Exploring better alternatives to size metrics for explainable software defect prediction

Chenchen Chai[1], Guisheng Fan[1*], Huiqun Yu[1*], Zijie Huang[1*], Jianshu Ding[1] and Yao Guan[1]

[1] Department of Computer Science and Engineering, East China University of Science and Technology, 130 Meilong Road, Shanghai, 200237, China .

*Corresponding author(s). E-mail(s): gsfan@ecust.edu.cn; yhq@ecust.edu.cn; hzj@mail.ecust.edu.cn;
Contributing authors: cc_chai@mail.ecust.edu.cn; djs@mail.ecust.edu.cn; gy_dazzling2333@mail.ecust.edu.cn;

Abstract

Delivering reliable software under the constraint of limited time and budget is a significant challenge. Recent progress in software defect prediction is helping developers to locate defect-prone code components, and allocate quality assurance resources more efficiently. However, practitioners' criticisms on defect predictors from academia are not practical since they rely heavily on size metrics such as Lines-Of-Code (LOC), which over-abstracts technical details and provides limited insights for software maintenance. Thus, the performance of predictors may be overclaimed. In response, based on a state-of-the-art defect prediction model, we (1) exclude size metrics and evaluate the impact on performance, (2) include new features such as network dependency metrics, and (3) explore which ones are better alternatives to size metrics using eXplainable Artificial Intelligence (XAI) technique. We find that excluding size metrics decreases model performance by 1.99% and 0.66% on AUC-ROC in within- and cross-project prediction respectively. The results show that two involved network dependence metrics (i.e., Betweenness and pWeakC(out)) and four other code metrics (i.e., LCOM, AVG(CC), LCOM3 and CAM) could effectively preserve or improve the prediction performance, even if we exclude size metrics. In conclusion, we suggest discarding size metrics and involving the up-mentioned network dependency metrics for better performance and explainability.

Keywords: Software Defect Prediction, Feature Engineering, eXplainable AI, Empirical Software Engineering

1 Introduction

Development of software has to meet certain quality standards within the constraint of time and budget. In recent years, software projects have become more complex and their development iterations are accelerated, causing a dramatic increase in maintenance costs [Reddivari and Raman, 2019]. Thus, practitioners need to allocate limited software quality assurance resources to defect-prone code components [Wan et al., 2020]. Software defect prediction techniques are introduced to automatically help locate, understand [Tantithamthavorn and Jiarpakdee, 2021], or even eliminate [Li et al., 2018] software defects. Since defect predictors have achieved ideal performance, researchers believe such models could be beneficial for software quality [Rajapaksha et al., 2022], if they are integrated into the development cycle properly.

However, software defect prediction models have been found to lack trust among practitioners, as reported by Papenmeier et al. [Papenmeier et al., 2022]. Previous studies, such as the work of Weyuker and Ostrand, attempted to investigate potential reasons for this lack of trust, including data privacy concerns [Weyuker and Ostrand, 2010]. Furthermore, the effectiveness of accurate defect prediction models in enhancing software testing efficacy remains a crucial question, as emphasized by Bell et al. [Bell et al., 2011]. In response to these challenges, federated learning (FL) has emerged as a promising solution. Wang et al. [Wang et al., 2022] stated that FL facilitates the training and building of models without exposing data privacy, enabling clients to train their models with local dataset and jointly builds a global model by transferring models' parameters to the server. Additionally, studies explored effort-aware ranked software modules according to defect density, which helps allocate testing resources more effectively, as shown by Rao et al. [Rao et al., 2021].

Recently, Antinyan from Volvo Car Group [Antinyan., 2021] criticized that defect prediction models were hypnotized by Lines-Of-Code (LOC) metric. The author believed such metrics over-abstracts technical details and provided limited insights for quality assurance. This study indicated that current models lack reliability and user acceptance, necessitating further elucidation on the role of size metrics in determining the final decision. Although size metrics are powerful for boosting performance, which aspects hidden behind LOC are unable to understand [Alpernas et al., 2020]. Moreover, such metrics could be manipulated by slightly modifying source codes, making the model less robust and reliable, e.g., by applying the T1: AddDeadCode transformation in the study [Henkel et al., 2022]. Indeed, oversized code indicates bad design which

may lead to buggy code. Zhang et al. [Zhang et al., 2017] found that excessive numbers of calls to other modules and lines of code are related to defect-proneness. However, oversize is not always a valid signal of defect, since Koru [Koru et al., 2009] found that different from the general belief that the value of size metrics and the emergence of defects are correlated, their study indicated that smaller modules also deserve more attention. Thus, from the perspective of practitioners, the involvement of size metrics is questionable.

The challenge has been met with a promising solution known as Explainable Artificial Intelligence (XAI). By generating feature importance for each local prediction, XAI provides insights into how models make decisions, by enhancing their interpretability [Cambria et al., 2023], and their outputs were stable according to recent empirical studies [Jiarpakdee et al., 2022, Yang et al., 2021b]. Unfortunately, such explanations could still be unreasonable [Gao et al., 2022], thus they would hinder the trust of developers in models. In contrast to the up-mentioned study that aimed to identify the causes of defects, which is not the primary objective of a defect prediction model, it is our conviction that the XAI outputs of such models should align with the primary expectations of the practitioners who employ them [Jiarpakdee et al., 2021b], i.e., it should explain why a code component is defect-prone or not. In contrast to empirical findings indicating that XAI outputs are deemed helpful by a substantial number of practitioners, our investigation reveals that there is potential for further enhancement.

Responding to the call of "dehypnotizing" [Antinyan., 2021] defect prediction models,we intend to cover the aspects that are measured by size metrics using other metrics. By doing so, we seek to reveal the performance and model behavior without the involvement of size metrics. Size metrics such as LOC summarize various aspects of code components about development effort, component importance, code structure (e.g., cohesion), code complexity, and so on. We perform an empirical case study on the gold-standard PROMISE dataset [Jureczko and Madeyski, 2010] that contains 30 open-source projects. First, we exclude size metrics and evaluate the performance impact on both within- and cross-project scenarios. Second, we preserve and introduce new metrics (e.g., network dependency metrics for code importance) related to the up-mentioned aspects that size metrics may summarize. Third, we explore the alternatives to size metrics using a model-agnostic XAI technique called SHAP (SHapley Additive ExPlanations) based on the feature importance of metrics. In conclusion, we propose the alternatives to size metrics for more explainable defect prediction.

The major contributions of our work are as follows.

(1) To the best of our knowledge, this is the first study to explore alternative metrics to size metrics responding to the concerns of practitioners.
(2) We reveal the explainability and practicability of size metrics, as well as the impact of removing them from models, to validate previous empirical observations of the importance of size metrics.

(3) We exploit SHAP to explain how and why certain metrics could be good alternatives to size metrics, and we propose suggestions for researchers and practitioners.
(4) Our experimental code and results have been shared in GitHub[1] for replication and further studies.

The remainder of this paper is organized as follows. Section 2 summarizes related work in software defect prediction and XAI. Section 3 introduces dataset construction. Section 4 proposes the research questions and research methods, while Section 5 analyses and discusses the experimental results. Section 6 outlines the threats to validity. Section 7 concludes this paper and introduces future work.

2 Related work

2.1 Metrics used in defect prediction

Metrics used traditionally for defect prediction were generally classified as code metrics and process metrics [Yang et al., 2021a].

2.1.1 Code metrics

Code metrics were primarily designed based on code size and complexity. In early defect prediction studies, McCabe metrics and Halstead metrics were used by researchers to predict defects [McCabe, 1976]. Since object-oriented design had become a major programming principle in recent years, metrics that capture characteristics such as inheritance and encapsulation have grown in popularity. Chidamber et al. [Chidamber and Kemerer, 1994] proposed CK metrics based on object-oriented development design and calculated metrics such as WMC, DIT, NOC, CBO, RFC, and LCOM from the perspectives of definition and evaluation. Jureczko et al. [Jureczko and Madeyski, 2010] proposed a dataset called PROMISE consisting of 20 object-oriented metrics and made it available online. This dataset has become the gold standard for software defect prediction and is also used in our study.

2.1.2 Process metrics

Many studies had designed process metrics to predict software defects, based one various characteristics of the software development process. These metrics were commonly categorized into three groups [Yang et al., 2021a], (1) code churns, (2) developer information and (3) the organization of the project team. Code churns reflect the changes made to the program code by the developer over time. Nagappan [Nagappan and Ball, 2005] proposed the use of relevant code churns to predict the defect density in a system and demonstrated that the metrics achieve high accuracy. Qiao et al. [Yu et al., 2020] introduced two new process metrics to predict defects based on the defect

[1] https://github.com/T-riple-C/code-SNA

rate of historical software packages and the degree of class change. They concluded that these metrics were better than those based on code churns and traditional code metrics for defect prediction. For developer aspect, Weyuker et al. [Weyuker et al., 2008] investigated the impact of the number of developers on defect prediction and showed that it did not significantly affect performance. In addition, Nagappan et al. [Nagappan et al., 2008] explored the influence of the team organizational structure on the tendency for defects in the software development process, proposing eight metrics based on factors such as organizational scale and group change. They found that using organizational structure metrics for defect prediction had a higher contribution than traditional code metrics and code churns.

2.1.3 SNA metrics

Understanding the structure and interconnections of software components is crucial for ensuring the overall quality of software systems. In software engineering, SNA (Social Network Analysis) can be used to analyze relationships between developers and projects. Numerous studies had applied it to defect prediction. Nguyen et al. emphasized the significance of dependency network metrics in influencing post-release failure [Nguyen et al., 2010]. Zimmerman et al. [Zimmermann and Nagappan, 2008] suggested that analyzing projects' dependency network through network analysis can extract dependency relationships and construct network dependency metrics. Results showed that utilizing these metrics along with code metrics can enhances the effectiveness of prediction models.

