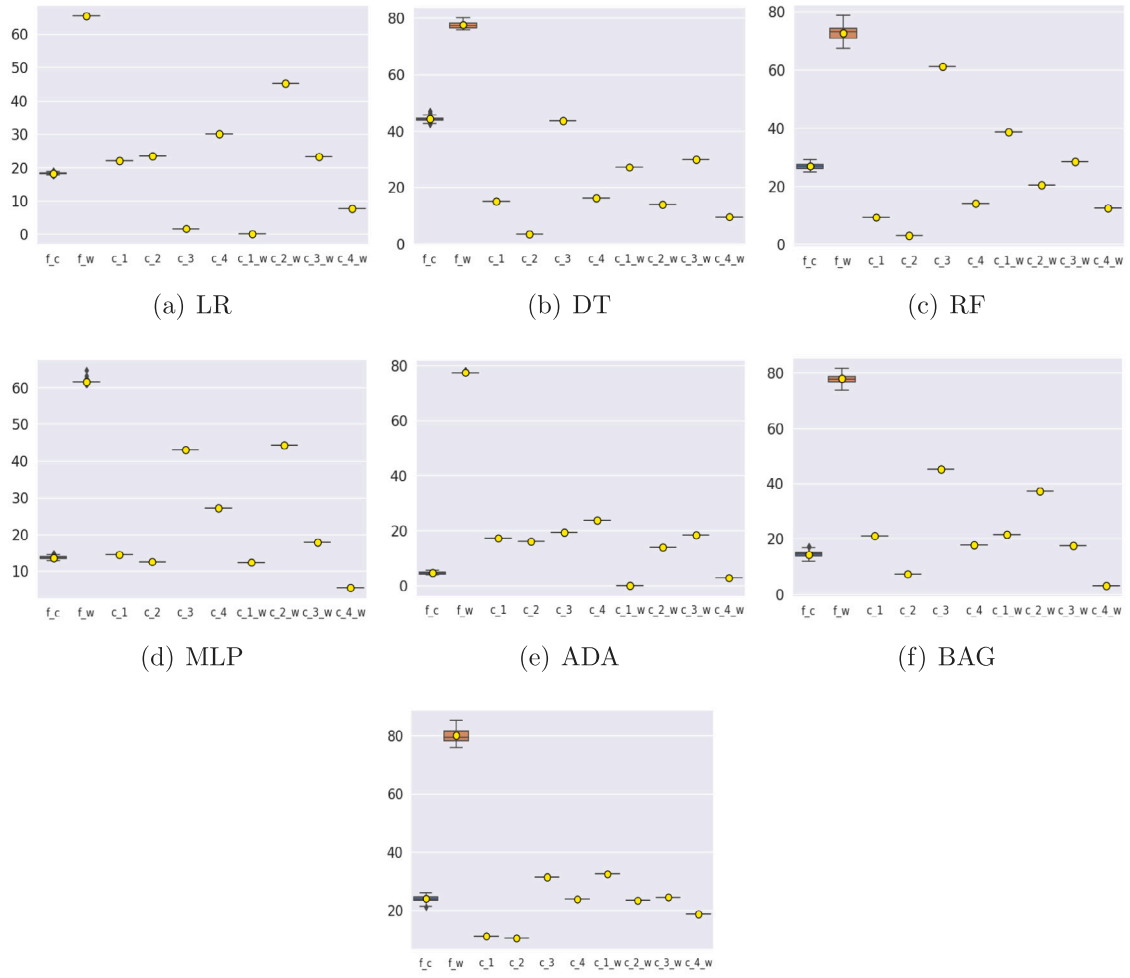


Fig. 14. The distribution of *%Reversed*, PCPD, PCPI, NCPD, and NCPI metrics, encompassing both correct and wrong predictions using LIME on the cross-project mobile apps dataset. Here, f_c and f_w represent the *%Reversed* metric for correct and wrong predictions, respectively, while c_1, c_2, c_3, c_4 and $c\_1\_w, c\_2\_w, c\_3\_w, c\_4\_w$ PCPD, PCPI, NCPD, NCPI evaluation metrics for correct and wrong predictions, respectively.

35 experimental combinations. While Pornprasit et al. (2021) only considered the proportion of correctly predicted instances in the test dataset for *What-if* analysis, our study analyzes both the correctly and wrongly predicted instances to enable extensive experimental analysis. For example, the RF model trained on the CLCDSA dataset correctly predicted 2821 samples and wrongly predicted 280 samples out of 3101 in total. Our study considers all 3101 instances. Thus, we first analyse the performance of our selected ML models, and then, we present the results of our study by addressing the following two research questions:

RQ1: Do rule-based XAI techniques generate reliable explanations for machine learning models in software analytics tasks?

RQ2: Do rule-based XAI techniques maintain consistency in generating explanations for machine learning models in software analytics tasks?

5.1. Analyse model performance

Section 4.2.2 outlines the process of fine-tuning the hyperparameters for our ML models during training. The Table 3 displays the hyperparameters and their optimized values for the ML models trained on the cross-project mobile apps dataset. For hyperparameter settings pertaining to other datasets, please refer to our replication package. Following the identification of these optimized hyperparameters, we proceed to train our ML models.

Table 4 represents the accuracy, F1-score, and AUC values of ML models trained on the different datasets. This table shows that the LR model exhibits the lowest AUC value (0.73), whereas the BAG model shows the highest AUC value (0.85) when trained on the cross-project mobile apps dataset. The ML models trained on the CLCDSA dataset show the highest AUC values, ranging from 0.82 to 0.96. Additionally, Table 4 demonstrates comparatively low AUC values for different ML models trained on the code review dataset. By examining Table 4, we can conclude that all the ML models demonstrate acceptable accuracy, F1-score, and AUC values, indicating their high accuracy and non-overfitting nature. Furthermore, all test-AUCs (except two cases and code review datasets) surpass 0.75, a threshold advocated by prior research as the minimum value necessary for ensuring the reliability of explanations (Roy et al., 2022; Lyu et al., 2021). Fig. C.17 depicts the AUC–ROC curves for all the ML models trained on the selected datasets, as presented in Appendix A.

5.2. Present experimental results

We present the results of our experimental study in relation to our two research questions. The detailed analysis and outcomes are elaborated in the following sections by addressing each research question individually.

RQ1: Do rule-based XAI techniques generate reliable explanations for machine learning models in software analytics tasks?

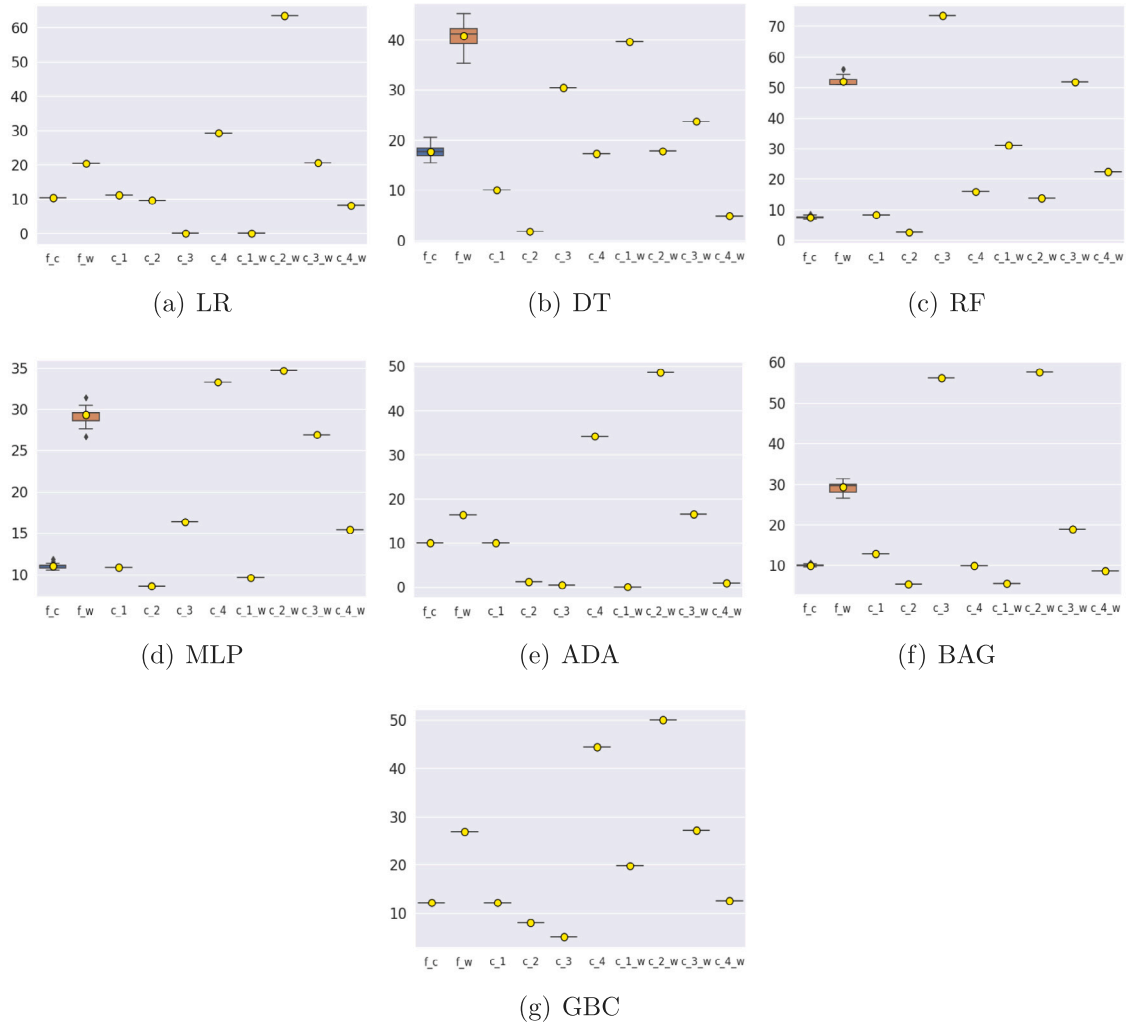


Fig. 15. The distribution of *%Reversed*, PCPD, PCPI, NCPD, and NCPI metrics, encompassing both correct and wrong predictions using LIME on the Java project dataset. Here, f_c and f_w represent the *%Reversed* metric for correct and wrong predictions, respectively, while c_1, c_2, c_3, c_4 and $c\_1\_w, c\_2\_w, c\_3\_w, c\_4\_w$ PCPD, PCPI, NCPD, NCPI evaluation metrics for correct and wrong predictions, respectively.

Table 3
Optimized hyperparameters for different ML models trained on the cross-project mobile apps dataset after tuning.

ML models	Hyperparameters
LR	C: 2.78, dual: True, max_iter: 130, penalty: 'l2', solver: 'liblinear'
DT	min_samples_split: 10, min_samples_leaf: 15, max_depth: 15, criterion: 'gini'
RF	bootstrap: False, max_depth: None, max_features: 'sqrt', min_samples_leaf: 2, min_samples_split: 2, n_estimators: 50
MLP	solver: 'adam', learning_rate: 'constant', hidden_layer_sizes: (10, 30, 10), alpha: 0.0001, activation: 'tanh'
ADA	n_estimators: 20, learning_rate: 1.04, algorithm: 'SAMME.R'
BAG	max_features: 5, max_samples: 100, n_estimators: 1200
GBC	learning_rate: 1, max_depth: 7, min_samples_leaf: 0.1, min_samples_split: 0.1, n_estimators: 200

Table 4
Performance of the ML models on different datasets considering different evaluation metrics such as Accuracy (Acc), F1-score (F-1), and AUC values.

