The Use of Summation to Aggregate Software Metrics Hinders the Performance of Defect Prediction Models

Article *in* IEEE Transactions on Software Engineering · January 2016

DOI: 10.1109/TSE.2016.2599161

The Use of Summation to Aggregate Software Metrics Hinders the Performance of Defect Prediction Models

Feng Zhang, Ahmed E. Hassan, *Member, IEEE*,
Shane McIntosh, *Member, IEEE*, and Ying Zou, *Member, IEEE*

Abstract—Defect prediction models help software organizations to anticipate where defects will appear in the future. When training a defect prediction model, historical defect data is often mined from a Version Control System (VCS, e.g., Subversion), which records software changes at the file-level. Software metrics, on the other hand, are often calculated at the class- or method-level (e.g., McCabe's Cyclomatic Complexity). To address the disagreement in granularity, the class- and method-level software metrics are aggregated to file-level, often using summation (i.e., McCabe of a file is the sum of the McCabe of all methods within the file). A recent study shows that summation significantly inflates the correlation between lines of code (SLOC) and cyclomatic complexity (CC) in Java projects. While there are many other aggregation schemes (e.g., central tendency, dispersion), they have remained unexplored in the scope of defect prediction. In this study, we set out to investigate how different aggregation schemes impact defect prediction models. Through an analysis of 11 aggregation schemes using data collected from 255 open source projects, we find that: (1) aggregation schemes can significantly alter correlations among metrics, as well as the correlations between metrics and the defect count; (2) when constructing models to predict defect proneness, applying only the summation scheme (i.e., the most commonly used aggregation scheme in the literature) only achieves the best performance (the best among the 12 studied configurations) in 11 percent of the studied projects, while applying all of the studied aggregation schemes achieves the best performance in 40 percent of the studied projects; (3) when constructing models to predict defect rank or count, either applying only the summation or applying all of the studied aggregation schemes achieves similar performance, with both achieving the closest to the best performance more often than the other studied aggregation schemes; and (4) when constructing models for effort-aware defect prediction, the mean or median aggregation schemes yield performance values that are significantly closer to the best performance than any of the other studied aggregation schemes. Broadly speaking, the performance of defect prediction models are often underestimated due to our community's tendency to only use the summation aggregation scheme. Given the potential benefit of applying additional aggregation schemes, we advise that future defect prediction models should explore a variety of aggregation schemes.

Index Terms—Defect prediction, aggregation scheme, software metrics

1 INTRODUCTION

SOFTWARE organizations spend a disproportionate amount of effort on the maintenance of software systems [1]. Fixing defects is one of the main activities in software maintenance. To help software organizations to allocate defect-fixing effort more effectively, defect prediction models anticipate where future defects may appear in a software system.

In order to build a defect prediction model, historical defect-fixing activity is mined and software metrics, which may have a relationship with defect proneness, are computed. The historical defect-fixing activity is usually mined from a Version Control System (VCS), which records change activity at the file-level. It is considerably easier for practitioners to build their models at the file-level [2], since it is often very hard to map a defect to a specific method even if the fixing change was applied to a particular method [3]. Instead, mapping defects to files ensures that the mapping was done to a more coherent and complete conceptual entity. Moreover, much of the publicly-available defect data sets (e.g., the PROMISE repository [4]) and current studies in literature [5], [6], [7], [8], [9], [10] are at the file-level. Understanding the impact of aggregation would benefit a large number of previously-conducted studies that build defect prediction models at the file-level.

