An Empirical Study of Model-Agnostic Techniques for Defect Prediction Models

Jirayus Jiarpakdee, Chakkrit (Kla) Tantithamthavorn, Hoa Khanh Dam, and John Grundy

Abstract—Software analytics have empowered software organisations to support a wide range of improved decision-making and policy-making. However, such predictions made by software analytics to date have not been explained and justified. Specifically, current defect prediction models still fail to explain why models make such a prediction and fail to uphold the privacy laws in terms of the requirement to explain any decision made by an algorithm. In this paper, we empirically evaluate three model-agnostic techniques, i.e., two state-of-the-art Local Interpretability Model-agnostic Explanations technique (LIME) and BreakDown techniques, and our improvement of LIME with Hyper Parameter Optimisation (LIME-HPO). Through a case study of 32 highly-curated defect datasets that span across 9 open-source software systems, we conclude that (1) model-agnostic techniques are needed to explain individual predictions of defect models; (2) instance explanations generated by model-agnostic techniques are mostly overlapping (but not exactly the same) with the global explanation of defect models and reliable when they are re-generated; (3) model-agnostic techniques take less than a minute to generate instance explanations; and (4) more than half of the practitioners perceive that the contrastive explanations are necessary and useful to understand the predictions of defect models. Since the implementation of the studied model-agnostic techniques is available in both Python and R, we recommend model-agnostic techniques be used in the future.

Index Terms—Explainable Software Analytics, Software Quality Assurance, Defect Prediction Models, Model-Agnostic Techniques.

1 Introduction

Software analytics have empowered many software organisations to improve software quality and accelerate software development processes. Such analytics are essential to guide operational decisions and establish quality improvement plans. For example, Microsoft leverages the advances of Artificial Intelligence and Machine Learning (AI/ML) capabilities to predict software defects [69]. In addition, prior studies have proposed techniques to estimate Agile story points [15], estimate software development costs [72], recommend a reviewer [105], recommend a developer to fix a software defect [4].

Despite the recent advances in software analytics, such decision-making based on AI/ML-based systems needs to be better justified and uphold privacy laws. Article 22 of the European Union's General Data Protection Regulation (GDPR) [82] states that the use of data in decision-making that affects an individual or group *requires an explanation for any decision made by an algorithm*. Recent work raises a concern about a lack of explainability of software analytics in software engineering [16]. Practitioners also share similar concerns that analytical models in software engineering must be explainable and actionable [16], [52]. For example, Google [52] argue that defect models should be more actionable to help software engineers debug their programs. Miller [66] also argue that human aspects should be taken into consideration when developing AI/ML-based systems. Thus, *Explainable Software Analytics*—a suite of AI/ML techniques that produce accurate predictions, while being able to explain such predictions—is vitally needed.

Thus, researchers often generated ***global explanations***, which refers to an explanation that summarises the predictions of black-box AI/ML learning algorithms. Such global explanations can be generated by model-specific explanation techniques of machine learning techniques (e.g., an ANOVA analysis for logistic regression and a variable importance analysis for random forests). Prior studies used these model interpretation techniques to understand the relationship between studied variables and an outcome. For example, Menzies *et al.* [63] investigated the impact of code attributes on software quality. Bird *et al.* [10] studied the correlation between human factors and software quality. McIntosh *et al.* [58] and Thongtanunam *et al.* [104] investigated the relationship between code review practices and post-release defects.

However, such global explanations cannot justify each individual prediction of the models on testing or unseen data. For example, an analytical model for software defects may generate a predicted probability of 0.9 for a testing instance, suggesting that a software module will be defective in the future. Such the predicted probability does not provide any explanation from the models as to why the machine learning techniques make that prediction. A lack of explanation of the predictions generated by such analytical models could lead to serious errors in decision- and policy-making, hindering the adoption of software analytics in industrial practices [16].

Recently, ***model-agnostic techniques*** have been proposed to explain the prediction of black-box AI/ML algorithms

- *J. Jiarpakdee, C. Tantithamthavorn, and J. Grundy are with the Faculty of Information Technology, Monash University, Australia.*
E-mail: {jirayus.jiarpakdee, chakkrit, john.grundy}@monash.edu.
- *H. K. Dam is with the School of Computing and Information Technology, Faculty of Engineering and Information Sciences, University of Wollongong, Australia.*
E-mail: hoa@uow.edu.au

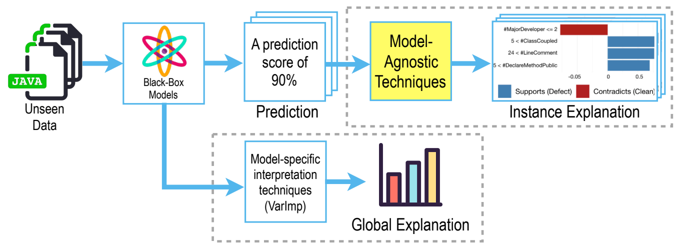


Figure 1: An illustration of model-agnostic techniques. Model-agnostic techniques are used to explain the predictions of unseen data, while the global explanation is derived from the trained models from training data. In other words, one model can have only one global explanation, but should have multiple instance explanations.

by identifying the contribution that each metric has on the prediction of an instance according to a trained model. Yet, such techniques have never been formally introduced and empirically evaluated in the context of software engineering. To address this challenge, this paper is the first to focus on generating ***instance explanations*** which refers to an explanation of the prediction of defect prediction models (see Figure 1), by answering a central question: *Should model-agnostic techniques be used to explain the predictions of defect models?*

In this paper, we empirically evaluate three model-agnostic techniques, i.e., two state-of-the-art Local Interpretability Model-agnostic Explanations (LIME) technique [85] and BreakDown [29], [90] technique, and our improvement of LIME with Hyper Parameter Optimisation (LIME-HPO) using a differential evolution algorithm. *LIME* constructs a local regression model surrounding the instance to be explained to identify the contribution of each metric to the prediction of the instance to be explained. On the other hand, *BreakDown* decomposes the prediction of the instance to be explained into parts that can be attributed to each studied metric as to their contribution to the prediction. We generate explanations of the predictions of defect models that are constructed from six classification techniques (i.e., logistic regression (LR), random forests (RF), C5.0, averaged neural network (AVNNet), gradient boosting machines (GBM), and extreme Gradient Boosting Trees (xGBTree)). Through a case study of 32 publicly-available defect datasets of 9 large-scale open-source software systems, we address the following six research questions:

(RQ1) Does LIME with Hyper Parameter Optimisation (LIME-HPO) outperform default LIME in terms of the goodness-of-fit of the local regression models?
LIME-HPO always outperform default LIME in terms of the goodness-of-fit (R^2) of the local regression models with an average improvement of 8% for all of the studied classification techniques.

(RQ2) Can model-agnostic techniques explain the predictions of defect models?
Model-agnostic techniques can explain the predictions of defect models. Given the same defect models, different predictions have different instance explanations. For example, one metric that appears at the top rank for one instance could appear at the rank 21 for another instance. Such high variation indicates that global explanations do not imply instance explanations (and vice versa), highlighting the need for model-agnostic techniques for explaining the predictions of defect models.

(RQ3) Do instance explanations generated by model-agnostic techniques overlap with the global explanation of defect models?
Despite the variation of the ranking of the top-10 important metrics for instance explanations (see RQ2), their overall ranking is mostly overlapping (but not exactly the same) with that of the global explanations. We find that, at the median, 7, 10, and 9 of the top-10 summarised important metrics of for instance explanations are overlapping with the top-10 global important metrics for LIME, LIME-HPO, and BreakDown, respectively.

(RQ4) Do model-agnostic techniques generate the same instance explanation when they are re-generated for the same instance?
Regardless of the studied classification techniques, LIME-HPO and BreakDown consistently generate the same instance explanation for the same instance. On the other hand, LIME generates different instance explanations when re-generating instance explanations of the same instance with a median rank difference of 7 ranks.

(RQ5) What is the computational time of model-agnostic techniques for explaining the predictions of defect models?
The computational time of LIME-HPO, BreakDown, and LIME techniques is less than a minute to generate instance explanations for all of the studied classification techniques, suggesting that all of the studied model-agnostic techniques are practical to be used in real-world deployments in the future.

(RQ6) How do practitioners perceive the contrastive explanations generated by model-agnostic techniques?
65% of the practitioners agree that model-agnostic techniques can generate a contrastive explanation within an object over time (Time-contrast) (e.g., why was file A not classified as defective in version 1.2, but was subsequently classified as defective in version 1.3?). In particular, 55% and 65% of the participants perceive that such Time-contrast explanations are necessary and useful, respectively.

Since the implementation of the studied model-agnostic techniques is readily available in both Python (LIME [83] and pyBreakDown [9]) and R (LIME [75] and BreakDown [8], [29]), we recommend model-agnostic techniques be used to explain the predictions of defect models.

Novelty & Contributions. The key contributions of this paper are as follows:

(1) An introduction to the explainability in software engineering from a perspective of psychological science (Section 2).
(2) An introduction to the state-of-the-art model-agnostic techniques (i.e., LIME and BreakDown) for generating instance explanations (Section 3).

(3) An improvement of LIME using Hyper Parameter Optimisation (LIME-HPO) with a differential evolution algorithm (Section 3.2.2) and empirical evidence (RQ1).
(4) An empirical study of the need (RQ2), trustworthiness (RQ3), reliability (RQ4), computational time (RQ5), and software practitioners' perception (RQ6) of model-agnostic techniques (Section 4).

Paper Organisation. Section 2 introduces the explainability in software engineering. Section 3 introduces model-agnostic techniques for generating instance explanation. Section 4 presents the design of our case study, while Section 5 discusses the results with respect to six research questions. Section 6 discusses the key differences between model-agnostic techniques and a simple tree-based technique. Section 7 discusses related work in order to situate the contributions of our paper with respect to explainable software analytics and analytical models for software defects. Section 8 discusses the threats to the validity of our study. Finally, Section 9 draws conclusions.

2 Explainability in Software Engineering

Software engineering is by nature a collaborative social practice. Collaboration among different stakeholders (e.g., users, developers, and managers) is essential in modern software engineering. As a part of the collaboration, individuals are often expected to explain decisions made throughout software development processes to develop appropriate trust and enable effective communication. Since tool support in software development processes is an integral part of this collaborative process, similar expectations are also applied. Such tools should not only provide insights or generate predictions for recommendation, but also be able to explain such insights and recommendations.

Recent automated and advanced software development tools heavily rely on Artificial Intelligence and Machine Learning (AI/ML) capabilities to predict software defects, estimate development effort, and recommend API choices. However, such AI/ML algorithms are often "black-box", which makes it hard for practitioners to understand how the models arrive at a decision. A lack of explainability of the black-box algorithms leads to a lack of trust in the predictions or recommendations produced by such algorithms.

2.1 A Theory of Explainability

According to philosophy, social science, and psychology theories, a common definition of *explainability or interpretability* is *the degree to which a human can understand the reasons behind a decision or an action* [67]. The explainability of AI/ML algorithms can be achieved by (1) making the entire decision-making process transparent and comprehensible and (2) explicitly providing an explanation for each decision [54] (since an explanation is not likely applicable to all decisions [50]). In order to make the entire decision-making process transparent, prior software engineering studies often use white-box AI/ML algorithms (e.g., decision trees and decision rules). While such white-box AI/ML algorithms can increase the explainability of the decision-making process, their predictions may not be as accurate as complex black-box AI/ML techniques (e.g., random forest, extreme gradient boosting trees). Hence, research has emerged to explore how to explain decisions made by complex, black-box models and how explanations are presented in a form that would be easily understood (and hence, accepted) by humans.

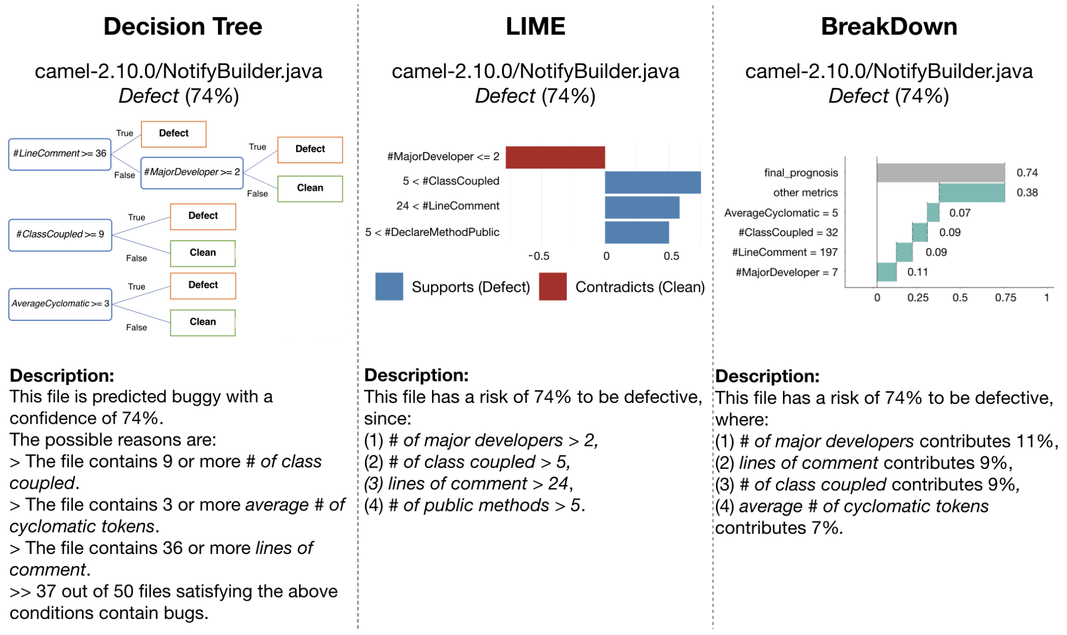
2.2 A Theory of Explanations

According to a philosophical and psychological theory of explanations, Salmon *et al.* [87] argue that explanations can be presented as a causal chain of causes that lead to the decision. Causal chains can be classified into five categories [35]: temporal, coincidental, unfolding, opportunity chains and pre-emptive. Each type of causal chain is thus associated with an explanation type. However, identifying the complete causal chain of causes is challenging, since most AI/ML techniques produce only correlations instead of causations.