ML model	Dataset														
	Cross project			Java project			Postgres			CLCDSA			Code review		
	Acc	F-1	AUC	Acc	F-1	AUC	Acc	F-1	AUC	Acc	F-1	AUC	Acc	F-1	AUC
LR	0.68	0.49	0.73	0.75	0.39	0.76	0.75	0.57	0.76	0.76	0.81	0.82	0.62	0.69	0.63
DT	0.79	0.60	0.83	0.80	0.49	0.80	0.78	0.59	0.79	0.86	0.88	0.93	0.67	0.73	0.68
RF	0.81	0.58	0.84	0.88	0.55	0.85	0.80	0.59	0.82	0.91	0.93	0.96	0.71	0.76	0.72
MLP	0.75	0.57	0.83	0.80	0.48	0.81	0.71	0.51	0.73	0.83	0.86	0.92	0.60	0.68	0.63
ADA	0.73	0.56	0.81	0.79	0.47	0.81	0.78	0.58	0.79	0.81	0.77	0.85	0.63	0.65	0.67
BAG	0.77	0.59	0.85	0.78	0.50	0.83	0.78	0.61	0.82	0.81	0.84	0.89	0.65	0.73	0.71
GBC	0.81	0.59	0.84	0.81	0.48	0.80	0.78	0.57	0.80	0.85	0.89	0.94	0.63	0.69	0.65

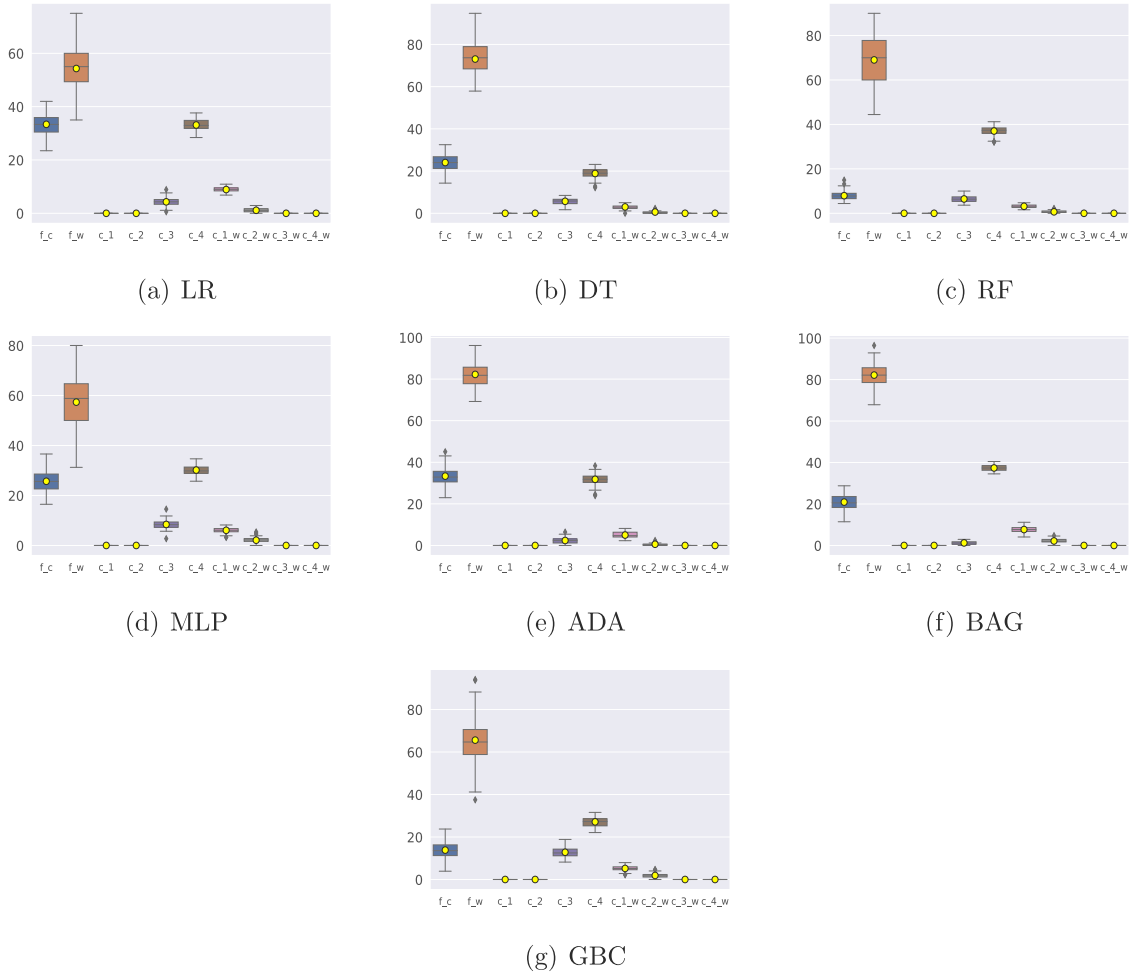


Fig. 16. The distribution of *%Reversed*, PCPD, PCPI, NCPD, and NCPI metrics, encompassing both correct and wrong predictions using PyExplainer on the Java project dataset. Here, f_c and f_w represent the *%Reversed* metric for correct and wrong predictions, respectively, while c_1, c_2, c_3, c_4 and $c\_1\_w, c\_2\_w, c\_3\_w, c\_4\_w$ PCPD, PCPI, NCPD, NCPI evaluation metrics for correct and wrong predictions, respectively.

Motivation: The quality and reliability of the explanations generated by the XAI techniques are crucial. In a scenario where a ML model predicts a defective commit with a 77% risk score as shown in Fig. 1, PyExplainer's visual interpretation suggests that changing feature values in the *green* zone should decrease the risk score and in the *red* zone, it should increase it. However, Fig. 4(b) contradicts this. For example, when the *LOC* value is 11, the risk score remains unchanged despite changing it to 21 in the *green* zone.

Similarly, in Fig. 3(a), changing feature values in the *red* zone increases the risk score from 77% to 79% (Fig. 1(c)). However, Fig. 4(a) reveals an exception to this. Changing *LOC* from 11 to 1 reduces the risk score from 77% to 68%, which is an anomaly in PyExplainer's explanations. We found similar anomalies in the explanations generated by LIME. These anomalies call for further investigations to evaluate the reliability of XAI's generated explanations.

Approach: We address this research question using *EvaluateXAI*, six evaluation metrics, and seven ML models trained on five datasets. For each testing sample of each dataset, we generate simulated instances. Then, we calculate the values of the six evaluation metrics. Finally, we report the experimental results using various tables, figures, and boxplots.

Results: Figs. 6 and 7 display the values of the *%Reversed* metric when PyExplainer and LIME are applied to simulated instances of the cross-project mobile apps, Java project, Postgres, CLCDSA, and code review datasets, respectively. The experiments are conducted with varying α values.

We start by evaluating the reliability of the explanations generated by PyExplainer and LIME for ML models focusing on correct predictions. From Fig. 6(a), we observe the highest *%Reversed* metric values for different α values in the case of the LR model when PyExplainer is run on simulated instances of the cross-project mobile apps dataset. In the case of other ML models, we observe a relatively low *%Reversed* metric value (e.g., $\ll 20\%$). Similar to PyExplainer, LIME exhibits a very similar trend. For example, we observe the highest *%Reversed* metric values for different α values in the case of the DT model when LIME is run on simulated instances of the cross-project mobile apps dataset. In the case of other ML models, we observe a relatively low *%Reversed* metric value (e.g., $\ll 29\%$).

Moving to wrong predictions, we observe improved *%Reversed* metric values for all ML models when PyExplainer and LIME are applied to simulated instances from the cross-project mobile apps dataset. For instance, with $\alpha = 1$, we find that the *%Reversed* metric values vary between 60.51% and 88.4% for PyExplainer and between 62.82% and 82.34% for LIME. Similar results are observed with the other four datasets. Overall, when considering the *%Reversed* metric, PyExplainer and LIME generate relatively reliable explanations for the ML models when they make wrong predictions. However, their performance is less impressive when ML models make correct predictions, with only a few exceptions.

Another important aspect of PyExplainer and LIME is that, as we increase the α values during the generation of simulated instances, we anticipate an increase in the *%Reversed* metric value. However,

the $\%Reversed$ metric values projected with the increase of α values, as shown in Figs. 6 and 7, contradict this expectation. For example, in the case of the GBC model trained on the cross-project mobile apps dataset, PyExplainer exhibits a downward trend in the $\%Reversed$ metric. Additionally, in the case of the RF model trained on the CLCDSA dataset, LIME exhibits a downward trend in the $\%Reversed$ metric. For some ML models, the $\%Reversed$ metric value increases from $\alpha = 1$ to $\alpha = 2$, and then decreases when $\alpha = 3$. Overall, we observe similar results for all the ML models trained on the other datasets. Therefore, the reliability of the explanation generated by PyExplainer and LIME is in question.

We further investigate the variations in prediction probabilities between the original and simulated instances using the $\%Prob\_diff$ metric. We present the experimental results for LR and RF models as these two models were extensively studied for JIT defect predictions (Pornprasit et al., 2021; Jiarpakdee et al., 2020; Roy et al., 2022). Additionally, we select two datasets, CLCDSA and code review, because, for these two datasets, our selected ML models show the highest and lowest accuracy and AUC values, respectively.

First, we evaluate the effectiveness of PyExplainer in generating reliable explanations using the $\%Prob\_diff$ metric. Figs. 8(a) and 8(b) demonstrate that, with the increase in α values, the mean value of $\%Prob\_diff$ consistently increases for the LR model trained on both the CLCDSA and code review datasets. For example, in the first execution with $\alpha = 1$ (e.g., r_1_a_1), the mean value is approximately 0.29 for correct predictions. In the first execution with $\alpha = 2$ (e.g., r_1_a_2), the mean value increases to approximately 0.37, and in the first execution with $\alpha = 3$ (e.g., r_1_a_3), the mean value further rises to approximately 0.43 for correct predictions. In this case, the mean values indicate a comparative difference between the original and the simulated instances. In contrast to the LR model, Figs. 8(e) and 8(f) demonstrate that, with the increase in α values, the mean value of $\%Prob\_diff$ remains almost the same for the RF model trained on both the CLCDSA and code review datasets. Furthermore, Fig. 8(f) clearly demonstrates that for some generated simulated instances, the $\%Prob\_diff$ values are negative, which contradicts the characteristic of the $\%Prob\_diff$ evaluation metric as described in Section 4.4.

Regarding LIME, when we examine the LR and RF models trained on the CLCDSA dataset, we notice a substantial difference in the prediction probabilities between the original and simulated instances, as depicted in Figs. 8(c) and 8(g). However, as we increase the α values, the $\%Prob\_diff$ metric remains relatively constant. In the case of the LR model trained on the code review dataset, we observe an increase in $\%Prob\_diff$ metric values with higher α values. Interestingly, in this case, we encounter some simulated instances with negative $\%Prob\_diff$ values, which contradicts the expected characteristics of the $\%Prob\_diff$ evaluation metric as described in Section 4.4. Finally, Fig. 8(h) illustrates that, for the RF model trained on the code review dataset, the $\%Prob\_diff$ metric value does not show a consistent increase with higher α values. Additionally, the probability difference between the original and simulated instances is insignificant. We observe similar experimental results for the LR and RF models, making wrong predictions. The details of the results can be found in Fig. C.18 in Appendix B.

Figs. 9 and 10 demonstrate the experimental results of four granular-level evaluation metrics (i.e., PCPD, PCPI, NCPD, and NCPI) for PyExplainer and LIME applied to ML models trained on the cross-project mobile apps dataset, both for correct and wrong predictions. In an ideal scenario, we expect an increase in each evaluation metric for each simulated instance. However, Fig. 9 clearly shows that, with PyExplainer, we observe very low values for PCPD, PCPI, NCPD, and NCPI metrics in the case of correct predictions, with only a few exceptions. For instance, when examining the LR model, we find a PCPD value of 24.59 and an NCPI value of 20.22. Conversely, we consistently observe very low PCPI and NCPD values for correct predictions when PyExplainer is used. Turning our attention to LIME, we observe similar results for the ML models making correct predictions.