2.2 Empirical studies on quality of features and defect dataset label

2.2.1 Quality of features used in defect prediction

Defect dataset often suffered from data class imbalance [Song et al., 2019] and feature redundancy [Ghotra et al., 2017], which can negatively impact prediction model performance. Defective and non-defective dataset were often naturally imbalanced. In terms of data balancing, data class imbalance learning algorithms could be classified into sampling-based methods, cost matrix-based training data, stacking learning methods, and problem-specific ones. Song et al. [Song et al., 2019] compared and analyzed the impact of the interactions between class imbalance algorithms, classifiers, and metrics on defect prediction. They concluded that dataset with a high degree of class imbalance had a greater impact on the prediction results. They also suggested that better results could be achieved by selecting appropriate class imbalance algorithms for different classifiers. As for the problem of feature redundancy, Jain et al. [Jain and Saha, 2021] combined stacking learning with a comparative analysis of hybrid feature selection methods based on three types of filters, wrappers, and embeddings. The results showed that the use of hybrid feature selection methods could improve model performance and stacking

learning methods performed better than individual classifiers. Jiarpakdee et al. [Jiarpakdee et al., 2018] proposed a feature selection method called AutoSpearman for the purpose of model interpretation. The results showed that the method could produce a highly consistent subset of features and improve model interpretability.

2.2.2 Quality of defect dataset label

In addition to the defect dataset itself, it is crucial to consider the quality of defect dataset labeling. This factor plays a vital role in both the predictive outcomes of the models and the acceptance of practitioners. Jurecako et al. [Jureczko et al., 2019] provided an example of code changes commit, analyzed the method that maps the defects to the code changes that resolve them, and identified the status of the defects as uncertain. Meanwhile, labeling issues caused by classical algorithms such as SZZ had been highlighted by numerous studies. For a instance, a study [Herbold et al., 2022] showed that inaccuracies in defect labels produced by the SZZ algorithm can severely compromise the effectiveness of the defect prediction. In our experiments, we utilize the gold-standard dataset [Jureczko and Madeyski, 2010], which is one of the most popular dataset has been extensively studied on defect prediction. Furthermore, the defect labels utilized in the study were gathered via a tool named BugInfo, which underwent comprehensive functional testing, as indicated by the original author.

2.3 XAI and its applications in software defect prediction

2.3.1 XAI in software engineering

The development of machine learning, particularly deep learning methods, has led to improved model performance. However, as models become more complex, understanding their behavior and interpreting their results has become more challenging. The comprehensibility and interpretability of models, however, are crucial factors that can impact users' trust and acceptance [Hoffman et al., 2018], which is why the demand for XAI technology has surged in recent years. The goal of XAI is to make the black-box model easier for users to comprehend and interpret [Burkart and Huber, 2021]. Molnar et al. [Molnar, 2020] classified machine learning model interpretability methods based on several criteria, i.e., model-specific and model-agnostic. Model-specific interpretation methods are only applicable to specific models. In the context of defect prediction, different interpretation methods are used based on the prediction model. For instance, decision rules, decision results, and feature weights can explain simple linear or Decision Tree models. Model-agnostic interpretation methods, on the other hand, can be used to interpret any machine learning model without being constrained by the interpretation format or input data.

2.3.2 Applications in software defect prediction

As previously mentioned, this section briefly introduces two different types of XAI methods and their applications. For model-specific methods, Jiarpakdee et al. [Jiarpakdee et al., 2021a] used Logistic Regression and Random Forest to construct defect models. They employed the ANOVA method for the Logistic Regression model and used the built-in GINI importance for the Random Forest model. However, these model-specific interpretation methods are not universal and may not be sufficient for interpreting individual prediction results. Meanwhile, there is no certain way to interpret some complex black-box models [Jiarpakdee et al., 2022]. A recent study [Yang et al., 2021b] found that the feature importance generated by model-specific interpretation methods could be unstable. They suggested the use of model-agnostic methods to generate model interpretations. For model-agnostic methods, Ribeiro et al. [Ribeiro et al., 2016] proposed Local Interpretable Model-agnostic Explanations (LIME), which is a local agent model that can be used to interpret individual prediction instances of black-box machine learning models. Staniak et al. [Staniak and Biecek, 2018] proposed BreakDown, which is designed to decompose models into specific variables based on a greedy policy decomposition model to predict specific variables in a partially attributable feature space. Aleithan et al. [Aleithan, 2021] applied LIME and iBreakDown methods to a commit granularity defect prediction model and found that the results of these interpretation techniques for defect prediction were not consistent with manual analysis. Lundberg et al. [Lundberg and Lee, 2017] proposed SHAP, which interprets the results of local instances by calculating the contribution of each feature to the predictions. Furthermore, the authors also provided the treeSHAP [Lundberg and Lee, 2017] method for tree-based models to improve computational speed. In our experiment, we utilized SHAP to interpret the model's prediction results and feature importance. It has been tested on defect prediction and has produced stable results.

3 Dataset construction

In this section, we explain our motivation for finding alternatives to size metrics, followed by an introduction to the projects and metrics of experimental dataset.

3.1 The gold-standard dataset and code metrics

Our experiment is based on a gold-standard dataset provided by Jureczko et al. [Jureczko and Madeyski, 2010], which also includes several object-oriented metrics. Our experiments require the extraction of source code dependency, thus we only consider the open-source projects. As demonstrated in Table 1, we finally selected 30 versions from 11 projects.

The dataset used in this study contains 20 object-oriented metrics, as listed in Table 2. The aim of this study requires the exclusion of size metrics. Thus we

Table 1 Experimental projects

Project	Version	Defect Rate	Project	Version	Defect Rate
Ant	1.4	15.09%	Ivy	2.0	8.39%
	1.5	7.98%	Poi	1.5	47%
	1.6	17.59%		2.0	9.59%
	1.7	15.57%		2.5	53.22%
JEdit	3.2.1	17.71%		3.0	52.92%
	4.0	12.38%	Tomcat	6.0.39	6.63%
	4.1	12.27%	Velocity	1.4	65.63%
	4.2	5.96%		1.5	57.72%
Log4j	1.0.4	22.22%		1.6.1	29.89%
Lucene	2.0	31.60%	Xalan	2.4.0	12.76%
	2.2	37.80%		2.5.0	40.95%
	2.4	37.87%		2.6.0	35.13%
Synapse	1.0	9.88%	Xerces	1.2.0	13.79%
	1.1	26.09%		1.3.0	12.66%
	1.2	31.97%		1.4.4	65.13%

should propose a protocol to identify size metrics firstly. We exclude a metric if it is (1) used to measure the size of a code component according to their names, i.e., LOC, NOC, and NPM, and (2) fitted with the definition of size metrics according to the metric classification framework of [Azeem et al., 2019], i.e., MOA and AMC. To clarify, the name of some metrics may not be related with code size that they actually measure. More specifically, MOA is calculated by the count of the number of class fields whose types are user defined classes, and AMC measures the average method size for each class [Jureczko and Madeyski, 2010]. Consequently, the set of size metrics identified in our study involves five metrics, called LOC, AMC, NPM, NOC, and MOA.

Table 2 Code metrics

Metrics	Definition	Code Metrics	Definition
WMC	Weighted methods per class	LOC	Lines of code
DIT	Depth of inheritance	DAM	Data access metric
NOC	Number of children	MOA	Measure of aggregation
CBO	Coupling between object classes	MFA	Measure of Functional Abstraction
RFC	Response for a class	CAM	Cohesion among methods of class
LCOM	Lack of cohesion in methods	LCOM3[1]	Lack of cohesion in methods
CA	Afferent couplings	CBM	Coupling between methods
CE	Efferent couplings	AMC	Average method complexity
NPM	Number of public methods	MAX(CC)	The greatest value of McCabe's cyclomatic complexity
IC	Inheritance coupling	AVG(CC)	The greatest value of McCabe's cyclomatic complexity

[1]Refers to LCOM3 suggested by Henderson-Sellers.

3.2 Potential alternatives to size metrics

Since size metrics' involvement in defect prediction models is questionable and they are not preferred by practitioners, we intend to explore whether and which alternative metrics can replace size metrics while not hindering performance.

Table 3 Aspects measured by non-size metrics from the PROMISE dataset

Aspect	Metrics
Effort	?
Importance	?
Code Structure (Cohesion)	LCOM, LCOM3, CAM
Code Structure (Coupling)	CBO, CA, CE, IC, CBM, RFC, DAM
Inheritance Complexity	DIT, IC, WMC
Code Complexity	MFA, MAX(CC), AVG(CC)

Apart from code size, size metrics (e.g., LOC) are often applied to measure various aspects of code component features including (1) effort of development [Yu et al., 2019, Alpernas et al., 2020], (2) code structure [Alpernas et al., 2020, Jureczko and Madeyski, 2010], (3) complexity [Alpernas et al., 2020], and (4) importance of code components, i.e., interactions with other components [Alpernas et al., 2020]. Table 3 classifies the non-size metrics extracted from Table 2 which also concern these aspects. In terms of effort, classes with large LOC may be relatively important and may represent more effort put into the code by the developer. In terms of structure, the quality of the code itself has been captured by object-oriented metrics such as cohesion and coupling. In terms of complexity, for example, the number of functions in the source code can stand in for it [Oram and Wilson, 2011]. In terms of components' importance, e.g. interactions with different components reflect it to some extent.