In contrast, Miller [67] argue that explanations can be presented as answers to why-questions. Similarly, Lipton *et al.* [53] also share a similar view of explanations as being *contrastive*. There are three components of why-questions [6]: (1) the event to be explained, also called the *explanandum* (e.g., file A is defective); (2) a set of similar events that are similar to the explanandum but did not occur (e.g., file A is clean); and (3) a request for information that can distinguish the occurrence of the explanandum from the non-occurrence of the other similar events (e.g., a large number of changes made to file A). Below, we describe four types of why-questions:

- **Plain-fact** is the properties of the object. *"Why does object a have property P?"*
  Example: Why is file A defective?
- **Property-contrast** is the differences in the Properties within an object. *"Why does object a have property P, rather than property P'?"*
  Example: Why is file A defective rather than clean?
- **Object-contrast** is the differences between two Objects. *"Why does object a have property P, while object b has property P'?"*
  Example: Why is file A defective, while file B is clean?
- **Time-contrast** is the differences within an object over Time. *"Why does object a have property P at time t, but property P' at time t'?"*
  Example: Why was file A not classified as defective in version 1.2, but was subsequently classified as defective in version 1.3?

Answers to the P-contrast, O-contrast and T-contrast why-questions form an explanation. *Contrastive explanations focus on only the differences on* ***Properties within an object*** *(Property-contrast), between* ***two Objects*** *(Object-contrast), and* ***within an object over Time*** *(Time-contrast) [107].* Answering a plain fact question is generally more difficult than generating answers to the contrastive questions [53]. For example, we could answer the Property-contrast question (e.g., "Why is file A classified as being defective instead of being clean?") by citing that there are a substantial number of defect-fixing commits that involve with the file. Information about the size, complexity, owner of the file,



Description:
This file is predicted buggy with a confidence of 74%.
The possible reasons are:
> The file contains 9 or more *# of class coupled*.
> The file contains 3 or more *average # of cyclomatic tokens*.
> The file contains 36 or more *lines of comment*.
>> 37 out of 50 files satisfying the above conditions contain bugs.

Description:
This file has a risk of 74% to be defective, since:
(1) *# of major developers > 2*,
(2) *# of class coupled > 5*,
(3) *lines of comment > 24*,
(4) *# of public methods > 5*.

Description:
This file has a risk of 74% to be defective, where:
(1) *# of major developers* contributes 11%,
(2) *lines of comment* contributes 9%,
(3) *# of class coupled* contributes 9%,
(4) *average # of cyclomatic tokens* contributes 7%.

Figure 2: An example of visual explanations for a decision tree model, and two model-agnostic techniques (i.e., LIME and BreakDown).

and so on are not required to answer this question. On the other hand, explaining why file A is defective in a non-contrastive manner would require us to use all causes. In addition, humans tend to be cognitively attached to digest contrastive explanations [67]. Thus, contrastive explanations may be more valuable and more intuitive to humans. These important factors from both social and computational perspectives should be considered when providing explainable models or tool support for software engineering.

Explanation is not only a *product*, as discussed above, but also a *process* [56]. In fact, generating explanations is a *cognitive process* which essentially involves four cognitive systems: (1) attention, (2) long-term memory, (3) working memory, and (4) metacognition [36], [50]. Recent work [67] further recognised the importance of considering explanation as being not only a cognitive process but also a *social process*, in which an explainer communicates knowledge to an explainee. Using this view, explanations should be considered as part of a conversation between the explainer and explainee. The theories, models, and processes of how humans explain decisions to one another are important to the work on explainable software analytics and the development of explainable tool support for software engineering in general.

3 Techniques for Generating Explanations

Prior studies often leverage *white-box AI/ML techniques*, such as decision trees [22] and decision rules [77]. The transparency of such white-box AI/ML techniques allows us to meaningfully understand the magnitude of the contribution of each metric on the learned outcomes by directly inspecting the model components. For example, the coefficients of each metric in a regression model, paths in a decision tree, or rules of a decision rule model. Figure 2 provides an example visual explanation of a white-box model (e.g., decision trees) and model-agnostic techniques.

In contrast, white-box AI/ML techniques are often less accurate than complex black-box AI/ML techniques and often generate generic explanations (e.g., one decision node may cover 100 instances). Recently, model-agnostic techniques (e.g., LIME [85] and BreakDown [29]) have been used to explain the predictions of any black-box AI/ML models at an instance level. Guiding by a theory of explanations in Section 2.2, this paper focuses on answering the why-questions at an instance level for any predictions made by a black-box model. Below, we present the formal definition of a black-box model, a global explanation, and an instance explanation.

Definition 1.1: A black-box model. A black-box model is a function $b : \mathcal{X}^{(m)} \rightarrow \mathcal{Y}$ where $\mathcal{X}^{(m)}$ is the feature space with m corresponding to the studied metrics (i.e., independent variables), and $\mathcal{Y}$ is the outcome space (e.g., defective or clean). Typically, training data D_{train} is used for training a black-box model $b(D_{\text{train}})$, and testing data D_{test} is used for evaluating the accuracy of a black-box model b.

There are a plethora of techniques for generating explanations from these black-box models where each technique has different definitions and targets of explanations. Below, we provide formal definitions and introduce techniques for explaining a black-box model and techniques for explaining an individual prediction made by a black-box model.

3.1 Explaining a black-box model

A **global explanation** (or a model explanation) refers to an explanation of the decisions of a black-box model which summarises the logic of a classification technique based on the conditional relationship between the independent variables (software metrics) and the dependent variable (an outcome) with respect to **a whole training data**. Below, we present the formal definition of global explanation and model-specific explanation techniques as follows:

Definition 1.2: global explanation. A global explanation $e_m = \varepsilon(b, D_{\text{train}})$, if a global explanation e_m is generated from an explanation function ε which summarises the logic of a black-box model b that is learned from a training dataset D_{train}.

Model-specific explanation techniques focus on explaining the entire decision-making process of a specific black-box model. For example, an ANOVA analysis for logistic regression and a variable importance analysis for random forests. However, such global explanations are often derived from black-box models that are constructed from training data, which are not specific enough to explain an individual prediction.

3.2 Explaining an individual prediction

An **instance explanation** (or a local explanation) refers to an explanation of the decision of a black-box model with respect to **a testing instance**. Below, we present the formal definition of instance explanation and model-agnostic techniques as follows:

Definition 1.3: instance explanation An instance explanation $e_o = \varepsilon(b, x)$, if an instance explanation e_o is generated from an explanation function ε for a prediction $b(x)$ of an instance $x \in D_{\text{test}}$ in a testing dataset D_{test}.

Model-agnostic techniques (i.e., local explanation techniques) focus on explaining an individual prediction by diagnosing a black-box model. Unlike model-specific explanation techniques discussed above, the great advantage of model-agnostic techniques is their flexibility. Such model-agnostic techniques can (1) interpret any classification techniques (e.g., regression, random forest, and neural networks); (2) are not limited to a certain form of explanations (e.g., feature importance or rules); and (3) are able to process any input data (e.g., features, words, and images [84]). According to the literature survey of model-agnostic techniques [30], we selected and discussed two state-of-the-art model-agnostic techniques (i.e., LIME and BreakDown). We also propose LIME-HPO—an improvement of LIME using Hyper Parameter Optimisation based on the differential evolution technique.

3.2.1 LIME: Explaining a local model that mimics the global model

LIME (i.e., Local Interpretable Model-agnostic Explanations) [85] is a model-agnostic technique that mimics the behaviour of the black-box model to generate the explanations of the predictions of the black-box model. Given a black-box model and an instance to explain, LIME performs 4 key steps to generate an instance explanation as follows:

- First, LIME randomly generates instances surrounding the instance of interest (*cf.* Line 3).
- Second, LIME uses the black-box model to generate predictions of the generated random instances (*cf.* Line 4).
- Third, LIME constructs a local regression model using the generated random instances and their generated predictions from the black-box model (*cf.* Line 7).
- Finally, the coefficients of the regression model indicate the contribution of each metric on the prediction of the instance of interest according to the black-box model (*cf.* Line 8).

Algorithm 1: LIME's algorithm [85]

```
Input  : $b$ is a black-box model,
         $x$ is an instance to explain,
         $n$ is a number of randomly generated
         instances, and
         $k$ is a length of explanation
Output: $e$ is a set of contributions of metrics on the
         prediction of the instance $x$.
1 $D = \varnothing$
2 for $i$ in $\{1, ..., n\}$ do
3     $d_i = \text{sample\_around}(x)$
4     $y'_i = \text{predict}(b, d_i)$
5     $D = D \cup \langle d_i, y'_i \rangle$
6 end
7 $l = \text{K} - \text{Lasso}(D, k)$
8 $e = \text{get\_coefficients}(l)$
9 return $e$
```

Figure 2 shows a visual explanation of LIME (the middle column). The blue bars indicate the supporting (+) scores of metrics towards the prediction as defective, while the red bars indicate the contradicting (-) scores of metrics towards the prediction as defective. In this example, the NotifyBuilder.java of the release 2.10.0 of the Apache Camel project is predicted (74%) as defective due to a supporting score of 0.75 for a condition of {#ClassCoupled $>$ 5}, a supporting score of 0.6 for a condition of {#LineComment $>$ 24}, and a supporting score of 0.5 for a condition of {#DeclareMethodPublic $>$ 5}. On the other hand, the remaining probability of 26% of not defective could be explained by a contradict score of 0.77 for a condition of {#MajorDeveloper $\leq$ 2}.

3.2.2 LIME-HPO: Optimising the hyperparameter settings of LIME

Since LIME involves parameter settings (e.g., the number of randomly generated instances that LIME uses to construct local regression models), we propose to optimise the parameter settings of the LIME algorithm using a Hyper Parameter Optimisation (*LIME-HPO*). We use a differential evolution algorithm [91] to find an optimal value of the number of randomly generated instances where the objective function is to maximise the goodness-of-fit (R^2) of the local regression models of LIME. We use the implementation of the differential evolution technique as provided by the `DEoptim` function of the `DEoptim` R package [68] using the following parameter settings:

- The lower boundary of 100 and the upper boundary of 10000 for the population of the number of randomly generated instances used by LIME.
- The number of population (NP) of 10.
- The crossover probability (cr) of 0.5.
- The differential weighting factor from interval (f) of 0.8.
- The number of procedure iterations of 10 for generating population.

Given a black-box model and an instance to explain, LIME-HPO performs 6 key steps to generate an instance explanation as follows:

- First, LIME-HPO randomly generates a set of candidate of size NP within the population boundary for the number of randomly generated instances surrounding the instance of interest.
- Second, for each ca ndidate, LIME-HPO uses the black-box model to generate predictions of the generated random instances and constructs local regression models (Steps 2 and 3 of LIME).
- Third, LIME-HPO find the best candidate of this generation for the number of randomly generated instances that produces the local regression model with the highest goodness-of-fit.
- Fouth, LIME-HPO generates a set of candidate for the next generation using the set of candidate of the current generation with the crossover probability (cr) and the differential weighting factor from interval (f).
- Fifth, LIME-HPO reiterates the procedure until reaching the number of procedure iterations.
- Finally, LIME-HPO derives the coefficients of the local regression model that yields the highest goodness-of-fit across all generations as the contribution of each metric on the prediction of the instance of interest according to the black-box model.

Since LIME involves random perturbation (Line 3 in Algorithm 1), different samplings may produce different instance explanations. To mitigate the randomisation of LIME when re-generating instance explanations, Ribeiro[1] suggests to set a random seed prior to applying LIME. Thus, our LIME-HPO follows this suggestion by setting a random seed.

3.2.3 BreakDown: Explaining a global model

BreakDown [29], [90] is a model-agnostic technique that uses the greedy strategy to sequentially measure contributions of metrics towards the expected prediction. Given a black-box model b, an instance to explain x, and training data used to construct the model D_{train}, BreakDown performs 5 key steps to generate an instance explanation as follows:

- First, BreakDown generates the predictions of all instances in the training data and computes the average estimation of such predictions (*cf.* Lines 3-4). In the first iteration, BreakDown uses the original training data as the syntactic training data for calculation (*cf.* Line 5).
- Second, BreakDown sequentially substitutes the values of each metric in the syntactic training data with the value of such metric appeared at the instance of interest (*cf.* Lines 7-9).
- Third, BreakDown generates the predictions of the substituted training data, and identify the metric that produces the greatest absolute difference between the expected predictions of the syntactic training data and the substituted training data (*cf.* Lines 10-13).
- Fourth, BreakDown allocates such greatest difference in expected predictions made by the metric as its contribution (*cf.* Line 14).
- Finally, BreakDown considers the set of expected predictions of the substituted training data with the greatest difference in expected predictions as the new set of expected predictions (*cf.* Lines 15-16) and reiterates to calculate the contributions of the metrics in which their contributions are not allocated (*cf.* Line 17).