It has been reported that predicting defective files is more effective than predicting defective packages for Java systems [11], [12], [13]. Typically, in order to train file-level defect prediction models, the method- or class-level software metrics are aggregated to the file-level. Such a process is illustrated in Fig. 1. Summation is one of the most commonly applied aggregation schemes in the literature [5], [6], [7], [8], [9], [10], [11], [14], [15], [16], [17]. However, Landman et al. [18] show that prior findings [19], [20] about the high correlation

- *F. Zhang and A.E. Hassan are with the School of Computing, Queen's University, Kingston, ON K7L 3N6, Canada. E-mail: {feng, ahmed}@cs.queensu.ca.*
- *S. McIntosh is with the Department of Electrical and Computer Engineering, McGill University, Montréal, QC H3A 0G4, Canada. E-mail: shane.mcintosh@mcgill.ca.*
- *Y. Zou is with the Department of Electrical and Computer Engineering, Queen's University, Kingston, ON K7L 3N6, Canada. E-mail: ying.zou@queensu.ca.*

Manuscript received 14 Aug. 2015; revised 6 June 2016; accepted 24 July 2016. Date of publication 9 Aug. 2016; date of current version 22 May 2017. Recommended for acceptance by M. Di Penta.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2016.2599161

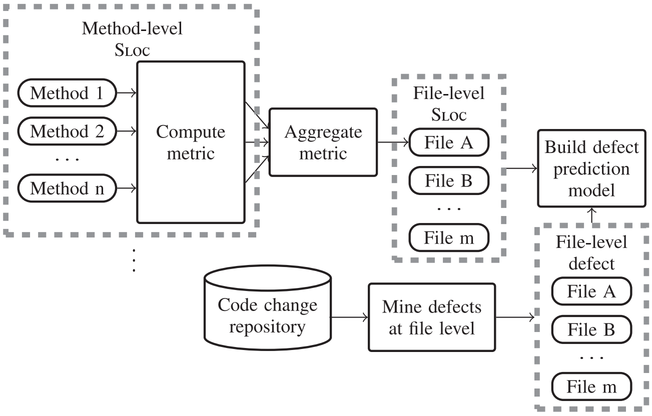


Fig. 1. A typical process to apply method-level metrics (e.g., SLOC) to build file-level defect prediction models.

between summed cyclomatic complexity (CC) [21] and summed lines of code (i.e., SLOC) may have been overstated for Java projects, since the correlation is significantly weaker at the method-level. We suspect that the high correlation between many metrics at the file-level may also be caused by the aggregation scheme. Furthermore, the potential loss of information due to the summation aggregation may negatively affect the performance of defect prediction models.

Besides summation, there are a number of other aggregation schemes that estimate the central tendency (e.g., arithmetic mean and median), dispersion (e.g., standard deviation and coefficient of variation), inequality (e.g., Gini index [22], Atkinson index [23], and Hoover index [24]), and entropy (e.g., Shannon's entropy [25], generalized entropy [26], and Theil index [27]) of a metric. However, the impact that aggregation schemes have on defect prediction models remains unexplored.

We, therefore, set out to study the impact that different aggregation schemes have on defect prediction models. We perform a large-scale experiment using data collected from 255 open source projects. First, we examine the impact that different aggregation schemes have on: (a) the correlation among software metrics, since strongly correlated software metrics may be redundant, and may interfere with one another; and (b) the correlation between software metrics and defect count, identifying candidate predictors for defect prediction models. Second, we examine the impact that different aggregation schemes have on the performance of four types of defect prediction models, namely:

(a) *Defect proneness models:* classify files as defect-prone or not;
(b) *Defect rank models:* rank files according to their defect proneness;
(c) *Defect count models:* estimate the number of defects in a given file;
(d) *Effort-aware models:* incorporate fixing effort in the ranking of files according to their defect proneness.

To ensure that our conclusions are robust, we conduct a 1,000-repetition bootstrap experiment for each of the studied systems. In total, over 12 million prediction models are constructed during our experiments.

