Revisiting the Impact of Dependency Network Metrics on Software Defect Prediction

Lina Gong, Gopi Krishnan Rajbahadur, Ahmed E. Hassan, *Fellow, IEEE*, and Shujuan Jiang, *Member, IEEE*

Abstract—Software dependency network metrics extracted from the dependency graph of the software modules by the application of Social Network Analysis (SNA metrics) have been shown to improve the performance of the Software Defect prediction (SDP) models. However, the relative effectiveness of these SNA metrics over code metrics in improving the performance of the SDP models has been widely debated with no clear consensus. Furthermore, some of the common SDP scenarios like predicting the number of defects in a module (Defect-count) in Cross-version and Cross-project SDP contexts remain unexplored. Such lack of clear directive on the effectiveness of SNA metrics when compared to the widely used code metrics prevents us from potentially building better performing SDP models. Therefore, through a case study of 9 open source software projects across 30 versions, we study the relative effectiveness of SNA metrics when compared to code metrics across 3 commonly used SDP contexts (Within-project, Cross-version and Cross-project) and scenarios (Defect-count, Defect-classification (classifying if a module is defective) and Effort-aware (ranking the defective modules w.r.t to the involved effort)). We find the SNA metrics by themselves or along with code metrics improve the performance of SDP models over just using code metrics on 5 out of the 9 studied SDP scenarios (three SDP scenarios across three SDP contexts). However, we note that in some cases the improvements afforded by considering SNA metrics over or alongside code metrics might only be marginal, whereas in other cases the improvements could be potentially large. Based on these findings we suggest that the future work should: consider SNA metrics alongside code metrics in their SDP models; as well as consider Ego metrics and Global metrics, the two different types of the SNA metrics separately when training SDP models as they behave differently.

Index Terms—Dependency network metrics, code metrics, software defect prediction, social network analysis (SNA), within-project, cross-version, cross-project, effort-aware

1 Introduction

As software scales and its complexity increases, more resources are required to test and inspect code. In practice, due to resource constraints such as time and cost, managers often allocate limited resources to test high-risk defective modules [23], [43], [76], [99]. Thus, effectively identifying high-risk defective modules to be tested becomes particularly critical. To address this, Software Defect Prediction (SDP) models trained on software metrics are commonly used to identify high-risk defective modules [61], [75], [82]. These SDP models are then used to effectively allocate testing resources. The use of software metrics (e.g., code metrics, SNA metrics, or process metrics) to train these SDP models play a crucial role in determining their performance.

In 2008, Zimmermann and Nagappan [103] demonstrated the effectiveness of using dependency network metrics along with source code metrics to improve the performance of SDP models. Such dependency network metrics (SNA metrics) were extracted from the dependency graph of the software modules by application of the Social Network Analysis. Zimmermann and Nagappan [103] captured the dependency relations among software modules along 60 SNA metrics. They hypothesized that these SNA metrics captured the dependency structure of the software system better and thereby identified defects more effectively. They reported a 10% improvement in the performance of the SDP model trained on SNA metrics for Windows Server 2003.

Since the early work of Zimmermann and Nagappan [103], several studies have explored the effectiveness of SNA metrics over code metrics for SDP and have come up with conflicting conclusions. Such prior studies examined the effectiveness of SNA metrics in various SDP contexts (i.e., Within-project, Cross-version and Cross-project) and scenarios (Defect-count (predicting the number of defects)), Defect-classification (classifying if a module is defective) and Effort-aware (ranking of defect prone modules w.r.t required effort)). The reached conclusions can be grouped into three categories: 1) studies that argue SNA metrics along with code metrics can indeed improve the performance of SDP models [60], [103], 2) studies that argue SNA metrics provide no significant advantage over using just the code metrics [66], 3) finally, studies that argue

- *Lina Gong is with the School of Computer Science and Technology, China University of Mining and Technology, Xuzhou, Jiangsu 221116, China, and also with the School of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China. E-mail: gonglina@nuaa.edu.cn.*
- *Gopi Krishnan Rajbahadur and Ahmed E. Hassan are with Software Analysis and Intelligence Lab (SAIL), School of Computing, Queen's University, Kingston, ON K7L 3N6, Canada. E-mail: {krishnan, ahmed}@cs.queensu.ca.*
- *Shujuan Jiang is with the School of Computer Science and Technology, China University of Mining and Technology, Xuzhou 221008, China. E-mail: shjjiang@cumt.edu.cn.*

Manuscript received 23 June 2021; revised 23 Nov. 2021; accepted 26 Nov. 2021. Date of publication 1 Dec. 2021; date of current version 12 Dec. 2022. This work was supported in part by the Fundamental Research Funds for the Central Universities under Grant 2019XKQYMS84. (Corresponding author: Gopi Krishnan Rajbahadur.) Recommended for acceptance by D. Hao. Digital Object Identifier no. 10.1109/TSE.2021.3131950

0098-5589 © 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:59:26 UTC from IEEE Xplore. Restrictions apply.

TABLE 1
Literature Survey of All the Prior Studies That Understand the Effectiveness of SNA Metrics Over Code Metrics

Context	Code metrics are more effective			SNA metrics are more effective			SM metrics are more effective		
	D-classification	D-count	E-aware	D-classification	D-count	E-aware	D-classification	D-count	E-aware
Within-project	[89] (small), [66]*,[94]*	Not as effective	Not as effective	[103]$^{\Delta}$, [67]$^{\Delta}$	Not as effective	[46]$^{\Delta}$	[103]$^{\Delta}$, [60], [89] (Large), [67]$^{\Delta}$, [66]*, [94]*	[103], [60]	[60], [46]$^{\Delta}$
Cross-version	[67]$^{\circ}$, [66]	Not studied	Not as effective	[67]$^{\circ}$	Not studied	[46]$^{\Delta}$	[67]$^{\circ}$	Not studied	[46]$^{\Delta}$
Cross-project	[67]$^{\bullet}$, [66]	Not studied	[46]$^{\bullet}$	[67]$^{\bullet}$	Not studied	[46]$^{\bullet}$	Not as effective	Not studied	Not as effective

1. Simple Combination (SM) metrics- Code and SNA metrics simply combined.
2. D-Classification- Defect-classification; *D-count-* Defect-count; *E-aware-* Effort-aware.
3. Small: Large performance impact was only found on small software projects.
4. Large: Large performance impact was only found on large software projects.
*5. *: code metrics and SM metrics were found to be equally effective; Δ: SNA metrics and SM metrics were found to be equally effective; •: code metrics and SNA metrics were found to be equally effective; ○: All three metrics were found to be equally effective (i.e., no difference was observed in effectiveness of the studied metrics).*

SNA metrics are effective for a limited number of SDP contexts and scenarios [46], [67], [89].

For instance, Nguyen *et al*. [60] showed that SNA metrics alongside code metrics indeed help in the improving the performance of SDP models in the Within-project context. Whereas, Prateek *et al*. [66] found that SNA metrics offered no significant advantage over code metrics in any of the SDP contexts. Different from both of these studies, Ma *et al*. [46] reported that SNA metrics offered a slight improvement over code metrics only in the Within-project and Cross-version SDP contexts. Such conflicting results prevents us from recommending the most appropriate metrics for building effective models. Table 1 presents a survey of prior SDP studies that explore SNA metrics (More details about the survey in Section 2).

We further note that, *none* of the prior studies attempt to rationalize the varying effectiveness of SNA metrics for a given SDP context and/or scenario. Hence our key contributions are:

1) We examine the effectiveness of SNA metrics across 3 common SDP contexts and scenarios on nine projects across 30 versions.
2) We attempt to rationalize the varying effectiveness of SNA metrics across the studied SDP contexts and scenarios.
3) In contrast to the prior studies, we study the effectiveness of separately considering the different types of SNA metrics (i.e., Global metrics (GN) and Ego metrics (EN)) for training a SDP model. In addition, we also study the effectiveness of using a simple filtering strategy to combine the code and SNA metrics (COM metrics) over simply combining them (SM metrics).

In our study, we focus only on comparing the effectiveness of SNA metrics over code metrics (and not other metric families like process metrics), as the history of a software project is not always readily available. In cases such as those, only metrics based on source code could be used to build models. We structure our study along the following research questions:

- *RQ1: Are SNA metrics more effective than code metrics for Within-project SDP?*

  Models trained on SM and COM metrics outperform models trained on all other metric families across Defect-Count and Defect Classification scenarios. In addition, models trained on SNA metrics outperform models trained on other metric families for Effort-aware scenarios. We observe that, such a result is because SNA metrics add additional information to the code metrics. Therefore, we recommend researchers and practitioners to consider the SNA metrics alongside code metrics in the Within-project context.
- *RQ2: Are SNA metrics more effective than code metrics for Cross-version SDP?*

  The results are varied for the Cross-version context across the studied scenarios. Models trained on code metrics outperform models trained on other metric families for the Defect-Count scenario. For the Defect-classification scenario, models trained on SNA metrics and code metrics perform similarly. Models trained on EN and COM metrics outperform models trained on other metric families for the Effort-aware scenarios. In addition, though code metrics are effective for Defect-count scenario, we observe that the models trained on GN metrics have better performance in identifying the number of defects on modules whose defectiveness changes across versions. These results indicate that considering SNA metrics alongside code metrics might be useful. In addition, these results highlight the need for considering GN and EN metrics separately.
- *RQ3: Are SNA metrics more effective than code metrics for Cross-project SDP?*

  Models trained on code metrics outperform models trained on other metric families for the Defect-Count scenario. For the Defect-classification scenario, models trained on SNA metrics (GN, SM and COM) metrics have similar performance as the models trained on code metrics on 3 of the 5 studied performance measures. Finally, the models trained on GN metrics outperform models trained on other metric families for the Effort-aware scenarios.

From the results we observe that considering SNA metrics by themselves or alongside code metrics is effective across all

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:59:26 UTC from IEEE Xplore. Restrictions apply.

the SDP contexts and 5 out of 9 studied SDP scenarios. However, we note that in some cases the improvements afforded by considering SNA metrics over or alongside code metrics might only be marginal, whereas in other cases the improvements could be potentially large. In light of these findings, we recommend researchers and practitioners to always consider including SNA metrics to train their models, particularly given the ease of extracting these metrics. In addition, to foster open science and to ensure that our results are replicable we provide a replication package for our study.[1]

Paper Organization. Section 2 motivates our study and discusses related work. Section 3 explains the metrics, models and evaluation measures used in our study. Section 4 presents our study design, including studied projects and experiment setup. Section 5 presents and discusses our experimental results. Section 7 presents the implications that can be inferred from our results. Section 8 discusses the threats to the validity of our observations. Finally, we conclude our study in Section 9.

2 Motivation and Related Work

2.1 Motivation

SDP has become a very active research field [6], [28], [39]. Researchers have proposed many metrics based on various aspects to build models, such as code metrics [7], [10], [57], [62], [102] and process metrics [47], [56], [68]. Code metrics leverage the source code of the software such as Halstead [29] and McCabe [48] metrics and Chidamber-Kemere (CK) [14] metrics. Process metrics leverage the software history such as change complexity [30], change coupling [17], change history [26], and developer metrics [9], [35], [49].

When the history of a software is not readily available, only metrics based on the source code can be used to build models. For instance, consider small companies or start ups where there is no established process to capture the software evolution [39]. In addition, companies might not always be willing to share history based metrics due to privacy concerns [64], [65]. Furthermore, as Nagappan *et al.* [58] argues process metrics are presumably more expensive to collect. Therefore, it is important to be able to leverage the information available in the source code of a software for SDP.

There exist many rich dependencies between the software modules in addition to the static details about the source code such as lines of code. For instance, data and call dependencies could be extracted from the source code. Several prior studies make use of such dependencies to build models [73], [77], [79], [103]. Most notably, Zimmermann and Nagappan [103] demonstrated the effectiveness of dependency network metrics that were extracted using SNA on the dependency graph of the software modules. Such SNA metrics have been the most widely adopted set of source code based dependency metrics in SDP. However, as we mention earlier, the effectiveness of SNA metrics to train models is inconclusive. To capture the common consensus on the effectiveness of SNA metrics, we conducted a thorough literature survey of prior SDP studies that examined the effectiveness of SNA metrics over code metrics. We conducted the literature survey following the process that was outlined by Wohlin *et al.* [93]. We first searched google scholar[2] with a single query for the terms "defect prediction" and "Social Network Analysis" then we went through the abstract of all the returned papers. We shortlisted the papers that compared SNA metrics to code metrics. Following which, we conducted a forward and reverse snowball search [93] for all the papers to which a given paper refers. We then went through each of them to locate any other relevant studies. We continued the forward and reverse snowball search for each of the collected papers recursively until we exhaust all the relevant papers. We finally arrived at 8 studies. We present the results of our literature survey in Table 1. We present and categorize each study based on the SDP context and scenario under which the effectiveness of the SNA metrics were studied. Each row of the Table 1 captures the SDP context under which a given study compares the effectiveness between SNA and code metrics. Each column of the Table 1 captures the metric family that a given study finds more effective. Additionally, each of these columns are further sub-divided 3 columns, which capture the SDP scenario under which a given study observed the metric family in this column to be more effective. If a given study found two metric families to be equally effective, we then present the study in both the columns and mark it with a symbol to present which two metric families were found to be equally effective. For instance, consider the study by Premraj and Herzig [67]. They found that for the Defect-classification scenario, in the Cross-version context, models trained on code, SNA and SM (code and SNA metrics) metrics perform similarly. Hence, we represent that in our Table 1 by having an entry for the study [67] on all the three columns pertaining to each metric family and SDP scenario. Furthermore, we mark the study with a "○" symbol represent the same.