1. https://github.com/marcotcr/lime/issues/119#issuecomment-344743006

Algorithm 2: BreakDown's algorithm [90]

```
Input : $b$ is a black-box model,
        $x$ is an instance to explain, and
        $D_{train}$ is a set of training instances used
        to construct the black-box model.
Output: $c$ is a set of contributions of metrics on the
        prediction of the instance $x$.
1  $M$ = independent variables in $x$
2  $M_{initial} = \varnothing$
3  $Y'_{train} = \text{predict}(b, D_{train})$
4  $EY_{initial} = \text{average}(Y'_{train})$
5  $D_{initial} = D_{train}$
6  for $i$ in $\{1, ..., \text{size}(M)\}$ do
7      for $j$ in $\{M - M_{initial}\}$ do
8          $D_{substituted} = D_{initial}$
9          $D_{substituted}[, j] = x[j]$
10         $Y'_{substituted,j} = \text{predict}(b, D_{substituted})$
11         $dy_j = \text{abs}(\text{average}(Y'_{substituted,j}) - EY_{initial})$
12     end
13     $dy_{m_{max}} = \text{find\_max}(dy_{\{M-M_{initial}\}})$
14     $c_{m_{max}} = dy_{m_{max}}$
15     $EY_{initial} = \text{average}(Y'_{substituted,m_{max}})$
16     $D_{initial}[, m_{max}] = x[m_{max}]$
17     $M_{initial} = M_{initial} \cup m_{max}$
18 end
19 return $c$
```

Figure 2 shows a visual explanation of BreakDown (the right column). The light blue bars indicate the supporting (+) probability of metrics towards the prediction as defective, while the light brown bars indicate the contradicting (-) probability of metrics towards the prediction as defective. In this example, the NotifyBuilder.java of the release 2.10.0 of the Apache Camel project is predicted (74%) as defective due to a supporting probability of 0.11 for #MajorDeveloper, a supporting probability of 0.09 for #LineComment, a supporting probability of 0.9 for #ClassCoupled, and a supporting probability of 0.07 for AverageCyclomatic.

4 Experimental Design

In this section, we discuss our criteria for selecting the studied datasets; and the design of the case study that we perform to address our six research questions. Figure 3 provides an overview of the design of our case study.

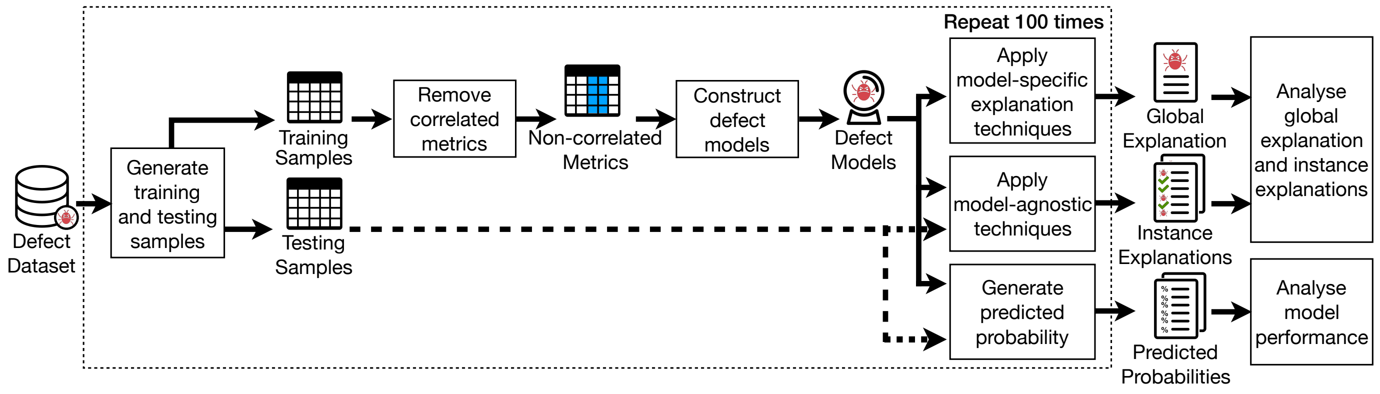

Repeat 100 times
Defect Dataset
Generate training and testing samples
Training Samples
Testing Samples
Remove correlated metrics
Non-correlated Metrics
Construct defect models
Defect Models
Apply model-specific explanation techniques
Apply model-agnostic techniques
Generate predicted probability
Global Explanation
Instance Explanations
Predicted Probabilities
Analyse global explanation and instance explanations
Analyse model performance

Figure 3: An overview diagram of the design of our case study.

4.1 Studied Datasets

Recently, Yatish *et al.* [112] showed that (1) some issue reports that are addressed within 6 months after a release do not realistically affect a studied release (false positive), while (2) some issue reports that realistically affect the studied release are addressed later than 6 months after the release (false negative). Thus, the approximation of post-release window periods (e.g., 6 months) that were popularly-used in many defect datasets may introduce biases to the construct to the validity of our results.

Thus, we select a corpus of publicly-available defect datasets provided by Yatish *et al.* [112] where the ground-truths were labelled based on the affected releases. The datasets consist of 32 releases that span 9 open-source software systems. Each dataset has 65 software metrics along 3 dimensions, i.e., 54 code metrics, 5 process metrics, and 6 human metrics. Table 1 shows a statistical summary of the studied datasets.

Code metrics describe the relationship between properties extracted from source code and software quality. These code metrics are extracted using the `Understand` tool from SciTools along 3 dimensions, i.e., complexity (e.g., McCabe Cyclomatic), volume (e.g., lines of code), and object-oriented (e.g., a coupling between object classes). Among these code metrics, some properties are extracted at the method level, and thus three aggregation schemes (i.e., minimum, average, and maximum) are used to summarise these metrics to the file level.

Process metrics describe the relationship between development process and software quality. For each instance, there are (1) the number of commits, (2) the number of added lines of code, (3) the number of deleted lines of code, (4) the number of active developers, and (5) the number of distinct developers. Similar to Rahman *et al.* [80], the number of added and deleted lines of code of each instance is normalized by the total number of added and deleted lines, respectively.

Human factors describe the relationship between the ownership of instances and software quality [10], [79], [104]. Ownership metrics are based on the traditional code ownership heuristics of Bird *et al.* [10], for each instance, the ownership of each developer is measured using the proportion of the code changes made by the developer on the total code changes. There are two granularities of code changes, i.e., lines of code level (LINE), and commit level (COMMIT), while there are two levels of ownership of an instance for developers, as recommended by Bird *et al.* [10]. That is, developers with low code ownership (i.e., less than 5% code contribution on an instance) are considered as minor authors. On the other hand, developers with high code ownership (i.e., more than 5% code contribution on an instance) are considered as major authors. Ownership metrics consist of (1) the number of the owner (i.e., the developer with the highest code contribution on an instance), (2) the number of the minor authors, and (3) the number of major authors with respect to the two granularities of code changes.

4.2 Generate Training and Testing Samples

To generate training and testing samples, we opt to use an out-of-sample bootstrap validation technique [19], [23], [34], [95], [101], which leverages aspects of statistical inference. We use the out-of-sample bootstrap validation technique (1) to ensure that the generated training samples are representative to the original dataset and (2) to ensure that the produced estimates are the least bias and most reliable [101]. We first generate a bootstrap sample of sizes N with replacement from the studied defect datasets. The generated sample is also of size N. We construct defect models using the bootstrap samples, while we interpret the samples that do not appear in the generated bootstrap samples at the instance-level. On average, 36.8% of the original dataset will not appear in the bootstrap samples, since the samples are drawn with replacement [19]. We repeat the out-of-sample bootstrap process for 100 times and report their average calculations.

4.3 Remove Correlated Metrics

Prior studies raise concerns that collinearity (i.e., correlated metrics) often impacts the global explanation of defect models [39], [41], [96], [100]. For example, a defect model that is constructed with correlated metrics could produce different global explanations when reordering the model formula of regression models. Recently, the bagging technique for random forest is proposed to mitigate collinearity (i.e., different trees are constructed with different subset of

Table 1: A statistical summary of the studied systems.

Name	Description	#DefectReports	No. of files	Defective Rate	KLOC	Studied Releases
ActiveMQ	Messaging and Integration Patterns server	3,157	1,884-3,420	6%-15%	142-299	5.0.0, 5.1.0, 5.2.0, 5.3.0, 5.8.0
Camel	Enterprise Integration Framework	2,312	1,515-8,846	2%-18%	75-383	1.4.0, 2.9.0, 2.10.0, 2.11.0
Derby	Relational Database	3,731	1,963-2,705	14%-33%	412-533	10.2.1.6, 10.3.1.4, 10.5.1.1
Groovy	Java-syntax-compatible OOP for JAVA	3,943	757-884	3%-8%	74-90	1.5.7, 1.6.0.Beta_1, 1.6.0.Beta_2
HBase	Distributed Scalable Data Store	5,360	1,059-1,834	20%-26%	246-534	0.94.0, 0.95.0, 0.95.2
Hive	Data Warehouse System for Hadoop	3,306	1,416-2,662	8%-19%	287-563	0.9.0, 0.10.0, 0.12.0
JRuby	Ruby Programming Lang for JVM	5,475	731-1,614	5%-18%	105-238	1.1, 1.4, 1.5, 1.7
Lucene	Text Search Engine Library	2,316	8,05-2,806	3%-24%	101-342	2.3.0, 2.9.0, 3.0.0, 3.1.0
Wicket	Web Application Framework	3,327	1,672-2,578	4%-7%	109-165	1.3.0.beta1, 1.3.0.beta2, 1.5.3

metrics), prior studies found that some trees are still constructed with correlated metrics [39], [92]. Since LIME builds a local regression model which is known to be sensitive to collinearity [39], [96], it is necessary to remove correlated metrics to mitigate such impact prior to constructing defect models.

Prior studies introduce many techniques to remove irrelevant metrics and correlated metrics (e.g., Correlation-based Feature Selection (CFS), Information Gain (IG), and stepwise regression) [5], [12], [18], [20], [44], [63], [73], [89]. Jiarpakdee *et al.* [42] found that such feature selection techniques cannot mitigate correlated metrics (e.g., CFS produces a subset of metrics that are highly correlated with the dependent variable while having the least correlation among themselves. Yet, some of the independent variables are still highly correlated), suggesting that correlation analyses must be applied. However, such correlation analyses often involve manual selection by a domain expert. To ensure the scalability of our framework, we apply the AutoSpearman technique on the training samples. AutoSpearman automatically selects one metric of a group of the highest correlated metrics that shares the least correlation with other metrics that are not in that group [42]. We use the implementation of the AutoSpearman technique as provided by the `AutoSpearman` function of the `Rnalytica` R package [41]. We observe that AutoSpearman mitigates correlated metrics and selects only 22-27 of 65 metrics. In other words, as high as 38-43 metrics share strong correlations among themselves, which are then removed by AutoSpearman.

4.4 Construct Defect Models

Shihab [88] and Hall *et al.* [32] show that logistic regression and random forests are the two most-popularly-used classification techniques in the literature of software defect prediction, since they are explainable and have built-in model explanation techniques (i.e., the ANOVA analysis for the regression technique and the variable importance analysis for the random forests technique). Recent studies [25], [99], [102] also demonstrate that automated parameter optimisation can improve the performance and stability of defect models. Using the findings of prior studies to guide our selection, we choose (1) the commonly-used classification techniques that have built-in model-specific explanation techniques (i.e., logistic regression and random forests) and (2) the top-5 classification techniques when performing automated parameter optimisation [99], [102] (i.e., random forests, C5.0, AVNNet, GBM, and xGBTree).

We use the implementation of Logistic Regression as provided by the `glm` function of the `stats` R package [103]. We use the implementation of automated parameter optimisation of Random Forests, C5.0, AVNNet, GBM, and xGBTree as provided by the `train` function of the `caret` R package [49] with the options of `rf`, `C5.0`, `avNNet`, `gbm`, and `xgbTree` for the method parameter, respectively. We neither re-balance nor normalize our training samples to preserve its original characteristics and to avoid any concept drift for the explanations of defect models [97].

4.5 Apply Model-specific Explanation Techniques

We apply model-specific explanation techniques to generate global explanations—what factors are associated with software quality. We use the ANOVA Type-II analysis for logistic regression and the scaled Permutation Importance analysis for random forests, as suggested by our recent work [39]. We use the *usage* (i.e., the percentage of training instances that satisfy all of the terminal nodes after the split which are associated with the metric) of metrics to the generate global explanation of C5.0 [78]. We use the combinations of the absolute values of the weights derived across hidden layers in neural networks to generate the global explanation of AVNNet [26]. We use the relative influence of metrics derived from boosted trees to generate the global explanation of GBM and xGBTree [24], [71].

We use the implementation of the ANOVA Type-II analysis as provided by the `Anova` function of the `car` R package [21]. We use the implementation of the scaled Permutation Importance analysis as provided by the `importance` function of the `randomForest` R package [11]. We use the implementation provided by the `varImp` function of the `caret` R package [49] to generate global explanation of C5.0, GBM, and xGBTree.

4.6 Apply Model-agnostic Techniques

We apply model-agnostic techniques to generate instance explanations of the predictions of defect models. We use three model-agnostic techniques, i.e., two state-of-the-art LIME (Local Interpretable Model-Agnostic Explanations) and BreakDown, and our improvement of LIME with Hyper Parameter Optimisation based on the differential evolution technique (LIME-HPO) The technical descriptions are presented in Section 3 We use the implementation of LIME as provided by the `lime` and `explain` functions of the `lime` R package [75]. We use the implementation of the differential evolution technique as provided by the `DEoptim` function of the `DEoptim` R package [68]. We use the implementation of BreakDown as provided by the `broken` function of the `breakDown` R package [8].