From Table 3, we can find that if size metrics are excluded, importance and effort metrics are missing from the dataset. Apart from LOC [Yu et al., 2019], effort measures rely on version control system statistics within a long range of time [Alpernas et al., 2020], which are not applicable to our dataset. Thus, the importance of code components are focused. We introduce network dependency metrics, which can capture the characteristics (e.g., centrality, weight) of nodes (i.e., code components) according to the dependencies between software modules. Dependency networks are graphs whose nodes are software modules and edges are dependencies between these modules [Gong et al., 2022]. The network dependency metrics could be calculated based on a network of code dependencies by social network analysis (SNA) [Zimmermann and Nagappan, 2008]. Several studies indicated that combining the SNA and the code metrics can improve the performance of the defect prediction model [Nguyen et al., 2010], while other sources argued that the SNA metrics are only effective in certain scenarios [Premraj, 2011]. Although the

conclusions of various studies on the utility of dependency network metrics for defect prediction are inconsistent, under the condition that we need extra metrics to make up for the lost information from excluding size metrics. This study focus on exploring their potential of replacing their functionalities of representing code component importance.

3.3 The introduced SNA metrics

In our study, we focus on binaries of individual projects [Shin et al., 2012] as well as SNA metrics that are obtained by applying the SNA methods to extract information on dependency networks of software projects.

In the analysis of social networks, there are mainly two perspectives, i.e., the global network and the ego network [Ma et al., 2016]. The global network involves the entire dependency graph and allows for a measure of the importance of the nodes in the entire software system. The ego network is usually composed of a node and its neighbor nodes. For the global network, our experiment focuses on two important features, i.e. structural holes and centrality [Jun, 2014]. Structural holes index generally takes effective size, efficiency, constraint, hierarchy and so on into consideration. For centrality, we involve four standard measures described as Table 4. In the ego network, for each node, there are three types of ego network, in, out, and undirected. The in (out) ego network only cares about the in (out) directed dependence between the ego node and its neighboring nodes. The undirected ego network contains the dependencies of both directions. Ego network basic measures calculated include size, ties, pairs, density, average distance, diameter, weak component, reach, broker and so on. Our experiment involves 9 SNA metrics in the global network and 24 SNA metrics in the ego network.

The process of extracting and calculating SNA metrics is as follows. First, we retrieve source code from their relative open-source version control systems according to the project information provided by the dataset. Second, we extract the dependency information of the source code by exploiting a tool called Understand[2]. The experiment employed version 5.1 of the tool, which has been purposefully developed for source code analysis and tracking. It enables developers to explore system architecture and design, comprehend the interconnections between various code components and evaluate the complexity of the code. We extract the dependency network for each version of each project. Third, we calculate SNA metrics for each node in terms of both the ego network and global network by using a tool named Ucinet[3]. This research employ Ucinet 6.5.0 for conducting social network analysis, which is a tool widely used in the social sciences. Moreover, it can also be applied to relational analysis of software systems. It is utilized in the study to analyze the outcomes from the Understand and extract relevant SNA metrics to support the experiment. Finally, we integrate the acquired SNA metrics into the original dataset. The SNA metrics involved are displayed in Table 4. To be consistent with the

[2]https://scitools.com/
[3]https://sites.google.com/site/ucinetsoftware/home

granularity of the original dataset, we extract class-level dependency relationships. In Table 4, the first column shows the two perspectives of networks, and the second column means different indicators in the dependency networks. Moreover, the third column displays the SNA metrics involved in our study.

Table 4 SNA metrics

Type	Indicators	Metrics
EN	In Out Un	Size, Ties, Pairs, Density, AvgDist, Diameter, nWeakComp pWeakComp, 2StepReach, 2StepP, ReachEfficency Brokerage, nBrokerage, EgoBetween, nEgoBetween
	Structural holes	EffSize, Efficiency, Constraint, Hierarchy, Degree EgoBetween, Ln(Constraint), Indirects, Density
GN	Structural holes	EffSize, Efficiency, Constraint, Hierarchy, Indirect
	Centrality	Degree, Closeness, Betweenness, Eigenvector

4 Empirical study design

The aim of this study is to identify more effective alternatives to size metrics, enhance the explainability of prediction models, and increase users' trust in these models. In pursuit of these objectives, we formulate three research questions. This section chiefly expounds upon the research questions, outlines the critical steps and methods involved in the experimental process.

4.1 Research questions

RQ1: Are size metrics essential for constructing a well-performed defect prediction model?

Motivation: Since size metrics could be significantly important metrics for defect prediction, we assess to what extent the impact on performance is if size metrics are removed. Meanwhile, we intend to recover the performance loss caused by excluding them in further experiments, and thus a baseline performance is needed for comparison.

Approach: To assess the impact of size metrics on performance, we should generate two defect prediction models built with and without size metrics in advance. The size metrics to remove are introduced in Section 3.2. The experimental process for model generation is showed on Fig.1. We compared the performance of prediction models constracted by with and without size metrics. The results were analyzed using the Scott-Knot Test.

RQ2: Can we improve performance by involving SNA metrics after removing size metrics?

Motivation: Based on Table 4, we involve SNA metrics and explore whether they can recover model performance after removing size metrics. We also examine whether the model performance is acceptable (e.g., AUC-ROC > 0.7 [Fawcett, 2006]) for an explanation.

Approach: To analyze whether performance can be improved by involving SNA metrics after removing size metrics, we generate another prediction model built with SNA and without size metrics. Except for dataset, the experimental process for model generation is similar. We compare the contribution of metrics and performance of models with the results of RQ1. Additionally we analyze the results using the Scott-Knot Test.

RQ3: Which metrics are better alternatives to size metrics?

Motivation: We explain and compare the behavior of models built in RQ1 and RQ2 by exploiting SHAP to reveal which metrics are making up for the absence of size metrics. We further discuss and conclude which metrics are better alternatives to size metrics.

Approach: To find the alternative metrics to size metrics, we should explore the impact and contribution of the features on the two prediction models constructed in RQ1 and one model constructed in RQ2. We use SHAP to analyze the difference and relationship between the potential alternative metrics on the model prediction results by comparing the feature importance and feature interaction with size metrics. Specifically, we use SHAP to obtain the Shapley feature importance values and summary plots of the prediction instances. To further explore the impact of feature interactions on model prediction, the experiment also analyzes the dependence plot using SHAP, focusing on the feature interactions of the size metrics as well as alternative metrics.

4.2 Methodology

Fig.1 depicts the experiment design, comprising three part: dataset construction, expounded upon in Section 3, defect prediction, and explanation. This section particularly accentuates the fundamental phases and approaches in constructing and interpreting the model, namely: 1) feature selection, 2) model validation, 3) data sampling, 4) classifier selection, 5) performance evaluation, and 6) introduction of SHAP.

4.2.1 Feature selection

The correlation and multicollinearity among features can significantly impact the performance [Katrutsa and Strijov, 2017] and explainability [Marcilio and Eler, 2020] of the prediction model. To select a reasonable set of features before constructing the model, we use AutoSpearman [Jiarpakdee et al., 2018] to mitigate the correlation between features, which is a feature selection approach aiming at reducing multicollinearity while preserving most features. We apply the implementation available in the Py-Explainer [Pornprasit et al., 2021]. Feature selection is performed for each available combination of metric sets and data validation.

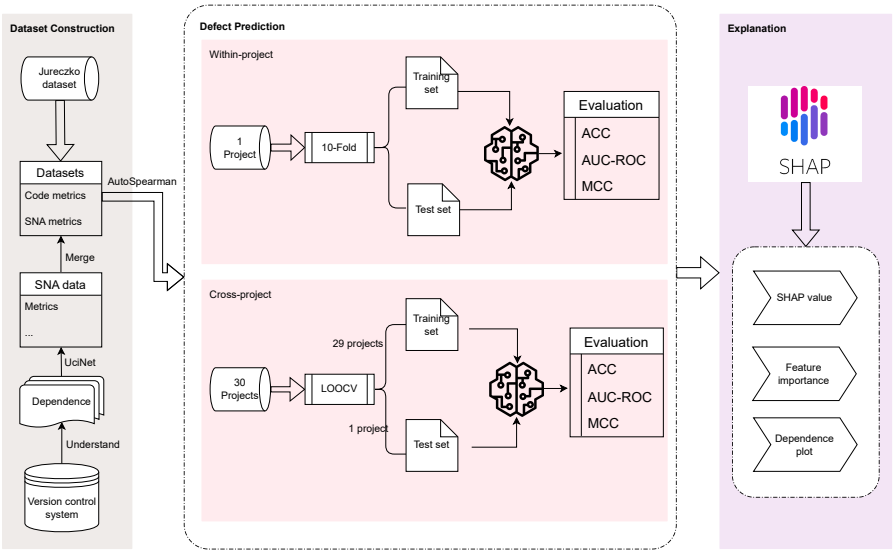


Figure 1 Experimental flow graph

4.2.2 Model validation

The validation approaches in software defect prediction can be classified as both within- and cross-project [Hosseini et al., 2019]. Within-project defect prediction refers to prediction models with training dataset and test dataset from the same project. Cross-project defect prediction typically refers to using the source project data to construct a prediction model to perform defect prediction on the target project data. To address the issue of the cold start problem for new dataset, and to improve the generalizability of prediction models, many efforts have been made by the research community to improve defect prediction in cross-project defect prediction scenarios. For instance, Nam et al. [Nam et al., 2013] proposed a TCA+ technique based on an extension of the TCA to mitigate data distribution differences and applied it to cross-project defect prediction using the feature mapping idea. Our study validates model in cross- and within-project scenarios. In terms of within-project prediction, 10-fold cross-validation is used, where for each project, the dataset is randomly divided into 10 parts, 9 of which are used for training and 1 for testing. Specifically, the process is repeated 10 times. In cross-project validation, we use a project-wide LOOCV (leave-one-out cross-over), i.e., for each iteration, 1 project is selected as the test dataset, and the others are regarded as the training dataset.