From the results presented in Table 1 we observe four shortcomings about prior studies of the effectiveness of SNA metrics.

First, there is no clear consensus on whether code metrics consistently outperform SNA metrics or the SM metrics. For instance, consider the Cross-version context. Prateek *et al.* [66] find that code metrics outperform SNA metrics for the Defect-classification scenario. Whereas, Premraj and Herzig [67] find that code, SNA metrics perform similarly for the same scenario across different classifiers. Such conflicts in the results could be observed across all the SDP context and scenarios in Table 1.

Second, prior studies mostly limit themselves to one or two SDP scenarios. For instance, consider the studies by Tosun *et al.* [89] and Song *et al.* [83] which consider only the within-project context and the Defect-classification scenario. Though some studies like Premraj and Herzig [67] and Ma *et al.* [46] span all the three common SDP contexts, they limit themselves to a single SDP scenario. For instance, Premraj and Herzig [67] focus on the Defect-classification scenario, while Ma *et al.* [46] focus on the Effort-aware scenario. Moreover, both the studies use different datasets and models making it impossible to reach a common finding that applies across the SDP scenarios.

1. https://github.com/glnmzx888/SNA

2. https://scholar.google.ca/

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:59:26 UTC from IEEE Xplore. Restrictions apply.

Third, the effectiveness of SNA metrics in SDP scenarios like Defect-count scenario in the Cross-version and Cross-project contexts remain unexplored. Table 1 denotes the SDP scenarios under which the effectiveness of SNA metrics remain unexplored. Lack of research in such SDP scenarios affects our ability to recommend SNA metrics widely.

Finally, none of the prior studies explore the effectiveness of different types of SNA metrics (GN and EN metrics). The GN metrics capture the dependency characteristics of a given module in the context of the whole software system. Whereas, EN metrics capture the dependency characteristics of a given software module with respect to its neighboring nodes [103] (see Section 3.1.2 for more details about the metrics). Despite the differences between GN and EN metrics, none of the prior studies outlined in Table 1 consider these separately. Therefore, in our study, we consider them separately and together to understand their effectiveness in relation to code metrics for building models.

2.2 Related Work

Studies Comparing Various Software Metrics. Prior studies have compared the effectiveness of various metric families for SDP. Moser *et al*. [52] demonstrated that process metrics are more effective than code metrics for SDP. Hassan [30] proposed an entropy based metric and examined its effectiveness relative to the number of prior faults and prior modifications. Arisholm *et al.* [4] evaluated the effectiveness of code and process metrics for SDP and observed that code metrics outperformed for the Defect-classification scenario, but under performed for Effort-aware scenario. Whereas, process metrics were effective in the Effort-aware scenario. Alternatively, Nagappan *et al.* [58] found that change bursts were more effective than complexity metrics, code churn, and organizational structure. However, Rahman and Devanbu [68] later observed that process metrics were more effective than code metrics.

Different from these studies, we study the effectiveness of SNA metrics over code metrics and investigate the rationale for the effectiveness of SNA metrics.

Studies of Other Source Code Based Dependency Metrics. Different from the SNA metrics computed on the dependency graph, other dependency metrics have been used for SDP. Schröter *et al.* [73] leveraged the import relationships between software components for SDP and showed that the imported software components could predict post-version defects more effectively than random guesses. Turhan *et al.* [90] proposed a static call graph based ranking (CGBR) framework, which was based on both static code metrics and the call dependency relations between software modules. Shin *et al.* [77], [79] investigated the effectiveness of the calling structure on SDP.

3 Background

In this section, we briefly describe the software metrics, SDP contexts and SDP scenarios considered in our study.

3.1 Software Metrics

Different software metrics capture details about different facets of a software project. We briefly describe the six different software metrics that we use in our study: Code metrics, SNA metrics, EN metrics, GN metrics, Simple Combination (SM) metrics and COM metrics.

3.1.1 Code Metrics

Code metrics quantify various properties of the source code of a software project. For instance, size of the project is typically quantified by the lines of code metric. These code metrics can be computed at various granularities: method-level, class-level and file-level.

Method-Level Metrics. These metrics quantify the various properties of methods. We use the Halstead [29] and McCabe [48] metrics to study method level properties as they are some of the most popular method-level metrics in SDP.

Class-Level Metrics. These metrics are observed at the class level and are only used for object-oriented (OO) projects. We use the Chidamber-Kemere (CK) [14] metrics (e.g., Coupling Between Object (CBO), Depth Of Inheritance Tree (DIT)) as the class-level metrics.

File-Level Metrics. File-level metrics are metrics derived from each file. They also include metrics that are generated by aggregating method-level or class-level metrics. For instances, we aggregate the Method-level metrics by max, min and mean at the file level.

In our study, we consider the metrics at all the three granularties above mentioned as *code metrics*. The 54 code metrics used in our study are listed in Table 2. For a detailed description about each of the studied code metrics, we refer the readers to `SciTools` website.[3]

3.1.2 SNA Metrics

SNA metrics are extracted by applying Social Network Analysis on the dependency network of software project [103]. Dependency networks are graphs whose nodes are software modules and edges are dependencies between these modules. There are two main families of dependency relationships: data dependency and call dependency. In such a dependency network that includes both call and data dependencies, each node belongs to two types of networks: ego network and global network.

Ego Network Metrics (EN). The Ego network of a node consists of the node itself and its neighboring nodes. For each node, there exists three types of ego networks: In, Out and Undirected. (In) ego of a node only considers itself and nodes which it depends on. (Out) ego of a node only considers itself and nodes that depend on it. (Undirected) ego of a node considers a combination of In and Out. In our study, we compute all three types of ego networks.

Global Network Metrics (GN). Global network refers to the entire dependency graph, which can measure the importance of nodes in the context of the entire software system. In our experiments, we compute structural holes and centrality as global network (GN) metrics.

SNA Metrics (SNA). Combination of EN and GN metrics together makes up the SNA metrics.

Table 2 describes the EN and GN metrics. We consider EN, GN and SNA metrics as three different metric families in our study. We present more details about each of the studied SNA metric in Table 9 of Appendix A, which can be

3. https://scitools.com/feature/metrics/

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:59:26 UTC from IEEE Xplore. Restrictions apply.

TABLE 2
Software Metrics Used in Our Study

Metric group	Metrics	Level	Metrics	Count
Code metrics	**Code**	**File**	AvgCyclomatic, AvgCyclomaticModified, AvgCyclomaticStrict, AvgEssential, AvgLine, AvgLineBlank, AvgLineCode, Avg-LineComment, CountDeclClass, CountDeclClassMethod, CountDeclClassVariable, CountDeclFunction, CountDeclInstanceMethod, CountDeclInstanceVariable, CountDeclMethod, CountDeclMethodDefault, CountDeclMethodPrivate, CountDeclMethodProtected, CountDeclMethodPublic, CountLine, CountLineBlank, CountLineCode, CountLineCodeDecl, CountLineCodeExe, CountLineComment, CountSemicolon, CountStmt, CountStmtDecl, CountStmtExe, MaxCyclomatic, MaxCyclomaticModified, MaxCyclomaticStrict, RatioCommentToCode, SumCyclomatic, SumCyclomaticModified, SumCyclomaticStrict, SumEssential	37
		Class	CountClassBase, CountClassCoupled, CountClassDerived, MaxInheritanceTree, PercentLackOfCohesion	5
		Method	CountInput {Min, Mean, Max}, CountOutput {Min, Mean, Max}, CountPath {Min, Mean, Max}, MaxNesting {Min, Mean, Max}	12
SNA metrics	**EN**	**File**	Size {in, out, un}, Ties {in, out, un}, Pairs {in, out, un}, Density {in, out, un}, WeakComp {in, out, un}, nWeakComp {in, out, un}, 2StepReach {in, out, un}, ReachEfficency {in, out, un}, Brokerage {in, out, un}, nBrokerage {in, out, un}, EgoBetween {in, out, un}, nEgoBetween {in, out, un}, EffSize(ego), Efficiency(ego), Constraint(ego), Hierarchy(ego)	40
	GN	**File**	Degree, Eigenvector, power, Closeness, Betweenness, Information, dwReach, EffSize, Efficiency, Constraint, Hierarchy	24
	SNA	**File**	EN+GN	64
Combined Metric	**SM**	**File**	Code+SNA	118
	COM	**File**	CountDeclMethodDefault, AvgLineComment, CountDeclMethodProtected, RatioCommentToCode, AvgLineBlank, MaxInheritanceTree, CountClassDerived, CountInput{Min}, CountOutput {Max}, CountOutput{Min}, MaxNesting{Min}, Densit(in), pWeakC(in), nBroke(in), nEgoBe(in), pWeakC(un), nEgoBe(un)	17

found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2021.3131950.

3.1.3 Combined Metrics

We consider two different ways of combining code metrics and SNA metrics to observe if a combination of these metrics outperform the individual metrics.

Simple Combination Metrics (SM). SM metrics are a simple combination of both code and SNA metrics. Code and SNA metrics are used without any metric being discarded. Such a way of combining the metrics has been used by all the prior studies listed in Table 1.

Common Metrics (COM). We take SM metrics and use a simple filtering strategy to filter metrics. We do so as, among the combined metrics, the stronger signal contained by some metrics could potentially be masked by the other collinear metrics. Therefore, we employ a Variance Inflation Factor (VIF) filter that we describe below to filter and combine the metrics.

Variance Inflation Factor (VIF) Filter. We use the Variance Inflation Factor (VIF) to remove multicollinearity among the studied metrics. The VIF score quantifies the collinearity of a metric by computing how well the given metric can be represented by the other studied metrics [36], [63]. The VIF score of a metric is computed by first building a linear regression model using the other studied metrics to predict the metric under examination. Then the $\mathbf{R^2}$ score [1] of the built regression model is computed. Finally, the VIF score of the metric under examination is given as $\frac{1}{1-R^2}$.

A large VIF score for a given metric indicates that the given metric is highly collinear with the other studied metrics. Therefore, for each project, we only consider the metrics whose VIF score is less than 10. We use a threshold of 10 since several prior studies [8], [46] consider the metrics that have a VIF score > 10 to be highly collinear with the other metrics.

The method of filtering metrics varies based on the context. In the Within-project context, we simply remove all the metrics in a given project whose metrics have a VIF > 10. In the Cross-version context, we define COM metrics as those SM metrics whose VIF < 10 in all versions of a project. In the Cross-project context, we use the COM metrics whose VIF < 10 across all the studied projects. We report the 17 metrics that are present in all projects and have a VIF < 10 in Table 2.

3.2 Software Defect Prediction (SDP) Models

In this sub-section, we detail the common SDP contexts and scenarios that we study.

3.2.1 SDP Scenarios

Models are typically constructed for the following three SDP scenarios: Defect-count, Defect-classification, and Effort-aware scenarios.

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:59:26 UTC from IEEE Xplore. Restrictions apply.

Defect-count scenario is where a model (Defect-count model) predicts the number of defects in a given software module [3], [71], [72].

Defect-classification scenario is where a model (Defect-classification model) classifies whether a given software module is defective or not [28], [50].

Effort-aware scenario is where a model (Effort-aware model) ranks software modules based on their probability of being defective with respect the effort required to identify the defects [6], [18], [27], [37]. For instance, ranking the software modules based on their probability of being defective and the lines of code that one needs to inspect in order to identify the defects.

3.2.2 SDP Contexts

SDP contexts can be divided into: Within-project context, Cross-version context and Cross-project context.

Within-project context is where the training data and testing data are derived from the same version of a project [85], [87]. Commonly, a part of the data from a version is used for training a model and the remaining is used for testing the trained model.

Cross-version context is where the training data to predict the defects in the current version of a project is comprised of data from its prior versions [96], [97]. For instance, in the Cross-version context, one uses versions $\mathrm{V}_1, \mathrm{V}_2, \ldots, \mathrm{V}_{\mathrm{n\text{-}1}}$ to train a model to predict defects in the version V_n.

Cross-project context is where the training data to predict the defects in the current version of a project is comprised of data from completely different projects [34], [95]. For instance, to predict the defects in the version V_k of a project P_B, the model in the Cross-project context is typically trained on the latest version V_m belonging to project P_A.

3.3 Model Evaluation Measures

In this sub-section, we detail the different performance measures that we use for evaluating our trained models.

3.3.1 Performance Measures for Defect-Count Models

To evaluate the effectiveness of Defect-count models, we use four different performance measures: Mean Absolute Error, Adjusted R square, Root mean squared error and Spearman correlation coefficient [42], [60]. We use these residual based performance measures over the commonly used performance measures like Magnitude of Relative Error (MRE) or Pred(X). We do so as several prior studies point out that MRE and Pred(X) are biased [38], [40], [53], [74]. In particular, Myrtveit *et al.* [53] showed that the MRE measure yields completely different values for two prediction models even when their residuals are similar. Due to these shortcomings, Myrtveit *et al.* [53] recommended the use of residual based performance measures (similar to the ones that we use) to evaluate the performance of regression models. We briefly explain these measures below.

Mean Absolute Error (MAE). MAE computes the mean of the absolute difference between the number of predicted defects by the Defect-count model and actual number of defects for each module [99]. The lower the MAE value, the better the effectiveness of the Defect-count model.