The main observations of our experiments are as follows:

- *Correlation analysis.* We observe that aggregation can significantly impact both the correlation among software metrics and the correlation between software metrics and defect count. Indeed, summation significantly inflates the correlation between SLOC and other metrics (not just CC). Metrics and defect count share a substantial correlation in 15-22 percent of the studied systems if metrics are aggregated using summation, while metrics and defect count only share a substantial correlation in 1-9 percent of the studied systems if metrics are aggregated using other schemes.
- *Defect prediction models.* We observe that using only the summation (i.e., the most commonly applied aggregation scheme) often does not achieve the best performance. For example, when constructing models to predict defect proneness, applying only the summation scheme only achieves the best performance in 11 percent of the studied projects, whereas applying all of the studied aggregation schemes achieves the best performance in 40 percent of the studied projects. Furthermore, when constructing models for effort-aware defect prediction, the mean or median aggregation schemes yield performance values that are significantly closer to the best ones than any of the other studied aggregation schemes. On the other hand, when constructing models to predict defect rank or count, either applying only the summation or applying all of the studied aggregation schemes achieves similar performance, with both achieving closer to best performance more often than the other studied aggregation schemes.

Broadly speaking, solely relying on summation tends to underestimate the predictive power of defect prediction models. Given the substantial improvement that could be achieved by using the additional aggregation schemes, we recommend that future defect prediction studies use the 11 aggregation schemes that we explore in this paper, and even experiment with other aggregation schemes. For instance, researchers and practitioners can generate the initial set of predictors (i.e., aggregated metrics, such as the summation, median, and standard deviation of lines of code) with all of the available aggregation schemes, and mitigate redundancies using PCA or other feature reduction techniques.

1.1 Paper Organization

The remainder of this paper is organized as follows. Section 2 summarizes the related work on defect prediction and aggregation schemes. We present the 11 studied aggregation schemes in Section 3. Section 4 describes the data that we use in our experiments. The approach and results of our case study are presented and discussed in Section 5. We discuss the threats to the validity of our work in Section 6. Conclusions are drawn and future work is described in Section 7.

2 Related Work

In this section, we discuss the related work with respect to defect prediction and aggregation schemes.

2.1 Defect Prediction

Defect prediction has become a very active research area in the last decade [9], [10], [28], [29], [30], [31], [32]. Defect prediction models can help practitioners to identify potentially

TABLE 1
List of the 11 Studied Aggregation Schemes

Category	Aggregation scheme	Formula
Summation	Summation	$\Sigma_m = \sum_{i=1}^{N} m_i$
Central tendency	Arithmetic mean	$\mu_m = \frac{1}{N}\sum_{i=1}^{N} m_i$
	Median	$M_m = \begin{cases} m_{\frac{n+1}{2}} & \text{if N is odd} \\ \frac{1}{2}\left(m_{\frac{n}{2}} + m_{\frac{n+2}{2}}\right) & \text{otherwise.} \end{cases}$
Dispersion	Standard deviation	$\sigma_m = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(m_i - \mu_m)^2}$
	Coefficient of variation	$\text{Cov}_m = \frac{\sigma_m}{\mu_m}$
Inequality index	Gini index [22]	$\text{Gini}_m = \frac{2}{N\Sigma_m}[\sum_{i=1}^{N}(m_i * i) - (N+1)\Sigma_m]$
	Hoover index [24]	$\text{Hoover}_m = \frac{1}{2}\sum_{i=1}^{N}\left\lvert\frac{m_i}{\Sigma_m} - \frac{1}{N}\right\rvert$
	Atkinson index [23]	$\text{Atkinson}_m = 1 - \frac{1}{\mu_m}\left(\frac{1}{N}\sum_{i=1}^{N}\sqrt{m_i}\right)^2$
Entropy	Shannon's entropy [25]	$E_m = -\frac{1}{N}\sum_{i=1}^{N}[\frac{freq(m_i)}{N} * ln\,\frac{freq(m_i)}{N}]$
	Generalized entropy [26]	$\text{GE}_m = -\frac{1}{N\alpha(1-\alpha)}\sum_{i=1}^{N}[(\frac{m_i}{\mu_m})^{\alpha} - 1], \alpha = 0.5$
	Theil index [27]	$\text{Theil}_m = \frac{1}{N}\sum_{i=1}^{N}[\frac{m_i}{\mu_m} * ln(\frac{m_i}{\mu_m})]$

In the formulas, m_i denotes the value of metric m in the ith method in a file that has N methods. Methods in the same file are sorted in the ascending order of the values of metric m.

defective modules so that software quality assurance teams can allocate their limited resources more effectively.