4.7 Generate Predicted Probability

We use defect models to generate predicted probabilities (i.e., defect-proneness) of testing instances. The predicted probabilities range from 0 (not defective) to 1 (likely to be defective). We use the classification threshold of 0.5 to map predicted probabilities to a binary decision of defect and clean. Predicted probabilities of above 0.5 indicate defect, otherwise clean.

4.8 Analyse Global Explanation and Instance Explanations

We analyse global explanation and instance explanations to address RQs 1, 2, 3, and 4. We also conduct a survey study of 20 practitioners to evaluation instance explanations generated by the studied model-agnostic techniques in RQ6. The motivation, approach, and results for each RQ are explained in detail in Section 5.

4.9 Analyse Model Performance

To ensure that the generated global explanation and instance explanations are derived from accurate defect models, we evaluate the model performance of the studied classification techniques.

First, we use the Area Under the receiver operator characteristic Curve (AUC) to measure the discriminatory power of our models, as suggested by recent research [27], [51], [80]. The axes of the curve of the AUC measure are the coverage of non-defective modules (true negative rate) for the x-axis and the coverage of defective modules (true positive rate) for the y-axis. The AUC measure is a threshold-independent performance measure that evaluates the ability of models in discriminating between defective and clean instances. The values of AUC range between 0 (worst), 0.5 (no better than random guessing), and 1 (best) [33].

Second, we use the Initial False Alarm (IFA) measure to identify the number of false alarms encountered before the first defective module [37]. To calculate the IFA measure, we sort the modules in descending order of their risk predicted by a model. Then, the IFA measure is computed as k, where k is the number of non-defective modules that are predicted as defective by a model before the first defective module. The values of IFA range from 1 (best) to n, where n is the number of all modules.

Third, we use the P_{opt} measure [43], [60], [111] to measure the effort-aware predictive performance of defect models. The P_{opt} measure is defined by the area Δ_{opt} between the effort-based cumulative lift charts of the optimal model and a defect model. The axes of the effort-based cumulative lift charts are the proportion of effort for the x-axis and the coverage of defective modules for the y-axis. For the optimal model, all modules are sorted in descending order of the actual defect density (i.e., the proportion of the number of defects and the lines of code of each module). On the other hand, for a defect model, all modules are sorted in decreasing order of the predicted probabilities of each module. The P_{opt} measure is computed as $P_{opt} = 1 - \Delta_{opt}$. The values of P_{opt} range between 0 (worst), 0.5 (no better than random guessing), and 1 (best).

Preliminary Analysis. Figure 4 shows the distributions of the model performance of all of the studied classification

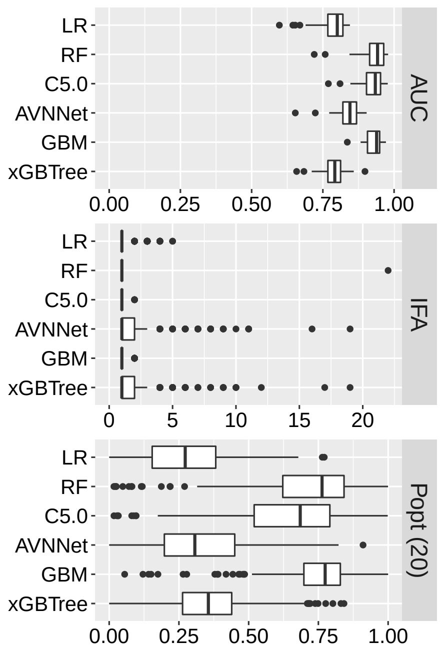


Figure 4: The distributions of model performance for all studied defect datasets of each classification technique.

techniques for all of the studied defect datasets. **Our studied classification techniques achieve a median AUC of 0.79-0.94, a median IFA of 1, and a median Popt20 of 0.27-0.77,** indicating that our studied classification techniques are highly accurate.

5 Case Study Results

We present the results of our case study with respect to our five research questions.

(RQ1) Does LIME with Hyper Parameter Optimisation (LIME-HPO) outperform default LIME in terms of the goodness-of-fit of the local regression models?

Motivation. Since LIME generates instance explanations from local regression models that are constructed using the randomly generated instances around the neighbours of the instance to be explained, we use the goodness-of-fit (R^2) of the local regression models as a proxy for measuring the performance of LIME when generating instance explanations. Prior studies [25], [99], [102] have shown that hyper parameter optimisation can be used to improve the performance of defect models. Yet, little is known about whether hyper parameter optimisation can improve the goodness-of-fit of the LIME algorithm when generating explanations for the predictions of defect models.

Approach. To address RQ1, we analyse the goodness-of-fit (R^2) of LIME and LIME-HPO when generating explanations for the predictions of defect models for all of the studied datasets. For each bootstrap sample of each defect dataset, we use the overview diagram (see Figure 3) to generate instance explanations of the testing instances. Then, we compute the goodness-of-fit of the local regression models that are used to generate these instance explanations for LIME

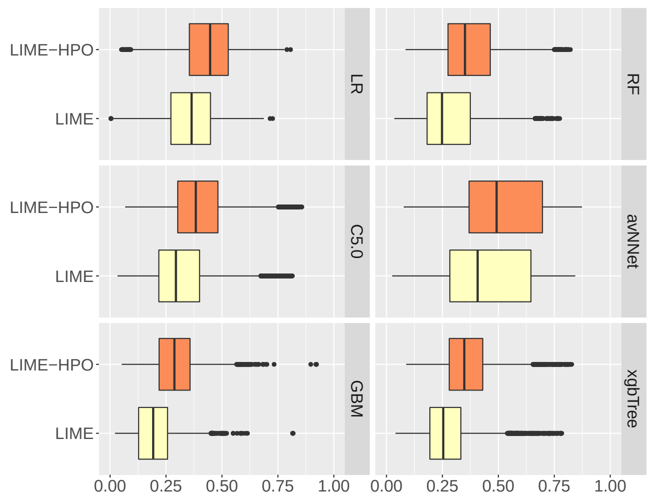


Figure 5: The distributions of the goodness-of-fit (R^2) of the local regression models constructed with LIME and LIME-HPO for all of the studied defect datasets and the studied classification techniques.

and LIME-HPO. We apply Wilcoxon signed-rank test [110] to identify whether distributions of the goodness-of-fit of the local regression models produced by LIME and LIME-HPO are statistically different. We also apply Cliff's $|\delta|$ effect size test to measure the magnitude of the differences. Finally, we report the results using boxplots in Figure 5.

<u>Results</u>. **LIME-HPO always outperform default LIME in terms of the goodness-of-fit of the local regression models with an average improvement of 8%.** Figure 5 shows the distributions of the goodness-of-fit of the local regression models constructed with LIME and LIME-HPO for all studied defect datasets. After performing hyper parameter optimisation (LIME-HPO), we find that LIME-HPO improves the goodness-of-fit of the local regression models by (at the median) 6.6% for LR, 7.4% for RF, 6.7% for C5.0, 6.1% for AVNNet, 8.0% for GBM, and 7.6% for xGBTree. We observe that the average R^2 improvement among six classification techniques is 8%. Moreover, the results of Wilcoxon signed-rank test confirm that the improvement in the goodness-of-fit of the local regression models of LIME-HPO over LIME are statistically significant for all of the studied classification techniques. The Cliff's $|\delta|$ effect size test also shows that the effect size of such differences are large for GBM; medium for LR, RF, C5.0, and xGBTree; and small for AVNNet.

(RQ2) Can model-agnostic techniques explain the predictions of defect models?

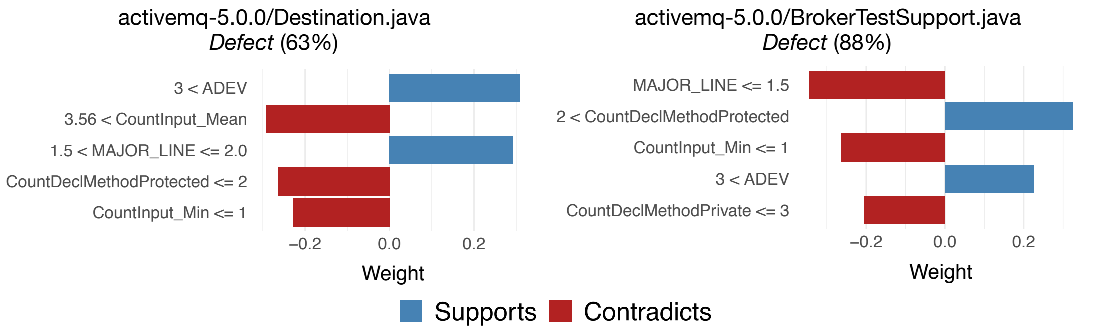
<u>Motivation</u>. Traditionally, model-specific explanation techniques (e.g., ANOVA) are used to generate global explanations. However, such global explanations cannot justify each individual prediction of defect models—i.e., why a software module is likely to be defective in the future. Recent research introduces model-agnostic techniques for explaining the predictions of any black-box models [29], [85]. Yet, these model-agnostic techniques have not been investigated in the context of software engineering (particularly for defect prediction).

<u>Approach</u>. To address RQ2, we investigate the variation of instance explanations generated by model-agnostic techniques for explaining the predictions of defect models. In other words, given a model trained from the same training data, do different predictions (i.e., test data) have different explanations. For each bootstrap sample of each defect dataset, we use the overview diagram (see Figure 3) to generate instance explanations of the testing instances. Instance explanations are the importance scores of each metric that contribute to the final probability of each prediction. For each instance explanation, we transform the scores of metrics into the ranking of metrics (e.g., from [ADEV = 0.8, MINOR_DEV = 0.15, CC = 0.05] to [1st = ADEV, 2nd = MINOR_DEV, 3rd = CC]). We then compute the rank differences of each metric among instance explanations of the correctly predicted defective instances. For example, given two instance explanations, the ranking of one instance explanation is [1st = ADEV, 2nd = MINOR_DEV, 3rd = CC], while that of another explanation is [1st = CC, 2nd = MINOR_DEV, 3rd = ADEV]. In this example, ADEV appears at the 1st rank in one instance explanation, while appearing at the 3rd rank in another instance explanation. Thus, the rank difference of ADEV is $|1-3| = 2$. We apply this calculation for all of the studied metrics for all instance explanations and report the results using box plots.

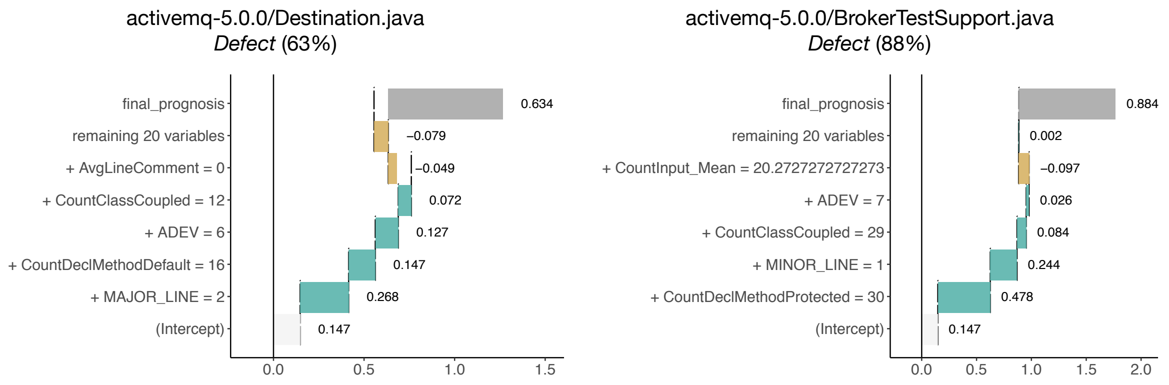
<u>Results</u>. **Model-agnostic techniques can explain the predictions of defect models.** To illustrate the visual explanation of our studied model-agnostic techniques, we select the Apache ActiveMQ 5.0.0 dataset and logistic regression (LR) as the subject of this illustrative example. Figures 6a and 6b presents an illustrative example of instance explanations of two testing instances that are correctly predicted as defective (i.e., Destination.java and BrokerTestSupport.java) produced by LIME and BreakDown, respectively. In this example, Destination.java and BrokerTestSupport.java are predicted as defective files with the predicted probability of 63% and 88%, respectively.

<u>LIME</u>. Figure 6a (left) shows the visual explanations for explaining the predictions of Destination.java that is generated by LIME. The blue bars indicate the supporting (+) scores of metrics towards the prediction as defective, while the red bars indicate the contradicting (-) scores of metrics towards the prediction as defective. Figure 6a (left) shows that Destination.java is predicted (63%) as defective due to a supporting score of 0.3 for a condition of $\{\text{ADEV} > 3\}$ and a supporting score of 0.29 for a condition of $\{1.5 < \text{MAJOR\_LINE} \leq 2\}$. On the other hand, the remaining probability of 37% of not defective could be explained by a contradict score of 0.29 for a condition of $\{\text{CountInput\_Mean} > 3.56\}$, a contradict score of 0.27 for a condition of $\{\text{CountDeclMethodProtected} \leq 2\}$, and a contradict score of 0.22 for a condition of $\{\text{CountInput\_Min} \leq 1\}$. These instance explanations indicate the most important conditions that support and contradict a file being predicted as defective.

<u>BreakDown</u>. Figure 6b (left) shows the visual explanations for explaining the predictions of Destination.java that is generated by BreakDown. The light blue bars indicate the supporting (+) probability of metrics towards the prediction as defective, while the light brown bars indicate the contradicting (-) probability of metrics towards the prediction as defective. Figure 6b (left) shows that Destination.java is



(a) An example instance explanation of LIME. The blue bars indicate supporting (positive) scores towards a file being predicted as defective, while the red bars indicate contradict (negative) scores towards its prediction.