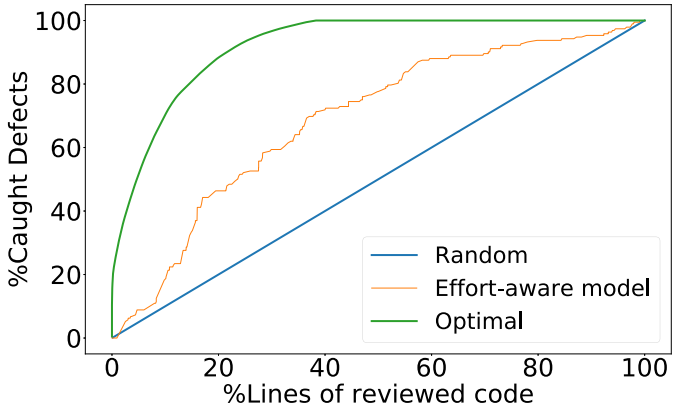

Fig. 1. Cost-effectiveness curve for an effort-aware model.

Root Mean Squared Error (RMSE). RMSE measures the square root of the average of squared differences between the number of predicted defects by a Defect-count model and actual number of defects for each module [11]. A lower RMSE score indicates better performance.

Adjusted R Square (Adj R^2). Adj R^2 measures the amount of variance of the dependent variable that the Defect-count model captures with respect to the number of used metrics [60]. The Adj R^2 indicates how good a defect count model fits the training data. A higher Adj R^2 value indicates a better performing model.

Spearman Correlation Coefficient (Spearman). Spearman correlation coefficient indicates the correlation between predicted number of defects and actual number of defects [60]. A high spearman values indicates that a Defect-count model can accurately predict the number of defects.

3.3.2 Performance Measures for Defect-Classification Models

To evaluate the effectiveness of Defect-classification models, we use the following performance measures such as: Area Under the Reciever-operator Characteristic Curve (AUC), Matthews correlation coefficient (MCC), Precision, Recall and Brier score similar to several prior studies [13], [23], [70], [88], [103]. A high value for AUC, Precision, Recall and MCC indicates a good performing model. A more detailed explanation about each of the studied performance measure is provided in the Table 10 of Appendix B, available in the online supplemental material.

3.3.3 Performance Measures for Effort-Aware Models

To evaluate the effectiveness of Effort-aware models, we use the cost-effectiveness (CE) curve [4], [46] and effort reduction (ER) measures [46], [78].

Area Under the Cost-Effectiveness Curve (CE). The area under the cost-effectiveness curve quantifies the required effort to identify defects. It does so by computing the area under the Cost-Effectiveness (CE) curve. Fig. 1 shows a CE curve for an Effort-aware model. The x-axis of the Fig. 1 is the percentage of reviewed LOC and y-axis is the percentage of discovered defects. If the majority of defects can be identified by reviewing a small percentage of the LOC then the model performance is optimal. Such an optimal scenario is depicted in Fig. 1 with the green Optimal line. Contrastingly, the worst

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:59:26 UTC from IEEE Xplore. Restrictions apply.

CE curve is one in which the Effort-aware model's ranking is completely random and the percentage of found defects linearly increases with the percentage of reviewed LOC. Such a scenario is depicted in Fig. 1 with a blue line. The orange line indicates how many defects could be caught with respect to the percentage of LOC that need to be reviewed for the prediction by the Effort-aware model. The closer the orange line is to the green Optimal line, the better. The area under the orange line is given as the Area under the CE curve. We calculate the Area under the CE curve with Formula (1) as suggested by Arisholm *et al.* [4]. A high CE value indicates that the Effort-aware model can identify most of the defects by investigating the least possible LOC. π is a cut-off value which shows the percentage of LOC at which the Area under the CE curve is calculated. In our experiments, we use $\pi = 1$ to report the CE values to present a holistic summary

$$CE_{\pi} = \frac{Area_{\pi}(m) - Area_{\pi}(Random)}{Area_{\pi}(Optimal) - Area_{\pi}(Random)}. \quad (1)$$

Effort Reduction (ER). ER indicates the ratio of the reduced LOC to be reviewed when using an Effort-aware model to LOC to be reviewed when using a random model to achieve the same recall as the Effort-aware model. We use the ER measure as outlined by Shin *et al.* [78] and Ma *et al.* [46]. We calculate the ER for our Effort-aware model (EM) with the following Formula (2)

$$ER(EM) = \frac{Effort(Random) - Effort(EM)}{Effort(Random)}. \quad (2)$$

In Formula (2), *Effort(EM)* is defined as the ratio of LOC to be inspected (i.e., where the Effort-aware model predicts the module to be defective) to the total LOC. We calculate the *Effort(EM)* with Formula (3). In Formula (3), $\mathrm{p_i}$ is 1 if the Effort-aware model considers the module defective and 0 otherwise. Similarly, to compute the effort for the random model *Effort(Random)*, we use Formula (4), where $\mathrm{d_i}$ is the number of defects in the software module i. Finally, we compute the ER with the Formula (2). A high ER values indicates that the Effort-aware model performs well [46]

$$Effort(EM) = \frac{\sum_{i=1}^{n} l_i \times p_i}{\sum_{i=1}^{n} l_i} \quad (3)$$

$$Effort(Random) = \frac{\sum_{i=1}^{n} d_i \times p_i}{\sum_{i=1}^{n} d_i}. \quad (4)$$

4 Study Design

In this section, we describe our study design to answer our research questions. An overview of our study design is shown in Fig. 2.

4.1 Studied Projects

We conduct our experiments on the same software projects as Yatish *et al.* [99]. These projects represent a carefully curated list of benchmark defect datasets, which consists of 32 versions of 9 open-source software projects including ActiveMQ, Camel, Derby, Groovy, HBase, Hive, JRuby, Lucene and Wicket. Table 3 provides a detailed overview of our studied projects. The column "defects" in Table 3 represents the total number of post-release defects that were identified each version. Yatish *et al.* [99] extracted the post-release defects that are associated with each version in two steps. First, they mined the closed issues reports (i.e., post-release defects) that are associated with the studied projects in the JIRA Issue management system. Second, they linked these issue reports to their corresponding versions by using the earliest available version from the affected versions field that is provided in the issue report. Each closed issue report represents a post release defect.

These chosen projects exhibit diversity along multiple dimensions. Such diversity helps us avoid conclusion instability and ensures that our obtained results generalize even on the datasets that are not a part of our study. We briefly explain below the various dimensions of diversity that our study exhibits.

Domain Diversity. We study projects that are from vastly different domains. For instance, we include projects that provide a NoSQL database (Hbase, Hive), search API (Lucene), an integration framework (camel) among our studied projects. The "Description" column in Table 3 provides a brief description of each of the studied projects.

Size Diversity. We study projects that vastly differ in size. From Table 3, we observe that the number of modules in our studied projects ranges from 559 to 8,714 and the total lines of code in our studied datasets range from 63.52k to 529.04K.

Defective Ratio Diversity. Our studied projects have defective ratios ranging from 2.20% to 33.80%. Such diversity ensures that we include projects of different quality in our study.

Release Diversity. We study projects that include a variety of versions defined by different durations. The "Duration (months)" column in our Table 3 provides the number of elapsed months between the studied successive versions. From Table 3, we observe that we have versions that have both shorter and longer elapsed time between the successive versions. For example in the ActiveMQ project, Version 5.1.0 was released in 2008-05-01, while version 5.2.0 was released almost after a year in 2009-06-30, and it was followed up with the next version 5.3.0 in another 4 months in 2009-10-23. While these are subsequent releases, we also include the Version 5.8.0 which was released 3 years in 2013-02-06 after Version 5.3.0 and has several other versions in between 5.3.0 and 5.8.0. Contrastingly, in the case of the HBase project, the duration between the studied versions are consistent and are only two months apart.

Finally, in addition to the code metrics that are provided by Yatish *et al.* [99], we extracted the SNA metrics for each version of each software project. To do so, we collected the source code of these projects. However, we weren't able to locate the source code for the versions Derby 10.3.1.4 and JRuby 1.7. Therefore, we do not include these two versions in our study.

4.2 Experiment Setup

4.2.1 Step 1: Metrics Extraction

To understand the effectiveness of the SNA metrics, we first extract the code and SNA metrics that we describe in Section 3.1 for each version of the studied projects.

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:59:26 UTC from IEEE Xplore. Restrictions apply.

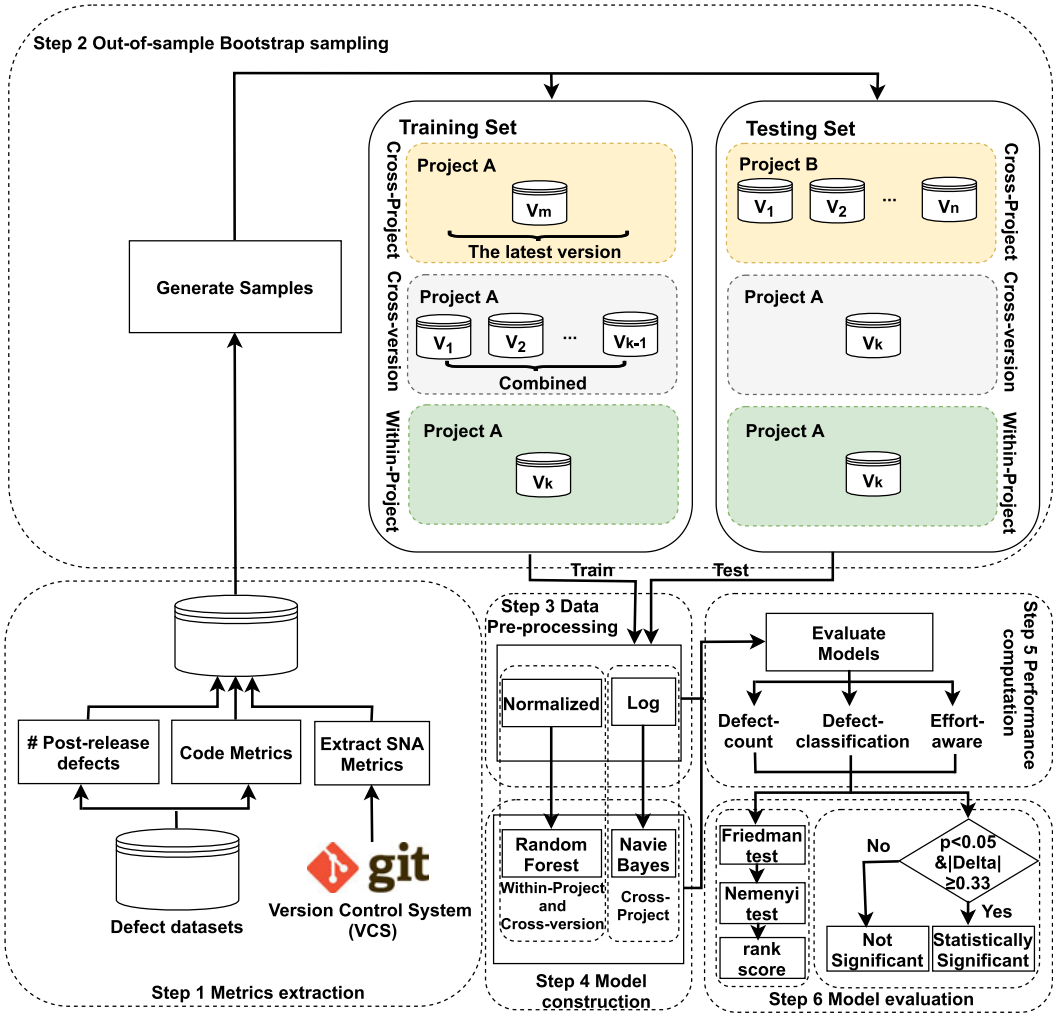

Fig. 2. Overview of our study design.

Code Metrics. We use the code metrics by Yatish *et al.* [99]. The 54 extracted code metrics for each version are provided in Table 2.

SNA Metrics. We extract the SNA metrics from the dependency network of a software system. We apply the Understand tool[4] to the source code of each version to construct the dependency network for each version of each project. Then we compute the SNA metrics of each node by using Ucinet tool.[5] In total, we extract 40 EN metrics and 24 GN metrics as shown in Table 2 in Section 3.1.2.

4.2.2 Step 2: Out-of-Sample Bootstrap Sampling

After we extract the metrics for each version of each studied project, we resample them 100 times with out-of-sample bootstrap sampling. We do so to ensure the statistical robustness of our results [85], [87].

The out-of-sample bootstrap procedure resamples the data points from the dataset with replacement to create a *train* set. The data points that were not sampled as a part of the *train* set are set aside as the *test* set. We repeat this procedure 100 times to create 100 *train* sets and 100 *test* sets. The generated *test* sets have the same distributional properties as the *train* set.

Even when the *test* sets can be obtained without such a resampling procedure, using the boostrap generated *train* sets has several advantages. It helps us account for the non-deterministic nature of the used machine learning models and helps us ensure the generalizability of the obtained results. For instance consider a commonly used SDP model like Random forest which is inherently non-deterministic. Therefore constructing the model just once on a given dataset might overestimate or underestimate the performance of the model. In addition, a bootstrap procedure helps us to non-parametrically approximate the underlying data distribution. By training our model on bootstrap generated *train* sets, we ensure that we do not just draw our insights based on the model trained on just the studied dataset. Rather we ensure that the generated insights are based on the multiple samples drawn from the underlying distribution that generated the original dataset [20], [21], [84]. In addition, as Harrell [21] argues performance estimates computed by the bootstrap procedure better quantify the variance in the computed performance estimates.

Therefore, we use the *train* sets to train the models and the *test* sets to compute the performance measures. Depending

4. https://scitools.com/
5. https://sites.google.com/site/ucinetsoftware/home

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:59:26 UTC from IEEE Xplore. Restrictions apply.