There are four main types of defect prediction models: (1) defect proneness models that identify defective software modules [10], [33], [34]; (2) defect rank models that order modules according to the relative number of defects expected [15], [28]; (3) defect count models that estimates the exact number of defects per module [7], [28]; and (4) effort-aware models are similar to defect rank models except that they also take the effort required to review the code in that file into account [35], [36].

Software modules analyzed by defect prediction models can be packages [15], [28], files [9], [10], [30], [34], classes [31], [37], methods [32], [38], or even lines [39]. Since software metrics are often collected at method- or class-levels, it is often necessary to aggregate them to the file-level.

While summation is one of the most commonly used aggregation schemes [8], [15], prior work has also explored others. For example, D'Ambros et al. [31] compute the entropy of both code and process metrics. Hassan [30] applies Shannon's entropy [25] to aggregate process metrics as a measure of the complexity of the change process. Vasilescu et al. [40] find that the correlation between metrics and defect count is impacted by the aggregation scheme that has been used. However, to the best of our knowledge, the impact that aggregation schemes have on defect prediction models has not yet been explored. Thus, we perform a large-scale experiment using data from 255 open source systems to examine the impact that aggregation schemes can have on defect prediction models.

2.2 Aggregation Scheme

While the most commonly used granularity of defect prediction is the file-level [9], [10], [30], [34], many software metrics are calculated at the method- or class-levels. The difference in granularity creates the need for aggregation of the finer method- and class-level metrics to file-level. Simple aggregation schemes, such as summation and mean, have been explored in the defect prediction literature [5], [6], [7], [8], [9], [10], [11], [14], [15], [16], [17]. However, recent work suggests that summation may distort the correlation among metrics [18], and the mean may be inappropriate if the distribution of metric values is skewed [41].

Apart from summation and mean, more advanced metric aggregation schemes have been also explored [41], [42], [43], [44], [45], [46], including the Gini index [22], the Atkinson index [23], the Hoover index [24], and the Kolm index [47]. For instance, He et al. [45] apply multiple aggregation schemes to construct various metrics about a project in order to find appropriate training projects for cross-project defect prediction. Giger et al. [44] use the Gini index to measure the inequality of file ownership and obtain acceptable performance for defect proneness models. In addition to correlations, we also study the impact that aggregation schemes have on defect prediction models. We investigate how 11 aggregation schemes impact the performance of four types of defect prediction models.

3 Aggregation Schemes

In this section, we introduce the 11 aggregation schemes that we studied for aggregating method-level metrics to the file-level (Fig. 1). We also discuss why we exclude some other aggregation schemes from our study. Table 1 shows the formulas of the 11 schemes. The details are presented as follows.

3.1 Summation

An important aspect of a software metric is the accumulative effect, e.g., files with more lines of code are more likely to have defects than files with few lines of code [48]. Similarly, files with many complex methods are more likely to have defects than files with many simple methods [49]. Summation captures the accumulative effect of a software metric.

Specifically, we study the *summation* scheme, which sums the values of a metric over all methods within the same file. The *summation* scheme has been commonly used in defect prediction studies [5], [6], [7], [8], [9], [10], [11], [14], [15], [16], [17].

3.2 Central Tendency

In addition to the accumulative effect, the average effect is also important. For example, it is likely easier to maintain a file with smaller methods than a file with larger ones, even if the total file size is equal. Computing the average effect can help to distinguish between files with similar total size, but different method sizes on average. The average effect of a software metric can be captured using central tendency metrics, which measure the central value in a distribution. In this paper, we study the *arithmetic mean* and *median* measures of central tendency.