(b) An example instance explanation of BreakDown. The x-axis presents the contribution (probability score) of each metric in the y-axis.

Figure 6: An illustrative example of instance explanations generated by LIME and BreakDown, respectively.

predicted (63%) as defective due to a supporting probability of 0.27 for MAJOR_LINE, a supporting probability of 0.15 for CountDeclMethodDefault, a supporting probability of 0.13 for ADEV, and a supporting probability of 0.07 for CountClassCoupled. In contrast, an CountClassCoupled value of 12 contradicts the prediction by a probability of 0.8, and an AvgLineComment value of 0 contradicts the prediction by a probability of 0.05.

Given the same defect models, different predictions have different instance explanations. Figure 7 shows the distributions of the rank difference of the metric among instance explanations for all of the studied defect datasets. We find that the rank differences of metrics among instance explanations are, at the median, 20 for LIME, 22 for LIME-HPO, and 21 for BreakDown. In other words, the most important metric of an instance may appear as the rank 21th-23th of another instance. We observe similar results for all studied classification techniques (the online appendix [1] provides the results of each classification technique). As shown in Figures 6a and 6b, we observe that while these two instances are predicted as defective by defect models, their instance explanations are different. According to this example, on the above sub-figure (LIME), the number of active developers (ADEV), which appears as the most important metric of the instance explanation of Destination.java, appears as the 4th important metric of the instance explanation of BrokerTestSupport.java. Similarly, on the below sub-figure (BreakDown), the lines of code contributed by major developers (MAJOR_LINE), which appears as the most important metric of the instance explanation of Destination.java, appears as the 2nd important metric of the instance explanation of BrokerTestSupport.java. The variation of instance explanations among the predictions of defect models indicates that one global explanation of defect models does not imply instance explanations (and vice versa), highlighting the need for model-agnostic techniques for explaining the predictions defect models.

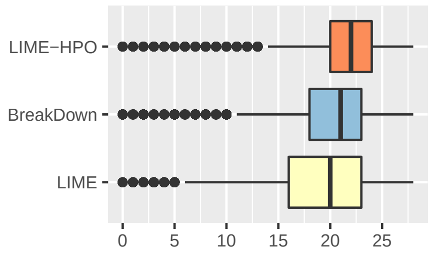


Figure 7: The distributions of rank differences of each metric across instances explanations for all studied defect datasets.

(RQ3) Do instance explanations generated by model-agnostic techniques overlap with the global explanation of defect models?

Motivation. Recent research raised concerns about the trustworthiness of instance explanations as generated by model-agnostic techniques. For example, Ribeiro *et al.* [85] argue that instance explanations must correspond to how the trained model behaves. Guidotti *et al.* [30] argue that the local approximation models should mimic the black-box models when predicting an unseen dataset. The results of RQ2 suggest that instance explanations generated by the studied model-agnostic techniques have a great variation among each prediction of defect models. Yet, little is known about whether these instance explanations generated by the studied model-agnostic techniques are overlapping with the global explanation of defect models.

Approach. To address RQ3, we investigate whether instance explanations generated by model-agnostic techniques are overlapping with the global explanation of defect models. For each bootstrap sample, we use the overview diagram (see Figure 3) to generate a global explanation from training instances and generate instance explanations from testing instances. Since the out-of-sample bootstrap validation technique leverages aspects of statistical inference [19], [23], [34], [95], [101], both training and testing samples are approximately equivalent to the population (i.e., the original dataset). Thus, explanations derived from both training and testing instances should also be approximately equivalent. To generate a global explanation, we use the ANOVA Type-II analysis for logistic regression, the scaled Permutation Importance analysis for random forests, the usage of metrics for C5.0, the combinations of the absolute weights across hidden layers for AVNNet, and the relative influence of metrics derived from boosted trees for GBM and xGBTree. To generate instance explanations for testing instances, we use the LIME, LIME-HPO, and BreakDown techniques.

While a defect model generates only one global explanation from training instances, model-agnostic techniques generate many instance explanations from testing instances. Thus, we need to summarise instance explanations to the model level prior to comparing with a global explanation of each bootstrap sample. To summarise instance explanations, we apply the Scott-Knott Effect Size Difference test (ScottKnottESD) [94] to identify the ranking of metrics that is statistically distinct across ranks while being non-negligible different within ranks. Then, we identify the top-k overlapping metrics between global explanations and the summary of instance explanations. The top-k overlapping metrics are the number of the top-k metrics of a global explanation that consistently appear in the top-k metrics of the summary of instance explanations. For example, the global ranking of metrics is [1st = LOC, 2nd = CC, 3rd = ADEV], while the summarised ranking of metrics for instance explanations is [1st = ADEV, 2nd = MINOR_DEV, 3rd = CC]. In this example, the top-3 overlapping metrics are 2 out of 3 metrics. Finally, we apply Wilcoxon signed-rank test [110] to identify whether the distributions of the top-k overlapping metrics between the global explanation and instance explanation produced by LIME, LIME-HPO, and BreakDown are statistically different. We also apply Cliff's $|\delta|$ effect size test to measure the magnitude of the differences.

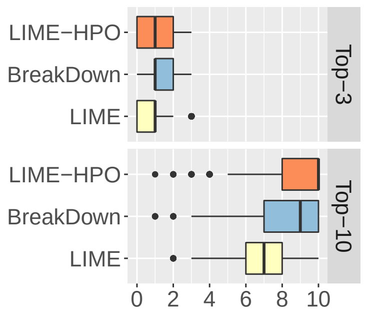


Figure 8: The distributions of the top-k overlapping metrics between the global explanation and instance explanations for all of the studied defect datasets and the studied classification techniques.

Results. **Despite the variation of the ranking of the top-10 important metrics for instance explanations in RQ2, their overall ranking is mostly overlapping with (but is not exactly the same) as that of the global explanations.** Figure 8 shows that, at the median, 7, 10, and 9 of the top-10 summarised important metrics for instance explanations are overlapping with the top-10 global important metrics for LIME, LIME-HPO, and BreakDown, respectively. The results of Wilcoxon signed-rank test show that the differences of the top-10 overlapping metrics are statistically significant among the studied model-agnostic techniques (for LIME and LIME-HPO, LIME-HPO and BreakDown, and LIME and BreakDown). The Cliff's $|\delta|$ effect size test also shows that the magnitude of such differences are large for LIME and LIME-HPO, medium for LIME and BreakDown, and negligible for LIME-HPO and BreakDown, suggesting that LIME-HPO is comparable to BreakDown. On the other hand, at the median, at least one of the top-3 summarised important metrics for instance explanations is overlapping with the top-3 global important metrics. The detailed results of each studied classification techniques are available in the online appendix [1].

(RQ4) Do model-agnostic techniques generate the same instance explanation when they are re-generated for the same instance?

Motivation. Recent research raised concerns about the reliability of instance explanations generated by model-agnostic techniques. For example, Lundberg *et al.* [57] argue that instance explanations must remain the same when they are re-generated for the same instance. Assuming that one wants to generate an explanation for a file predicted as defective, model-agnostic techniques (that involve random perturbation like LIME) might generate different synthetic instances, leading to different explanations for the same instance. Thus, little is known about whether model-agnostic techniques generate the same instance explanation when they are re-generated for the same instance and the same defect model.

Approach. To address RQ4, we analyse the reliability of the instance explanations generated by LIME, LIME-HPO,

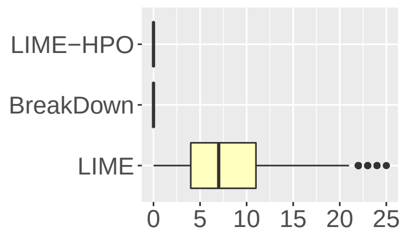


Figure 9: The distributions of rank differences of each metric when re-generating instance explanations for all of the studied defect datasets.

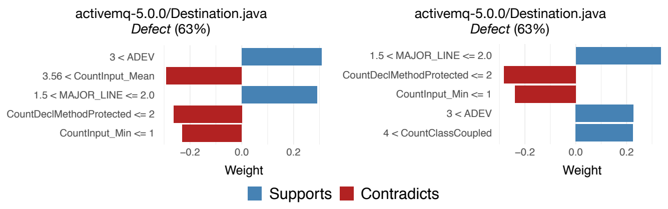


Figure 10: An illustrative example of instances explanations of a defective testing instance when regenerating instance explanations with LIME.

and BreakDown. Ideally, re-generating instance explanations of the same instance from the same model using the same model-agnostic technique should produces the same explanation. Therefore, we use the variation in instance explanations when re-generating with the same setting as a proxy for measuring the reliability of model-agnostic techniques. Rather than generating instance explanations of all testing instances, we randomly select only one testing instance as the instance of interest. Similar to RQs 1, 2, and 3, we use the overview diagram (see Figure 3) to generate instance explanations of this instance of interest. We re-generate the instance explanation of the selected instance for 100 repetitions. We compute the rank differences of each metric when re-generating instance explanations and report the results of all studied defect datasets using boxplots.

Results. **Regardless of the studied classification techniques, LIME-HPO and BreakDown consistently generate the same instance explanation for the same instance. On the other hand, LIME generates different instance explanations when re-generating instance explanations of the same instance.** Figure 9 shows the distributions of rank differences of each metric when re-generating instance explanations for all studied defect datasets. Ideally, instance explanations of an instance should be the same when re-generating using the same model and the same model-agnostic technique. Regardless of the studied classification techniques, we find that LIME-HPO and BreakDown consistently produce the same instance explanation for the same instance. On the other hand, we find that LIME produces inconsistent instance explanations across repetitions with the median rank differences of 7. We report the detailed results of rank differences for each studied classification technique in the online appendix [1].

To further illustrate the variation of instance explanations generated by LIME across repetitions, similar to RQ2, we select the Apache ActiveMQ 5.0.0 dataset as the subject

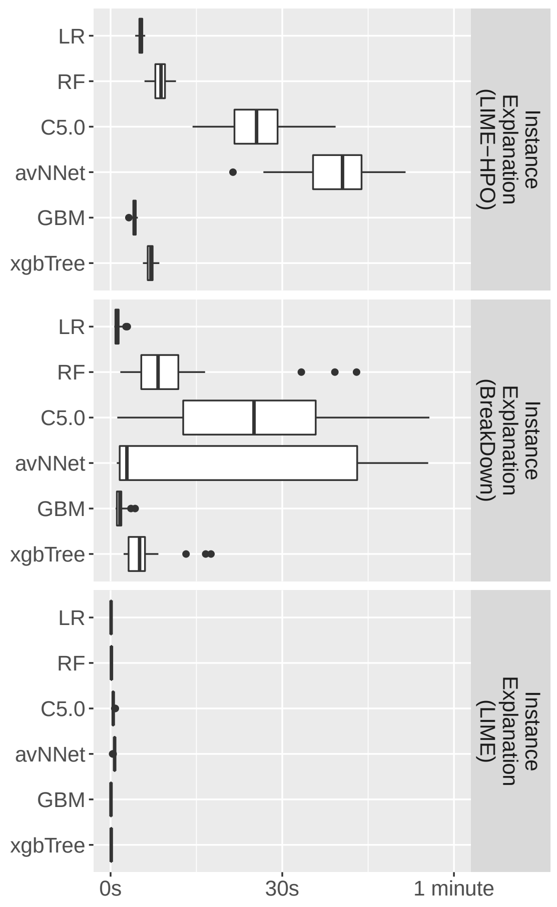


Figure 11: The distributions of computational time of model-agnostic techniques for explaining the predictions of defect models for all of the studied defect datasets.

of this illustrative example. Figure 10 shows an illustrative example of instance explanations of a defective testing instance when re-generating instance explanations with LIME. We observe that the model consistently identifies the instance as defective with the same predicted probability of 0.63 for both repetitions. However, while these instance explanations of the same instances are generated from the same model using the same model-agnostic technique (LIME), such explanations—the top-5 important metrics—vary greatly.

(RQ5) What is the computational time of model-agnostic techniques for explaining the predictions of defect models?

Motivation. The computational time is one of the most important criteria when deploying software analytics in practice. Model-agnostic techniques may introduce significant overhead to the current practices of defect modelling workflow. Yet, little is known about whether the computational time of model-agnostic techniques is practical to be deployed by practitioners.

Approach. To address RQ4, we analyse the computational time of model-agnostic techniques for explaining the predictions of defect models. To do so, similar to RQs 1, 2, 3, and 4, we use the overview diagram (see Figure 3) to construct defect models and generate instance explanations. For each studied defect dataset, we generate one set of bootstrap training and testing instances. We construct a defect model using bootstrap training instances. Then, we randomly select one testing instance and generate an instance explanation using model-agnostic techniques to measure their computation time. The computational time is based on a standard computing machine with an Intel Core i7-8700K processor and 32GB of RAM. Finally, we report the results using box plots.

Results. **The computational time of LIME-HPO, Break-Down, and LIME is less than a minute to generate instance explanations for all of the studied classification techniques.** Figure 11 shows the distributions of computational time of model-agnostic techniques for explaining the predictions of defect models for all of the studied defect datasets. We find that, regardless of the studied classification techniques, the computational time that the studied model-agnostic techniques take to generate an instance explanation is at most one minute for all of the studied defect datasets. This finding suggests that all of the studied model-agnostic techniques is practical to be used in real-world deployments.

(RQ6) How do practitioners perceive the contrastive explanations generated by model-agnostic techniques?