TABLE 3
Overview of the Studied Software Projects

Project	Description	Versions	KLOC	#modules	#edges	#defective module	#defects	Duration (#month)
ActiveMQ	Open source messaging server.	5.0.0	141.42	1,869	9,192	291 (15.57%)	603	2007-12-07
		5.1.0	152.70	1,943	9,640	153 (7.87%)	287	5
		5.2.0	167.50	2,011	9,961	218 (10.84%)	478	14
		5.3.0	159.74	2,011	11,968	231 (11.49%)	681	4
		5.8.0	228.26	2,339	18,957	173 (7.40%)	346	39
Camel	Enterprise integration framework to integrates systems producing and consuming data.	1.4.0	74.21	1,481	7,608	280 (18.90%)	430	2009-01-19
		2.9.0	381.86	7,020	45,871	199 (2.83%)	518	35
		2.10.0	425.44	7,805	50,861	230 (2.95%)	581	6
		2.11.0	480.81	8,714	56,523	192 (2.20%)	464	10
Derby	Relational database software.	10.2.1.6	410.24	1,941	20,347	656 (33.80%)	2013	2006-10-06
		10.5.1.1	529.04	2,672	27,318	383 (14.33%)	1445	31
Groovy	A programming language	1.6.0.Beta1	71.38	608	3,275	63 (10.36%)	142	2008-05-02
		1.5.7	63.52	559	2,787	25 (4.47%)	111	5
		1.6.0.Beta2	78.73	670	3,794	75 (11.19%)	184	5
HBase	A non-relational distributed datasbase.	0.94.0	245.92	1,041	8,175	216 (20.70%)	792	2012-05-14
		0.95.0	503.63	1,600	12,308	378 (23.63%)	1115	11
		0.95.2	501.28	1,790	14,193	482 (26.92%)	1262	4
Hive	A Data warehouse software built on top of Hadoop.	0.9.0	225.15	1,311	8,595	247 (18.80%)	377	2012-04-30
		0.10.0	282.84	1,444	9,772	174 (12.05%)	351	8
		0.12.0	463.55	2,493	16,894	208 (8.34%)	339	9
JRuby	The Ruby Programming Language on the JVM.	1.1	104.71	689	5,849	87 (12.63%)	173	2008-04-05
		1.4	148.61	915	8,975	180 (19.67%)	376	19
		1.5	167.73	1,063	10,094	79 (7.43%)	180	6
Lucene	Text search engine library.	2.3.0	101.10	798	3,051	195 (24.40%)	428	2008-01-23
		2.9.0	171.84	1,363	9,518	272 (19.96%)	540	20
		3.0.0	169.24	1,331	9,459	155 (11.65%)	347	2
		3.1.0	239.54	1,816	14,346	93 (5.12%)	176	16
Wicket	Component oriented web application framework.	1.3.0-beta1	104.53	1,665	8,822	101 (6.07%)	169	2007-04-23
		1.3.0.beta2	108.34	1,756	9,372	130 (7.40%)	194	3
		1.5.3	164.95	2,565	14,737	105 (4.10%)	244	51

In each of the project, in the "Duration (months)" column, for the first studied version, we only provide its release date.

on the SDP context, the composition of the dataset (the way the software projects/versions are grouped together) and the method of conducting the 100 out-of-sample bootstrap varies as follows:

Within-Project Context. The *train* and *test* sets are generated by conducting a 100 out-of-sample bootstrap on each individual version. Each version of a project is considered as a dataset in the Within-project context.

Cross-Version Context. The *train* set is generated by conducting a 100 out-of-sample bootstrap on the dataset that is created by combining all versions of project that are released prior to the current version of the project. As described in Section 3.2.2, the *train* set is generated by resampling the dataset made by combining versions $V_1, V_2, \ldots, V_{k-1}$ of project A, where V_n is the version of Project A for which defects are to be predicted. The *test* remains the same in this context for all the 100 times. Therefore, a model is trained 100 times on 100 different resampled *train* sets, but is tested 100 times against the same data points from the version V_k of Project A.

Cross-Project Context. The *train* set is generated by conducting a 100 out-of-sample bootstrap on the latest available version from one project (For example Project A). The latest available version of one project constitutes as the dataset in the Cross-project context similar to the studies by Rahman and Devanbu [68] and Ma *et al*. [46]. The *test* set is created by considering each version of the target project (for example, version V_n of Project B) i.e., the project for which the defects are to be predicted. Similar to the Cross-version context, while there are 100 different resampled *train* sets, the *test* set remains the same.

4.2.3 Step 3: Data Pre-Processing

In the Within-project and Cross-version contexts, as recommended by several prior studies [92], [104], we normalize the *train* and *test* sets. By doing so we ensure that all the studied metric values vary between 0 and 1.

In the Cross-project context, we perform a log transformation to standardize the studied metrics across the *train* and *test* sets. We do so as we follow the approach outlined by Cruz and Ochimizu [15]. Such a procedure accounts for the concept drift that is common in the Cross-project context. We provide more details on why we choose the approach outlined by Cruz and Ochimizu [15] is provided in the following step.

4.2.4 Step 4: Model Construction

Models are constructed 100 times on the resampled datasets from each of the studied projects with hyperparameter

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:59:26 UTC from IEEE Xplore. Restrictions apply.

tuning to ensure that our models fit the datasets well [88]. Therefore, six models are constructed for each scenario on the six metric families (for details about metrics refer to Section 3.1) on each *train* set for each project per bootstrap iteration. Therefore, for each SDP scenario, for each project, we build 600 models.

We investigate three different SDP contexts as outlined in Section 3.2.2 in our study. In order to ensure the validity of our findings, we choose the best baseline model in each of the studied SDP context to study the effectiveness of SNA metrics across the SDP contexts as follows:

Within-Project Context. In the Within-project context, we use the Random Forest learner. We do so as Ghotra *et al.* [24] through a comprehensive benchmarking study, found Random Forest models to be among one of the best performing models in the Within-project context. In addition, Random Forest model is widely used by prior studies for the Within-Project defect prediction context [69], [99]. We use the `RandomForestRegressor` function from the `Scikit-learn`[6] Python package to build our Defect-count models, and the `RandomForestClassifier` function from the `Scikit-learn`[7] python package to build our Defect-classification and Effort-aware models.

Cross-Version Context. In the Cross-version context, we use the same Random Forest learner as the Within-project context. We do so as Amasaki [2] recently found that the difference in performance between the model constructed with the Random Forest learner and the top performing model in the Cross-version context (LACE2+NaiveBayes proposed by Peters *et al.* [65]) to be small. Moreover, the Random Forest model is more accessible to practitioner and simpler than the LACE2+NaiveBayes model.

Cross-Project Context. In the Cross-project context, we use the log+NavieBayes model proposed by Cruz and Ochimizu [15]. We do so as Herbold *et al.* [33] found it to be the best performing model in the Cross-project context. The log +NaiveBayes model first performs a log transformation to standardize the studied metrics across the training and testing projects as we outline in the previous step. Following which, we build a Naive Bayes model on the log transformed data. We use the `BayesianRidge` function from the `Scikit-learn`[8] Python package to build our Defect-count models, and the `GaussianNB` function from the `Scikit-learn`[9] python package to build our Defect-classification and Effort-aware models.

Hyperparameter Tuning the Studied Models. To ensure that the constructed models fit the datasets well, we use `GridSearchCV` from the `Scikit-learn`[10] Python package to tune the hyperparameters of learners. For Random Forest learner, we tune the "n_estimators" in the range $\{50, 100, 200, 500, 600\}$, and "max_depth" in the range $\{3, 5, 10, 15\}$. Similarly, for the Bayesian Ridge learner, we tune the the "alpha_1", "lambda_1", "alpha_2" and "lambda_2" in the range $\{1e-7, 1e-6, 1e-5, 1e-4, 1e-3\}$. The `GaussianNB` function does not present any hyperparameters to tune. We tune these hyperparmaters for each bootstrap iteration. Then on each bootstrap iteration we choose the best performing hyperparameters for the learners.

4.2.5 Step 5: Performance Computation

We compute the performance measures for the constructed models across the three studied SDP scenarios in each of the SDP contexts. We use the performance measures detailed in Section 3.3. For each SDP scenario, the performance of the model constructed on each of the six metric families is noted. In both the Within-project and Cross-version contexts, on each bootstrap iteration the 600 constructed models are tested on *test* sets (100 models per *train* and *test* set for each studied metric family). These tests result in 600 performance scores for each of the studied performance measure. However, in the Cross-project context, the 600 models that are constructed for a given project are tested on each version present in each of the other 8 studied projects (since the *test* set in a Cross-project context is each version from all the other studied projects). Therefore, for each project in the Cross-project context, 4,800 performance scores are computed (six metric families $\times$ 100 iterations $\times$ 8 projects).

4.2.6 Step 6: Model Evaluation

To determine which metrics yield the best performing model for the studied SDP scenarios, we use a Friedman test with a post-hoc Nemenyi test, as suggested by Demšar *et al.* [19]. The Friedman test [22] checks if there are statistically significant differences between performance scores generated by the models trained on the studied metric families. If the Friedman test finds statistically significant differences, we employ the post-hoc Nemenyi test [59] to obtain a ranking of the computed performance scores. Such a ranking helps us understand the metric family that generates the best performing models.

However, the post-hoc Nemenyi test [59] typically yields overlapping ranks [32], which makes it harder to rank the effectiveness of the studied metrics. Therefore, we post process the results of the post-hoc Nemenyi test as suggested by Herbold *et al.* [33] to generate a distinct *rankscore* for each model trained on the studied metric families. The rankscore ranges between 0 and 1. A high rankscore for a model trained on a given metric family signifies that the given metric family generates the best performing model compared to the other metric families on the studied performance measure.

As mentioned in the previous step, for each dataset, 100 performance scores are computed for each constructed model on each of the metric families (In the Cross-project context, 800 performance scores, across the 8 other studied projects). Then for each dataset, we first compute the median of the performance scores. We then apply the Friedman test on the median performance scores across the studied datasets. If the results are significant, we then apply the post-hoc Nemenyi test to the generated performance scores. We then postprocess the results of the post-hoc Nemenyi test and generate the rankscore of each metric family for a

6. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html
7. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
8. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.BayesianRidge.html
9. https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html
10. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:59:26 UTC from IEEE Xplore. Restrictions apply.

TABLE 4
The Rankscore Results as Computed on the Median Performance Measures Across the Bootstrap Iterations

Context	Measures	Code	SNA	EN	GN	SM	COM
Within-Project	**Defect-Count model**						
	Adj R^2	0.60	0.20 (2/30)	0.40 (0/30)	0.80 (9/30)	0.00 (1/30)	**1.00** (11/30)
	RMSE	0.60	0.40 (3/30)	0.00 (0/30)	0.20 (1/30)	**1.00** (10/30)	0.80 (7/30)
	Spearman	0.60	0.20 (3/30)	0.00 (0/30)	0.20 (1/30)	**1.00** (8/30)	0.80 (6/30)
	MAE	0.40	0.20 (7/30)	0.00 (4/30)	0.60 (6/30)	**1.00** (13/30)	0.80 (10/30)
	Defect-classification model						
	AUC	0.60	0.40 (4/30)	0.00 (0/30)	0.20 (4/30)	**1.00** (19/30)	0.80 (17/30)
	MCC	0.40	0.60 (7/30)	0.00 (3/30)	0.20 (5/30)	**1.00** (16/30)	0.80 (14/30)
	Recall	0.20	0.40 (9/30)	0.00 (7/30)	0.60 (9/30)	**1.00** (15/30)	0.80 (12/30)
	Brier score	0.60	0.40 (5/30)	0.00 (1/30)	0.20 (3/30)	0.80 (11/30)	**1.00** (13/30)
	Precision	0.60	0.40 (3/30)	0.00 (0/30)	0.20 (2/30)	0.80 (9/30)	**1.00** (11/30)
	Effort-aware model						
	ER	0.00	**1.00** (21/30)	0.80 (18/30)	0.40 (15/30)	0.20 (10/30)	0.60 (16/30)
	CE	0.00	**1.00** (21/30)	0.80 (19/30)	0.60 (18/30)	0.20 (14/30)	0.40 (16/30)
Cross-version	**Defect-Count model**						
	Adj R^2	**0.50**	0.00 (4/21)	**0.50** (7/21)	0.00 (4/21)	0.00 (2/21)	**0.50** (12/21)
	RMSE	**1.00**	0.50 (5/21)	**1.00** (7/21)	0.50 (2/21)	**1.00** (7/21)	**1.00** (7/21)
	Spearman	**0.50**	0.00 (1/21)	0.00 (2/21)	0.00 (4/21)	**0.50** (5/21)	**0.50** (7/21)
	MAE	**1.00**	0.33 (0/21)	0.67 (1/21)	0.33 (0/21)	0.67 (1/21)	0.67 (5/21)
	Defect-classification model						
	AUC	**0.50**	0.00 (2/21)	0.00 (3/21)	0.00 (3/21)	**0.50** (9/21)	**0.50** (10/21)
	MCC	**0.50**	**0.50** (7/21)	0.00 (4/21)	**0.50** (5/21)	**0.50** (8/21)	**0.50** (8/21)
	Recall	**0.50**	0.00 (4/21)	**0.50** (3/21)	0.00 (1/21)	**0.50** (6/21)	**0.50** (4/21)
	Brier score	0.50	0.50 (9/21)	0.50 (8/21)	0.50 (6/21)	**1.00** (15/21)	0.50 (15/21)
	Precision	0.00	**0.50** (13/21)	0.00 (4/21)	**0.50** (17/21)	**0.50** (12/21)	0.00 (10/21)
	Effort-aware model						
	ER	0.00	0.00 (6/21)	**0.50** (10/21)	0.00 (7/21)	0.00 (7/21)	**0.50** (11/21)
	CE	0.00	0.00 (5/21)	**0.50** (8/21)	0.00 (6/21)	0.00 (4/21)	0.00 (6/21)
Cross-Project	**Defect-Count model**						
	Adj R^2	**0.50**	**0.50** (0/30)	0.00 (0/30)	**0.50** (0/30)	**0.50** (0/30)	**0.50** (1/30)
	RMSE	**1.00**	0.50 (0/30)	0.50 (0/30)	0.50 (0/30)	**1.00** (0/30)	0.50 (1/30)
	Spearman	**0.50**	0.00 (4/30)	0.00 (4/30)	**0.50** (3/30)	**0.50** (3/30)	0.00 (1/30)
	MAE	**1.00**	0.50 (3/30)	**1.00** (3/30)	**1.00** (1/30)	0.50 (0/30)	**1.00** (2/30)
	Defect-classification model						
	AUC	**0.50**	0.00 (0/30)	0.00 (1/30)	**0.50** (10/30)	0.00 (0/30)	**0.50** (2/30)
	MCC	**0.50**	0.00 (0/30)	**0.50** (4/30)	0.00 (0/30)	**0.50** (4/30)	**0.50** (4/30)
	Recall	**0.67**	0.33 (0/30)	0.33 (1/30)	0.33 (0/30)	**0.67** (3/30)	**0.67** (2/30)
	Brier score	0.33	0.67 (12/30)	0.67 (6/30)	**1.00** (24/30)	0.33 (0/30)	0.67 (5/30)
	Precision	0.00	0.00 (10/30)	**0.50** (13/30)	**0.50** (19/30)	0.00 (0/30)	0.00 (4/30)
	Effort-aware model						
	ER	0.00	**0.50** (12/30)	**0.50** (14/30)	**0.50** (16/30)	0.00 (1/30)	0.00 (7/30)
	CE	0.00	0.00 (2/30)	0.00 (1/30)	**0.50** (10/30)	0.00 (0/30)	0.00 (2/30)