3.3 Dispersion

Dispersion measures the spread of values of a particular metric, with respect to some notion of central tendency. For example, in a file with low dispersion, methods have similar sizes, suggesting that the functionalities of the file are balanced across methods. On the other hand, in a file with high dispersion, some methods have much larger sizes than the average, while some methods have much smaller sizes than the average. The large methods may contain too much functionality while small methods may contain little functionality. We study the *standard deviation* and the *coefficient of variation* measures of dispersion.

3.4 Inequality Index

An inequality index explains the degree of imbalance in a distribution. For example, a large degree of inequality shows that most lines of code of a file belong to only a few methods. Such methods contain most of the lines of code, and thus, have a higher chance of falling victim to the "Swiss Army Knife" anti-pattern. Inequality indices are often used by economists to measure income inequality in a specific group [43]. In this paper, we study the Gini [22], Hoover [24], and Atkinson [23] inequality indices. These indices have previously been analyzed in the broad context of software engineering [41], [43].

Each index captures different aspects of inequality. The Gini index measures the degree of inequality, but cannot identify the unequal part of the distribution. The Atkinson index can indicate which end of the distribution introduces the inequality. The Hoover index represents the proportion of all values that, if redistributed, would counteract the inequality. The three indices range from zero (perfect equality) to one (maximum inequality).

3.5 Entropy

In information theory, entropy represents the information contained in a set of variables. Larger entropy values indicate greater amounts of information. In the extreme case, from the code inspection perspective, files full of duplicated code snippets contain less information than files with only unique code snippets. It is easier to spot defects in many code snippets that are duplicated than in many code snippets that are different from each other. Hence, a file with low entropy is less likely to experience defects than a file with high entropy. In this paper, we study the *Shannon's entropy* [25], *generalized entropy* ($\alpha = 0.5$) [26], and the *Theil index* [27]. Shannon's (and generalized) entropy measure redundancy or diversity in the values of a particular metric. The Theil index, an enhanced variant of the generalized entropy, measures inequality or lack of diversity.

3.6 Excluded Aggregation Schemes

Distribution shape is another widely used family of aggregation schemes. Skewness and kurtosis are two commonly used measures that capture the shape of a distribution. In the formulas for computing skewness and kurtosis, the denominator is the standard deviation. If the standard deviation is zero, the skewness and kurtosis are both undefined. In our data set, we observe that a large number of methods have exactly the same value of a particular metric, producing zero variance. Hence, we exclude skewness and kurtosis from our analysis, since they are undefined for many files.

Kolm index [47] is another candidate scheme that measures the absolute inequality of a distribution. However, the computation of Kolm index requires the exponentiation of metric values. Since many of our metrics have values larger than 1,000, the Kolm index becomes uncomputable. Therefore, it is not suitable for our study.

4 Experimental Data

In this section, we describe our experimental data, including the characteristics of the dataset, the defect data, and the software metrics that we use.

4.1 Dataset Characteristics

In this study, we begin with the dataset that was initially collected by Mockus [50]. The dataset contains 235 K open source systems hosted on SourceForge and GoogleCode. However, there are many systems that have not yet accumulated a sufficient amount of historical data to train defect models. Similar to our prior work [10], we apply a series of filters to exclude such systems from our analysis. Specifically, we exclude the systems that:

- (F1) Are not primarily written in C, C++, C#, Java, or Pascal, since the tool [51] that we use to compute the software metrics only supports these languages.
- (F2) Have a small number of commits (i.e., less than the 25th percentile of the number of commits across all remaining systems), as systems with too few commits have not yet accumulated enough historical data to train a defect model.
- (F3) Have a lifespan of less than one year, since most defect prediction studies collect defect data using two consecutive six-month time periods [15]. The first six-month period is used to collect defect data and metrics for building a defect prediction model, and the second six-month period is used to evaluate the performance of the model.
- (F4) Have a limited number of fix-inducing and non-fixing commits (i.e., less than the 75th percentile of the number of fix-inducing and non-fixing commits across all remaining systems, respectively). We do so to ensure that we have enough data to train stable defect models.
- (F5) Have less than 100 usable files (i.e., without undefined values of aggregated metrics). This ensures that we have sufficient instances for bootstrap model validation.