For wrong predictions, we observe granular-level evaluation metric values that are relatively higher or similar, except for the NCPI metric, when we apply LIME to the simulated instances for all the ML models compared to the correct predictions. From Figs. 9 and 10, it is evident that considering both correct and wrong predictions, the four granular-level evaluation metrics often yield low values, raising concerns about the reliability of the explanations generated by PyExplainer and LIME.

Similar to the $\%Reversed$ and $\%Prob\_diff$ metrics, we can assess whether the values of granular-level evaluation metrics such as PCPD, PCPI, NCPD, and NCPI increase with the rise of α when PyExplainer and LIME are applied to the simulated instances. Figs. 9 and 10 demonstrate that, as α increases, the granular-level evaluation metrics do not consistently increase for all the ML models. For example, in the case of the NCPD metric, we observe either a downward trend or a steady trend with the increase of α values. Therefore, the reliability of the explanation generated by PyExplainer and LIME is in question. Please refer to Tables C.8 and C.9 in Appendix C for the detailed results of granular-level metrics for different ML models trained on the Java project, Postgres, CLCDSA, and code review datasets.

Table 4 reveals that, for certain datasets and ML models, F1-scores are comparatively low due to data imbalance. For example, the Java project test dataset (e.g., after the train-test split) contains 2191 samples of the negative class (e.g., clean commits) and 323 samples of the positive class (e.g., bug-inducing commits). Therefore, we perform additional experiments by balancing the test set to better assess the reliability and consistency of the explanations generated by rule-based XAI techniques (e.g., PyExplainer) using EvaluateXAI. Table 5 reveals that after balancing the test set, F1-scores improve for different ML models trained on the Java project dataset.

Fig. 11 shows that, in the case of PyExplainer, none of the evaluation metrics reach 100% when considering all examples of the balanced test set of the Java project dataset. Additionally, as the α value increases, the metric values do not increase accordingly. Therefore, after balancing the test set, we still obtain similar results to those without balancing the test set of the Java project dataset. We also observe similar results for the $\%Prob\_diff$ metric with the balanced test set of the Java project dataset. The detailed results can be found in our replication package, following the instructions in the README.md file.

Ideally, we expect an increase in the value of each evaluation metric for every test instance. For instance, when the $\%Reversed$ metric value approaches 100%, it indicates the effectiveness of PyExplainer and LIME in producing reliable explanations for ML models. However, the relatively low values of each evaluation metric in our experiments demonstrate that PyExplainer and LIME often fail to generate reliable explanations for ML models in software analytics tasks.

In this study, we introduce a novel framework that employs six evaluation metrics to evaluate the reliability of explanations generated by XAI techniques. Our experiments reveal that XAI techniques do not consistently provide reliable explanations for ML models. This underscores the need for further research to enhance the effectiveness of XAI techniques in generating reliable explanations for ML models in software analytics tasks.

> **Result RQ1**: Our extensive study, employing *EvaluateXAI* on five datasets and seven ML models, concludes that none of the evaluation metrics reach 100% when considering all the testing instances for explanation generation in our experiments. This indicates that rule-based XAI techniques, such as PyExplainer and LIME, often fail to generate reliable explanations for ML models in software analytics tasks.

RQ2: Do rule-based XAI techniques maintain consistency in generating explanations for machine learning models in software analytics tasks?

Table 5
Performance of the ML models on Java project dataset considering different evaluation metrics such as Accuracy (Acc), F1-score (F-1), and AUC values.

ML model	Acc		F-1		AUC	
	Before balancing	After balancing	Before balancing	After balancing	Before balancing	After balancing
LR	0.75	0.69	0.37	0.67	0.76	0.76
DT	0.80	0.74	0.49	0.72	0.80	0.79
RF	0.88	0.76	0.55	0.70	0.85	0.87
MLP	0.80	0.73	0.48	0.70	0.81	0.80
ADA	0.79	0.76	0.47	0.75	0.81	0.81
BAG	0.78	0.75	0.50	0.75	0.83	0.82
GBC	0.81	0.72	0.48	0.68	0.80	0.79

Motivation: Lundberg and Lee (2017) argued that instance explanation generation must remain consistent upon regeneration for the same instance. However, Fig. 1 shows that PyExplainer (while PyExplainer involves randomization in the neighbour generation process, Pornprasit et al. (2021) claimed that **"we repeated the experiment five times, the conclusion of our paper remains the same"**) provides four different explanations for the same instance upon re-execution. On the other hand, Figs. 2(a) and 2(b) demonstrate that LIME (which involves random perturbation) provides two different explanations for the same instance upon re-execution. Furthermore, we notice that among the five features, PyExplainer utilizes four of them (e.g., Line of Code (*LOC*), *nCommit*, *CommentToCodeRatio*, and *nCoupledClass*) to generate explanations for various executions. Additionally, in Fig. 1(a), PyExplainer generates an explanation for cases where the feature value *LOC* is less than 11.0, while in Fig. 1(c), it generates explanations for instances where the feature value *LOC* is greater than 11.0. Lastly, Fig. 1(b) illustrates a scenario in which PyExplainer fails to generate an explanation. These inconsistent explanations may confuse practitioners and discourage them from using XAI techniques to explain the results of ML models in software analytics tasks (Roy et al., 2022). Therefore, further research is warranted to evaluate the consistency of the explanations generated by XAI techniques.

Approach: Jiarpakdee et al. (2020) assessed inconsistency by running 100 consecutive iterations on a single instance and measuring the rank difference of each metric. We employ a similar approach but consider 500 instances instead of a single one. Thus, we address this research question by conducting the same experimental analysis performed for the first research question (**RQ1**) multiple times. We compare the results from each execution using six different evaluation metrics. For the evaluation metric $\%Prob\_diff$, to measure whether the distributions of the $\%Prob\_diff$ between two executions are statistically different, we apply the Wilcoxon Signed-Rank Test (Wilcoxon, 1992). Additionally, we utilize Cliff's $|\delta|$ effect size (Cliff, 1993) to quantify the extent of the differences. Finally, we report the experimental results using several tables and boxplots.

Results: We evaluate the consistency of PyExplainer and LIME by examining the results of six evaluation metrics. We expect that when we run PyExplainer and LIME multiple times on the same test instances with the same experimental settings, we should obtain consistent results across different evaluation metrics. Any discrepancies among successive executions indicate the inconsistency of PyExplainer and LIME in generating explanations for ML models.

Jiarpakdee et al. (2020) regenerated instance explanations 100 times by randomly selecting only one instance to observe the consistency of the explanations generated by *LIME*, *LIME-HPO*, and *BreakDown* for JIT defect prediction datasets regarding rank difference metric. In our study, we adopt a similar approach but consider 500 test instances instead of just one to evaluate the consistency of PyExplainer and LIME in explanation generation using *EvaluateXAI*. We calculate the $\%Reversed$ metric and values for four granular-level (e.g., PCPD, PCPI, NCPD, NCPI) metrics for 100 iterations to plot the distribution as shown in Figs. 12, 13, 14, and 15 and observe any differences. Additionally, we consider three different executions with varying α values: $\alpha = 1$, $\alpha = 2$, and $\alpha = 3$ to calculate the $\%Prob\_diff$ metric for the CLCDSA and code review datasets. Keeping similarity with Jiarpakdee et al. (2020), we consider two defect prediction datasets, such as cross-project mobile apps and Java project, to assess the consistency of PyExplainer and LIME in explanation generation regarding the evaluation metrics except $\%Prob\_diff$.

Table 6
Interpretation of Cliff's $|\delta|$ effect size.

Interval	Interpret effect
$\delta < 0.147$	Negligible effect
$0.147 \leq 0.33$	Small effect
$0.33 \leq 0.474$	Medium effect
$\delta \geq 0.474$	Large effect

First, we analyse the variation in $\%Reversed$ metric values across different executions for correct and wrong predictions. The distribution of $\%Reversed$ metric values in Figs. 12 and 13 reveals notable differences, irrespective of the studied datasets and ML models, when PyExplainer is employed for explanation generation. Similar patterns are observed with LIME, as depicted in Figs. 14 and 15, with a few exceptions. For example, LR and ADA models trained on the cross-project mobile apps dataset show no difference in $\%Reversed$ metric values for wrong predictions. Similarly, LR, ADA, and GBC models trained on the Java project datasets exhibit no variation in $\%Reversed$ metric values for both correct and wrong predictions. Regarding the $\%Reversed$ metric, PyExplainer does not generate consistent explanations, while LIME demonstrates relatively more consistency, although it is not entirely consistent in explanation generation.

Secondly, we examine whether the $\%Prob\_diff$ metric values differ across multiple executions for correct predictions. By visually inspecting Figs. 8(a) and 8(b), we observe differences among the executions for the ML models trained on the CLCDSA and code review datasets when we apply PyExplainer to the simulated instances. With LIME, we observe differences only for the LR model trained on the code review dataset. Although it is difficult to discern the differences among the executions for the ML models by visual inspection, we quantitatively measure the differences using the Wilcoxon signed-rank test and the magnitude of the difference with Cliff's $|\delta|$ effect size. We interpret the Cliff's $|\delta|$ effect size as shown in Table 6. Table 7 presents the corresponding p-values and Cliff's $|\delta|$ effect size for assessing the differences.

To evaluate the consistency of the explanations generated by rule-based XAI techniques, we start with PyExplainer and subsequently present the experimental results of LIME. From Table 7, we see that in the case of the RF model trained on the code review dataset, there is no significant $\%Prob\_diff$ metric difference when the value of α is 1 for multiple executions. For the same model, we observe a difference when the value of α is 1 between the first and third executions, although the effect size is small. In contrast, for all other cases, the differences between multiple executions are clearly visible despite the small effect size values. Moving to LIME, we find no significant $\%Prob\_diff$ metric difference between the first and second executions for the RF model trained on the CLCDSA dataset. Furthermore, for both models trained on the code review dataset, there is no significant difference between

Table 7
*p*_values and Cliff's $|\delta|$ effect size between different executions on the same instance across different datasets under the same α value. Here, *R_i_R_j* denotes the difference between *i*th and *j*th execution.

XAI technique	ML model	α values	Dataset											
			CLCDSA						Code review					
			R_1_R_2		R_1_R_3		R_2_R_3		R_1_R_2		R_1_R_3		R_2_R_3	
			p_val	Cliff's $\|\delta\|$	p_val	Cliff's $\|\delta\|$	P_val	Cliff's $\|\delta\|$	p_val	Cliff's $\|\delta\|$	p_val	Cliff's $\|\delta\|$	p_val	Cliff's $\|\delta\|$
PyExplainer	LR	$\alpha = 1$	0.00058	0.22	0.00099	0.31	0.00025	0.11	0.00043	0.23	0.00033	0.29	0.00054	0.11
		$\alpha = 2$	0.0002	0.23	0.00015	0.33	0.000022	0.11	0.00011	0.15	0.00045	0.26	0.00011	0.15
		$\alpha = 3$	0.00001	0.25	0.000017	0.34	0.000033	0.12	0.00051	0.18	0.00014	0.29	0.00026	0.14
	RF	$\alpha = 1$	0.00039	0.02	0.00069	0.03	0.00018	0.01	**0.7464**	**0.01**	**0.2012**	**0.04**	**0.0665**	**0.03**
		$\alpha = 2$	0.00011	0.02	0.00068	0.03	0.00026	0.01	**0.0643**	**0.02**	0.017	0.04	**0.0972**	**0.024**
		$\alpha = 3$	0.00033	0.01	0.00052	0.02	0.00073	0.08	0.029	0.036	0.0037	0.068	0.0036	0.03
LIME	LR	$\alpha = 1$	0.00016	0.04	0.00272	0.05	0.00061	0.01	**0.0536**	**0.01**	0.0038	0.001	0.0003	0.02
		$\alpha = 2$	0.00045	0.03	0.00065	0.04	0.00037	0.01	0.035	0.006	0.0062	0.006	0.0014	0.013
		$\alpha = 3$	0.00032	0.03	0.00041	0.04	0.00048	0.01	0.0250	0.003	0.0051	0.002	0.0016	0.02
	RF	$\alpha = 1$	**0.237**	**0.01**	**0.621**	**0.01**	0.0169	0.001	0.015	0.05	0.00046	0.07	0.0315	0.02
		$\alpha = 2$	**0.8455**	**0.01**	**0.378**	**0.01**	0.0174	0.01	0.0244	0.046	0.00039	0.067	0.0185	0.02
		$\alpha = 3$	**0.419**	**0.01**	**0.524**	**0.01**	0.00045	0.01	**0.072**	**0.03**	0.00082	0.058	0.0036	0.026

the first and second executions. On the other hand, in all other cases, differences between multiple executions are noticeable despite having negligible and small effect size values. Lundberg and Lee (2017) argued that "instance explanation generation must remain consistent upon re-generation for the same instance", emphasizing that even negligible or small effect sizes can impact the consistency of explanations generated by XAI techniques. However, when considering the $\%Prob\_diff$ metric, the experimental results mentioned above appear to contradict this statement.