Each value in the table is represented as R (x/y).
R- rankscore of the model trained on the metrics (given by the column).
x- No. of datasets on which the model trained on the metric family (given by the column) compared to the model trained on the code metrics in the given context have a larger than medium effect size. Please note that it is common for two metric families with similar rankscore R to have slightly different number of datasets).
y- Total number of datasets for the given context in our study. The total number of datasets vary as the way in which the train *set is created varies based on context as outlined in Section 3.2.2.*

given scenario across the studied datasets. We do so for all the studied SDP contexts and scenarios.

We also compute the Wilcoxon-signed rank test [82] and Cliff's Delta effect size test [98] between the models trained on code metrics and each of other studied metric families. However unlike the Friedman and post-hoc Nemenyi test, we compute the Wilcoxon-signed rank test and Cliff's delta effect size test per dataset (not across the studied datasets). Therefore we conduct the Wilcoxon-signed rank test and the Cliff's Delta effect size test directly on the bootstrap results of each of the studied datasets. That is across 100 bootstrap iterations for the Within-project and Cross-version context and 800 iterations for the Cross-project context. Moreover, since we perform multiple pairwise-comparisons, we correct the obtained p-values from the Wilcoxon-signed rank test with Bonferroni correction [5] to control for false positives. We do so to statistically quantify the number of datasets on which models trained on other metric families outperform models trained on code metrics with a statistical significance (P-value < 0.05) and a greater than medium effect size (i.e., if magnitude of Cliff's Delta ≥ 0.33).

5 Results

In this section, we present and discuss our study results related to our research questions. For all the RQs, we follow the experiment setup (as applicable to each SDP context that the RQ deals with) outlined in Section 4.2.

5.1 (RQ1) are SNA Metrics More Effective Than Code Metrics for Within-Project SDP?

5.1.1 Results

Result 1) In Within-project context, for both Defect-count and Defect-classification scenarios, models trained on SM and COM metrics outperform the models trained on other metric families. Table 4 lists the results of the rankscores (refer to

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:59:26 UTC from IEEE Xplore. Restrictions apply.

Section 4.2.6) for the models trained for all the SDP scenarios across the studied datasets. From Table 4, we observe that models trained on SM and COM metrics (both of which contain SNA metrics as a part of it) are consistently ranked at the top for Defect-count scenario. However, the number of datasets on which the models Defect-count models trained on SM and COM metrics outperform the models trained on code metrics with larger than medium effect size remain small (at most 13 out of the 30 studied datasets).

In contrast, for the Defect-classification scenario, we observe that on at least 9 of the 30 studied datasets, the Defect-classification models trained on SM and COM metrics outperform the models trained on code metrics with a larger than medium effect size. In particular, in terms of AUC, MCC and Recall, there are at least 15 out of the 30 studied datasets on which Defect-classification models trained on SM metrics outperform models trained on code metrics with statistical significance and a larger than medium effect size. Such results highlight the effectiveness of SNA metrics alongside with code metrics for the Defect-classification scenario.

Result 2) Models trained on SNA metrics outperform models trained on other metric families for the Effort-aware scenarios across all the studied performance measures. Furthermore, at least on 21 of the 30 studied datasets Effort-aware models trained on SNA metrics outperform the models trained on code metrics with statistical significance and a larger than medium effect size. From Table 4, we also observe that models trained on EN and GN metrics are also consistently ranked above models trained on code metrics (with a statistical significance and a larger than medium effect size on at least 15 out of the 30 studied datasets). Such results highlight the effectiveness of SNA metrics for the Effort-aware scenario.

5.1.2 Analysis

From the above results, we observe that models trained on SM and COM metrics consistently outperform models trained on other metric families for Defect-count and Defect-classification scenarios. We hypothesize that such a result might be because SM and COM metrics might capture richer information about software projects than the other metric families. We arrive at such a hypothesis as prior studies [46], [60], [89], [103] theorize that SNA metrics might provide information that is different or complimentary to code metrics. In addition, combing SNA metrics with code metrics could improve the performance of SDP models.

In order to verify our hypothesis and quantify the information captured by the different metric families, we employ a Principal Component Analysis (PCA)-based analysis similar to Ghotra *et al.* [25]. We quantify the informativeness of each metric family in a given dataset by calculating the number of components that are needed to account for 95% of the data variance information in each metric family. The higher the number of components that are required to account for 95% of data variance information, the higher the informativeness of the metric family. We present the median number of components required by each studied metric family to account of 95% of the data variance information across the 30 studied datasets (presented in Table 3) in Table 5.

TABLE 5
The Median Number of Components Required by Each Metric Family to Account for 95% of Data Variance Information

Metric family	Code	SNA	EN	GN	SM	COM
Median number	22	21	17	8	38	34

Result 3) The SM and COM metrics exhibit higher informativeness than the other metric families (see Table 5). Such a result supports our hypothesis that SNA metrics contained in the SM and COM metrics adds additional information to the code metrics. Which in turn results in models trained on SM and COM metrics outperforming models trained on the other studied metric families.

5.2 (RQ2) are SNA Metrics More Effective Than Code Metrics for Cross-Version SDP?

5.2.1 Results

Result 4) In Cross-version context, Defect-count models trained on code metrics outperform the Defect-count models trained on other metric families across three out the four studied performance measures. From Table 4 we observe that, the number of datasets on which Defect-count models trained on other metric families significantly outperform Defect-count models trained on code metrics in terms of Adj R^2, MAE, RMSE and Spearman is only at most 12. We note that Defect-count models trained on COM metrics (which combine SNA and code metrics) perform slightly better than the Defect-count models trained on code metrics according to Adj R^2 measure.

Result 5) Defect-classification models trained on SM and COM metrics have a similar rankscore to that of models trained on code metrics.. On 3 of the studied performance measures (i.e., AUC, MCC, Recall) the Defect-classification models trained on SM and COM metrics have a similar rankscore to that of models trained on code metrics. However, Defect-classification models trained on SM metrics outperform Defect-Classification models trained on code metrics across 2 out of 5 studied performance measures (i.e., Brier score and Precision). On these two performance measures, from Table 4, we observe that Defect-classification models trained on SM metrics outperform the models trained on code metrics on at least 12 out of the 21 studied datasets with statistical significance and a larger than medium effect size. These results, demonstrate the potential usefulness of considering the SNA metrics alongside code metrics for the Defect-classification scenario in Cross-version context.

Result 6) Effort-aware models trained on EN and COM metrics outperform Effort-aware models trained on other metric families. From Table 4, we observe that EN and COM metrics rank at the top. Interestingly, we observe that Effort-aware models trained on all the other metric families are ranked similarly. Such a result indicates that except for EN and COM metrics, other metric families yield Effort-aware models of similar performances. Such a result indicates that EN and COM metrics improve the performance of Effort-aware models.

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:59:26 UTC from IEEE Xplore. Restrictions apply.

5.2.2 Analysis

From the above results, we observe that for the Defect-count scenario, code metrics are more effective than the other metric families. Such a result is in contrast with the observed results in the Within-project context and other SDP scenarios in the Cross-version context. For the other SDP scenarios of the Cross-version context, where SNA (particularly, EN metrics) metrics by themselves or in combination with code metrics improve the performance of models over models built using code metrics only. We hypothesize that such a result might be because, as software systems evolve, the defective status of a large number of modules across multiple versions remains unchanged due to a couple of reasons. First, the modules that had defect(s) in one version tend to be defective across multiple versions [80]. Therefore, some modules remain defective across multiple versions. Second, many modules are not modified across multiple versions, therefore their defective status typically remains unchanged (especially for non-defective modules). For instance, many of the modules which were not defective in version 1.0 are typically not modified for version 1.1 (since they are already performing their intended function without any defects) therefore their defective status remains 'non-Defective' across both versions 1.0 and 1.1. Therefore only a few modules change their defective status across versions. We refer to these modules whose defective status remains unchanged across versions as "unchanged-status modules". Because of the aforementioned reasons, a large number of the data points have exactly the same defective status across two successive versions (i.e., the train and test data).

As Rahman and Devanbu [68] show that code metrics typically have a higher stasis (i.e., they do not change much) on unchanged-status modules. Which would imply that there exists a large number of modules across successive versions (i.e., the train and test data) which are more similar in terms of code metrics across successive releases than in terms of the SNA metrics. In turn, Defect-count models trained on code metrics could easily predict the defective status of these unchanged-status modules. In other words, the models trained on just code metrics would repeatedly predict the same modules to be defective over and over again across the versions. Therefore, if the number of unchanged-status modules across versions is high, then the performance of code metrics on these modules might dominate the performance of the Defect-count models trained on metric families which contain SNA metrics. Furthermore, if the Defect-count models trained on code metrics perform well only on the modules that exhibit the same defective status across versions, and not on the other modules, the SNA metrics might still be useful by themselves are in combination with code metrics in predicting the number of defects in Cross-version context.

To validate our hypothesis, we do three things: First we compute the number of software modules that have the same defective status (unchanged-status modules) across each version of a given project. We also compute the number of modules that changed their defective status (changed-status modules) and the number of newly added modules (new modules). We do so to identify if there exists a significant number of unchanged-status modules across each version for the studied software projects.

Second, we verify if the stasis of code metrics are indeed higher than that of SNA metrics for these unchanged-status modules. To do so, we compare the stasis of code and SNA metrics on the unchanged-status modules across each version of the studied projects. We compute the stasis of the metrics using the approach outlined by Rahman and Devanbu [68]. We compute the spearman correlation between each metric from both the code and SNA metric families across each two successive versions and average them. We present these average values as the stasis of code and SNA metric families respectively. We do so to see if the superior performance of Defect-count models trained on code metrics are due to the high stasis of code metrics on unchanged-status modules.

Third, for each Defect-count model trained on the Cross-version train set (refer Section 3.2.2), we split the Cross-version test set into unchanged-status, changed-status, and new software modules. We then calculate the performance of the Defect-count models on them separately. We do so for the Defect-count models trained on all the metric families across all the studied projects. We do so to identify if the Defect-count models trained on code metrics outperform other studied metric families only on the unchanged-status modules or across all the types of modules. If the code metrics outperform the SNA metrics only on unchanged-status modules (and does not outperform the other metric families on the changed-status modules) then such a result would support our hypothesis and vice versa.

Except for ActiveMQ 5.8.0, Camel 2.9.0, HBase 0.95.0, JRuby 1.4, Lucene 2.9.0 and Wicket 1.5.3, all the versions across the studied projects (15 of 21) have 50-87.6% unchanged-status modules. Table 6 shows the number of unchanged-status, changed-status and new software modules per version. The results in Table 6 supports our hypothesis that the presence of a high number of unchanged-status modules favours the Defect-count models that are trained on code metrics.

Across all the studied projects, we observe that the code metrics exhibit significantly higher stasis than SNA metrics for the unchanged-status modules. Such a result, along with the high number of unchanged-status modules across the different versions, explain the superior performance of Defect-count models trained on code metrics in the Cross-version context. We compute the stasis values of the code and SNA metrics in the unchanged-status modules across the studied projects, and observe that code metrics exhibit an extremely high median stasis of 0.97 on the unchanged-status modules with a very limited range between 0.6 and 1. In contrast, the SNA metrics have a median stasis of 0.86 and show a significantly wider range varying from -1 to 1. Furthermore, a Wilcoxon Signed Rank test between the computed stasis values of the code metrics and SNA metrics of the unchanged-status modules across the studied project exhibits a statistically significant difference (p-value $<$ 0.05) with a small cliff's delta effect size of 0.3.