TABLE 2
The Descriptive Statistics of Our Dataset

Programming language	# of systems	# of files	# of methods	Defect ratio (mean ± sd)
C	34	8,140	167,146	43% ± 26%
C++	85	20,649	479,907	40% ± 27%
C#	15	2,951	666,046	38% ± 23%
Java	121	32,531	527,203	37% ± 27%
All	255	64,271	1,840,302	39% ± 27%

Table 2 provides an overview of the 255 systems that survive our filtering process.

4.2 Defect Data

In general, defect data is mined from commit messages. Since these commit messages can be noisy, data mined from commit messages are often corroborated using data mined from Issue Tracking Systems (ITSs, e.g., Bugzilla[1]) [15]. However, we find that only 53 percent of the studied systems are using ITSs. Hence, to treat every studied system equally, we mine defect data solely based on commit messages. While this approach may introduce bias into our dataset [52], [53], [54], prior work has shown that this bias can be offset by increasing the sample size [55]. There are 255 subject systems in our dataset, which is larger than most defect prediction studies to date [56].

Similar to our prior study [57], we consider that a commit is related to a defect fix if the commit message matches the following regular expression:

$(bug|fix|error|issue|crash|problem|fail|defect|patch)$

To further reduce the impact that noise in commit messages may introduce, we clean up noisy words like “debug” and “prefix” by removing all words that end with “bug” or “fix”. A similar strategy was used by Mockus and Votta [58] and is at the core of the popular SZZ algorithm [59]. In addition, similar to prior work [15], we use a six-month time period to collect defect data, i.e., we check for defect-fixing commits that occur in a six-month time period after a software release has occurred. Unfortunately, many systems on SourceForge or GoogleCode have not recorded their release dates. Hence, we simply choose the date that is six months prior to the last recorded commit of each system as the split date. Defect data is collected from commit messages in the six-month period after the split date.

4.3 Software Metrics

We group software metrics into three categories, i.e., traditional metrics, object-oriented metrics, and process metrics. In the scope of defect prediction, Radjenović et al. [60] perform a systematic review and report that traditional metrics are often collected at the method-level, object-oriented metrics are often collected at the class-level, and process metrics are often collected at the file-level. In this paper, we study traditional metrics, so that we can focus on investigating how the studied aggregation schemes impact defect prediction models.

1. http://www.bugzilla.org/

TABLE 3
List of Software Metrics at Method-Level

Metric	Description
SLOC	Source lines of code, excluding comments and blank lines.
CC	McCabe’s cyclomatic complexity.
EVG	Essential complexity is a modified version of cyclomatic complexity.
NPATH	The number of possible execution paths in a method.
FANIN	The number of inputs, including parameters, global variables, and method calls.
FANOUT	The number of outputs, such as updating parameters and global variables.

In this study, we choose six method-level metrics that are known to perform well in defect prediction models [38]. Table 3 provides an overview of the studied metrics. *Source Lines Of Code* (SLOC) is a measure of the size of a method. *Cyclomatic complexity* and *essential complexity* (EVG) are measures of the complexity of a method. The *number of possible paths* (NPATH) measures the complexity of the control flow of a method. The *number of inputs* (FANIN) and the *number of outputs* (FANOUT) are used by Giger et al. [38] to measure the control flow of a method only, but we use the original definition [61] of these two metrics to measure the information flow (i.e., both data and control flow) of a method.

To compute these metrics, we use the *Understand* [51] tool on the release (or split) code snapshot of each studied system. This code snapshot is the historical version of the studied system at the date just before the six-month time period used for collecting the defect data.