4.2.3 Data sampling

The defect prediction dataset are imbalanced naturally (e.g., the distribution of defects fits with the Pareto principle). However, overly imbalanced dataset tend to impose a significant negative impact on model performance [Song et al., 2019]. Therefore, it is necessary to preprocess the imbalanced dataset before constructing the model. We use the SMOTE method, which is an oversampling technique for generating more minority class data by interpolating between the nearest neighbors of the minority class to synthesize the creation of new minority class instances [Naufal and Kusuma, 2019].

4.2.4 Classifier selection

To choose the best-performed classifier, several machine learning algorithms are applied such as RF (Random Forest), LR (Logistic Regression), NB (Naive Bayes), SVM (Support Vector Machine), and XGB (XGBoost) to construct prediction models for defect prediction and compares the experimental results based on evaluation metrics.

4.2.5 Performance evaluation

ACC, AUC-ROC, and MCC are used in the experiment for performance evaluation. ACC is a direct representation of the model's accuracy and is widely used to evaluate model performance. AUC-ROC[Fawcett, 2006] is measured by the area under the ROC curve, whose value is usually between 0.5 and 1. MCC (Matthews correlation coefficient ([Chicco et al., 2021]) is an evaluation metric that measures the relationship between the predicted class and the actual class, and it can yield stable results when applied to imbalanced dataset. To verify whether the experimental results are statistically significant, we exploit SK-ESD (Scott-Knott Effect Size Difference)[Tantithamthavorn et al., 2017] test, which is based on hierarchical cluster analysis and divides the results into statistically distinct groups. The SK-ESD test does not require the input to be normally distributed, since it can correct for the non-normal distribution of the input data. We apply the original SK-ESD implementation available in an R package[4].

4.2.6 SHAP

As we mentioned in Section 2.3.2, SHAP could be exploited to interpret the model's decision, which is an estimation method for Shapley values and based on the game's theoretically optimal Shapley value. Firstly the Shapley value is an alliance game theoretic approach to the average marginal contribution of feature values across all combinations of features. It can be used to illustrate a fair distribution of feature contributions to the total outcome. Moreover, it is applicable to classification and regression problems and can be used to calculate a solution to feature contributions for a single prediction instance

[4] https://github.com/klainfo/ScottKnottESD

of any machine learning model. For example, assuming that there exists a feature j of prediction instance z. The interpretation of the Shapley value for feature j is that the value of the j-th feature contributes Φ_j to the prediction of the instance z. Since the interpretation given by the Shapley value always employs all features, calculating the Shapley value takes a plant of time if the feature combination space is too large. Therefore a number of estimation methods for Shapley values such as SHAP have emerged. The goal of SHAP is to interpret the prediction of instance x by calculating the contribution of each feature to the prediction of x. The interpretation of the Shapley values in the SHAP algorithm is represented as an additive feature imputation method. For example, in a linear model, SHAP specifies the interpretation using the equation (1).

$$g(X) = \phi_0 + \sum_{i=1}^{M} \phi_i x_i \tag{1}$$

SHAP transforms the features of the instance into simple binary features as input and constructs a linear explanatory model g. As in the above equation, $x \in \{0,1\}^M$ is the feature vector, M is the feature space size, and Φ_i is the feature imputation Shapley value for feature i, and Φ_0 is the average predictive value of the model. Specifically, when x_i is equal to 1, it means that the i-th feature of the predicted example x exist in the feature union space. While x_i is equal to 0, it means that the i-th feature of the predicted example x not exist in the feature union space. $\|\Phi_i\|$ is the SHAP feature importance score. Larger $\|\Phi_i\|$ indicates greater influence of the i-th feature on the positive prediction result of the model. In contrast to other local interpretation techniques, the global interpretation of SHAP is consistent with the local interpretation. In addition, the authors proposed an more efficient estimation method based on the tree model called treeSHAP, which can obtain accurate Shapley values or correctly estimate them when features are correlated. It is also utilized for our experiments.

5 Result and discussion

In this section, we introduce experimental results and outline the findings and implications.

5.1 RQ1: Removing size metrics

To enable a more comprehensive comparative analysis of both research questions, in Table 5, we present the experimental outcomes related to RQ1 and RQ2. Table 5 displays metric subsets M1, M2, and M3. Experiments for M1 and M2 were conducted in the context of RQ1, M3 was explored under the purview of RQ2. The detailed account of the metrics M1, M2 and M3 is as follow. M1 represents including size metrics, M2 represents code metrics removing

size metrics, and M3 represents adding SNA metrics. For RQ1, we focus on the scenarios with M1 and M2. In terms of the within-project validation (upper half part of Table 5), the performance of the models constructed without size metrics decreases for all methods except NB for AUC-ROC and ACC. For MCC, the performance of the models decreases for all five algorithms. In terms of the cross-project validation (bottom half part of Table 5), the performance of the predictive models constructed by all five machine learning algorithms decreases for AUC-ROC, ACC, and MCC after removing size metrics.

However, although a decline in performance is observed, the removal of size metrics does not impose a significant impact on AUC-ROC performance (i.e., less than 2% in most cases). The statistical significance of the results will be discussed in subsection 5.2 together with SNA metrics. In within-project prediction, we can still construct good predictors using RF with regards to performance. Size metrics are helpful for defect prediction, but they are not essential for building a well-performed model in the majority of cases.

Finding 1. After removing size metrics, the performance of defect prediction models is very likely to decline in both within- and cross-project validation, and thus size metrics are helpful for defect prediction. However, since the impact to performance is limited, they are not essential for building a well-performed model. Considering the preference of practitioners, we believe size metrics should be removed, and alternative metrics should be involved to make up for the performance decline.

Table 5 Model Performance Results

ML	AUC-ROC			ACC			MCC		
	M1	M2	M3	M1	M2	M3	M1	M2	M3
RF	0.77873	0.76744	**0.81581**	0.77091	0.76213	**0.79179**	0.36186	0.34591	**0.38839**
XGB	0.77934	0.76398	0.79431	0.74885	0.74258	0.76160	0.38085	0.35142	0.38737
SVM	0.70452	0.65267	0.72572	0.68231	0.67109	0.69758	0.25115	0.22376	0.28668
NB	0.71895	0.72904	0.75434	0.65145	0.66922	0.68347	0.28472	0.28256	0.31262
LR	0.73419	0.71363	0.72879	0.70006	0.69139	0.69326	0.31211	0.28082	0.30191
RF	0.64871	0.64216	0.67900	0.62933	0.61909	0.63685	0.19501	0.19229	0.22649
XGB	0.66523	0.66262	0.68680	0.64237	0.63424	0.64091	0.19235	0.19176	0.23171
SVM	0.67443	0.66549	0.67262	0.63817	0.62656	0.61085	0.20932	0.20306	0.20522
NB	0.66201	0.65806	0.68900	0.63815	0.62962	0.64521	0.19127	0.19081	0.23460
LR	0.66706	0.66375	0.67527	0.63912	0.62783	0.61929	0.19944	0.19678	0.21199

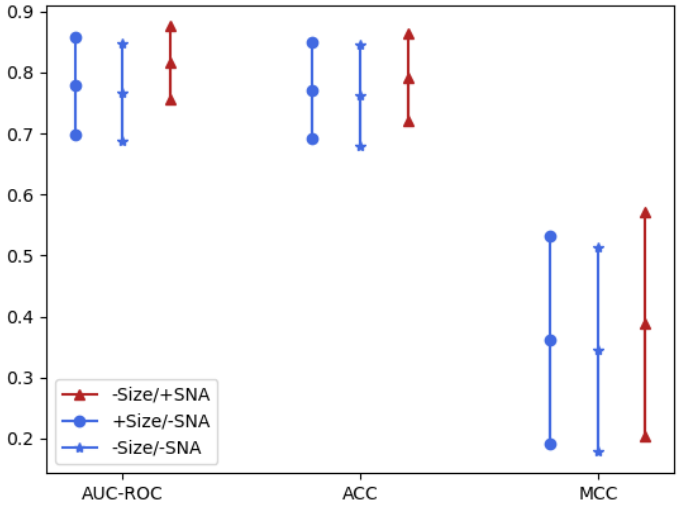
5.2 RQ2: Involving SNA metrics

For RQ2, in Table 5, we focus on the scenarios with M2 and M3. In terms of within-project prediction, for AUC-ROC, ACC and MCC, the performance of the models constructed with SNA metrics compared to those constructed with the removal of the size metrics improves for all five machine learning algorithms. Except for LR, the models constructed with the SNA metrics are

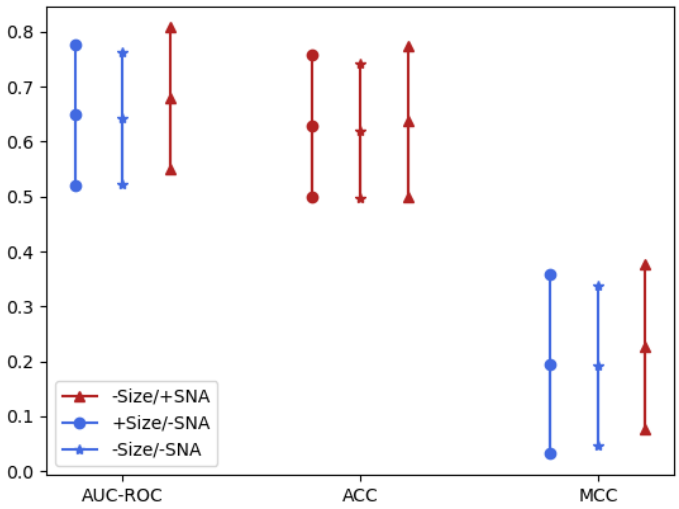
superior to those constructed using only the code metrics. In terms of cross-project prediction, for AUC-ROC and MCC, the performance of the models constructed by adding the SNA metrics compared to those constructed by removing the size metrics is improved for all five machine learning algorithms. For ACC, the models constructed by incorporating the SNA metric demonstrated enhanced performance across all methods, except for SVM, when compared to the models constructed by removing the size metrics. For AUC-ROC and MCC, the performance of the models constructed with the SNA metrics are improved compared to those constructed using only the code metrics, except for the SVM. For ACC, the performance of the models constructed with the SNA metrics is improved compared to those constructed using only the code metric, except for the SVM and LR. The RF model has the best average results compared to other machine learning models, and the prediction results of the random forest model are used for discussion and analysis in the subsequent experiment in RQ3.