Results 7) Defect-count models trained on code metrics do not outperform models trained on other metric families for changed-status modules across all the studied versions and performance measures. Table 7 shows the results of the rankscore for the Defect-count models trained on unchanged-status, changed-status and new modules for all the studied metric families. We note that, while the Defect-count model trained on code metrics

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:59:26 UTC from IEEE Xplore. Restrictions apply.

TABLE 6
The Number of Unchanged-Status, Changed-Status and New Modules per Version Across All the Studied Projects

Project	Versions	# Unchanged	#Changed	#New
ActiveMQ	5.1.0	81.7%	13.7%	4.6%
	5.2.0	84.3%	10.0%	5.7%
	5.3.0	72.2%	10.7%	17.1%
	5.8.0	2.6%	0.4%	97.0%
Camel	2.9.0	12.5%	3.3%	84.2%
	2.10.0	86.5%	2.8%	10.7%
	2.11.0	85.4%	3.5%	11.1%
Derby	10.5.1.1	50.3%	17.3%	32.4%
Groovy	1.5.7	83.6%	6.1%	10.3%
	1.6.0.Beta2	83.7%	5.4%	10.9%
HBase	0.95.0	40.9%	13.6%	45.5%
	0.95.2	60.5%	22.1%	17.4%
Hive	0.10.0	74.9%	15.6%	9.5%
	0.12.0	50.0%	6.3%	43.7%
JRuby	1.4	42.7%	10.9%	46.4%
	1.5	70.0%	14.9%	15.1%
Lucene	2.9.0	38.1%	10.3%	51.6%
	3.0.0	87.6%	10.8%	1.6%
	3.1.0	55.2%	9.7%	35.1%
Wicket	1.3.0.Beta2	82.2%	7.8%	10.0%
	1.5.3	18.1%	1.3%	80.6%

• *Unchanged - Unchanged-status modules.*
• *Changed - Changed-status modules.*
• *The first version from each software project is not included as it is not possible to identify the Unchanged-status, Changed-status and new modules for the first version of the software where all the modules are New.*

performs well on unchanged-status and new modules, it does not perform as well on changed-status modules. These results in addition with the high number of unchanged-status and new modules for each version across the studied projects could potentially indicate the reason for the superior overall performance of the Defect-count models trained on code metrics.

Results 8) Defect-count models trained on EN, GN and SNA metrics are useful at identifying the number of defects in changed-status modules across the software versions on three of the four studied performance measures. In addition, From Table 7, we observe that across all the performance measures on at least 11 out of the 21 studied datasets, Defect-count models trained on EN, GN and SNA metrics significantly outperform Defect-count models trained on other metric families (including code metrics). Such a result indicates that SNA metrics indeed improve the performance for the Defect-count scenario in the Cross-version context. Therefore these results reiterate the need for considering SNA metrics even as a part of the Defect-count scenario, where at first glance, it appears that SNA metrics might not be as effective.

5.3 (RQ3) are SNA Metrics More Effective Than Code Metrics for Cross-Project SDP?

5.3.1 Results

Result 9) In the Cross-project context, Defect-count models trained on the code metrics outperform the Defect-count models trained on other metric families across all the studied performance measures. From Table 4 we observe that the number of datasets on which the Defect-count models trained on other metric families that outperform the Defect-count models trained on code metrics is less than 4. Even the SM and COM metrics, which contain code metrics are not as effective for the Defect-count scenario in the Cross-project context.

Result 10) Defect-classification models trained on the studied SNA metrics do not outperform the Defect-classification models trained on code metrics on 3 out the 5 studied performance measures (i.e., AUC, MCC and Recall). Across these three performance measures, only Defect-classification models trained on COM metrics have similar performance (not higher) as the Defect-classification models trained on code metrics.

Defect-classification models trained on GN metrics outperform the Defect-classification models trained on all the other metric families on 2 out of 5 the studied performance measures. From Table 4 we observe that the models trained on GN metrics exhibit superior performance than models trained on code metrics with a larger than medium effect size on at least 19 out of the 30 studied datasets in terms of Brier-score and Precision. Such a result showcases that while there might be some slight advantage that can be potentially realized by using SNA metrics (i.e., GN metrics). However, we cannot conclude that using SNA metrics

TABLE 7
The Rankscore Results as Computed on the Defect-Count Model for Unchanged-Status, Changed-Status and New Modules Across All the Studied Datasets

Measures		Code	SNA	EN	GN	SM	COM
Unchanged-status	Adj R^2	**0.50**	0.00 (1/21)	0.00 (2/21)	0.00 (2/21)	0.00 (1/21)	**0.50** (7/21)
	RMSE	**1.00**	0.50 (1/21)	0.50 (1/21)	0.50 (1/21)	**1.00** (2/21)	**1.00** (4/21)
	Spearman	**0.50**	0.00 (1/21)	0.00 (1/21)	0.00 (1/21)	**0.50** (2/21)	**0.50** (5/21)
	MAE	**1.00**	0.33 (0/21)	0.67 (0/21)	0.33 (0/21)	0.67 (0/21)	0.67 (3/21)
Changed-status	Adj R^2	0.33	0.33 (11/21)	**0.67** (18/21)	**0.67** (21/21)	0.00 (1/21)	**0.67** (21/21)
	RMSE	0.50	**1.00** (19/21)	**1.00** (17/21)	**1.00** (20/21)	**1.00** (19/21)	**1.00** (14/21)
	Spearman	0.00	**0.50** (16/21)	**0.50** (18/21)	**0.50** (18/21)	0.00 (6/21)	0.00 (8/21)
	MAE	0.50	**1.00** (18/21)	**1.00** (20/21)	**1.00** (18/21)	0.50 (8/21)	0.50 (6/21)
New	Adj R^2	**0.50**	0.00 (3/21)	**0.50** (12/21)	**0.50** (13/21)	0.00 (4/21)	**0.50** (15/21)
	RMSE	**1.00**	**1.00** (8/21)	**1.00** (7/21)	**1.00** (7/21)	**1.00** (10/21)	**1.00** (10/21)
	Spearman	**1.00**	**1.00** (5/21)	**1.00** (5/21)	**1.00** (8/21)	**1.00** (8/21)	**1.00** (8/21)
	MAE	**1.00**	0.50 (4/21)	**1.00** (8/21)	0.50 (4/21)	0.50 (6/21)	**1.00** (10/21)

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:59:26 UTC from IEEE Xplore. Restrictions apply.

TABLE 8
The Observed Effectiveness of SNA Metrics Over Code Metrics Across the Studied SDP Scenarios and Contexts in Our Study

Metric family	Within-project			Cross-version			Cross-project		
	D-classification	D-count	E-aware	D-classification	D-count	E-aware	D-classification	D-count	E-aware
Code metrics				√	√		√	√	
SNA metrics			√						
EN metrics						√			
GN metrics									√
SM metrics	√	√							
COM metrics	√	√				√			

would consistently yield better results than using training models with just code metrics for Defect-classification scenario in the cross-project context.

Result 11) Effort-aware models trained on GN metrics outperform the Effort-aware models trained on all the other metric families. From Table 4 we observe that the models trained on GN metrics exhibit superior performance than models trained on code metrics with a larger than medium effect size on atleast 10 out of the 30 studied datasets.

5.3.2 Analysis

Models trained on code metrics outperform models trained on other metrics for the Defect-count and Defect-classification scenario. Such a result is not particularly surprising as Hosseini *et al.* [34] through a meta analysis find that models trained on code metrics typically perform the best in the Cross-project context. However, in contrast to the findings of Hosseini *et al.* [34] we find that models trained on GN metrics outperform the models trained on code metrics for Defect-classification (on 2 out of the 5 studied performance measures) and Effort-aware scenarios. In addition, for the Defect-classification scenario, on the MCC and Recall measures, models trained on SM and COM metrics (which contain SNA metrics) perform at least as well as models trained on code metrics, if not outperform them.

We argue that the different information in the SNA metrics (in particular the GN metrics) help break the ceiling effect [51] exhibited by code metrics. As Menzies *et al.* [51] explain, code metrics contain only limited information content. To improve the performance of models in the Cross-project context, Herbold *et al.* [32] argue that metrics other than code metrics need to be considered to train Defect-classification models. They argue that other metrics might help us break the ceiling effect exhibited by code metrics due to their limited information content. Since the SNA metrics capture information that is not typically captured by the code metrics, we argue that SNA metrics (in particular the GN metrics) help break the ceiling effect and in turn provides comparable and at times better performance for the Defect-classification and Effort-aware scenarios.

6 Discussion

Table 8 summarizes the results that we observed from Section 5. From Table 8, we see that SNA metrics are useful in all the studied SDP contexts and several of the studied SDP scenarios. Such a result is in contrast with results presented in Table 1. Table 1 highlights that there is no clear consensus on the effectiveness of SNA metrics over the code metrics across the different SDP contexts and scenarios. In this section, we highlight potential reasons why prior studies portray a mixed picture of the effectiveness of SNA metrics over code metrics across the three SDP contexts.

6.1 Within-Project Context

Prior studies that do not find SNA metrics to be more effective than code metrics in the Within-project context only use a small subset of SNA metrics in their studies. From Table 8 we clearly observe that the models trained on SM and COM metric families perform better than models trained on code metrics across the Defect-count and Defect-classification scenario. For the Effort-aware scenario, SNA metrics prove to be more effective than code metrics. While most prior studies come to a similar conclusion as ours [46], [60], [67], [103], Prateek *et al.* [66] and Wu [94] find code metrics to be equally or more effective than SNA/SM metrics. We argue that these studies come to a different conclusion as they used only a small subset of the available SNA metrics in their studies. Prateek *et al.* [66] used only 12 SNA metrics and Wu [94] used only five SNA metrics in their studies (in contrast we used 64 SNA metrics). However the studies that found SNA metrics to be more effective than code metrics typically used more than 60 SNA metrics (including our study).

6.2 Cross-Version and Cross-Project Contexts

We argue that prior studies do not observe the effectiveness of SNA metrics over code metrics in both the Cross-version and Cross-project contexts because they do not consider EN and GN metrics separately. As we observe from Table 1, prior studies [46], [66], [67] state that the models trained on code metrics perform similarly or better than the models trained on SNA metrics. They do so for the Effort-aware scenarios of both the Cross-version and Cross-project contexts. From Table 4 we could observe similar findings if we only consider SNA and SM metrics in our study. However, the effectiveness of the SNA metrics in the Cross-version and Cross-project contexts can be observed only when the GN, EN and COM metrics are considered separately (as we can see from Table 8).

We observe that considering these metric families separately showcase the effectiveness of SNA metrics because GN and EN metrics capture different characteristics of a software system. We argue that considering GN and EN metrics separately helps us avoid Simpson's paradox [91]. Simpson's paradox is a phenomenon where combining separate views of data does not necessarily result in a more

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:59:26 UTC from IEEE Xplore. Restrictions apply.

informative grouping. Rather, it might in fact weaken individual effects exhibited by each view [91]. In fact, such a grouping may even present a signal that is not faithful to either of the underlying views [91]. In our case, GN metrics capture the dependency characteristics of a given module in the context of the whole software system (i.e., a global view). Whereas, EN metrics capture the dependency characteristics of a given software module with respect to its neighboring nodes (i.e., a local view). Therefore, we argue that considering the GN and EN metrics together as SNA metrics might weaken the strength of the signal that they may individually contain. Therefore, we argue that to avoid Simpson's paradox, future studies should not only consider the combined effects of different views of data. Instead, they should also separately consider the impact of each view by itself in addition to the combined view.

Summary
We argue that prior studies do not clearly capture the effectiveness of SNA metrics because of two key reasons: 1) Some studies only consider a small subset of the available SNA metrics in their study 2) They do not consider the GN and EN metrics separately in their studies.

7 Implications

In this section, we outline the implications that can be inferred from our results. We present these implications to guide researchers and practitioners who seek to build SDP models in the future.

Implication 1) Researchers and practitioners should consider using SNA metrics by themselves or alongside code metrics when building models for any of the studied SDP scenarios. From the results presented in Section 5, we observe that SNA metrics improve the performance of models for 5 out of the 9 studied SDP scenarios (three SDP scenarios across three SDP contexts). More specifically, except for the Defect-count and Defect-classification scenario for both Cross-version and Cross-project SDP contexts, we find that models trained on SNA metrics by themselves (GN or EN metrics) or SNA metrics alongside code metrics (SM or COM metrics) perform equally or at least slightly better (if not more).

However, none of SDP studies in the recent years consider SNA metrics in their studies when building models. For instance, consider these handful of studies published in ICSE or TSE 2018 and 2019. None of these studies that built models for the Within-project, consider the SNA (or GN metrics) [34], [44], [81], [88], [99]. In addition, the effort of computing SNA metrics is not significantly greater than computing just the code metrics. For instance, prior studies [99] typically use the Understand tool to compute code metrics. To compute SNA metrics, as we explain in Section 4.2, in addition to using the Understand tool, we use the UciNet tool. Therefore, given the potential performance improvements and the ease of computing the SNA metrics, we urge the researchers and practitioners to consider using SNA metrics alongside the code metrics in their SDP studies.

Implication 2) EN and GN metrics should be considered separately. EN and GN metrics have different impact on models depending on the context and scenario. For instance, in the Cross-project context, for the Effort-aware scenario, models trained on just GN metrics have better performance compared to models trained on other metric families. Furthermore, in the Cross-version context, Effort-aware models trained on the EN metrics exhibit improved performance over the models trained on other metric families. These results indicate the necessity of considering the EN and GN metrics separately for SDP.