Finally, we expect PyExplainer and LIME to produce the same output for other granular-level (e.g., PCPD, PCPI, NCPD, NCPI) metrics for multiple executions, considering correct and wrong predictions. However, Figs. 12 and 13 demonstrate that regardless of the studied datasets and ML models, PyExplainer generates inconsistent instance explanations. Upon visual inspection of these figures, it becomes evident that there are noticeable differences in the four granular-level evaluation metrics (e.g., PCPD, PCPI, NCPD, NCPI), both for correct and wrong predictions. In contrast, as depicted in Figs. 14 and 15, we observe consistent values for PCPD, PCPI, NCPD, and NCPI metrics, indicating that LIME exhibits greater consistency in generating explanations for ML models in software analytics. Table 7 quantitatively demonstrates that PyExplainer provides $(31/36) * 100 = 86.11\%$ inconsistent explanations for all the test instances of CLCDSA and code review datasets. In contrast, LIME provides $(28/36) * 100 = 77.78\%$ inconsistent explanations.

Similar to evaluating the reliability of the explanations generated by PyExplainer using the balanced test set of the Java project dataset for addressing RQ1, we conduct the same experiments originally performed on the imbalanced test dataset to assess consistency using the balanced test data. Fig. 16 shows the distributions of $\%Reversed$, PCPD, PCPI, NCPD, and NCPI metrics. From this figure, we observe notable differences across multiple executions, considering different metrics, irrespective of the studied ML models when PyExplainer is employed for explanation generation. We also observe similar results for the $\%Prob\_diff$ metric with the balanced test set of the Java project dataset. The detailed results can be found in our replication package, following the instructions in the README.md file. Therefore, despite using the balanced test set, PyExplainer still produces inconsistent explanations.

Therefore, using *EvaluateXAI* and considering the $\%Reversed$, $\%Prob\_diff$ and four granular-level evaluation metrics (e.g., PCPD, PCPI, NCPD, NCPI), we can conclude that PyExplainer does not provide consistent explanations for ML models in software analytics tasks. In addition, although LIME offers a bit more consistent explanation regarding granular-level evaluation metrics, its inconsistency can be assessed using *EvaluateXAI* by considering the $\%Reversed$ and $\%Prob\_diff$ metrics.

Result RQ2: In the case of 36 experimental combinations involving the CLCDSA and code review datasets, PyExplainer and LIME provide inconsistent explanations for 31 and 28 cases, representing 86.11% and 77.78%, respectively. These findings corroborate the inconsistent explanations produced by PyExplainer and LIME for ML models in software analytics tasks.

6. Key findings & guidelines

In this study, we design a novel framework called *EvaluateXAI*, along with six evaluation metrics, to evaluate the effectiveness of rule-based XAI techniques in generating reliable and consistent explanations for ML models trained on tabular data. We summarize the key findings from our study as follows:

1. PyExplainer and LIME generate completely different explanations, as shown in Figs. 1 and 2 for the same instance regarding multiple executions. These inconsistent explanations may confuse its practitioners and discourage them from using PyExplainer and LIME to explain the outcome of JIT defect prediction models. Therefore, further research is necessary to generate consistent explanations across multiple executions of the same instance.
2. While manually investigating the consistency of the explanations generated by PyExplainer and LIME, we found that LIME generates more consistent rules in each explanation than PyExplainer. For instance, in Figs. 2(a) and 2(b), we observe two different explanations from LIME for the same instance. However, the generated rules are the same, with only a change in orientation. In contrast, Fig. 1 shows that PyExplainer produces four different explanations with distinct rules for the same instance. Additionally, we observe the rule values are different, as shown in Figs. 1(a) and 1(c). For example, in Fig. 1(a), PyExplainer generates the explanation where the feature value *LOC* is less than 11.0. On the other hand, in Fig. 1(c), PyExplainer generates the explanation where the feature value *LOC* is more than 11.0. Our experimental results also indicate that LIME provides more consistent rules and explanations than PyExplainer across four granular-level evaluation metrics: PCPD, PCPI, NCPD, and NCPI. Therefore, generating consistent rules is crucial for ensuring the effectiveness of XAI techniques in practical applications.
3. Our study only considers the rule-based XAI techniques PyExplainer and LIME. We adopt two metrics from the existing study by Pornprasit et al. (2021) and design four other granular-level evaluation metrics to evaluate the reliability and consistency of generated explanations by PyExplainer and LIME. However, the in-depth experimental results possibly threaten the effectiveness of the PyExplainer and LIME in providing reliable and consistent

explanations for the JIT defect prediction models. In addition, the empirical findings also demonstrate that PyExplainer does not always perform well in generating explanations for the ML models trained on the CLCDSA and code review datasets, meaning it is less generalizable. Therefore, further actions are warranted to advance the XAI research in software analytics.

4. Pornprasit et al. (2021) changed the original feature value by one STD (a standard deviation of that particular feature found from the training dataset) from the rule threshold. We also apply the same technique to change the feature values by one STD while generating simulated instances. However, there is a high chance of generating invalid feature values. For example, for the given instance, as shown in Table 1, the value of *LOC* feature is 11.0, and the approximate threshold value is 16 found from the explanation as shown in Fig. 1(c). Now, consider one STD value of this feature is 50, and for "*more than*" rule, we have to subtract one STD from the rule threshold value to generate a simulated instance. Any further subtraction operation will yield a negative feature value (e.g., $16 - 50 = -34$), which is not accepted as the *Line of Code (LOC)* cannot be negative in a file. Therefore, further research is essential for the effective generation of simulated instances.
5. We design *EvaluateXAI* and the granular-level evaluation metrics based on the PyExplainer. However, our simulated instance generation process for LIME, along with comprehensive experimental results, demonstrates that *EvaluateXAI* is adaptable to any rule-based XAI technique. This adaptability allows us to evaluate their effectiveness in generating reliable and consistent explanations for ML models trained on tabular data in software analytics tasks.

Finally, based on the in-depth experiments conducted to evaluate the effectiveness of PyExplainer and LIME in generating reliable and consistent explanations using *EvaluateXAI* and evaluation metrics, we offer the following recommendations to XAI researchers and users:

1. Ensure that the generated explanations remain consistent even with multiple executions on the same instance.
2. Ensure that the generated rules in each explanation remain consistent even with multiple executions on the same instance.
3. The rule-based XAI techniques must satisfy the granular-level evaluation metrics (e.g., PCPD, PCPI, NCPD, NCPI) outlined in our study to ensure the reliability of the generated explanations.
4. If practitioners use JIT defect prediction models to prioritize high-risk commits, we suggest exercising caution when allocating SQA resources based on the explanations provided by rule-based XAI techniques (e.g., PyExplainer or LIME).

7. Threats to validity

This section briefly explains the internal and external threats associated with our research and the methods we employed to address them.

7.1. Internal threat

The first potential internal threat is the accuracy of the selected ML models. However, we addressed this issue by adopting the parameter and hyperparameter configurations outlined in previous studies (Yatish et al., 2019; Pornprasit et al., 2021; Jiarpakdee et al., 2020; Roy et al., 2022). To handle imbalanced datasets and multicollinearity issues in training ML models, we also utilized the SMOTE and Autospearman parameter settings defined in Tantithamthavorn et al. (2018). Furthermore, we applied random search, grid search, and Bayesian optimization to select the best hyper-parameter combinations, resulting in highly accurate and non-overfitted ML models. Finally, we tested our trained models based on AUC values, confirming their high accuracy. Thus, we successfully mitigated this internal threat.

Another internal threat could be generating simulated instances to flip predictions, find the prob-diff, and calculate different granular-level evaluation metrics (e.g., PCPD, PCPI, NCPD, NCPI) values. Nevertheless, we addressed this threat by changing the feature values by one standard deviation from the threshold values found by PyExplainer, as described in Pornprasit et al. (2021).

7.2. External threat

The generalizability of our experimental results can be considered a potential threat to the external validity of *EvaluateXAI*. However, we addressed this issue by selecting seven state-of-the-art ML models and five datasets previously used in other studies (Catolino et al., 2019; Pornprasit et al., 2021; Roy et al., 2022; Nguyen et al., 2023). Additionally, our extensive experiments considered $7 \times 5 = 35$ combinations, indicating that our findings are possibly sufficiently robust to generalize the performance of *EvaluateXAI*. The generalizability of the granular-level evaluation metrics could present an external threat to our study. Nevertheless, we mitigated this threat by adopting two metrics from Pornprasit et al. (2021) and designing four additional granular-level metrics based on the inconsistencies and anomalies found in PyExplainer. Furthermore, we demonstrated the adaptability of these metrics in the case of another XAI technique called LIME to assess the reliability and consistency of the explanations generated by LIME. Therefore, we believe that these granular-level evaluation metrics are applicable to any rule-based XAI techniques as long as they adhere to the conventions of the generated rules (Almutairi et al., 2021).

8. Related work

There has been a growing interest in Explainable AI (XAI) in recent years, specifically focusing on making ML models interpretable and transparent to practitioners. At the same time, more research is being conducted to propose methods for formally assessing and contrasting various XAI methods. Therefore, obtaining practical knowledge to evaluate interpretability across multiple tools and establish a collaborative agreement among the community for future progress is crucial. This section briefly discusses existing works that aim to evaluate XAI methods.

In computer vision and image processing, many XAI methods such as saliency map (Simonyan et al., 2013), occlusion sensitivity (Zeiler and Fergus, 2014), integrated gradient (Sundararajan et al., 2017), Grad-CAM (Selvaraju et al., 2017), and SmoothGrad (Smilkov et al., 2017) have been proposed to explain the outcome of ML models. However, Ghorbani et al. (2019) and Kindermans et al. (2019) demonstrated that it is possible to attack the saliency map methods by manipulating the derived explanations. Samek et al. (2016) and Montavon et al. (2018) both suggested a method for evaluating the effectiveness of saliency map methods by introducing perturbations to the input. Adebayo et al. (2018) proposed a practical framework for assessing the types of explanations the saliency map method is capable and incapable of providing. By conducting thorough experiments, the authors demonstrated that certain saliency map techniques are not dependent on either the model or the process of generating data. The above research evaluates the effectiveness of different saliency map methods in generating explanations in computer vision targeting image data. Additionally, those studies did not consider classical ML models trained on tabular data when assessing the effectiveness of rule-based XAI techniques. However, this study aims to evaluate the reliability and consistency of rule-based XAI techniques in generating explanations for classical ML models trained on tabular data for software analytics tasks.