Motivation. Referring to a theory of explanations described in Section 2.2, Miller [67] and Lipton *et al.* [53] argue that explanations can be presented as answers to why-questions and humans tend to be cognitively attached to digest contrastive explanations. *Contrastive explanations* focus on only the differences on properties within an object (Property-contrast), between two Objects (Object-contrast), and within an object over Time (Time-contrast) [107]. Thus, contrastive explanations may be more valuable and more intuitive to humans. Yet, little is known about whether contrastive explanations generated by model-agnostic techniques can answer why-questions.

Approach. To address RQ6, we conducted a survey study of 20 software practitioners to investigate their perceptions of instance explanations generated by model-agnostic techniques. As suggested by Kitchenham and Pfleeger [45], we performed the following steps when conducting this survey study:

(Step-1) Setting the objectives: The survey aimed to investigate whether instance explanations generated by model-agnostic techniques (1) can be used to answer the why-questions (i.e., Property-contrast, Objective-contrast, and Time-contrast); (2) build appropriate trusts of the predictions of defect models; and (3) are necessary and useful.

(Step-2) Survey design: We first introduced a concept of explainable defect models with respect to the literature of eXplainable Artificial Intelligence (XAI). Then, we used the Releases 2.10.0 and 2.11.0 of the Apache Camel project as the subject of the study to generate instance explanations. We presented 3 types of explanations for investigation, i.e., **Property-contrast explanation** (e.g., why was file A predicted as defective rather than clean?), **Object-contrast explanation** (e.g., why was file A predicted as defective, while file B was predicted as clean?), and **Time-contrast explanation** (e.g., why was file A predicted as defective in version 1.2, but was subsequently predicted as clean in version 1.3?). Figure 12 illustrates an example of the Time-contrast explanations generated by model-agnostic techniques, while other examples of the Property-contrast and Object-contrast explanations are provided in the online appendix [1].

The survey design is a cross-sectional study where participants provide their responses at one fixed point in time. The survey consists of demographic and three sets of questions with respect to the 3 objectives of the study. There are 11 closed-ended questions and 20 open-ended questions. The survey takes approximately 15 minutes to complete and is anonymous.

(Step-3) Developing a survey instrument: For closed-ended questions, we used agreement and evaluation ordinal scales. To mitigate the inconsistency of the interpretation of numeric ordinal scales, we labelled each level of the ordinal scales with words as suggested by Krosnick [48], i.e., strongly disagree, disagree, neutral, agree, and strongly agree. The format of our survey instrument was an online questionnaire. We used an online questionnaire service provided by Google Forms.[2] When accessing the survey, each participant was provided with an explanatory statement that describes the purpose of the study, why the participant is chosen for this study, possible benefits and risks, and confidentiality.

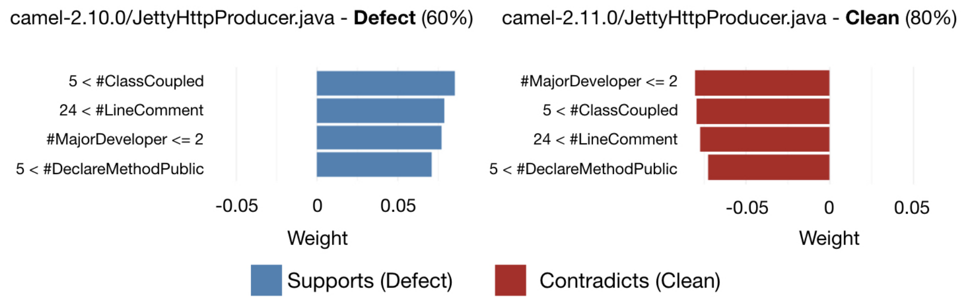
(Step-4) Evaluating the survey instrument: We carefully evaluated the survey (i.e., pre-testing [55]) prior to recruiting participants. We evaluated it with focus groups (i.e., practitioners) to assess the reliability and validity of the survey. We re-ran it to identify and fix potential problems (e.g., missing, unnecessary, or ambiguous questions) until reaching a consensus among the focus groups. Finally, the survey has been rigorously reviewed and approved by Monash University Human Research Ethics Committee (MUHREC Project ID: 22542). We also provide the ethics approval certificate in our online appendix [1].

(Step-5) Obtaining valid data: The target population of our study is software practitioners. To reach the target population, we used a recruiting service provided by the Amazon Mechanical Turk to recruit 20 participants as a representative subset of the target population. We also applied participant filter options of "Employment Industry - Software & IT Services" and "Job Function - Information Technology" to ensure that the recruited participants are valid samples representing the target population.

(Step-6) Analysing the data: To analyse the data, we first checked the completeness of the collected data (i.e., whether all questions are appropriately answered). Then, we manually analysed the answers to the open-ended questions to extract in-depth insights. For closed-ended questions, we summarised and presented key statistical results.

Results. **65% of the participants agree that model-agnostic techniques can generate the Time-contrast explanation to answer the why-questions.** Similarly, we found that 55% and 50% of the participants agree (and strongly agree) that model-agnostic techniques can generate the Property-contrast and Object-contrast explanations, respectively.

2. https://www.google.com/forms



T-contrast Explanation: *JettyHttpProducer.java* is predicted as defective in release 2.10.0, but is predicted as clean in release 2.11.0.
The reasons are (1) 5 < # *of class coupled*, (2) 24 < *lines of comment*, (3) # *of major developers* <= *2*, and (4) 5 < # *of public methods*.

Figure 12: An example of the Time-contrast explanations generated by model-agnostic techniques for explaining the predictions of defect models.

55% and 65% of the participants perceived that the Time-contrast explanations generated by model-agnostic techniques are necessary and useful, respectively. Similarly, we found that 40% and 30% of the participants perceive that the Property-contrast and Object-contrast explanations generated by model-agnostic techniques are necessary, respectively. We found that 55% and 40% of the participants perceive that the Property-contrast and Object-contrast explanations generated by model-agnostic techniques are useful, respectively. Finally, we found that 50%, 45%, and 70% of the participants agree (and strongly agree) that instance explanations generated by model-agnostic techniques can build appropriate trusts of the predictions of defect models for the Property-contrast, Object-contrast, and Time-contrast explanations.

6 Discussion

In this section, we discuss the key differences between model-agnostic techniques and a tree-based technique.

6.1 A Comparison of Model-Agnostic Techniques with a Tree-based Technique

There are many approaches and granularity levels to explain the predictions of defect models (i.e., global explanation, subgroup explanation, instance explanation). Traditionally, we can use tree-based models to predict and explain the characteristics of defective files. For example, Tan *et al.* [93] use Alternative Decision Tree technique (ADTree) as provided by Weka [31] to explain the predictions of defect models. Chen *et al.* [14] use Fast-and-Frugal Trees (FFT) technique to construct comprehensible defect models. An explanation of each prediction can be generated by deriving a decision node of the decision tree that matches with the instance to explain. Below, we select the ADTree technique as the subject of this discussion and discuss the strengths and weakness of the tree-based technique with respect to the model accuracy, the locality of the explanation, and the visual explanation.

6.1.1 Model Accuracy

Practitioners often make decisions whether defect models should be deployed in practice based on their model accuracy. We first evaluate the model accuracy of the decision tree technique for predicting defective files with respect to three performance measures (i.e., AUC, Initial False Alarm (IFA), and Popt(20)) for all of the 32 studied defect datasets. We report the evaluation results of ADTree in the online appendix [1].

We find that ADTree achieves a median AUC of 0.75, a median IFA of 53, and a median Popt(20) of 0.02. When comparing the model accuracy to other classification techniques as shown in Figure 4, we find that ADTree is the least top-performing classification technique in terms of AUC, IFA, and Popt(20), raising concerns that the explanations that are derived from such inaccurate models could be misleading. However, such black-box AI/ML-based classification techniques are complex and hard to explain. Thus, model-agnostic techniques play a key role in explaining the predictions of highly accurate yet complex classification techniques.

6.1.2 Visual Explanation

Practitioners often make a decision whether the predictions should be trusted based on the understand-ability of the visual explanations. We conduct a preliminary survey with 20 practitioners to better understand which visual explanations are the most preferred by practitioners. We find that the visual explanation of ADTree is the most preferred by practitioners (60% of practitioners agree or strongly agree), as such visual explanation is simple to digest and involves logical reasoning. While the visual explanation of LIME also involves logical reasoning, practitioners are confused about the bar colours of LIME that explain the supporting and contradict scores (i.e., LIME uses the red colour to explain contradicting scores, which imply that such metrics contribute towards a prediction as clean). Despite the advantages of BreakDown that decompose the final probability score into each score for each feature, practitioners raise concerns that the visual explanation lacks necessary details (e.g., optimal threshold values for each metric) and is difficult to understand. Therefore, future studies should develop a novel visual explanation that is understandable to domain experts using human-centred design approaches (e.g., a co-creation design session and the Wizard-of-Oz prototyping technique).

6.1.3 The Locality of Explanation

Practitioners often make a decision as to whether the predictions should be trusted based on the locality of the explanations. The locality of explanations refers to the scope that such explanations are derived from. For example, an explanation generated by a variable importance technique of the random forest technique is derived from the global level of the prediction models. On the other hand, an explanation generated by a model-agnostic technique (e.g., LIME) is derived from a local model that is constructed from instances around their neighbours. Similarly, an explanation generated by a decision tree technique is derived from a decision node which can cover N instances. Although the locality of explanation between LIME and a decision tree is similar, the key difference is the flexibility of the visual representation and the choice of AI/ML-based classification techniques. In other words, a decision node can be used to explain only the decision tree technique. While the model accuracy of such decision tree technique is not as competitive as complex AI/ML-based classification techniques, model-agnostic techniques can be used to explain any classification techniques.

Summary. Decision tree is easy to understand, but not as accurate as other complex AI/ML learning algorithms. Complex AI/ML learning algorithms (e.g., xGBTree, neural network) are more accurate, but it is very hard to understand their predictions. Thus, the main goal of our paper is to leverage model-agnostic techniques to explain the predictions of any accurate yet complex AI/ML learning algorithms. However, practitioners perceive that decision tree is the most preferred visual explanation, suggesting that future studies should invent new visual explanations that are directed towards practitioners' needs.

7 Related Work

We discuss key related work in order to situate the contributions of our paper with respect to explainable software analytics and analytical models for software defects.

7.1 Explainable Software Analytics

Despite the advances in analytical modelling in software engineering, recent work raises a concern about a lack of explainability of analytical models in software engineering [16]. Practitioners also share similar concerns that analytical models in software engineering must be explainable and actionable in order to be of practical use [16], [52], [65]. For example, Dam *et al.* [16] argue that making software analytics models explainable to software practitioners is as important as achieving accurate predictions. Lewis *et al.* [52] emphasize that defect modelling should be more actionable to help Google engineers debug their programs. Menzies and Zimmermann [65] also emphasize that software analytics must be actionable.

Key Difference. To the best of our knowledge, little research in software analytics explores a theory of global explanations, the differences of the goals, scopes, and targets of global explanations and instance explanations. Moreover, model-agnostic techniques, i.e. techniques for explaining an individual prediction, have not yet been introduced in the context of software engineering.

7.2 Analytical Models for Software Defects (i.e., Defect Models)

Analytical models for software defects play a foundational role in optimising the limited resources of software quality assurance activities and in building empirical theories of software quality. Below, we discuss the literature with respect to the two main purposes of defect models.

To predict software defects. First, defect models are used to optimize the limited quality assurance resources on the most risky software modules [17], [63], [99], [102]. There are a plethora of studies focus on investigating advanced features and advanced techniques in order to improve the predictive ability of defect models. For example, Wang *et al.* [109] leverage a multiple kernel learning to produce a multiple kernel classifier through ensemble learning method, which has the advantages of both multiple kernel learning and ensemble learning. Wang *et al.* [108] use a deep belief network (DBN) to automatically learn semantic features to improve the predictive ability of defect models.

The improvement of the predictive ability of defect models is critical to practitioners when deploying defect models in practice.

Key Difference. Unlike prior studies that focused on improving the predictive ability of defect models, this paper focuses on investigating techniques to explain software defect predictions.

To explain software defects. Second, defect models are used (1) to understand factors that are associated with software defects, (2) to establish effective quality improvement plans, (3) to provide actionable guidance to avoid pitfalls that lead to software defects in the past, and (4) to build empirical theories of software quality [10], [58], [104]. There are a plethora of studies focusing on investigating the best modelling workflow and advanced techniques in order to improve the explainability of defect models, which will be discussed below.

7.2.1 Investigating the best modelling workflow to improve the explainability of defect models

Explanations of defect models may not be accurate and reliable if care is not taken in the analytical modelling workflow for software defects. Hall *et al.* [32] raised a critical concern that different researchers often develop different analytical workflows, which makes it hard to derive practical guidelines of the best defect modelling workflows. Recent studies demonstrate that if pitfalls are not mitigated when collecting defect datasets [98], [112] and designing the analytical workflow [64], [95], [96] for software defects, the predictions and explanations that are derived from defect models may be inaccurate and unreliable. For example, Menzies and Shepperd [64] raised concerns that there are many components that could potentially impact the predictions of defect models.

Recent studies reveal many components of the analytical workflow that impact the predictions and explanations of defect models [96]. For example, noise in defect datasets [27], the quality of issue reports [98], defect labelling techniques [112], feature selection techniques [28], [42], collinearity analysis [39], [40], [42], class rebalancing techniques [97], model construction [27], parameter optimisation [2], [3], [25], [99], [102], model evaluation [101], and model interpretation [39].