Fig. 2 shows the SK-ESD rankings of the three generated models. The horizontal axes represents the performance metrics, i.e., AUC-ROC, ACC, and MCC. The vertical axes corresponds to the results of the SK-ESD. The data point in the middle presents the mean value of SK-ESD and the top one presents the mean value plus standard deviation and the bottom one presents the mean value minus standard deviation. The shape of the data points, as shown in the legend, distinguishes the three different sets of metrics, which are sets including size metrics but without SNA metrics (+Size/-SNA), sets without size metrics and without SNA metrics (-Size/-SNA), and sets without size metrics but including SNA metrics (-Size/+SNA). In addition, the group ranking of the prediction models is marked by different colors, e.g., red for the 1st rank and blue for the 2nd rank. The metric sets marked in red are superior in terms of performance. In within-project validation, as Fig.2(a), the mean value calculated by SK-ESD for the prediction models constructed with the SNA metrics (i.e., the -Size/+SNA metric set) is better than other metric sets. For AUC-ROC and MCC, the SK-ESD group ranking shows that the prediction models constructed with the addition of SNA metrics are superior to those constructed with the other two metric sets. In the cross-project validation, as Fig.2(b), the findings are similar to those described above. However, for ACC, the group rankings of the three models are consistent, which means the performance difference is not statistically significant.

Finding 2. For most machine learning algorithms, in both within- and cross-project prediction, defect prediction models constructed with the SNA metrics are better than those constructed with and without size metrics. In addition, according to the SK-ESD rankings, the models constructed by adding the SNA metrics also outperformed the models constructed by removing the size metrics and using only the code metrics. Thus, after removing size metrics SNA metrics are suggested to involve for prediction.



(a)



(b)

Figure 2 SK-ESD rankings of prediction results. (a) Within-project prediction results, (b) Cross-project prediction results.

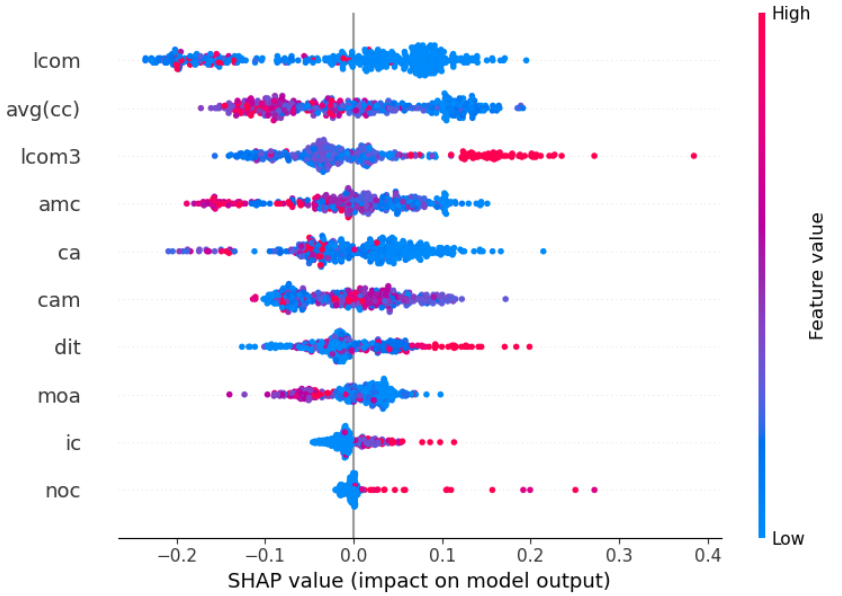
5.3 RQ3: Better alternative metrics to size metrics

In this section, we focus on the 3 prediction models constructed in the previous RQs and analyze feature importance and feature interaction in these models. We assess the feature importance of size metrics and SNA metrics through SHAP value. First of all, we analyze the influence of size metrics and SNA metrics on the prediction results through a summary plot of one project instance. We also focus on the overall feature importance through the mean SHAP value of the aggregated data. Finally, we use dependence plots to explore the effect of the interaction of specific metrics on the prediction results from one instance.

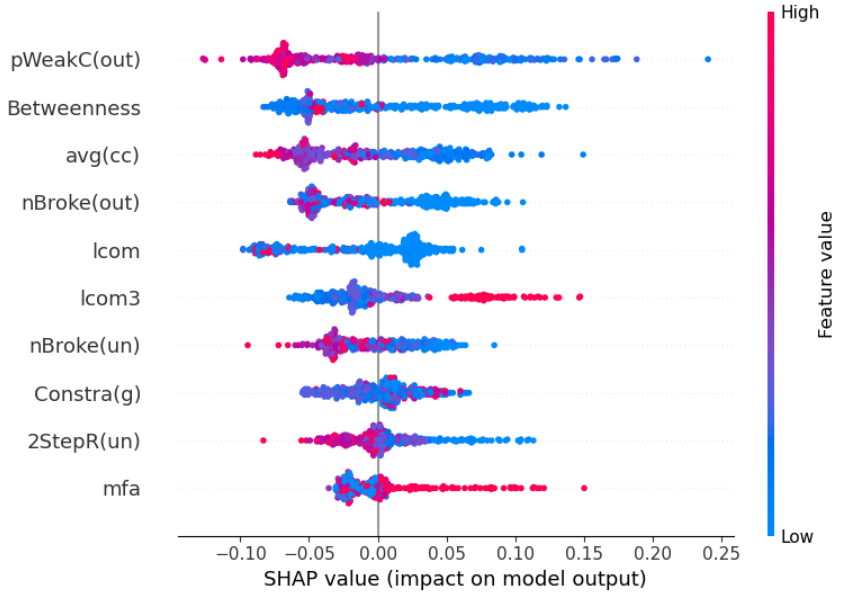
For the jEdit project, we analyze by summary plots in Fig.3 and Fig.4 the importance of the size metrics and SNA metrics, as well as, their impact on the prediction model in two validations. Summary plots of other instances are included in the online appendix [1]. For the summary plot, all sample points are presented and the color bar reflects features value, i.e. red (high) and blue (low). Additionally, a vertical line shows that points emerging along the right (left) are contributing to increasing (decreasing) the defect probability [Esteves et al., 2020]. Fig.3 shows the top 10 important features of experimental results for within-project validation. For Fig.3(a), taking size metric (amc) as an example, the higher amc values (red points), the lower the SHAP values are. That means high amc values decrease the chance of models predicting defects in the instance. For Fig.3(b), the top-ranked metric is pWeakC(out) and it means that this metric is the most influential in the instance. Similarly, the higher pWeakC(out) values (red points), the lower the SHAP values are and higher pWeakC(out) values decrease the chance of models predicting defects. Fig.4 shows the top 10 important features of experimental results for cross-project validation. For Fig.4(a) and Fig.4(b), we can obtain similar summary plot analysis for the same metric i.e., amc and pWeakC(out).

The experimental results of the feature importance rankings under both validations in all projects are shown in Table 6 and Table 7. Both of them are given in descending order of the mean SHAP value. The feature importance of the size metrics is displayed on the left part of the table, and the SNA metrics are on the right. Table 6 shows the results of the comparison in within-project validation, the left side of the table contains two size metrics and the right side of the table contains four SNA metrics. Three SNA metrics are higher in importance ranking compared with other remaining code metrics. Table 7 shows the comparison results in the cross-project. In the top-10 SHAP value importance ranking, the left side of the table contains 3 size metrics and the right side of the table contains 5 SNA metrics.