8 Threats to Validity

In this section, we discuss the threats to validity of our presented results and inferences.

Construct Validity. The number of observed post-release defects in our studied datasets and the version in which the defects were introduced might not be accurate. Da Costa *et al.* [16] suggested that the realistic estimation of the earliest affected version for a given defect depends heavily on development team of the software project. In practice, a development team may provide incomplete or wrong estimation of the number of defects or the version in which the defect was originally introduced. Such inconsistencies may generate false negatives. To mitigate such instances, we construct our experiments on a dataset provided by Yatish *et al.* [99], where they calculated the number of post-release defects for each module with a realistic approach over the traditionally used heuristic approach. The used approach seeks to eliminate some of the associated noise with aforementioned inconsistencies. However, some amount of noise in the dataset cannot be avoided. Tantithamthavorn *et al.* [86] observed that such noise impacted the recall of the SDP classifiers. Therefore, we suggest that future studies should revisit the effectiveness of the SNA metrics across various SDP contexts and scenarios with an approach similar to ours using datasets devoid of noise.

Another threat to construct validity is that we do not consider multiple methods outlined by Zhang *et al.* [101] to aggregate the code metrics at the file-level. Zhang *et al.* [101] showed when building Defect-classification models, using code metrics aggregated through several aggregation schemes is better than using summation. However, in our study, though we do not use all the aggregation methods that Zhang *et al.* [101] used to aggregate code metrics, we still use the mean, min and max methods of aggregation in addition to the summation method. We observe that our use of several aggregation methods helps mitigate the threat outlined by Zhang *et al.* [101] for the Defect-classification scenario. Furthermore, for the case of Defect-count and Effort-aware scenarios, Zhang *et al.* [101] showed that using code metrics aggregated through summation is as effective as using code metrics aggregated through several different aggregation schemes. Therefore, we argue that our results are still relevant to the current state-of-practice. However, we acknowledge that investigating the effectiveness of SNA metrics over code metrics aggregated through several aggregation schemes for the Defect-classification scenario is an interesting avenue for future research.

Another threat to construct validity is that we quantify the informativeness of each metric family by calculating the number of components that are needed to account for 95% of the data variance information similar to Ghotra *et al.* [25]. However, on the off chance that any studied metric family is purely

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:59:26 UTC from IEEE Xplore. Restrictions apply.

composed of random values, our procedure for calculating the informativeness of the metric families could be impacted by that metric family composed of random values. As such, it presents a threat to the analysis that we conduct in Section 5.1.2. However, such a result is unlikely, since the metric families that we find as the most informative (i.e., SM and COM metrics) are indeed the best performing metrics across all the studied datasets for the Within-project context. Such a result would not be possible if the SM and COM metrics were random values. Nevertheless, we encourage the future studies to revisit the informativeness of our studied metrics with more robust methods of calculating informativeness.

Finally, prior studies show that the results obtained from Defect-count and Defect-classification models could potentially be influenced by the module size [41]. In particular, if the size of a module heavily correlates with the number of defects or defectiveness of a module, then a more trivial SDP models could simply predict large modules as defective. Therefore, to investigate if such a threat exists in our studied projects, we compute the spearman correlation between the lines of code and the number of defects in a module, across all the studied datasets. We observe that the lines of code metric is not highly correlated with the number defects across the studied datasets. We observe that the median spearman correlation across all the studied datasets is only 0.3. Such a result asserts that the Defect-count and Defect-classification models do not rely solely on the size of the inspected modules in our studied projects.

External Validity. We analyze the effectiveness of the SNA metrics over code metrics on nine fixed open source software projects developed in JAVA. Though the studied projects are diverse, there is a threat that our findings may not generalize for projects with different module sizes and version definitions. In addition, as Zhang *et al.* [100] even if different software projects appear seemingly different, their software metrics might be similarly or differently distributed depending on several contextual features. Therefore it is possible that the observed usefulness of SNA metrics on our studied projects might not generalize for projects that are of a different nature. Therefore, we encourage future studies to explore the impact of the nature, module size and version definitions of a studied software project on the usefulness of SNA metrics across different SDP contexts and scenarios.

Another threat to external validity is that we only use Area Under the Cost-Effectiveness Curve and Effort Reduction measures (which rely only on LOC) to quantify effort. Herbold [31] specified a cost model that takes more than LOC into account to quantify effort. We argue that our studied datasets provided by Yatish *et al.* [99] only include the static code metrics and number of defects present in each module, which make our study unable to adapt the cost model. However, we invite future research to explore the use of this cost model to understand the effectiveness of SNA metrics for the Effort-aware scenarios.

Finally, we recommend researchers and practitioners to consider SNA metrics alongside code metrics in their SDP studies. Considering both the metric families together, particularly when building SDP models for small projects with limited data points might result in over fitted SDP models. These SDP models might not perform well in practice and as such it is a threat to the external validity of our recommendation. To mitigate this threat, we urge the researchers and practitioners to use robust feature selection methods [12] and model validation methods like out-of-sample bootstrap [87] as a part of the experimental setup that they use to construct their SDP models.

Internal Validity. A key threat to internal validity is that we only study the effectiveness of SNA metrics compared to code metrics for SDP and do not consider other metric families like process metrics. Rahman and Devanbu [68] showed the models built on process metrics outperform the models built on code metrics. Similarly several other metric families show the superiority of change metrics [17], [26], [45], churn metrics [54], [55] for SDP. However, we do not consider other metrics, as in the absence of history of a software project, only the source code of a software is available. In cases such as those, we can only extract code metrics and SNA metrics. Therefore, we argue that our findings help researchers and practitioners better leverage the source code for SDP. Especially, when the history of a project is not available. Furthermore, we also consider all combinations of the metrics extracted from the source and thereby eliminate potential confounders. However, we invite the future research to understand the effectiveness of SNA metrics in the context of the other commonly used metric families in SDP.

9 Conclusion

The effectiveness of the SNA metrics for models across the common SDP contexts and scenarios has been widely debated. Some prior studies observe that SNA metrics are more effective than the widely used code metrics [60], [103], whereas several other prior studies observe that SNA metrics are no more effective than the code metrics [66]. Therefore it is pivotal to understand the effectiveness of SNA metrics in comparison to code metrics across the various common SDP contexts and scenarios. As these are the only few metrics other than code metrics that can be extracted when the history of a software project is not available (e.g., new project or not historical tracking).

Therefore, in this paper, we revisit the effectiveness of SNA metrics for the common SDP scenarios in different SDP contexts through a case study on 30 versions of 9 open source software projects. The results of our case study suggest that considering SNA metrics by themselves or along with the code metrics could potentially improve the performance of models for at least 5 out of the 9 studied SDP scenarios (three SDP scenarios across three SDP contexts). However, we do note that in some cases, the improvements in performance by using SNA metrics alongside or over simply using code metrics might be marginal. Whereas in other cases it might be large.

We do not claim the generalization of our observed results. Instead, we simply observe that there exist software projects in which using SNA metrics by themselves or along with code metrics could yield better performing SDP models across several SDP contexts and scenarios. Hence, we suggest that future research should consider using SNA metrics alongside code metrics when building SDP models for various SDP contexts and scenarios. Finally, we suggest that EN and GN metrics (that make up the SNA metrics) should be considered separately along with being considered together.

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:59:26 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] C. A. Colin and F. A. G. Windmeijer, "An R-squared measure of goodness of fit for some common nonlinear regression models," *J. Econometrics*, vol. 77, no. 2, pp. 329–342, 1997.

[2] S. Amasaki, "Cross-version defect prediction: Use historical data, cross-project data, or both?," *Empir. Softw. Eng.*, vol. 25, pp. 1573–1595, 2020.

[3] A. S. Andreou and S. P. Chatzis, "Software defect prediction using doubly stochastic poisson processes driven by stochastic belief networks," *J. Syst. Softw.*, vol. 122, pp. 72–82, 2016.

[4] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *J. Syst. Softw.*, vol. 83, no. 1, pp. 2–17, 2010.

[5] R. A. Armstrong, "When to use the Bonferroni correction," *Ophthalmic Physiological Opt.*, vol. 34, no. 5, pp. 502–508, 2014.

[6] S. Assar, M. Borg, and D. Pfahl, "Using text clustering to predict defect resolution time: A conceptual replication and an evaluation of prediction accuracy," *Empir. Softw. Eng.*, vol. 21, pp. 1437–1475, 2016.

[7] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Softw. Eng.*, vol. 22, no. 10, pp. 751–761, Oct. 1996.

[8] N. Bettenburg and A. E. Hassan, "Studying the impact of social structures on software quality," in *Proc. Int. Conf. Program Comprehension*, 2010, pp. 124–133.

[9] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code! Examining the effects of ownership on software quality," in *Proc. 19th ACM SIGSOFT Symp. Found. Softw. Eng.*, 2011, pp. 4–14.

[10] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," *J. Syst. Softw.*, vol. 51, no. 05, pp. 245–273, 2000.

[11] T. Chai and R. Draxler, "Root mean square error (RMSE) or mean absolute error (MAE)? - Arguments against avoiding RMSE in the literature," *Geosci. Model Develop.*, vol. 7, no. 3, pp. 1247–1250, 2014.

[12] Y. Chen, Y. Li, X.-Q. Cheng, and L. Guo, "Survey and taxonomy of feature selection algorithms in intrusion detection system," in *Proc. Int. Conf. Inf. Secur. Cryptol.*, 2006, pp. 153–167.

[13] D. Chicco and G. Jurman, "The advantages of the matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation," *BMC Genomics*, vol. 21, no. 6, pp. 1–13, 2020.

[14] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object-oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[15] A. E. C. Cruz and K. Ochimizu, "Towards logistic regression models for predicting fault-prone code across software projects," in *Proc. 3rd Int. Symp. Empir. Softw. Eng. Meas.*, 2009, pp. 460–463.

[16] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes," *IEEE Trans. Softw. Eng.*, vol. 43, no. 7, pp. 641–657, Jul. 2017.

[17] M. D'Ambros, M. Lanza, and R. Robbes, "On the relationship between change coupling and software defects," in *Proc. Int. Conf. Reverse Eng.*, 2009, pp. 135–144.

[18] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison," *Empir. Softw. Eng.*, vol. 17, pp. 531–577, 2012.

[19] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *J. Mach. Learn. Res.*, vol. 7, pp. 1–30, 2006.

[20] B. Efron, "The bootstrap and modern statistics," *J. Amer. Statist. Assoc.*, vol. 95, no. 452, pp. 1293–1296, 2000.

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

[22] M. Friedman, "A comparison of alternative test of significance for the problem of m rankings," *Ann. Math. Statist.*, vol. 11, no. 1, pp. 86–92, 1940.

[23] B. Ghostra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *Proc. Int. Conf. Softw. Eng.*, 2015, pp. 789–800.

[24] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *Proc. Int. Conf. Softw. Eng.*, 2015, pp. 789–800.

[25] B. Ghotra, S. McIntosh, and A. E. Hassan, "A large-scale study of the impact of feature selection techniques on defect classification models," in *Proc. IEEE/ACM 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 146–157.

[26] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.*, vol. 26, no. 7, pp. 653–661, Jul. 2000.

[27] S. Gupta and A. Gupta, "A set of measures designed to identify overlapped instances in software defect prediction," *Computing*, vol. 99, pp. 889–914, 2017.

[28] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Countsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1276–1304, Nov./Dec. 2012.

[29] M. Halstead, *Elements of Software Science*. New York, NY, USA: Elsevier, 1977.

[30] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. Int. Conf. Softw. Eng.*, 2009, pp. 78–88.

[31] S. Herbold, "On the costs and profit of software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 47, no. 11, pp. 2617–2631, Nov. 2021.

[32] S. Herbold, A. Trautsch, and J. Grabowski, "A comparative study to benchmark cross-project defect prediction approaches," *IEEE Trans. Softw. Eng.*, vol. 44, no. 9, pp. 811–833, Sep. 2018.

[33] S. Herbold, A. Trautsch, and J. Grabowski, "Correction of "A comparative study to benchmark cross-project defect prediction approaches"," *IEEE Trans. Softw. Eng.*, vol. 45, no. 6, pp. 632–636, Jun. 2019.

[34] S. Hosseini, B. Turhan, and D. Gunarathna, "A systematic literature review and meta-analysis on cross project defect prediction," *IEEE Trans. Softw. Eng.*, vol. 45, no. 2, pp. 111–147, Feb. 2019.

[35] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2013, pp. 279–289.

[36] J. Jiarpakdee, C. Tantithamthavorn, and C. Treude, "AutoSpearman: Automatically mitigating correlated software metrics for interpreting defect models," in *Proc. Int. Conf. Softw. Maintenance Evol.*, 2018, pp. 92–103.

[37] Y. Kamei *et al.*, "A large-scale empirical study of just-in-time quality assurance," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 757–773, Jun. 2013.

[38] B. A. Kitchenham, L. M. Pickard, S. G. MacDonell, and M. J. Shepperd, "What accuracy statistics really measure [software estimation]," *IEE Proc. Softw.*, vol. 148, no. 3, pp. 81–85, 2001.

[39] E. Klotins, M. Unterkalmsteiner, and T. Gorschek, "Software engineering in start-up companies: An analysis of 88 experience reports," *Empir. Softw. Eng.*, vol. 24, pp. 68–102, 2019.

[40] E. Kocaguneli, T. Menzies, J. Keung, D. Cok, and R. Madachy, "Active learning and effort estimation: Finding the essential content of software effort estimation data," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1040–1053, Aug. 2013.