Doshi-Velez and Kim (2017) suggested three primary methods for evaluating interpretability: application-based, human-based, and

function-based. These methods vary from having human participation to no human involvement at all. Robnik-Šikonja and Bohanec (2018) proposed several properties (e.g., Accuracy, Fidelity, Consistency, Stability, Comprehensibility, Certainty, Importance, Novelty, and Representatives) to assess the generated explanation in a functional context. However, there is uncertainty about evaluating these properties, and how useful they are for XAI needs to be clarified. Miller (2019) conducted an extensive investigation to assess the explanation regarding constructiveness, selectivity, social aspects, emphasis on abnormality, truthfulness, consistency, and generality. Sokol and Flach (2020) systematically assessed XAI techniques across five significant dimensions: functional, operational, usability, safety, and validation. Their study involved comparing XAI methods to understand better how they differ. The above studies evaluate different model-agnostic XAI methods considering tabular data and ML models. ElShawi et al. (2021) evaluated three model-agnostic interpretability techniques (e.g., SHAP, LIME, and Anchors) on healthcare data using different evaluation metrics: identity, stability, separability, similarity, execution time, and bias detection. Hailemariam et al. (2020) conducted an empirical study on SHAP and LIME to gain insight into their operations on Artificial Neural Networks (ANNs) and Convolutional Neural Networks (CNNs). Their findings show that SHAP performed slightly better than LIME regarding Identity, Stability, and Separability. None of the studies described above consider classical ML models trained on tabular data for various software analytics tasks. Furthermore, our proposed novel framework and six evaluation metrics differ from the criteria proposed in the studies above. These criteria include accuracy, fidelity, consistency, stability, comprehensibility, certainty, importance, novelty, and representativeness to assess the generated explanation in a functional context. Finally, we demonstrate how our framework is utilized to evaluate the reliability and consistency of rule-based XAI techniques for classical ML models trained on tabular data through an extensive study of software analytics tasks.

Jiarpakdee et al. (2020) conducted an empirical study to evaluate three model-agnostic techniques for defect prediction models. Their experimental results suggest that contrastive explanations are both essential and valuable in comprehending the predictions of defect models. Hase and Bansal (2020) carried out human subject tests utilizing both tabular and text data to evaluate five XAI methods: LIME, Anchors, Decision Boundary, a Prototype Model, and a Composite approach that combines explanations from each technique. Gao et al. (2022) evaluated the effectiveness of local explanation techniques on defect prediction models trained on source code. Their study found that local explanation techniques are effective on token frequency-based models but not on deep learning-based models. Ledel and Herbold (2022) evaluated the quality of the explanations generated by both SHAP and LIME on the bug and non-bug issues. Their study assessed the quality of the explanations based on human expectations. Shin et al. (2023) conducted an empirical study on the stability of two XAI techniques, i.e., LIME-HPO and BreakDown for defect prediction models using two evaluation metrics, i.e., *hit_rate* and *rank_diff*. However, their experiment did not consider PyExplainer for stability check in explanation generation. Roy et al. (2022) showed how to resolve the disagreement problems among different XAI techniques for classical ML models trained on tabular data for JIT defect prediction tasks. However, they did not evaluate the reliability and consistency of the explanations generated by rule-based XAI techniques.

All the studies above evaluated different XAI methods, considering deep learning and ML models trained on image, text, and tabular data in various domains. However, to the best of our knowledge, studies have yet to consider evaluating the reliability and consistency of rule-based XAI techniques such as LIME and PyExplainer through the development of a framework and granular-level evaluation metrics. Furthermore, the evaluation criteria and proposed techniques significantly differ from our novel framework, and the inclusion of six evaluation metrics makes our work distinctive.

9. Conclusion

Machine learning (ML)-based approaches have enhanced various software analytics tasks in software maintenance and evolution. However, the lack of explainability behind the reasoning behind ML models can hinder practitioners from effectively applying ML-based approaches to improve software analytics tasks. To address this issue, novel rule-based XAI techniques, such as PyExplainer and LIME, have been utilized to elucidate the predictions of ML models. This allows practitioners to understand why a particular decision is made and take appropriate actions when needed. In this paper, we assess the ability of these techniques, particularly the state-of-the-art PyExplainer and LIME, to generate reliable and consistent explanations for ML models across various software analytics tasks. To do so, we propose a novel framework called *EvaluateXAI* and six evaluation metrics to assess the reliability and consistency of the explanations generated by rule-based XAI techniques for ML models in software analytics tasks. Our extensive investigations using *EvaluateXAI* conclude that none of the evaluation metrics reached 100%, indicating the unreliability of the explanations generated by XAI techniques. Moreover, PyExplainer and LIME could not offer consistent explanations in 86.11% and 77.78% of the experimental combinations. Finally, we show that *EvaluateXAI* could be adapted to any rule-based XAI techniques for evaluating their effectiveness in generating reliable and consistent explanations for the ML models in software analytics tasks. Our findings warrant further efforts to advance XAI research in software analytics.

CRediT authorship contribution statement

Md. Abdul Awal: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Chanchal K. Roy:** Writing – review & editing, Funding acquisition, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

We have shared the replication package on Zenodo.[10]

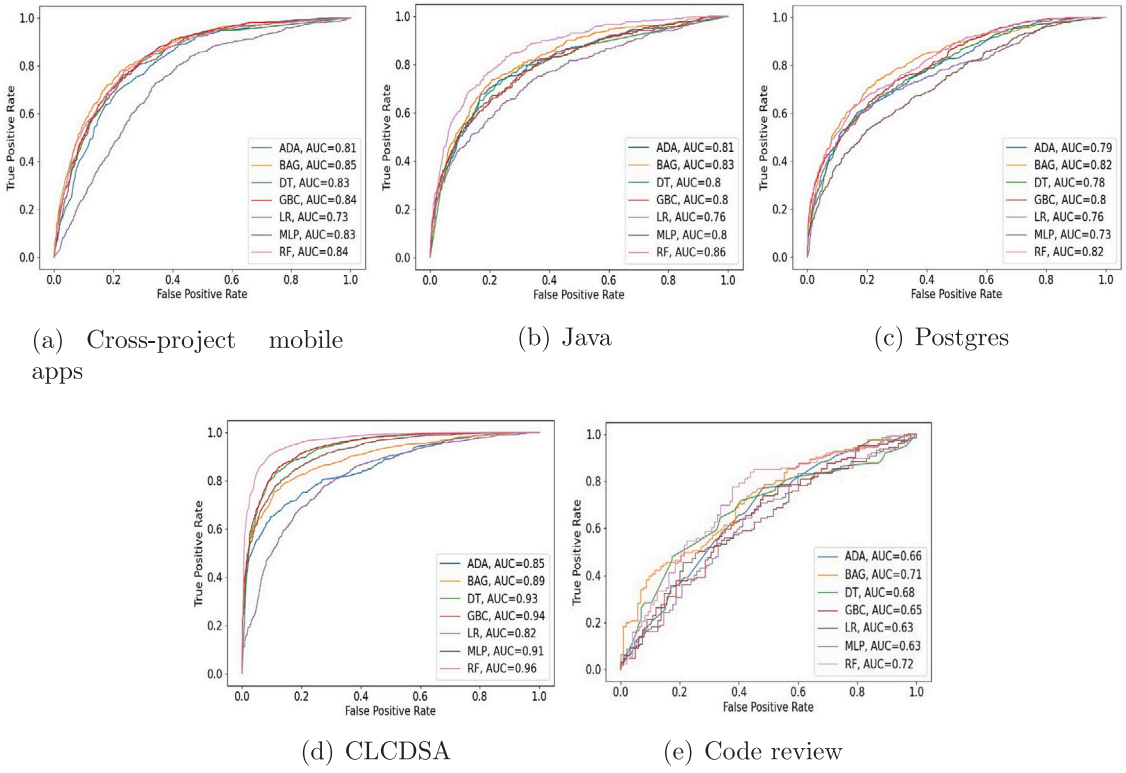
Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work, the author(s) used Grammarly[8] and ChatGPT 3.5[9] to find grammatical mistakes and improve sentence clarity/presentation. After using these tools/services, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the publication.

Acknowledgements

This research is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), and by the industry-stream NSERC CREATE in Software Analytics Research (SOAR).

[10] Replication-package
[8] https://app.grammarly.com/.
[9] https://chat.openai.com/.



(a) Cross-project mobile apps (b) Java (c) Postgres

(d) CLCDSA (e) Code review

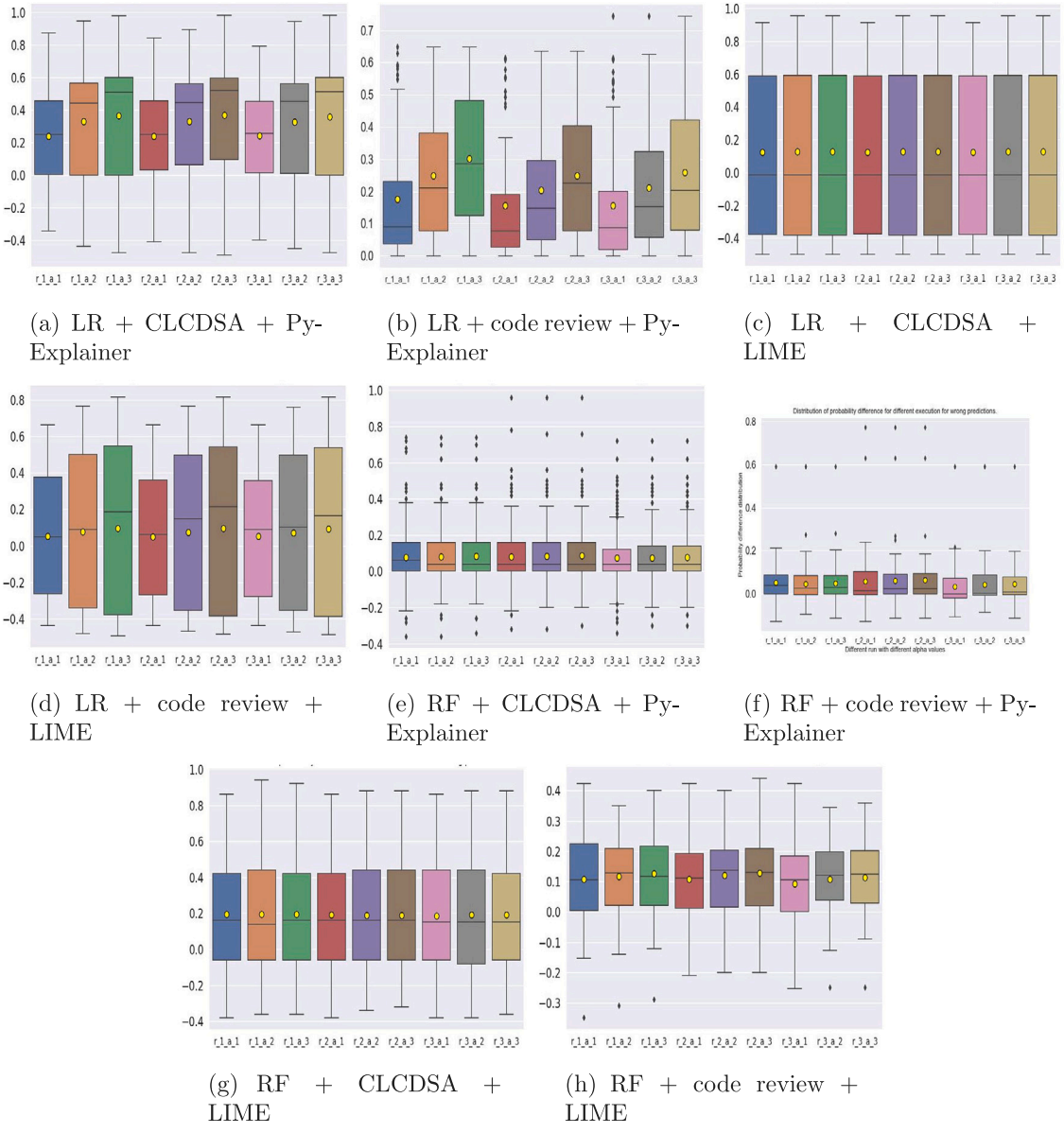
Fig. C.17. Performance of ML models across different datasets based on AUC-ROC curve. Figs. C.17(a), C.17(b), C.17(c), C.17(d), and C.17(e) depict the AUC-ROC curves for ML models trained on the cross-project mobile apps, Java project, Postgres, CLCDSA, and code review datasets, respectively.