On the other hand, metrics with higher consistency in the effect of feature interactions on prediction outcomes are more likely to be used as alternative metrics. Therefore, the experiment explores the effect of feature interactions on prediction outcomes with the same instance (jEdit) through dependence plots. Initially, the study examine the impact of a single feature on the prediction outcomes to determine whether the SNA metric and the size metric displayed

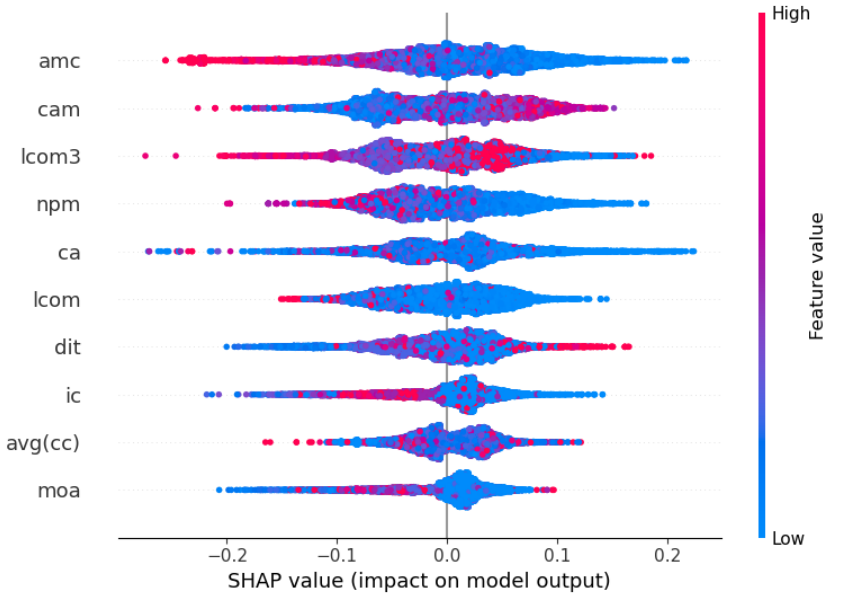


(a)

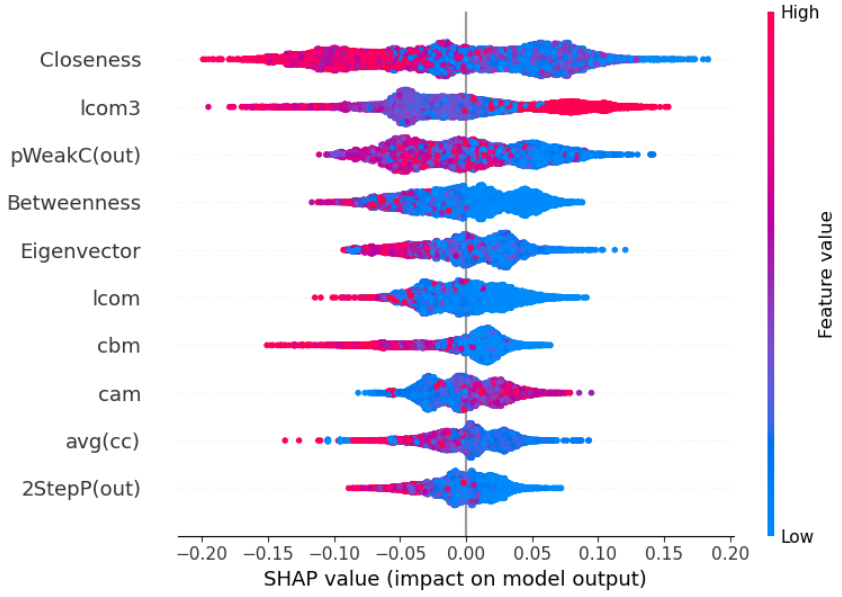


(b)

Figure 3 SHAP summary plot for within-project prediction. (a) Dataset with size metrics, (b) Dataset with SNA metrics and without size metrics.

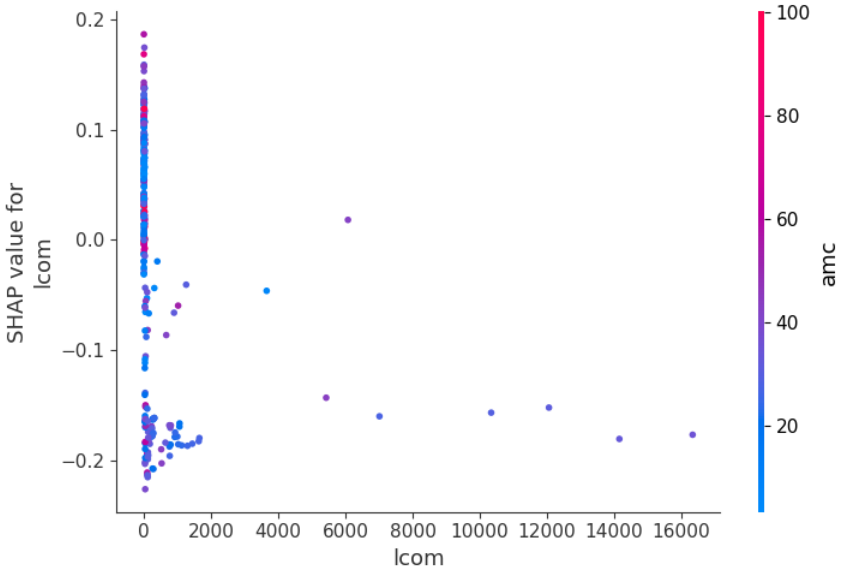


(a)

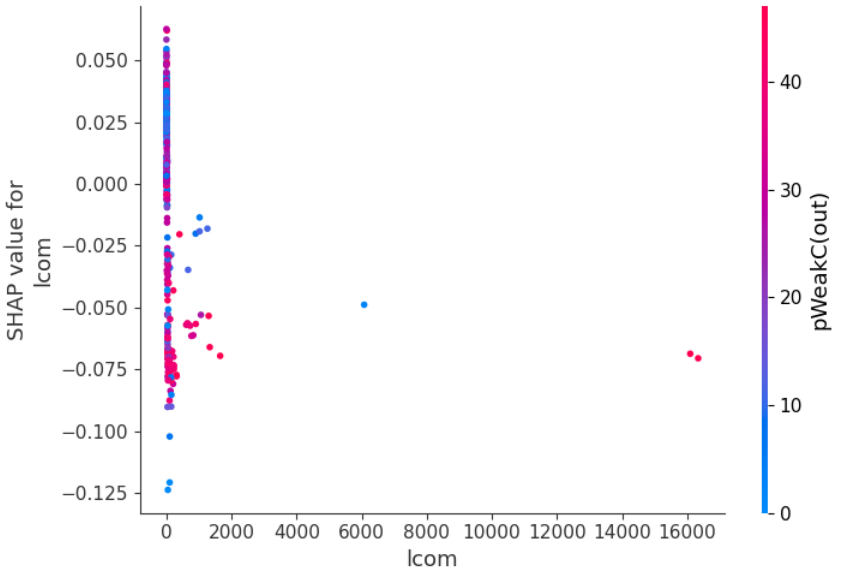


(b)

Figure 4 SHAP summary plot for cross-project prediction. (a) Dataset with size metrics, (b) Dataset with SNA metrics and without size metrics.

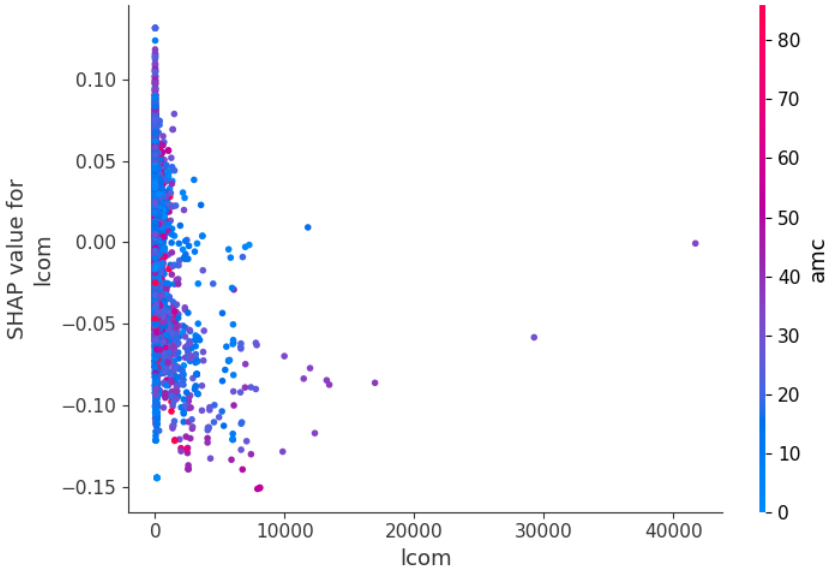


(a)

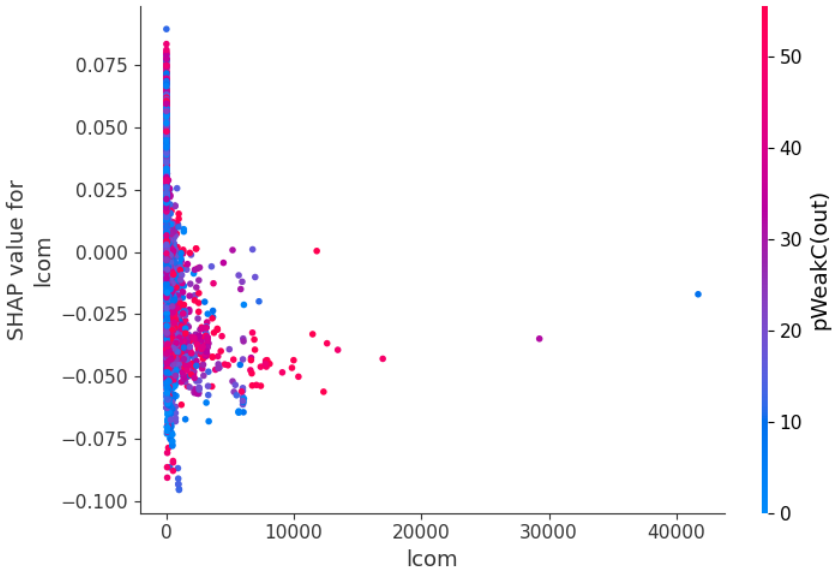


(b)

Figure 5 Two-feature interaction plots. (a) AMC and LCOM in within-project validation, (b) pWeakC(out) and LCOM in within-project validation.



(a)



(b)

Figure 6 Two-feature interaction plots. (a) AMC and LCOM in cross-project validation, (b) pWeakC(out) and LCOM in cross-project validation.