Key Difference. While these studies focused on developing practical guidelines to develop the most accurate and reliable analytical models to predict and explain software defects, this paper focused on investigating advanced techniques to improve the explainability of defect models.

7.2.2 Investigating advanced techniques to improve the explainability of defect models

As discussed in Section 3, there are techniques to generate explanations with different goals, scopes, and target of explanations. Below, we discuss prior studies focused on (1) explaining a black-box model; (2) explaining a group of predictions; and (3) explaining an individual prediction.

Explaining a black-box model

Prior studies have been leveraged well-established explainable classification techniques, such as regression models [80], [81], random forests [44], [76], decision trees [113], decision rules [86]. In addition, Chen *et al.* [14] point out that Fast-and-Frugal Tree is more accurate, comprehensible, and operational than the well-established explainable classification techniques in the context of software defect prediction. Moreover, a domain-specific classification technique like Bellwether [47] has shown to mitigate the conclusion instability when transferring knowledge from one software project to another.

Despite the recent advances of well-established explainable classification techniques and domain-specific classification techniques in the context of software engineering, such techniques only derive knowledge of the learned models from the whole training dataset without justifying an individual prediction. Instead of explaining a black-box model, prior studies [7], [62] proposed techniques with an attempt to explain a smaller group of predictions with similar data characteristics in order to improve the predictive ability and explainability of defect models, such techniques still cannot justify each individual prediction and uphold the privacy laws (i.e., GDPR).

Explaining an individual prediction

Recently, model-agnostic algorithms that treat the original model as a black-box have been proposed to explain the predictions of any learners at the instance level. For example, Ribeiro *et al.* [85] proposed a Local Interpretable Model-Agnostic Explanations (LIME) that leverages the approximation of a simple linear model locally around the prediction.

Key Difference. Unlike prior studies that focus on explaining black-box models or a group of predictions, this paper is the first to investigate model-agnostic techniques for explaining an individual prediction from testing instances in the domain of software engineering.

8 Threats to Validity

8.1 Construct Validity

Prior studies show that the parameters of learners have an impact on the performance of defect models [25], [46], [59], [61], [99]. While we use a default parameter setting of 100 trees for random forests, recent work [38], [99], [106] find that the parameters of random forests are insensitive to the performance of defect models. Thus, the parameters of random forests do not pose a critical threat to the validity of our study.

Due to the technical limitations of our studied classification techniques, correlated metrics must be removed prior to explaining the prediction models. One might suggest that LASSO should be used to penalise collinearity (i.e., correlated metrics). We found that the top-rank metric that is correlated with another will be less important by half when using LASSO (as they penalise the important score by half to another correlated metric). Yet, they are still correlated. We noted that this is still an open problem for ML domains. Thus, software practitioners should not draw implications solely from the most important metric, but should also consider its group of correlated metrics.

We use 100 iterations to draw reliable bootstrap estimates of the studied measures in the experiments. However, such high bootstrap iterations often come with a high computation cost. Thus, 100 iterations of bootstrap validation may not be practical for compute-intensive AI/ML algorithms like deep learning.

To ensure that our survey is reliable and valid, we carefully evaluated the survey (i.e., pre-testing [55]) prior to recruiting participants. We evaluated it with focus groups (i.e., practitioners) to assess the reliability and validity of the survey. We re-ran it to identify and fix potential problems (e.g., missing, unnecessary, or ambiguous questions) until reaching a consensus among the focus groups. Finally, the survey has been rigorously reviewed and approved by Monash University Human Research Ethics Committee (MUHREC Project ID: 22542).

8.2 Internal Validity

We studied a limited number of model-specific explanation techniques (i.e., the ANOVA Type-II analysis for logistic regression and the scaled Permutation Importance analysis for random forests) and model-agnostic techniques (i.e., BreakDown and LIME). Our results, therefore, may not generalise to other defect explainers. However, other techniques for generating explanations can be investigated in future studies. In this paper, we provide a detailed methodology for others who wish to revisit our study with other techniques for generating explanations.

8.3 External Validity

In this study, our experiments rely on one defect prediction scenario, i.e., within-project defect prediction. However, there are a variety of defect prediction scenarios in the literature (e.g., cross-project defect prediction [13], [113], just-in-time defect prediction [43], [74], and heterogenous defect prediction [70]). Therefore, the results may differ in other scenarios. Future studies should revisit our study in other defect prediction scenarios.

The number of our studied datasets is limited and thus may produce results that cannot be generalised to other datasets and domains. However, it is a carefully curated dataset where the ground truths were labelled based on the affected releases. Future work can revisit and replicate our study with other datasets.

The number of survey participants is limited to a recruitment of 20 software practitioners from Amazon Mechanical Turk. Thus, the results of the survey may not be representative to the perceptions of all software practitioners. Future work can revisit and replicate the survey study with other groups of software practitioners.

9 Conclusions

We investigate model-agnostic techniques for explaining the predictions of defect models. Through a case study of 32 publicly-available defect datasets of 9 large-scale open-source software systems, the experimental results lead us to conclude that (1) model-agnostic techniques are needed to explain individual predictions of defect models; (2) instance explanations generated by model-agnostic techniques are mostly overlapping with the global explanation of defect models (except for LIME) and reliable when they are re-generated (except for LIME); (3) model-agnostic techniques take less than a minute to generate instance explanations; and (4) more than half of the practitioners perceive that the instance explanations are necessary and useful to understand the predictions of defect models. Since the implementation of the studied model-agnostic techniques is readily available in both Python and R, we recommend model-agnostic techniques be used to explain the predictions of defect models.

References

[1] "Online Appendix for "An Empirical Study of Model-Agnostic Techniques for Defect Prediction Models"," https://github.com/awsm-research/model-agnostic-online-appendix.

[2] A. Agrawal and T. Menzies, "Is Better Data Better Than Better Data Miners?: On the Benefits of Tuning SMOTE for Defect Prediction," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2018, pp. 1050–1061.

[3] A. Agrawal, T. Menzies, L. L. Minku, M. Wagner, and Z. Yu, "Better Software Analytics via" DUO": Data Mining Algorithms Using/Used-by Optimizers," *arXiv preprint arXiv:1812.01550*, 2018.

[4] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2006, pp. 361–370.

[5] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A Systematic and Comprehensive Investigation of Methods to Build and Evaluate Fault Prediction Models," *Journal of Systems and Software*, vol. 83, no. 1, pp. 2–17, 2010.

[6] C. V. F. Bas, *The Scientific Image*. Oxford University Press, 1980.

[7] N. Bettenburg, M. Nagappan, and A. E. Hassan, "Think locally, act globally: Improving defect and effort prediction models," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2012, pp. 60–69.

[8] P. Biecek and A. Grudziaz, "breakDown: Model Agnostic Explainers for Individual Predictions. R package version 0.1.6." *Software available at URL: https://cran.r-project.org/package=breakDown*.

[9] ——, "pyBreakDown: Python implementation of R package breakDown," *Github Repository https://github.com/MI2DataLab/pyBreakDown*, 2017.

[10] C. Bird, B. Murphy, and H. Gall, "Don't Touch My Code ! Examining the Effects of Ownership on Software Quality," in *Proceedings of the European Conference on Foundations of Software Engineering (ESEC/FSE)*, 2011, pp. 4–14.

[11] L. Breiman, A. Cutler, A. Liaw, and M. Wiener, "randomForest : Breiman and Cutler's Random Forests for Classification and Regression. R package version 4.6-12." *Software available at URL: https://cran.r-project.org/package=randomForest*.

[12] J. Cahill, J. M. Hogan, and R. Thomas, "Predicting Fault-prone Software Modules with Rank Sum Classification," in *Proceedings of the Australian Software Engineering Conference (ASWEC)*, 2013, pp. 211–219.

[13] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Multi-objective Cross-project Defect Prediction," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2013, pp. 252–261.

[14] D. Chen, W. Fu, R. Krishna, and T. Menzies, "Applications of Psychological Science for Actionable Analytics," in *Proceedings of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018, pp. 456–467.

[15] E. Coelho and A. Basu, "Effort Estimation in Agile Software Development using Story Points," *International Journal of Applied Information Systems (IJAIS)*, vol. 3, no. 7, 2012.

[16] H. K. Dam, T. Tran, and A. Ghose, "Explainable Software Analytics," in *Proceedings of the International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 2018, pp. 53–56.

[17] M. D'Ambros, M. Lanza, and R. Robbes, "An Extensive Comparison of Bug Prediction Approaches," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2010, pp. 31–41.

[18] ——, "Evaluating Defect Prediction Approaches: A Benchmark and an Extensive Comparison," *Empirical Software Engineering (EMSE)*, vol. 17, no. 4-5, pp. 531–577, 2012.

[19] B. Efron and R. J. Tibshirani, *An Introduction to the Bootstrap*. Boston, MA: Springer US, 1993.

[20] K. O. Elish and M. O. Elish, "Predicting Defect-prone Software Modules using Support Vector Machines," *Journal of Systems and Software*, vol. 81, no. 5, pp. 649–660, 2008.

[21] J. Fox, S. Weisberg, and B. Price, "car: Companion to Applied Regression. R package version 3.0-2," *Software available at URL: https://cran.r-project.org/web/packages/car*.

[22] S. French, "Decision Analysis," *Wiley StatsRef: Statistics Reference Online*, 2014.

[23] J. Friedman, T. Hastie, and R. Tibshirani, *The Elements of Statistical Learning*. Springer series in statistics, 2001, vol. 1.

[24] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of Statistics*, pp. 1189–1232, 2001.

[25] W. Fu, T. Menzies, and X. Shen, "Tuning for Software Analytics: Is it really necessary?" *Information and Software Technology*, vol. 76, pp. 135–146, 2016.

[26] M. Gevrey, I. Dimopoulos, and S. Lek, "Review and comparison of methods to study the contribution of variables in artificial neural network models," *Ecological Modelling*, vol. 160, no. 3, pp. 249–264, 2003.

[27] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the Impact of Classification Techniques on the Performance of Defect Prediction Models," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2015, pp. 789–800.

[28] B. Ghotra, S. Mcintosh, and A. E. Hassan, "A Large-scale Study of the Impact of Feature Selection Techniques on Defect Classification Models," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2017, pp. 146–157.

[29] A. Gosiewska and P. Biecek, "iBreakDown: Uncertainty of global explanations for Non-additive Predictive Models," *arXiv preprint arXiv:1903.11420*, 2019.

[30] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, D. Pedreschi, and F. Giannotti, "A Survey Of Methods For Explaining Black Box Models," vol. 51, no. 5, pp. 1–45, 2018. [Online]. Available: http://arxiv.org/abs/1802.01933

[31] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[32] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A Systematic Literature Review on Fault Prediction Performance in Software Engineering," *Transactions on Software*

Engineering (TSE), vol. 38, no. 6, pp. 1276–1304, 2012. [Online]. Available: http://ieeexplore.ieee.org.pc124152.oulu.fi:8080/xpls/abs{_}all.jsp?arnumber=6035727

[33] J. A. Hanley and B. J. McNeil, "The meaning and use of the area under a receiver operating characteristic (ROC) curve." *Radiology*, vol. 143, no. 4, pp. 29–36, 1982.

[34] F. E. Harrell Jr, *Regression Modeling Strategies : With Applications to Linear Models, Logistic and Ordinal Regression, and Survival Analysis*. Springer, 2015.

[35] D. J. Hilton, J. J. McClure, and B. R. Slugoski, "The psychology of counterfactual thinking," D. R. Mandel, D. J. Hilton, and P. Catellani, Eds. Routledge, Abingdon, Oxon, UK: Routledge Research International Series in Social Psychology, 2005, ch. The course of events: counterfactuals, causal sequences, and explanation, pp. 44–60.

[36] Z. Horne, M. Muradoglu, and A. Cimpian, "Explanation as a cognitive process," *Trends in Cognitive Sciences*, 1 2019.

[37] Q. Huang, X. Xia, and D. Lo, "Supervised vs Unsupervised Models: A Holistic Look at Effort-Aware Just-In-Time Defect Prediction," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 159–170.

[38] Y. Jiang, B. Cukic, and T. Menzies, "Can Data Transformation Help in the Detection of Fault-prone Modules?" in *Proceedings of the International Workshop on Defects in Large Software Systems (DEFECTS)*, 2008, pp. 16–20.

[39] J. Jiarpakdee, C. Tantithamthavorn, and A. E. Hassan, "The Impact of Correlated Metrics on Defect Models," *Transactions on Software Engineering (TSE)*, p. To Appear, 2019.

[40] J. Jiarpakdee, C. Tantithamthavorn, A. Ihara, and K. Matsumoto, "A Study of Redundant Metrics in Defect Prediction Datasets," in *Proceedings of the International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2016, pp. 51–52.

[41] J. Jiarpakdee, C. Tantithamthavorn, and C. Treude, "Artefact: An R Implementation of the AutoSpearman Function," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, 2018, p. To appear.

[42] ——, "AutoSpearman: Automatically Mitigating Correlated Software Metrics for Interpreting Defect Models," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 92–103.

[43] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A Large-Scale Empirical Study of Just-In-Time Quality Assurance," *Transactions on Software Engineering (TSE)*, vol. 39, no. 6, pp. 757–773, 2013.

[44] A. Kaur and R. Malhotra, "Application of Random Forest in Predicting Fault-prone Classes," in *Proceedings of International Conference on the Advanced Computer Theory and Engineering (ICACTE)*, 2008, pp. 37–43.

[45] B. A. Kitchenham and S. L. Pfleeger, "Personal opinion surveys," in *Guide to Advanced Empirical Software Engineering*. Springer, 2008, pp. 63–92.