Table C.8
Granular-level evaluation metric values for PyExplainer executed on the same instance across the entire Java project, Postgres, CLCDSA, and code review datasets, with varying α values.

ML model	Java project																							
	PCPD						PCPI						NCPD						NCPI					
	$\alpha=1$		$\alpha=2$		$\alpha=3$		$\alpha=1$		$\alpha=2$		$\alpha=3$		$\alpha=1$		$\alpha=2$		$\alpha=3$		$\alpha=1$		$\alpha=2$		$\alpha=3$	
	Cort.	Wrng.	Cort.	Wrng.	Cort.	Wrng.	Cort.	Wrng.	Cort.	Wrng.	Cort.	Wrng.	Cort.	Wrng.	Cort.	Wrng.	Cort.	Wrng.	Cort.	Wrng.	Cort.	Wrng.	Cort.	Wrng.
LR	3.97	61.69	3.73	55.72	3.51	53.9	0.45	10.78	0.5	8.96	0.53	8.46	3.11	2.32	3.54	2.32	3.63	2.49	29.44	14.59	27.34	12.94	27.03	12.77
DT	0.97	23.91	0.94	21.83	0.85	21.41	0.51	7.69	0.58	8.32	0.55	8.73	5.23	3.53	4.9	2.91	4.81	2.7	15.27	7.48	14.78	7.48	14.6	7.69
RF	2.63	35.42	2.76	40.28	2.74	38.19	0.66	11.11	0.41	5.21	0.34	5.21	6.67	12.85	5.89	10.42	5.71	10.42	32.18	30.56	32.14	31.6	32.02	31.25
MLP	3.35	46.39	3.31	47.19	3.45	47.19	1.65	24.3	1.59	21.89	1.42	21.49	8.32	5.22	7.63	4.42	7.15	4.62	23.81	13.86	22.91	13.05	22.84	12.45
ADA	1.79	22.39	1.51	20.88	1.42	20.08	0.14	2.9	0.16	3.47	0.19	3.28	2.07	0.97	2.68	0.97	2.54	0.97	29.41	12.55	27.01	11.97	27.39	11.97
BAG	3.55	43.96	3.71	40.11	3.46	38.64	0.76	11.9	0.44	11.54	0.46	11.17	1.45	0.73	1.29	0.92	1.34	0.92	34.21	15.75	33.54	14.65	33.43	14.65
GBC	2.56	39.91	2.16	36.36	2.19	35.03	0.92	15.96	0.92	15.3	0.78	15.3	11.05	10.42	11.24	8.43	11.1	8.43	23.42	14.86	21.51	13.75	21.47	13.53
Postgres																								
LR	7.47	47.33	7.55	48.4	7.55	47.33	0.78	5.12	0.61	3.62	0.55	4.26	5.62	7.46	5.62	7.46	5.62	7.25	18.06	21.54	15.63	18.98	14.96	19.19
DT	4.36	25.42	4.39	25.96	3.96	24.36	2.66	11.86	2.39	12.95	2.85	11.25	11.38	15.25	12.39	14.65	11.39	16.58	16.71	20.34	15.67	19.84	14.78	21.81
RF	2.36	15.69	2.14	14.29	2.14	14.82	4.29	10.56	4.39	8.85	4.81	10.92	14.62	18.59	12.39	17.89	15.62	20.85	17.35	20.39	17.21	21.52	18.54	22.63
MLP	7.41	44.44	7.41	66.67	6.17	55.56	0.0	22.22	0.0	0.0	0.0	11.11	6.17	0.0	4.94	0.0	4.94	0.0	13.58	22.2	11.11	22.22	9.88	22.22
ADA	3.51	21.45	3.51	22.14	3.6	22.28	1.04	3.96	1.04	3.73	1.04	3.5	15.51	22.84	16.2	23.31	16.94	24.71	8.92	12.35	8.35	11.89	7.1	10.26
BAG	6.36	48.15	6.94	51.85	6.36	55.56	1.16	14.81	0.58	7.41	0.58	3.7	2.31	3.7	1.73	3.7	1.16	3.7	19.08	22.22	19.08	22.22	18.5	22.22
GBC	5.28	27.93	4.12	23.73	3.87	22.03	1.97	12.47	2.66	10.17	2.66	10.17	9.88	13.47	10.9	15.25	10.41	16.95	15.56	24.69	14.04	20.34	13.08	18.64
CLCDSA																								
LR	24.59	40.89	24.37	41.03	24.22	40.89	1.96	3.51	1.83	2.97	1.88	2.83	2.16	7.15	2.16	7.69	2.2	7.69	9.89	34.68	9.47	33.87	9.41	33.87
DT	9.28	13.3	9.0	13.07	9.15	13.07	2.24	3.44	2.18	3.67	2.13	3.67	1.87	13.3	1.8	12.61	1.8	12.61	3.98	17.2	4.09	17.89	4.07	17.89
RF	23.85	29.86	24.22	29.51	24.36	28.82	1.23	9.38	0.92	9.03	0.85	8.33	2.48	11.11	2.27	9.38	2.2	9.03	10.86	29.86	10.92	30.9	11.0	31.25
MLP	17.98	19.96	16.49	19.58	15.87	18.63	9.25	10.46	10.44	10.65	11.0	11.6	4.35	22.24	4.39	21.29	4.56	19.96	9.04	34.41	8.69	34.41	8.47	35.55
ADA	15.23	13.92	13.06	11.91	13.31	12.2	1.72	3.01	8.87	13.06	8.83	13.06	1.12	5.31	2.34	10.47	2.34	10.47	6.33	22.96	4.98	18.94	4.98	18.51
BAG	24.07	24.52	24.05	24.17	23.98	24.17	1.45	1.74	1.2	2.26	1.33	2.09	1.88	10.26	1.52	9.22	1.38	8.52	10.93	39.65	11.11	41.22	11.22	41.74
GBC	17.44	16.03	17.44	16.03	17.44	16.03	6.98	10.77	6.98	10.77	6.98	10.77	5.12	22.97	5.12	22.97	5.12	22.97	8.47	27.27	8.47	27.27	8.47	27.27
Code review																								
LR	16.67	35.37	18.08	45.12	17.51	45.12	6.21	18.29	3.11	8.54	2.82	6.1	1.98	1.22	1.98	3.66	1.98	3.66	10.45	37.8	10.45	35.37	10.45	36.59
DT	5.43	13.89	7.07	13.89	6.52	13.89	5.16	8.33	4.35	6.94	4.35	6.94	1.09	4.17	1.09	4.17	1.09	4.17	5.98	22.22	5.98	22.22	5.98	20.83
RF	14.78	40.0	14.51	40.0	16.09	44.62	8.97	18.46	8.44	15.38	6.86	9.23	3.17	7.69	2.9	10.77	2.64	10.77	8.44	21.54	8.71	18.46	8.18	18.46
MLP	11.01	31.71	10.71	31.71	10.71	31.71	10.12	17.07	10.12	14.63	9.82	14.63	4.76	12.2	4.76	14.63	4.46	10.98	5.95	23.17	5.95	20.73	5.95	21.95
ADA	10.85	20.48	9.68	18.07	9.68	18.07	0.29	3.61	0.29	3.61	0.29	3.61	2.05	1.2	2.05	1.2	2.05	1.2	10.85	38.55	10.85	38.55	10.85	38.55
BAG	16.44	40.26	15.89	29.87	15.07	24.68	9.32	32.47	9.32	38.96	9.86	38.96	1.92	1.3	1.92	2.6	2.19	2.6	6.58	15.58	6.3	14.29	6.03	12.99
GBC	8.72	17.95	8.14	19.23	7.56	20.51	10.47	23.08	9.88	24.36	9.88	23.08	3.49	10.26	3.78	8.97	3.49	8.97	8.14	14.1	7.85	14.1	7.85	14.1

Appendix A. AUC–ROC curve

Fig. C.17 shows the AUC_ROC curves for the selected ML models trained on different datasets. From Fig. C.17(a), we observe that the LR model exhibits the lowest AUC value (0.73), whereas the BAG model shows the highest AUC value (0.85) when trained on the cross-project mobile apps dataset. The ML models trained on the CLCDSA dataset show the highest AUC values, ranging from 0.82 to 0.96. Additionally, Fig. C.17(e) demonstrates comparatively low AUC values for different ML models trained on the code review dataset. However, Fig. C.17 indicates that the AUC value varies from 0.63 to 0.96.



(a) LR + CLCDSA + PyExplainer (b) LR + code review + PyExplainer (c) LR + CLCDSA + LIME

(d) LR + code review + LIME (e) RF + CLCDSA + PyExplainer (f) RF + code review + PyExplainer

(g) RF + CLCDSA + LIME (h) RF + code review + LIME

Fig. C.18. Distribution of the probability difference between the original and the simulated instances. Figs. C.18(a) and C.18(b) show the probability difference distribution for the LR model trained on CLCDSA and code review datasets when we use PyExplainer for explanation generation. Figs. C.18(c) and C.18(d) show the probability difference distribution for the LR model trained on the CLCDSA and code review datasets when we use LIME for explanation generation. Figs. C.18(e) and C.18(f) show the probability difference distribution for the RF model trained on CLCDSA and code review datasets when we use PyExplainer for explanation generation. Figs. C.18(g) and C.18(g) show the probability difference distribution for the RF model trained on the CLCDSA and code review datasets when we use LIME for explanation generation. Here, r_i_a_j denotes the ith execution with $\alpha = j$.

Appendix B. *%Prob_diff metric*

Fig. C.18 shows the distribution of the probability difference between the original instances and the simulated instances for the ML models making wrong predictions. Similar to the results of the $\%Prob_diff$ metric for correct predictions, as described in Section 5.2, we observe similar patterns for the ML models making wrong predictions.

Appendix C. Granular-level evaluation metrics

Table C.8 presents the experimental results of four granular-level evaluation metrics (i.e., PCPD, PCPI, NCPD, and NCPI) for PyExplainer applied to ML models, both for correct and wrong predictions. In an ideal scenario, we expect an increase in each evaluation metric for each simulated instance. However, from Table C.8, it is evident that we observe very low values for PCPD, PCPI, NCPD, and NCPI metrics in the case of correct predictions, with only a few exceptions. For example, when considering the LR model trained on the CLCDSA dataset, we find a PCPD value of 24.59 and an NCPI value of 34.68. In contrast, we consistently observe very low PCPI and NCPD values for correct predictions.

For wrong predictions, we observe relatively high *PCPD* values for all the ML models. However, this is not the case for the *PCPI* metric, with only a few exceptions. For example, the *PCPI* values for the BAG and GBC models are 32.47 and 23.08, respectively. Similarly, the *NCPD* values are low for the simulated instances, with only a few exceptions. Finally, we observe comparatively high *NCPI* values for all the models trained on the CLCDSA dataset. In contrast, for the cross-project mobile apps dataset, the *NCPI* values are low. In summary, when considering both correct and wrong predictions, the four granular-level evaluation metrics consistently yield low values,

Table C.9
Granular-level evaluation metric values for LIME executed on the same instance across the entire Java project, Postgres, CLCDSA, and code review datasets, with varying α values.