Table 6 SHAP value of all projects (within-project)

Size	SHAP value (mean)	SNA	SHAP value (mean)
AMC	0.073215	**pWeakC(out)**	0.041868
LCOM	0.063169	**Betweenness**	0.039147
CAM	0.061237	**2StepP(out)**	0.035751
LCOM3	0.060809	AVG(CC)	0.034491
AVG(CC)	0.058955	LCOM	0.034378
DAM	0.058013	DAM	0.030776
NPM	0.053092	CAM	0.030754
MFA	0.050671	LCOM3	0.028906
CA	0.044052	CBO	0.027125
CBM	0.041999	**ReachE(un)**	0.024551
DIT	0.034897	Indirec(g)	0.022581
IC	0.024945	2StepR(un)	0.022032
MOA	0.021315	MFA	0.021722
NOC	0.008150	Densit(out)	0.021035

Table 7 SHAP value of all projects (cross-project)

Size	SHAP value (mean)	SNA	SHAP value (mean)
AMC	0.052028	**Closeness**	0.056397
CAM	0.048131	LCOM3	0.042383
NPM	0.046602	**pWeakC(out)**	0.037451
LCOM3	0.045439	**Betweenness**	0.037135
CA	0.038213	**Eigenvector**	0.026029
LCOM	0.038174	CAM	0.025075
MOA	0.035131	CBM	0.023857
DIT	0.033895	LCOM	0.022769
IC	0.033053	AVG(CC)	0.018861
AVG(CC)	0.027356	**2StepP(out)**	0.017755
NOC	0.009310	2StepR(un)	0.015017

comparable distributions regarding feature interactions. Subsequently, we explore the effects of interactions between features on the prediction results to assess whether the size metrics and the SNA metrics exhibited similar distributions when interacting with non-size code metric features. Based on the aforementioned experimental results, we focus on the most significant SNA

metrics (i.e., pWeakC(out), Betweenness) and size metrics (i.e., AMC, NPM, MOA). As an example, while considering the space, we summarize comparative results of the impact of the interaction between two features on the prediction results using the combination of AMC, pWeakC(out), and LCOM. Fig.5 and Fig.6 shows the experimental results of the dependence plots in two validations and each data point represents a specific sample. For the two-feature interaction plots, the vertical axes on the left side of the plot means the SHAP value of the feature and the horizontal corresponds to the feature value. The feature value is reflected in different colors (red for high, and blue for low), as shown on the right side of the plot. Taking the AMC (LCOM) and pWeakC(out) (LCOM) metrics as examples, the comparison results for the interaction between two features reveal that AMC and pWeakC(out) have a relatively similar effect on model prediction. Therefore, it could be concluded that pWeakC(out) can be used as a substitute metric for the AMC. We also observe similar results in other combinations of metrics (e.g., Betweenness and MOA and they are included in the online appendix[1]).

Finding 3. According to feature importance and ranking results, the SNA metrics called Betweenness and pWeakC(out) have high importance in both validation scenarios. In addition, feature interaction results show that Betweenness is consistent with the size metric MOA for model prediction, and pWeakC(out) is more similar to the size metric AMC in model prediction. Therefore, we believe that these two SNA metrics can be used as alternative metrics to the size metrics.

6 Threats to validity

The threats to validity introduced in this section may affect our results and conclusions.

6.1 External validity

External validity can hinder the generalization of the findings. Only limited open-source projects are considered in the experiment, and we can not guarantee the conclusions remain consistent on other projects. However, we used the gold-standard PORMISE dataset studied widely in defect prediction, and we believe the projects in this dataset are classical. Moreover, the projects used in the experiment are all Java projects, and the experiment only involved discussion and analysis of the Java-based applications, instead of other commonly used programming languages such as C/C++ and Python. We suggest replicating our results in dataset of other programming languages.

6.2 Internal validity

Internal validity refers to the impact of independent variables on dependent variables in an experiment. The experiment in this study is primarily based on open-source code metrics dataset, which may be affected by computational

errors. However, these dataset are widely validated for defect prediction, and we believe their validity could be an assurance to a great extent. Additionally, the SNA metrics considered in the experiment are obtained through source code information extraction and dependency network analysis using network metrics calculation tools. It is not guaranteed that these methods and tools could completely reflect the dependency networks of the projects.

6.3 Construct validity

Construct validity refers to the extent to which the experimental results can be inferred to represent concepts or theories. On the one hand, there are numerous types of feature selection methods, such as filter, wrapper, and embedded methods. To generate a highly consistent subset of metrics, our experiment uses the AutoSpearman method for feature selection. AutoSpearman may not generate the best performance compared to other feature selection methods. However, we apply it since it mitigates multicollinearity to a great extent and generates stable results for XAI. Moreover, it has been proven reliable for defect prediction in related empirical studies [Jiarpakdee, 2019]. On the other hand, to understand the impact of feature importance and feature interactions on model prediction, we apply SHAP in the experiment. However, other model interpretation methods such as LIME and Breakdown are not considered, and it is not guaranteed that consistent conclusions can be drawn from the results using these methods. Recent work [Jiarpakdee et al., 2022, Yang et al., 2021b] has shown a high level of consistency in their results.

7 Conclusion and future work

In this study, we explore the use of SNA metrics and existing code metrics as alternatives for size metrics in defect prediction based on a gold-standard open-source software dataset. We compare the performance and apply SHAP to interpret the model from the perspectives of feature importance and feature interaction to explain model behavior and the impact of features on defect prediction. Based on our findings, we conclude that (1) size metrics are helpful but not essential for building well-performed models, and (2) two SNA metrics and several existing code metrics can be used as alternatives to size metrics. Since practitioners may not prefer size metrics [Antinyan., 2021], we suggest discarding them and introducing alternatives to preserve the information that size metrics may capture.

In the future, we plan to (1) evaluate the acceptance of defect prediction models without size metrics, (2) replicate our results in commercial and projects in other programming languages, and (3) continuously improve the performance of the prediction model and explore the potential of other types of metrics.

8 Acknowledgements

This work was partially supported by the National Natural Science Foundation of China (Grant no. 62276097), and the Natural Science Foundation of Shanghai (Grant no. 21ZR1416300).

9 Declarations

- Chenchen Chai: Conceptualization, Methodology, Software, Validation, Formal Analysis, Investigation, Resources, Data curation, Writing – original draft, Visualization. Guisheng Fan: Supervision, Project Administration, Funding acquisition. Huiqun Yu: Supervision, Project Administration. Zijie Huang: Formal analysis, Methodology. All authors reviewed the manuscript.
- No conflict of interest exists in the submission of this manuscript. The manuscript is approved by all authors for publication.
- The part of dataset analysed during the current study are available in http://purl.org/MarianJureczko/MetricsRepo.
- Our experimental code and results are open-sourced for replication and further studies as footnote1.

References

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

[Alpernas et al., 2020] Alpernas, K., Feldman, Y. M. Y., and Peleg, H. (2020). The wonderful wizard of loc: Paying attention to the man behind the curtain of lines-of-code metrics. In *Proc. 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 146–156.

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

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

[Bell et al., 2011] Bell, R. M., Weyuker, E. J., and Ostrand, T. J. (2011). Assessing the impact of using fault prediction in industry. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 561–565.

[Burkart and Huber, 2021] Burkart, N. and Huber, M. F. (2021). A survey on the explainability of supervised machine learning. *Journal of Artificial Intelligence Research*, 70:245–317.

[Cambria et al., 2023] Cambria, E., Malandri, L., Mercorio, F., and et al. (2023). A survey on xai and natural language explanations. *Information Processing & Management*, 60(1):103111.

[Chicco et al., 2021] Chicco, D., Warrens, M. J., and Jurman, G. (2021). The matthews correlation coefficient (MCC) is more informative than cohen's kappa and brier score in binary classification assessment. *IEEE Access*, 9:78368–78381.

[Chidamber and Kemerer, 1994] Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

[Esteves et al., 2020] Esteves, G., Figueiredo, E., Veloso, A., and et al (2020). Understanding machine learning software defect predictions. *Automated Software Engineering*, 27(3):369–392.

[Fawcett, 2006] Fawcett, T. (2006). An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874.

[Gao et al., 2022] Gao, Y., Zhu, Y., Yu, Q., and et al. (2022). Evaluating the effectiveness of local explanation methods on source code-based defect prediction models. In *Proc. 2022 19th ACM/IEEE International Conference on Mining Software Repositories (MSR)*, pages 640–645.

[Ghotra et al., 2017] Ghotra, B., McIntosh, S., and Hassan, A. E. (2017). A large-scale study of the impact of feature selection techniques on defect classification models. In *Proc. 2017 14th ACM/IEEE International Conference on Mining Software Repositories (MSR)*, pages 146–157.

[Gong et al., 2022] Gong, L., Rajbahadur, G. K., Hassan, A. E., and et al. (2022). Revisiting the impact of dependency network metrics on software defect prediction. *IEEE Transactions on Software Engineering*, 48(12):5030–5049.

[Henkel et al., 2022] Henkel, J., Ramakrishnan, G., Wang, Z., and et al. (2022). Semantic robustness of models of source code. In *Proc. 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 526–537.

[Herbold et al., 2022] Herbold, S., Trautsch, A., Trautsch, F., and Ledel, B. (2022). Problems with SZZ and features: An empirical study of the state of practice of defect prediction data collection. *Empirical Software Engineering*, 27(2):42.

[Hoffman et al., 2018] Hoffman, R. R., Mueller, S. T., Klein, G., and et al (2018). Metrics for explainable ai: Challenges and prospects.

[Hosseini et al., 2019] Hosseini, S., Turhan, B., and Gunarathna, D. (2019). A systematic literature review and meta-analysis on cross project defect prediction. *IEEE Transactions on Software Engineering*, 45(2):111–147.