[46] A. G. Koru and H. Liu, "An Investigation of the Effect of Module Size on Defect Prediction Using Static Measures," *Software Engineering Notes (SEN)*, vol. 30, pp. 1–5, 2005.

[47] R. Krishna and T. Menzies, "Simpler Transfer Learning (Using "Bellwethers")," pp. 1–23, 2017. [Online]. Available: http://arxiv.org/abs/1703.06218

[48] J. A. Krosnick, "Survey research," *Annual Review of Psychology*, vol. 50, no. 1, pp. 537–567, 1999.

[49] M. Kuhn, J. Wing, S. Weston, A. Williams, C. Keefer, A. Engelhardt, T. Cooper, Z. Mayer, B. Kenkel, R. Team *et al.*, "caret: Classification and regression training. R package version 6.0–78," *Software available at URL: https://cran.r-project.org/web/packages/caret*, 2017.

[50] D. B. Leake, *Evaluating Explanations: A Content Theory*. Psychology Press, 2014.

[51] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings," *Transactions on Software Engineering (TSE)*, vol. 34, no. 4, pp. 485–496, 2008.

[52] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr, "Does Bug Prediction Support Human Developers? Findings from a Google Case Study," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2013, pp. 372–381.

[53] P. Lipton, "Contrastive explanation," *Royal Institute of Philosophy Supplement*, vol. 27, pp. 247–266, 1990.

[54] Z. C. Lipton, "The mythos of model interpretability," in *Proceedings of the 2016 ICML Workshop on Human Interpretability in Machine Learning*, 2016, pp. 96–100.

[55] M. S. Litwin, *How to Measure Survey Reliability and Validity*. Sage, 1995, vol. 7.

[56] T. Lombrozo, "The structure and function of explanations," *Trends in Cognitive Sciences*, vol. 10, no. 10, pp. 464–70, 2006 Oct 2006. [Online]. Available: https://doi.org/10.1016/j.tics.2006.08.004

[57] S. M. Lundberg and S.-I. Lee, "A Unified Approach to Interpreting Model Predictions," in *Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 4765–4774.

[58] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The Impact of Code Review Coverage and Code Review Participation on Software Quality," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2014, pp. 192–201.

[59] T. Mende, "Replication of Defect Prediction Studies: Problems, Pitfalls and Recommendations," in *Proceedings of the International Conference on Predictive Models in Software Engineering (PROMISE)*, 2010, pp. 1–10.

[60] T. Mende and R. Koschke, "Revisiting the evaluation of defect prediction models," in *Proceedings of the International Conference on Predictor Models in Software Engineering (PROMISE)*, 2009, p. 7.

[61] ——, "Revisiting the Evaluation of Defect Prediction Models," *Proceedings of the International Conference on Predictive Models in Software Engineering (PROMISE)*, pp. 7–16, 2009.

[62] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann, "Local Versus Global Lessons for Defect Prediction and Effort Estimation," *Transactions on Software Engineering (TSE)*, vol. 39, no. 6, pp. 822–834, 2013.

[63] T. Menzies, J. Greenwald, and A. Frank, "Data Mining Static Code Attributes to Learn Defect Predictors," *Transactions on Software Engineering (TSE)*, vol. 33, no. 1, pp. 2–13, 2007.

[64] T. Menzies and M. Shepperd, "Special Issue on Repeatable Results in Software Engineering Prediction," *Empirical Software Engineering (EMSE)*, pp. 1–17, 2012.

[65] T. Menzies and T. Zimmermann, "Software Analytics: So What?" *IEEE Software*, no. 4, pp. 31–37, 2018.

[66] K. W. Miller and D. K. Larson, "Agile software development: human values and culture," *IEEE Technology and Society Magazine*, vol. 24, no. 4, pp. 36–42, 2005.

[67] T. Miller, "Explanation in artificial intelligence: Insights from the social sciences," *Artifical Intelligence*, vol. 267, pp. 1–38, 2019.

[68] K. Mullen, D. Ardia, D. L. Gil, D. Windover, and J. Cline, "DEoptim: An R package for global optimization by differential evolution," *Journal of Statistical Software*, vol. 40, no. 6, pp. 1–26, 2011.

[69] N. Nagappan, T. Ball, and A. Zeller, "Mining Metrics to Predict Component Failures," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2006, pp. 452–461.

[70] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan, "Heterogeneous Defect Prediction," *Transactions on Software Engineering (TSE)*, p. In Press, 2017.

[71] A. Natekin and A. Knoll, "Gradient boosting machines, a tutorial," *Frontiers in Neurorobotics*, vol. 7, p. 21, 2013.

[72] E. A. Nelson, "Management Handbook for the Estimation of Computer Programming Costs," SYSTEM DEVELOPMENT CORP SANTA MONICA CA, Tech. Rep., 1967.

[73] A. Okutan and O. T. Yıldız, "Software Defect Prediction using Bayesian Networks," *Empirical Software Engineering (EMSE)*, vol. 19, no. 1, pp. 154–181, 2014.

[74] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-Grained Just-In-Time Defect Prediction," *Journal of Systems and Software (JSS)*, 2018.

[75] T. L. Pedersen and M. Benesty, "lime: Local Interpretable Model-Agnostic Explanations. R package version 0.4.0," *Software available at URL: https://cran.r-project.org/web/packages/lime*.

[76] D. Petkovic, M. Sosnick-Pérez, K. Okada, R. Todtenhoefer, S. Huang, N. Miglani, and A. Vigil, "Using the random forest classifier to assess and predict student learning of Software Engineering Teamwork," in *The Frontiers in Education Conference (FIE)*, 2016, pp. 1–7.

[77] J. R. Quinlan, "Simplifying Decision Trees," *International Journal of Man-machine Studies*, vol. 27, no. 3, pp. 221–234, 1987.

[78] ——, *C4.5: Programs for Machine Learning*, 1993.

[79] F. Rahman and P. Devanbu, “Ownership, experience and defects: a fine-grained study of authorship,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2011, pp. 491–500.

[80] ——, “How, and Why, Process Metrics are Better,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2013, pp. 432–441.

[81] G. K. Rajbahadur, S. Wang, Y. Kamei, and A. E. Hassan, “The Impact of Using Regression Models to Build Defect Classifiers,” in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2017, pp. 135–145.

[82] G. D. P. Regulation, “Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the Protection of Natural Persons with regard to the Processing of Personal Data and on the Free Movement of such Data, and Repealing Directive 95/46,” *Official Journal of the European Union (OJ)*, vol. 59, no. 1-88, p. 294, 2016.

[83] M. Ribeiro, “lime: Explaining the predictions of any machine learning classifier,” *Github Repository https://github. com/marcotcr/lime*, 2016.

[84] M. T. Ribeiro, S. Singh, and C. Guestrin, “Model-agnostic Interpretability of Machine Learning,” *arXiv preprint arXiv:1606.05386*, 2016.

[85] ——, “Why should I trust you?: Explaining the Predictions of Any Classifier,” in *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDDM)*, 2016, pp. 1135–1144.

[86] D. Rodríguez, R. Ruiz, J. C. Riquelme, and J. S. Aguilar-Ruiz, “Searching for Rules to Detect Defective Modules: A Subgroup Discovery Approach,” *Information Sciences*, vol. 191, pp. 14–30, 2012.

[87] W. C. Salmon, *Scientific explanation and the causal structure of the world*. Princeton University Press Princeton, N.J, 1984.

[88] E. Shihab, “An Exploration of Challenges Limiting Pragmatic Software Defect Prediction,” Ph.D. dissertation, Queen’s University, 2012.

[89] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim, “Reducing Features to Improve Code Change-Based Bug Prediction,” *Transactions on Software Engineering (TSE)*, vol. 39, no. 4, pp. 552–569, 2013.

[90] M. Staniak and P. Biecek, “Explanations of Model Predictions with live and breakDown Packages,” *arXiv preprint arXiv:1804.01955*, 2018.

[91] R. Storn and K. Price, “Differential Evolution–A simple and efficient heuristic for global optimization over continuous spaces,” *Journal of Global Optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[92] C. Strobl, A.-L. Boulesteix, T. Kneib, T. Augustin, and A. Zeileis, “Conditional Variable Importance for Random Forests,” *BMC Bioinformatics*, vol. 9, no. 1, p. 307, 2008.

[93] M. Tan, L. Tan, S. Dara, and C. Mayeux, “Online Defect Prediction for Imbalanced Data,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, vol. 2, 2015, pp. 99–108.

[94] C. Tantithamthavorn, “ScottKnottESD : The Scott-Knott Effect Size Difference (ESD) Test. R package version 2.0,” *Software available at URL: https://cran.r-project.org/web/packages/ScottKnottESD*.

[95] ——, “Towards a Better Understanding of the Impact of Experimental Components on Defect Prediction Modelling,” in *Companion Proceeding of the International Conference on Software Engineering (ICSE)*, 2016, pp. 867—-870.

[96] C. Tantithamthavorn and A. E. Hassan, “An Experience Report on Defect Modelling in Practice: Pitfalls and Challenges,” in *In Proceedings of the International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2018, pp. 286–295.

[97] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto, “The Impact of Class Rebalancing Techniques on The Performance and Interpretation of Defect Prediction Models,” *Transactions on Software Engineering (TSE)*, 2019.

[98] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto, “The Impact of Mislabelling on the Performance and Interpretation of Defect Prediction Models,” in *Proceeding of the International Conference on Software Engineering (ICSE)*, 2015, pp. 812–823.

[99] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, “Automated Parameter Optimization of Classification Techniques for Defect Prediction Models,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2016, pp. 321–332.

[100] ——, “Comments on “Researcher Bias: The Use of Machine Learning in Software Defect Prediction”,” *Transactions on Software Engineering (TSE)*, vol. 42, no. 11, pp. 1092–1094, 2016.

[101] ——, “An Empirical Comparison of Model Validation Techniques for Defect Prediction Models,” *Transactions on Software Engineering (TSE)*, vol. 43, no. 1, pp. 1–18, 2017.

[102] ——, “The Impact of Automated Parameter Optimization on Defect Prediction Models,” *Transactions on Software Engineering (TSE)*, p. In Press, 2018.

[103] R. C. Team and contributors worldwide, “stats : The R Stats Package. R Package. Version 3.4.0.”

[104] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, “Revisiting Code Ownership and its Relationship with Software Quality in the Scope of Modern Code Review,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2016, pp. 1039–1050.

[105] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto, “Who Should Review My Code? A File Location-based Code-reviewer Recommendation Approach for Modern Code Review,” in *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 141–150.

[106] A. Tosun and A. Bener, “Reducing False Alarms in Software Defect Prediction by Decision Threshold Optimization,” in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2009, pp. 477–480.

[107] J. Van Bouwel and E. Weber, “Remote causes, bad explanations?” *The Journal for the Theory of Social Behaviour*, vol. 32, pp. 437–449, 2002.

[108] S. Wang, T. Liu, J. Nam, and L. Tan, “Deep Semantic Feature Learning for Software Defect Prediction,” *Transactions on Software Engineering (TSE)*, 2018.

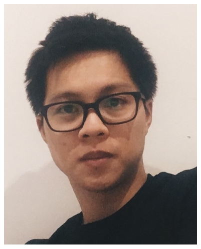
[109] T. Wang, Z. Zhang, X. Jing, and L. Zhang, “Multiple Kernel Ensemble Learning for Software Defect Prediction,” *Automated Software Engineering*, vol. 23, no. 4, pp. 569–590, 2016.

[110] F. Wilcoxon, “Individual Comparisons by Ranking Methods,” in *Breakthroughs in statistics*, 1992, pp. 196–202.

[111] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, “Effort-Aware Just-In-Time Defect Prediction: Simple unsupervised models could be better than supervised models,” in *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2016, pp. 157–168.

[112] S. Yathish, J. Jiarpakdee, P. Thongtanunam, and C. Tantithamthavorn, “Mining Software Defects: Should We Consider Affected Releases?” in *In Proceedings of the International Conference on Software Engineering (ICSE)*, 2019, p. To Appear.

[113] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, “Cross-project Defect Prediction,” in *Proceedings of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2009, pp. 91–100.



Jirayus Jiarpakdee is a Ph.D. candidate at Monash University, Australia. His research interests include empirical software engineering and mining software repositories (MSR). The goal of his Ph.D. is to apply the knowledge of statistical modelling, experimental design, and software engineering to improve the explainability of defect prediction models.

Chakkrit Tantithamthavorn is an ARC DECRA Fellow and a lecturer at the Faculty of Information Technology, Monash University, Australia. His work has been published at several top-tier software engineering venues (e.g., TSE, ICSE, EMSE). His research interests include empirical software engineering and mining software repositories (MSR). He received the B.E. degree from Kasetsart University, Thailand, the M.E. and Ph.D. degrees from NAIST, Japan. More about Chakkrit and his work is available online at http://chakkrit.com.

Hoa Khanh Dam is Associate Professor in the School of Computing and Information Technology, University of Wollongong (UOW) in Australia. He is Associate Director for the Decision System Lab at UOW, heading its Software Analytics research program. His research interests lie primarily in the intersection of Software Engineering and Artificial Intelligence (AI). He develops AI solutions for project managers, software engineers, QA and security teams to improve software quality/cybersecurity and accelerate productivity. His research also focuses on methodologies and techniques for engineering autonomous AI multi-agent systems.

John Grundy is Australian Laureate Fellow and Professor of Software Engineering at Monash University, Australia. He has published widely in automated software engineering, domain-specific visual languages, model-driven engineering, software architecture, and empirical software engineering, amoung many other areas. He is Fellow of Automated Software Engineering and Fellow of Engineers Australia.