[41] A. G. Koru and H. Liu, "An investigation of the effect of module size on defect prediction using static measures," in *Proc. Workshop Predictor Models Softw. Eng.*, 2005, pp. 1–5.

[42] W. B. Langdon, J. Dolado, F. Sarro, and M. Harman, "Exact mean absolute error of baseline predictor, MARP0," *Inf. Softw. Technol.*, vol. 73, no. 5, pp. 16–18, 2016.

[43] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, Jul./Aug. 2008.

[44] Z. Li, X.-Y. Jing, X. Zhu, H. Zhang, B. Xu, and S. Ying, "On the multiple sources and privacy preservation issues for heterogeneous defect prediction," *IEEE Trans. Softw. Eng.*, vol. 45, no. 4, pp. 391–411, Apr. 2019.

[45] J. Liu, Y. Zhou, Y. Yang, H. Lu, and B. Xu, "Code churn: A neglected metric in effort-aware just-in-time defect prediction," in *Proc. Int. Symp. Empir. Softw. Eng. Meas.*, 2017, pp. 11–19.

[46] W. Ma, L. Chen, Y. Yang, Y. Zhou, and B. Xu, "Empirical analysis of network measures for effort-aware fault-proneness prediction," *Inf. Softw. Technol.*, vol. 69, pp. 50–70, 2016.

[47] L. Madeyski and M. Jureczko, "Which process metrics can significantly improve defect prediction models? An empirical study," *Softw. Qual. J.*, vol. 23, no. 1, pp. 393–422, 2015.

[48] T. J. McCabe , "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:59:26 UTC from IEEE Xplore. Restrictions apply.

[49] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting failures with developer networks and social network analysis," in *Proc. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2008, pp. 13–23.

[50] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 2–13, Jan. 2007.

[51] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang, "Implications of ceiling effects in defect predictors," in *Proc. 4th Int. Workshop Predictor Models Softw. Eng.*, 2008, pp. 47–54.

[52] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proc. Int. Conf. Softw. Eng.*, 2008, pp. 181–190.

[53] I. Myrtveit, E. Stensrud, and M. Shepperd, "Reliability and validity in comparative studies of software prediction models," *IEEE Trans. Softw. Eng.*, vol. 31, no. 5, pp. 380–391, May 2005.

[54] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proc. Int. Conf. Softw. Eng.*, 2005, pp. 284–292.

[55] N. Nagappan and T. Ball, "Using software dependencies and churn metrics to predict field failures: An empirical case study," in *Proc. Int. Symp. Empir. Softw. Eng. Meas.*, 2007, pp. 364–373.

[56] N. Nagappan, T. Ball, and B. Murphy, "Using historical in-process and product metrics for early estimation of software failures," in *Proc. Int. Symp. Softw. Rel. Eng.*, 2006, pp. 62–74.

[57] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proc. Int. Conf. Softw. Eng.*, 2006, pp. 452–461.

[58] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change bursts as defect predictors," in *Proc. Int. Symp. Softw. Rel. Eng.*, 2010, pp. 309–318.

[59] P. Nemenyi, "Distribution-free multiple comparisons," PhD dissertation, Princeton Univ., 1963.

[60] T. H. D. Nguyen, B. Adams, and A. E. Hassan, "Studying the impact of dependency network measures on software quality," in *Proc. Int. Conf. Softw. Maintenance*, 2010, pp. 1–10.

[61] A. Okutan and O. T. Yildiz, "Software defect prediction using Bayesian networks," *Empir. Softw. Eng.*, vol. 19, pp. 154–181, 2014. [Online]. Available: https://doi.org/10.1007/s10664-012-9218-8

[62] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," *ACM SIGSOFT Softw. Eng. Notes*, vol. 29, no. 4, pp. 86–96, 2004.

[63] F. Palomba, M. Zanoni, F. A. Fontana, A. D. Lucia, and R. Oliveto, "Toward a smell-aware bug prediction model," *IEEE Trans. Softw. Eng.*, vol. 45, no. 2, pp. 194–218, Feb. 2019.

[64] F. Peters, T. Menzies, L. Gong, and H. Zhang, "Balancing privacy and utility in cross-company defect prediction," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1054–1068, Aug. 2013.

[65] F. Peters, T. Menzies, and L. Layman, "LACE2: Better privacy-preserving data sharing for cross project defect prediction," in *Proc. Int. Conf. Softw. Eng.*, 2015, pp. 801–811.

[66] S. Prateek, A. Pasala, and L. M. Aracena, "Evaluating performance of network metrics for bug prediction in software," in *Proc. Asia-Pacific Softw. Eng. Conf.*, 2013, pp. 124–131.

[67] R. Premraj and K. Herzig, "Network versus code metrics to predict defects: A replication study," in *Proc. Int. Symp. Empir. Softw. Eng. Meas.*, 2011, pp. 215–224.

[68] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 432–441.

[69] G. K. Rajbahadur, S. Wang, Y. Kamei, and A. E. Hassan, "The impact of using regression models to build defect classifiers," in *Proc. IEEE/ACM 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 135–145.

[70] G. K. Rajbahadur, S. Wang, Y. Kamei, and A. E. Hassan, "Impact of discretization noise of the dependent variable on machine learning classifiers in software engineering," *IEEE Trans. Softw. Eng.*, vol. 47, no. 7, pp. 1414–1430, Jul. 2021.

[71] S. S. Rathore and S. Kumar, "Towards an ensemble based system for predicting the number of software faults," *Expert Syst. Appl.*, vol. 82, no. 1, pp. 357–382, 2017.

[72] S. S. Rathore and S. Kumar, "An empirical study of some software fault prediction techniques for the number of faults prediction," *Soft Comput.*, vol. 21, no. 1, pp. 7417–7434, 2017.

[73] A. Schröter, T. Zimmermann, and A. Zeller, "Predicting component failures at design time," in *Proc. Int. Symp. Empir. Softw. Eng.*, 2006, pp. 18–27.

[74] M. Shepperd and S. G. MacDonell, "Evaluating prediction systems in software project estimation," *Inf. Softw. Technol.*, vol. 54, no. 8, pp. 820–827, 2012.

[75] M. Shepperd, Q. Song, Z. Sun, and C. Mair, "Data quality: Some comments on the NASA software defect datasets," *IEEE Trans. Softw. Eng.*, vol. 39, no. 9, pp. 1208–1215, Sep. 2013.

[76] M. Shepperd, D. Bowes, and T. Hall, "Researcher bias: The use of machine learning in software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 40, no. 6, pp. 603–616, Jun. 2014.

[77] Y. Shin, B. Robert, T. J. Ostrand, and E. J. Weyuker, "Does calling structure information improve the accuracy of fault prediction?," in *Proc. Int. Conf. Mining Softw. Repositories*, 2009, pp. 61–70.

[78] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, Nov./Dec. 2011.

[79] Y. Shin, R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "On the use of calling structure information to improve fault prediction," *Empir. Softw. Eng.*, vol. 17, no. 4/5, pp. 390–423, 2012.

[80] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?," in *Proc. Int. Workshop Mining Softw. Repositories*, 2005, pp. 1–5.

[81] J. Sohn and S. Yoo, "Empirical evaluation of fault localisation using code and change metrics," *IEEE Trans. Softw. Eng.*, vol. 47, no. 8, pp. 1605–1625, Aug. 2021.

[82] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu, "A general software defect-proneness prediction framework," *IEEE Trans. Softw. Eng.*, vol. 37, no. 3, pp. 356–370, May/Jun. 2011.

[83] Q. Song, Y. Guo, and M. Shepperd, "A comprehensive investigation of the role of imbalanced learning for software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 45, no. 12, pp. 1253–1269, Dec. 2019.

[84] E. W. Steyerberg and F. E. Harrell Jr., "Prediction models need appropriate internal, internal-external, and external validation," *J. Clin. Epidemiol.*, vol. 69, pp. 245–247, 2016.

[85] C. Tantithamthavorn, "Towards a better understanding of the impact of experimental components on defect prediction modelling," in *Proc. Int. Conf. Softw. Eng.*, 2016, pp. 867–870.

[86] C. Tantithamthavorn, S. MaIntosh, A. E. Hassan, A. Ihara, and K. Matsumo, "The impact of mislabelling on the performance and interpretation of defect prediction models," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, 2015, pp. 812–823.

[87] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 1–18, Jan. 2017.

[88] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "The impact of automated parameter optimization for defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 45, no. 7, pp. 683–711, Jul. 2019.

[89] A. Tosun, B. Turhan, and A. Bener, "Validation of network measures as indicators of defective modules in software systems," in *Proc. Int. Conf. Predictor Models Softw. Eng.*, 2009, pp. 1–9.

[90] B. Turhan, G. Kocak, and A. Bener, "Software defect prediction using call graph based ranking (CGBR) framework," in *Proc. Euromicro Conf. Softw. Eng. Adv. Appl.*, 2008, pp. 191–198.

[91] C. H. Wagner, "Simpson's paradox in real life," *Amer. Statistician*, vol. 36, no. 1, pp. 46–48, 1982.

[92] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 46, no. 12, pp. 1267–1293, Dec. 2020.

[93] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Berlin, Germany: Springer, 2012.

[94] F. Wu, "Comparative analysis of software network and CK metrics: Implications for pre-and post-release faults," *J. Softw.*, vol. 9, no. 3, pp. 541–552, 2014.

[95] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang, "HYDRA: Massively compositional model for cross-project defect prediction," *IEEE Trans. Softw. Eng.*, vol. 42, no. 10, pp. 977–998, Oct. 2016.

[96] Z. Xu *et al.*, "Cross version defect prediction with representative data via sparse subset selection," in *Proc. Int. Conf. Program Comprehension*, 2018, pp. 132–143.

[97] Z. Xu *et al.*, "TSTSS: A two-stage training subset selection framework for cross version defect prediction," *J. Syst. Softw.*, vol. 154, no. 1, pp. 59–78, 2019.

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:59:26 UTC from IEEE Xplore. Restrictions apply.

[98] X. Yang, D. Lo, X. Xia, and J. Sun, "TLEL: A two-layer ensemble learning approach for just-in-time defect prediction," *Inf. Softw. Technol.*, vol. 87, pp. 206–220, 2017.
[99] S. Yatish, J. Jiarpakdee, P. Thongtanunam, and C. Tantithamthavorn, "Mining software defects: Should we consider affected releases?," in *Proc. Int. Conf. Softw. Eng.*, 2019, pp. 654–665.
[100] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards building a universal defect prediction model with rank transformed predictors," *Empir. Softw. Eng.*, vol. 21, pp. 2107–2145, 2016.
[101] F. Zhang, A. E. Hassan, S. McIntosh, and Y. Zou, "The use of summation to aggregate software metrics hinders the performance of defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 5, pp. 476–491, May 2017.
[102] Y. Zhou, B. Xu, and H. Leung, "On the ability of complexity metrics to predict fault-prone classes in object-oriented systems," *J. Syst. Softw.*, vol. 83, no. 4, pp. 660–674, 2010.
[103] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proc. Int. Conf. Softw. Eng.*, 2008, pp. 531–540.
[104] O. F. Arar and K. Ayan, "A feature dependent naive bayes approach and its application to the software defect prediction problem," *Appl. Soft Comput.*, vol. 59, no. 1, pp. 197–209, 2017.

Lina Gong received the BE degree in software engineering from the China University of Petroleum, Beijing, China, and the PhD degree in the computer software and theory from the China University of Mining and Technology, Beijing, China. She is currently a lecturer with the School of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China. She also studied as a visitor one year in the Software Analysis and Intelligence Lab (SAIL), School of Computing, Queen's University, Canada. Her research interests include machine learning, software analysis, software testing, and mining software repositories.

Gopi Krishnan Rajbahadur received the BE degree in computer science and engineering from the SKR Engineering College, Anna University, Chennai, Tamil Nadu, India, and the PhD degree in computer science from Queen's University, Kingston, Canada. His research interests include software engineering for artificial intelligence (SE4AI), artificial intelligence for software engineering (AI4SE), mining software repositories (MSR), and explainable AI (XAI).

Ahmed E. Hassan (Fellow, IEEE) received the PhD degree in computer science from the University of Waterloo, Waterloo, Canada. He is an ACM SIGSOFT influential educator, an NSERC steacie fellow, the Canada research chair (CRC) in software analytics, and the NSERC/BlackBerry software engineering chair with the School of Computing, Queen's University, Canada. His research interests include mining software repositories, empirical software engineering, load testing, and log mining. He spearheaded the creation of the Mining Software Repositories (MSR) conference and its research community. He also serves/d on the editorial boards of the *IEEE Transactions on Software Engineering*, *Springer Journal of Empirical Software Engineering*, and *PeerJ Computer Science*. For more information, please visit http://sail.cs.queensu.ca

Shujuan Jiang (Member, IEEE) received the BS degree in computer science from the Computer Science Department, East China Normal University, Shanghai, China, in 1990, the MS degree in computer science from the China University of Mining and Technology, Xuzhou, Jiangsu, China, in 2000, and the PhD degree in computer science from the Southeast University, Nanjing, Jiangsu, China, in 2006. From 1995 to today, she has been a teaching assistant, lecturer, associate professor and professor with the Computer Science Department, China University of Mining and Technology, Xuzhou, Jiangsu, China. She is the author of more than 80 articles. Her research interests include software engineering, program analysis and testing, software maintenance, etc.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: Queen's University. Downloaded on July 08,2023 at 00:59:26 UTC from IEEE Xplore. Restrictions apply.