[Jain and Saha, 2021] Jain, S. and Saha, A. (2021). Improving performance with hybrid feature selection and ensemble machine learning techniques for code smell detection. *Science of Computer Programming*, 212:102713.

[Jiarpakdee, 2019] Jiarpakdee, J. (2019). Towards a more reliable interpretation of defect models. In *Proc. 2019 the 41st International Conference on Software Engineering (ICSE)*, pages 210–213.

[Jiarpakdee et al., 2021a] Jiarpakdee, J., Tantithamthavorn, C., and Hassan, A. E. (2021a). The impact of correlated metrics on the interpretation of

defect models. *IEEE Transactions on Software Engineering*, 47(2):320–331.

[Jiarpakdee et al., 2018] Jiarpakdee, J., Tantithamthavorn, C., and Treude, C. (2018). Autospearman: Automatically mitigating correlated software metrics for interpreting defect models. In *Proc. 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 92–103.

[Jiarpakdee et al., 2022] Jiarpakdee, J., Tantithamthavorn, C. K., Dam, H. K., and et al. (2022). An empirical study of model-agnostic techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 48(2):166–185.

[Jiarpakdee et al., 2021b] Jiarpakdee, J., Tantithamthavorn, C. K., and Grundy., J. (2021b). Practitioners' perceptions of the goals and visual explanations of defect prediction models. In *Proc. 2021 18th ACM/IEEE International Conference on Mining Software Repositories (MSR)*, pages 432–443.

[Jun, 2014] Jun, L. (2014). *Global network analysis*. Truth and Wisdom Press.

[Jureczko and Madeyski, 2010] Jureczko, M. and Madeyski, L. (2010). Towards identifying software project clusters with regard to defect prediction. In *Proc. 2010 6th International Conference on Predictive Models in Software Engineering (PROMISE)*, pages 1–10.

[Jureczko et al., 2019] Jureczko, M., Nguyen, N. T., Szymczyk, M., and Unold, O. (2019). Towards implementing defect prediction in the software development process. *Journal of Intelligent & Fuzzy Systems*, 37(6):7223–7238.

[Katrutsa and Strijov, 2017] Katrutsa, A. and Strijov, V. (2017). Comprehensive study of feature selection methods to solve multicollinearity problem according to evaluation criteria. *Expert Systems with Applications*, 76:1–11.

[Koru et al., 2009] Koru, A. G., Zhang, D., Emamand, K. E., and et al. (2009). An investigation into the functional form of the size-defect relationship for software modules. *IEEE Transactions on Software Engineering*, 35(2):293–304.

[Li et al., 2018] Li, Z., Jing, X., and Zhu, X. (2018). Progress on approaches to software defect prediction. *IET Software*, 12(3):161–175.

[Lundberg and Lee, 2017] Lundberg, S. and Lee, S. I. (2017). A unified approach to interpreting model predictions. In *Proc. 2017 30th Annual Conference on Neural Information Processing Systems (NIPS)*, pages 4765–4774.

[Ma et al., 2016] Ma, W., Chen, L., Yang, Y., and et al (2016). Empirical analysis of network measures for effort-aware fault-proneness prediction. *Information and Software Technology*, 69:50–70.

[Marcilio and Eler, 2020] Marcilio, W. E. and Eler, D. M. (2020). From explanations to feature selection: Assessing shap values as feature selection mechanism. In *Proc. 2020 33rd Conference on Graphics, Patterns and Images (SIBGRAPI)*, pages 340–347.

[McCabe, 1976] McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320.

[Molnar, 2020] Molnar, C. (2020). *Interpretable machine learning.* "Lulu.com".

[Nagappan and Ball, 2005] Nagappan, N. and Ball, T. (2005). Use of relative code churn measures to predict system defect density. In *Proc. 2005 27th International Conference on Software Engineering (ICSE)*, pages 284–292.

[Nagappan et al., 2008] Nagappan, N., Murphy, B., and Basili, V. R. (2008). The influence of organizational structure on software quality. In *Proc. 2008 30th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 521–530.

[Nam et al., 2013] Nam, J., Pan, S. J., and Kim, S. (2013). Transfer defect learning. In *Proc. 2013 35th International Conference on Software Engineering (ICSE)*, pages 382–391.

[Naufal and Kusuma, 2019] Naufal, M. F. and Kusuma, S. F. (2019). Software defect detection based on selected complexity metrics using fuzzy association rule mining and defective module oversampling. In *2019 16th International Joint Conference on Computer Science and Software Engineering*, pages 330–335.

[Nguyen et al., 2010] Nguyen, T. H. D., Adams, B., and Hassan, A. E. (2010). Studying the impact of dependency network measures on software quality. In *Proc. 26th IEEE International Conference on Software Maintenance (ICSM)*, pages 1–10.

[Oram and Wilson, 2011] Oram, A. and Wilson, G. (2011). *Making software: What really works, and why we believe It.* O'Reilly Media, Inc.

[Papenmeier et al., 2022] Papenmeier, A., Dagmar Kern, D., Englebienne, G., and et al. (2022). It's complicated: The relationship between user trust, model accuracy and explanations in ai. *ACM Transactions on Computer-Human Interaction*, 29(4):1–33.

[Pornprasit et al., 2021] Pornprasit, C., Tantithamthavorn, C., Jiarpakdee, J., and et al (2021). Pyexplainer: Explaining the predictions of just-in-time defect models. In *Proc. 2021 36th ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 407–418.

[Premraj, 2011] Premraj, K. R. (2011). Network versus code metrics to predict defects: A replication study. In *Proc. 2011 5th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 215–224.

[Rajapaksha et al., 2022] Rajapaksha, D., Tantithamthavorn, C., Jiarpakdee, J., and et al. (2022). Sqaplanner: Generating data-informed software quality improvement plans. *IEEE Transactions on Software Engineering*, 48(8):2814–2835.

[Rao et al., 2021] Rao, J., Yu, X., Zhang, C., Zhou, J., and Xiang, J. (2021). Learning to rank software modules for effort-aware defect prediction. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 372–380.

[Reddivari and Raman, 2019] Reddivari, S. and Raman, J. (2019). Software quality prediction: An investigation based on machine learning. In *Proc. 2019 IEEE 20th International Conference on Information Reuse and*

Integration for Data Science (IRI), pages 115–122.

[Ribeiro et al., 2016] Ribeiro, M. T., Singh, S., and Guestrin, C. (2016). "Why should I trust you?": Explaining the predictions of any classifier. In *Proc. 2016 22nd ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 1135–1144.

[Shin et al., 2012] Shin, Y., Bell, R. M., Ostrand, T. J., and et al. (2012). On the use of calling structure information to improve fault prediction. *Empirical Software Engineering*, 17(4-5):390–423.

[Song et al., 2019] Song, Q., Guo, Y., and Shepperd, M. (2019). A comprehensive investigation of the role of imbalanced learning for software defect prediction. *IEEE Transactions on Software Engineering*, 45(12):1253–1269.

[Staniak and Biecek, 2018] Staniak, M. and Biecek, P. (2018). Explanations of model predictions with live and breakdown packages. *The R Journal*, 10(2):395.

[Tantithamthavorn et al., 2017] Tantithamthavorn, C., McIntosh, S., Hassan, A. E., and et al (2017). An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 43(1):1–18.

[Tantithamthavorn and Jiarpakdee, 2021] Tantithamthavorn, C. K. and Jiarpakdee, J. (2021). Explainable ai for software engineering. In *Proc. 2021 36th ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 1–2.

[Wan et al., 2020] Wan, Z., Xia, X., E.Hassan, A., and et al. (2020). Perceptions, expectations, and challenges in defect prediction. *IEEE Transactions on Software Engineering*, 46(11):1241–1266.

[Wang et al., 2022] Wang, A., Zhao, Y., Li, G., Zhang, J., Wu, H., and Iwahori, Y. (2022). Heterogeneous defect prediction based on federated reinforcement learning via gradient clustering. *IEEE Access*, 10:87832–87843.

[Weyuker and Ostrand, 2010] Weyuker, E. and Ostrand, T. J. (2010). An automated fault prediction system. *Making Software: What Really Works, and Why We Believe It,"O'Reilly Media, Inc*, pages 145–160.

[Weyuker et al., 2008] Weyuker, E. J., Ostrand, T. J., and Bell, R. M. (2008). Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13(5):539–559.

[Yang et al., 2021a] Yang, F., Huang, Y., Zhou, S., and et al. (2021a). Research progress of software defect prediction based on multiple metrics. *Computer Engineering and Applications*, 57(5):10–24.

[Yang et al., 2021b] Yang, X., Yu, H., Fan, G., and et al. (2021b). An empirical study of model-agnostic interpretation technique for just-in-time software defect prediction. In *Proc. 2021 17th EAI International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 420–438.

[Yu et al., 2020] Yu, Q., Jiang, S., Qian, J., and et al. (2020). Process metrics for software defect prediction in object-oriented programs. *IET Software*,

14(3):283–292.

[Yu et al., 2019] Yu, X., Bennin, K. E., Liu, J., Keung, J. W., Yin, X., and Xu, Z. (2019). An empirical study of learning to rank techniques for effort-aware defect prediction. In *Proc. 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 298–309.

[Zhang et al., 2017] Zhang, X., Zhou, Y., and Zhu, C. (2017). An empirical study of the impact of bad designs on defect proneness. In *Proc. 2017 International Conference on Software Analysis, Testing and Evolution (SATE)*, pages 1–9.

[Zimmermann and Nagappan, 2008] Zimmermann, T. and Nagappan, N. (2008). Predicting defects using network analysis on dependency graphs. In *Proc. 2008 30th International Conference on Software Engineering (ICSE)*, pages 531–540.