ML model	Java project																							
	PCPD						PCPI						NCPI						NCPD					
	$\alpha=1$		$\alpha=2$		$\alpha=3$		$\alpha=1$		$\alpha=2$		$\alpha=3$		$\alpha=1$		$\alpha=2$		$\alpha=3$		$\alpha=1$		$\alpha=2$		$\alpha=3$	
	Cort.	Wrng.	Cort.	Wrng.	Cort.	Wrng.	Cort.	Wrng.	Cort.	Wrng.	Cort.	Wrng.	Cort.	Wrng.	Cort.	Wrng.	Cort.	Wrng.	Cort.	Wrng.	Cort.	Wrng.	Cort.	Wrng.
LR	10.64	0.31	10.64	0.79	10.64	0.94	8.83	66.46	9.53	47.4	10.01	45.04	0.0	19.37	0.0	19.37	0.0	19.37	29.59	11.02	45.93	18.9	45.98	19.21
DT	11.41	34.17	11.46	33.59	11.06	28.16	2.9	19.61	2.85	20.39	3.2	23.5	30.22	21.17	39.27	21.36	40.17	20.78	16.26	6.8	15.66	6.8	14.91	7.77
RF	8.33	33.55	8.33	42.76	8.28	45.07	1.99	23.68	1.63	9.21	1.45	4.93	73.67	42.43	77.1	42.43	79.23	42.43	16.88	22.04	17.29	26.32	16.83	26.64
MLP	11.84	8.25	11.89	10.36	11.84	11.32	9.43	44.53	10.39	23.03	10.49	19.58	16.11	25.72	17.21	25.91	9.43	44.53	32.61	14.97	25.24	22.84	32.61	14.97
ADA	12.14	0.0	12.14	0.0	12.14	0.0	2.73	51.58	7.03	44.32	7.08	44.59	0.86	18.62	0.05	18.62	0.0	18.44	29.89	0.93	24.18	9.87	24.28	9.87
BAG	13.59	8.74	13.59	16376	13.59	20.4	6.31	54.28	4.53	31.88	4.53	29.14	53.18	17.3	58.07	17.3	60.31	17.3	11.2	5.1	10.94	8.93	10.08	8.93
GBC	11.03	24.39	11.03	22.15	11.03	25.41	7.62	45.33	7.27	37.6	7.57	36.79	5.64	30.28	6.38	30.28	7.62	29.47	45.5	15.24	37.93	15.24	37.93	16.26
Postgres																								
LR	19.91	11.83	19.91	15.56	19.91	11.83	16.58	25.52	14.6	23.44	14.79	22.61	16.58	25.52	0.45	43.36	0.06	43.36	0.19	43.15	68.31	39.21	68.63	43.36
DT	18.05	41.39	17.54	36.97	17.98	39.5	8.35	10.5	8.8	11.55	8.48	11.55	13.97	27.31	11.1	29.62	8.67	26.05	28.25	16.39	28.44	14.5	29.09	14.92
RF	18.4	27.45	18.4	26.97	18.4	27.21	2.03	11.22	1.6	10.74	1.23	8.59	68.62	55.13	67.32	55.13	66.52	54.89	37.54	22.2	36.86	26.25	35.94	24.82
MLP	21.74	8.7	21.74	7.88	21.74	9.85	18.33	38.26	17.98	37.11	18.33	37.44	5.71	38.59	8.01	38.59	9.69	38.59	61.6	31.2	61.05	32.84	57.77	33.33
ADA	19.72	11.47	19.79	15.58	19.85	18.83	7.77	23.38	7.65	21.0	7.65	23.59	1.07	47.4	0.76	47.62	0.57	47.62	49.37	15.58	56.32	15.8	56.01	15.8
BAG	21.59	36.59	21.59	42.57	21.59	45.45	7.28	13.53	5.02	7354	4.27	7.32	16.82	37.25	20.21	37.25	23.23	37.03	45.51	9.31	37.85	8.2	34.84	7.98
GBC	18.67	24.35	18.8	21.98	18.73	24.35	11.27	22.41	10.57	19.83	10.76	20.26	41.01	45.69	36.84	46.77	36.33	46.12	49.11	21.55	50.38	22.2	51.56	20.91
CLCDSA																								
LR	65.66	0.0	65.66	0.54	65.66	0.27	25.81	35.44	32.3	33.29	33.53	33.42	0.0	50.74	0.0	50.6	0.0	50.74	29.24	33.96	27.84	36.51	27.33	36.38
DT	59.11	21.23	59.41	21.46	59.56	22.6	7.17	11.19	7.1	10.05	7.13	9.36	15.85	46.58	12.77	44.52	13.18	44.98	8.11	30.14	8.34	30.59	8.6	30.82
RF	52.25	35.07	62.25	31.94	62.25	32.99	6.93	10.07	5.37	8.68	5.12	8.33	28.87	48.96	28.4	49.31	28.79	50.35	9.42	28.12	8.5	27.78	7.75	26.39
MLP	62.94	14.1	63.02	10.34	62.98	9.96	24.25	15.79	29.93	17.67	33.01	17.86	23.39	58.27	17.56	62.59	15.96	63.72	25.22	45.49	26.66	45.3	27.56	44.92
ADA	61.29	0.97	61.29	1.53	61.29	1.95	16.58	21.14	16.62	21.7	16.58	21.84	0.13	67.32	0.59	67.18	0.5	67.18	21.2	11.96	23.55	21.7	23.64	22.11
BAG	60.48	16.61	60.29	15.22	60.29	16.09	14.82	6.57	12.72	7.09	13.0	7.61	24.49	69.72	29.65	70.42	30.36	69.9	13.99	54.84	11.06	33.39	10.54	30.45
GBC	60.93	20.67	61.04	20.19	61.16	20.19	19.81	18.05	19.78	17.58	19.7	17.81	25.86	44.18	18.92	44.18	18.25	43.23	18.32	45.37	17.84	46.56	17.72	46.56
Code review																								
LR	62.41	2.41	61.7	7.23	61.7	7.23	48.23	34.94	59.57	31.33	62.41	33.73	4.26	44.58	4.26	44.58	3.55	44.58	24.82	42.17	21.99	38.35	19.86	36.14
DT	46.67	21.62	49.33	21.62	49.33	21.62	8.0	8.11	9.33	16.22	9.33	17.57	13.33	45.95	12.67	48.65	10.67	47.3	6.67	12.16	7.33	12.16	7.33	13.51
RF	63.29	27.27	61.39	28.79	62.66	34.85	28.48	28.79	29.11	22.73	27.85	18.18	15.82	36.36	25.95	36.36	27.22	36.36	25.95	21.21	22.15	21.21	17.72	22.73
MLP	64.18	17.78	67.91	23.33	67.91	20.0	41.79	38.89	45.52	33.33	38.06	32.22	15.67	40.0	9.7	38.89	8.21	38.89	26.87	33.33	23.88	33.33	25.37	34.44
ADA	40.0	16.67	32.86	16367	36.43	17.86	32.86	15.48	33.57	14.29	34.29	20.24	4.29	44.05	4.29	45.24	4.29	44.05	4.29	28.57	4.29	28.57	16.43	28.57
BAG	73.79	22.78	71.72	15.19	73.1	15.19	35.86	48.1	55.86	51.9	42.76	51.9	12.41	16.46	17.24	16.46	17.93	6.33	11.72	8.86	7.59	6.33	7.59	6.33
GBC	62.68	10.98	61.27	10.98	62.68	12.2	48.59	47.56	26.76	51.22	23.94	51.22	5.63	39.02	11.27	36.59	6.34	39.02	21.13	8.54	21.83	7.32	22.54	7.32

raising concerns about the reliability of the explanations generated by PyExplainer.

We observe similar results for LIME, as shown in Table C.9. However, the granular-level evaluation metric values do not consistently increase with the rise of α values for LIME.

References

Abdou, A., Darwish, N., 2024. Severity classification of software code smells using machine learning techniques: A comparative study. J. Softw.: Evol. Process 36 (1), e2454.

Adebayo, J., Gilmer, J., Muelly, M., Goodfellow, I., Hardt, M., Kim, B., 2018. Sanity checks for saliency maps. Adv. Neural Inf. Process. Syst. 31.

Ali, M., Mazhar, T., Arif, Y., Al-Otaibi, S., Ghadi, Y.Y., Shahzad, T., Khan, M.A., Hamam, H., 2024. Software defect prediction using an intelligent ensemble-based model. IEEE Access.

Ali, M., Mazhar, T., Shahzad, T., Ghadi, Y.Y., Mohsin, S.M., Akber, S.M.A., Ali, M., 2023. Analysis of feature selection methods in software defect prediction models. IEEE Access.

Almutairi, M., Stahl, F., Bramer, M., 2021. Reg-rules: an explainable rule-based ensemble learner for classification. IEEE Access 9, 52015–52035.

Alon, U., Brody, S., Levy, O., Yahav, E., 2018. code2seq: Generating sequences from structured representations of code. arXiv preprint arXiv:1808.01400.

Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2019. code2vec: Learning distributed representations of code. Proc. ACM Program. Lang. 3 (POPL), 1–29.

Bacchelli, A., Bird, C., 2013. Expectations, outcomes, and challenges of modern code review. In: 2013 35th International Conference on Software Engineering. ICSE, IEEE, pp. 712–721.

Begum, M., Shuvo, M.H., Ashraf, I., Al Mamun, A., Uddin, J., Samad, M.A., 2023. Software defects identification: Results using machine learning and explainable artificial intelligence techniques. IEEE Access.

Bergstra, J., Bengio, Y., 2012. Random search for hyper-parameter optimization. J. Mach. Learn. Res. 13 (2).

Brughmans, D., Melis, L., Martens, D., 2024. Disagreement amongst counterfactual explanations: how transparency can be misleading. TOP 1–34.

Catolino, G., Di Nucci, D., Ferrucci, F., 2019. Cross-project just-in-time bug prediction for mobile apps: An empirical assessment. In: 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems. MOBILESoft, IEEE, pp. 99–110.

Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P., 2002. SMOTE: synthetic minority over-sampling technique. J. Artificial Intelligence Res. 16, 321–357.

Choi, J., Manikandan, S., Ryu, D., Baik, J., 2022. An empirical analysis on just-in-time defect prediction models for self-driving software systems. In: International Conference on Web Engineering. Springer, pp. 34–45.

Cliff, N., 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. Psychol. Bull. 114 (3), 494.

D'Ambros, M., Lanza, M., Robbes, R., 2010. An extensive comparison of bug prediction approaches. In: 2010 7th IEEE Working Conference on Mining Software Repositories. MSR 2010, IEEE, pp. 31–41.

Dewangan, S., Rao, R.S., Chowdhuri, S.R., Gupta, M., 2023. Severity classification of code smells using machine-learning methods. SN Comput. Sci. 4 (5), 564.

Doshi-Velez, F., Kim, B., 2017. Towards a rigorous science of interpretable machine learning. arXiv preprint arXiv:1702.08608.

ElShawi, R., Sherif, Y., Al-Mallah, M., Sakr, S., 2021. Interpretability in healthcare: A comparative study of local machine learning interpretability techniques. Comput. Intell. 37 (4), 1633–1650.

Feng, S., Suo, W., Wu, Y., Zou, D., Liu, Y., Jin, H., 2024. Machine learning is all you need: A simple token-based approach for effective code clone detection.

Fernandes, P., Allamanis, M., Brockschmidt, M., 2018. Structured neural summarization. arXiv preprint arXiv:1811.01824.

Gao, Y., Zhu, Y., Yu, Q., 2022. Evaluating the effectiveness of local explanation methods on source code-based defect prediction models. In: Proceedings of the 19th International Conference on Mining Software Repositories. pp. 640–645.

Ghorbani, A., Abid, A., Zou, J., 2019. Interpretation of neural networks is fragile. In: Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 33, pp. 3681–3688.

Ghotra, B., McIntosh, S., Hassan, A.E., 2015. Revisiting the impact of classification techniques on the performance of defect prediction models. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. Vol. 1, IEEE, pp. 789–800.

Gosiewska, A., Biecek, P., 2019. IBreakDown: Uncertainty of model explanations for non-additive predictive models. arXiv preprint arXiv:1903.11420.

Hailemariam, Y., Yazdinejad, A., Parizi, R.M., Srivastava, G., Dehghantanha, A., 2020. An empirical evaluation of AI deep explainable tools. In: 2020 IEEE Globecom Workshops. GC Wkshps, IEEE, pp. 1–6.

Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S., 2011. A systematic literature review on fault prediction performance in software engineering. IEEE Trans. Softw. Eng. 38 (6), 1276–1304.

Hanley, J.A., McNeil, B.J., 1982. The meaning and use of the area under a receiver operating characteristic (ROC) curve. Radiology 143 (1), 29–36.

Hase, P., Bansal, M., 2020. Evaluating explainable AI: Which algorithmic explanations help users predict model behavior? arXiv preprint arXiv:2005.01831.

Hu, X., Li, G., Xia, X., Lo, D., Jin, Z., 2018. Deep code comment generation. In: Proceedings of the 26th Conference on Program Comprehension. pp. 200–210.

Huang, Z., Yu, H., Fan, G., Shao, Z., Li, M., Liang, Y., 2024. Aligning XAI explanations with software developers' expectations: A case study with code smell prioritization. Expert Syst. Appl. 238, 121640.

Jiarpakdee, J., Tantithamthavorn, C.K., Dam, H.K., Grundy, J., 2020. An empirical study of model-agnostic techniques for defect prediction models. IEEE Trans. Softw. Eng. 48 (1), 166–185.

Jiarpakdee, J., Tantithamthavorn, C.K., Grundy, J., 2021. Practitioners' perceptions of the goals and visual explanations of defect prediction models. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories. MSR, IEEE, pp. 432–443.

Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N., 2012. A large-scale empirical study of just-in-time quality assurance. IEEE Trans. Softw. Eng. 39 (6), 757–773.

Kindermans, P.-J., Hooker, S., Adebayo, J., Alber, M., Schütt, K.T., Dähne, S., Erhan, D., Kim, B., 2019. The (un) reliability of saliency methods. In: Explainable AI: Interpreting, Explaining and Visualizing Deep Learning. Springer, pp. 267–280.

Kumar, R., Chaturvedi, A., 2023. Software bug prediction using reward-based weighted majority voting ensemble technique. IEEE Trans. Reliab..

Ledel, B., Herbold, S., 2022. Studying the explanations for the automated prediction of bug and non-bug issues using LIME and SHAP. arXiv preprint arXiv:2209.07623.

Lessmann, S., Baesens, B., Mues, C., Pietsch, S., 2008. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. IEEE Trans. Softw. Eng. 34 (4), 485–496.

Liashchynskyi, P., Liashchynskyi, P., 2019. Grid search, random search, genetic algorithm: a big comparison for NAS. arXiv preprint arXiv:1912.06059.

Lundberg, S.M., Lee, S.-I., 2017. A unified approach to interpreting model predictions. Adv. Neural Inf. Process. Syst. 30.

Lyu, Y., Rajbahadur, G.K., Lin, D., Chen, B., Jiang, Z.M., 2021. Towards a consistent interpretation of aiops models. ACM Trans. Softw. Eng. Methodol. (TOSEM) 31 (1), 1–38.

Malhotra, R., Meena, S., 2023. Empirical validation of feature selection techniques for cross-project defect prediction. Int. J. Syst. Assur. Eng. Manag. 1–13.

McIntosh, S., Kamei, Y., 2018. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. In: Proceedings of the 40th International Conference on Software Engineering. pp. 560–560.

Mehta, S., Patnaik, K.S., 2021. Improved prediction of software defects using ensemble machine learning techniques. Neural Comput. Appl. 33 (16), 10551–10562.

Miller, T., 2019. Explanation in artificial intelligence: Insights from the social sciences. Artificial Intelligence 267, 1–38.

Montavon, G., Samek, W., Müller, K.-R., 2018. Methods for interpreting and understanding deep neural networks. Digit. Signal Process. 73, 1–15.

Nadim, M., Mondal, D., Roy, C.K., 2022. Leveraging structural properties of source code graphs for just-in-time bug prediction. Autom. Softw. Eng. 29 (1), 27.

Nadim, M., Roy, B., 2023. Utilizing source code syntax patterns to detect bug inducing commits using machine learning models. Softw. Qual. J. 31 (3), 775–807.

Nafi, K.W., Kar, T.S., Roy, B., Roy, C.K., Schneider, K.A., 2019. Clcdsa: cross language code clone detection using syntactical features and api documentation. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 1026–1037.

Nguyen, G., Biswas, S., Rajan, H., 2023. Fix fairness, don't ruin accuracy: Performance aware fairness repair using AutoML. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 502–514.

Pornprasit, C., Tantithamthavorn, C.K., 2021. JITLine: A simpler, better, faster, finer-grained just-in-time defect prediction. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories. MSR, IEEE, pp. 369–379.

Pornprasit, C., Tantithamthavorn, C., Jiarpakdee, J., Fu, M., Thongtanunam, P., 2021. Pyexplainer: Explaining the predictions of just-in-time defect models. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 407–418.

Rahman, F., Devanbu, P., 2013. How, and why, process metrics are better. In: 2013 35th International Conference on Software Engineering. ICSE, IEEE, pp. 432–441.

Rahman, M.M., Roy, C.K., Kula, R.G., 2017. Predicting usefulness of code review comments using textual features and developer experience. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories. MSR, IEEE, pp. 215–226.

Rajapaksha, D., Tantithamthavorn, C., Jiarpakdee, J., Bergmeir, C., Grundy, J., Buntine, W., 2021. SQAPlanner: Generating data-informed software quality improvement plans. IEEE Trans. Softw. Eng. 48 (8), 2814–2835.

Rajbahadur, G.K., Wang, S., Oliva, G.A., Kamei, Y., Hassan, A.E., 2021. The impact of feature importance methods on the interpretation of defect classifiers. IEEE Trans. Softw. Eng. 48 (7), 2245–2261.

Rao, R.S., Dewangan, S., Mishra, A., Gupta, M., 2023. A study of dealing class imbalance problem with machine learning methods for code smell severity detection using PCA-based feature selection technique. Sci. Rep. 13 (1), 16245.

Ribeiro, M.T., Singh, S., Guestrin, C., 2016. "Why should i trust you?" Explaining the predictions of any classifier. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 1135–1144.

Ribeiro, M.T., Singh, G., Sameer, C., 2018. Anchors: High-precision model-agnostic explanations. In: Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 32.

Robnik-Šikonja, M., Bohanec, M., 2018. Perturbation-based explanations of prediction models. In: Human and Machine Learning: Visible, Explainable, Trustworthy and Transparent. Springer, pp. 159–175.

Roy, C.K., Cordy, J.R., 2007. A Survey on Software Clone Detection Research. TR 541, Queen's School of computing, pp. 64–68.

Roy, S., 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: 2022 IEEE International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 444–448.

Saha, D., Chatterjee, S., 2022. Optimized decision tree-based early phase software dependability analysis in uncertain environment. In: 2022 International Interdisciplinary Conference on Mathematics, Engineering and Science. MESIICON, IEEE, pp. 1–6.

Samek, W., Binder, A., Montavon, G., Lapuschkin, S., Müller, K.-R., 2016. Evaluating the visualization of what a deep neural network has learned. IEEE Trans. Neural Netw. Learn. Syst. 28 (11), 2660–2673.

Selvaraju, R.R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., Batra, D., 2017. Grad-cam: Visual explanations from deep networks via gradient-based localization. In: Proceedings of the IEEE International Conference on Computer Vision. pp. 618–626.

Sharma, T., Jatain, A., Bhaskar, S., Pabreja, K., 2023. Ensemble machine learning paradigms in software defect prediction. Procedia Comput. Sci. 218, 199–209.

Shihab, E., 2012. An Exploration of Challenges Limiting Pragmatic Software Defect Prediction. Queen's University, Canada.

Shin, J., Aleithan, R., Nam, J., Wang, J., Shiri Harzevili, N., Wang, S., 2023. An empirical study on the stability of explainable software defect prediction. Available at SSRN 4328070.

Shrimankar, R., Kuanr, M., Piri, J., Panda, N., 2022. Software defect prediction: A comparative analysis of machine learning techniques. In: 2022 International Conference on Machine Learning, Computer Systems and Security. MLCSS, IEEE, pp. 38–47.

Simonyan, K., Vedaldi, A., Zisserman, A., 2013. Deep inside convolutional networks: Visualising image classification models and saliency maps. arXiv preprint arXiv:1312.6034.

Singh, M., Chhabra, J.K., 2024. Improved software fault prediction using new code metrics and machine learning algorithms. J. Comput. Lang. 78, 101253.

Smilkov, D., Thorat, N., Kim, B., Viégas, F., Wattenberg, M., 2017. Smoothgrad: removing noise by adding noise. arXiv preprint arXiv:1706.03825.

Sokol, K., Flach, P., 2020. Explainability fact sheets: A framework for systematic assessment of explainable approaches. In: Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency. pp. 56–67.

Sundararajan, M., Taly, A., Yan, Q., 2017. Axiomatic attribution for deep networks. In: International Conference on Machine Learning. PMLR, pp. 3319–3328.

Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K., 2016a. Automated parameter optimization of classification techniques for defect prediction models. In: Proceedings of the 38th International Conference on Software Engineering. pp. 321–332.

Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K., 2016b. An empirical comparison of model validation techniques for defect prediction models. IEEE Trans. Softw. Eng. 43 (1), 1–18.

Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K., 2018. The impact of automated parameter optimization on defect prediction models. IEEE Trans. Softw. Eng. 45 (7), 683–711.

Wahono, R.S., 2015. A systematic literature review of software defect prediction. J. Softw. Eng. 1 (1), 1–16.

Wang, Z., Zhou, Y., Qiu, M., Haque, I., Brown, L., He, Y., Wang, J., Lo, D., Zhang, W., 2023. Towards fair machine learning software: Understanding and addressing model bias through counterfactual thinking. arXiv preprint arXiv:2302.08018.

Wilcoxon, F., 1992. Individual comparisons by ranking methods. In: Breakthroughs in Statistics: Methodology and Distribution. Springer, pp. 196–202.

Wu, J., Chen, X.-Y., Zhang, H., Xiong, L.-D., Lei, H., Deng, S.-H., 2019. Hyperparameter optimization for machine learning models based on Bayesian optimization. J. Electron. Sci. Technol. 17 (1), 26–40.

Xiao, Y., Zhang, J., Liu, Y., Mousavi, M., Liu, S., Xue, D., 2004. MirrorFair: Fixing fairness bugs in machine learning software via counterfactual predictions. In: Proceedings of the ACM International Conference on the Foundations of Software Engineering. FSE 2024, ACM.

Yang, L., Shami, A., 2020. On hyperparameter optimization of machine learning algorithms: Theory and practice. Neurocomputing 415, 295–316.

Yatish, S., Jiarpakdee, J., Thongtanunam, P., Tantithamthavorn, C., 2019. Mining software defects: Should we consider affected releases? In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE, pp. 654–665.

Zeiler, M.D., Fergus, R., 2014. Visualizing and understanding convolutional networks. In: Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I 13. Springer, pp. 818–833.

Zhang, W., Li, Y., Wen, M., He, R., 2023. Comparative study of ensemble learning methods in just-in-time software defect prediction. In: 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion. QRS-C, IEEE, pp. 83–92.