5 Case Study

In this section, we report the results of our case study along two dimensions. First, we study the impact that different aggregations have on the correlation among software metrics and the correlation between software metrics and defect counts. Second, we evaluate the impact of aggregations on four types of defect prediction models, i.e., defect proneness, defect rank, defect count, and effort-aware models. Finally, we provide comprehensive guidelines regarding the choice of aggregation schemes for future studies.

5.1 Correlation Analysis

Correlation analysis can be used to investigate how the relationship between any two particular metrics vary after aggregation, regardless how the aggregation is computed. When choosing software metrics to build a defect prediction model, it is a common practice to explore the correlations among software metrics, and the correlations between software metrics and defects [7], [15], [28], [62], [63]. Strongly correlated software metrics may be redundant, and may interfere with one another if used together to train a defect prediction model. Furthermore, a substantial correlation between a software metric and defect count may identify good candidate predictors for defect prediction models.

Aggregation schemes are required to lift software metrics to the file-level. However, aggregation schemes may distort the correlation between SLOC and CC in Java projects [18]. If

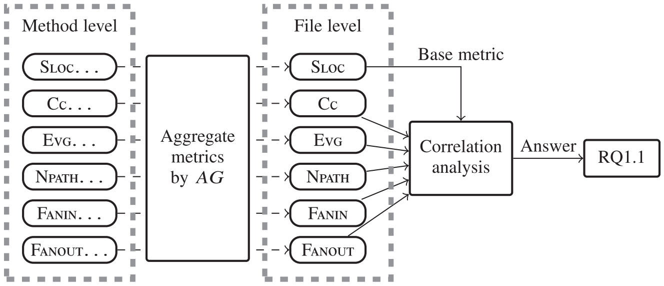


Fig. 2. Our approach to analyze the impact of aggregations on the correlations between software metrics (RQ1.1).

two metrics have a much stronger correlation after aggregation, it is unclear if the two metrics are actually strongly correlated, or if the aggregation has distorted one or both of the metrics.

Understanding the impact that aggregation schemes have can prevent the removal of useful metrics. Hence, we want to examine the impact that aggregations have in order to avoid potential loss of information in the model construction step.

5.1.1 Research Questions

To study the impact that aggregations have on the correlation among software metrics and their correlation with the defect count, we formulate the following two research questions:

RQ1.1 Do aggregation schemes alter the correlation between software metrics?

RQ1.2 Do aggregation schemes alter the correlation between software metrics and defect count?

5.1.2 Experimental Design

1) *Correlation Among Metrics*. In this study, we use Spearman's ρ [64] to measure correlation. Spearman's ρ measures the similarity between two ranks, instead of the exact values of the two assessed variables. Unlike parametric correlation techniques (e.g., Pearson correlation [64]), Spearman correlation does not require that the input data follow any particular distribution. Since Spearman correlation is computed on rank-transformed values, it is more robust to outliers than Pearson correlation [65]. Furthermore, in the presence of ties, Spearman's ρ is preferred [66] over other nonparametric correlation techniques, such as Kendall's τ [64]. Spearman's ρ ranges from -1 to +1, where -1 and +1 indicate the strongest negative and positive correlations, respectively. A value of zero indicates that the two input variables are entirely independent.

TABLE 4
The Percentage of the Studied Systems That Do Not Have Strong Correlations Among All Six Metrics at the Method-Level

Metric	Cc	Npath	Fanin	Fanout	Evg
Sloc	58%	59%	100%	39%	100%
Cc	-	0%	100%	96%	99%
Npath	-	-	100%	96%	99%
Fanin	-	-	-	100%	100%
Fanout	-	-	-	-	100%

TABLE 5
The Cliff's δ of the Difference in Correlation Values Between Sloc and Other Metrics Before and After Aggregation

Scheme	Cc	Npath	Fanin	Fanout	Evg
(1) Sum	**−0.881**	**−0.655**	**−0.884**	**−0.907**	**−0.969**
(2) Mean	−0.363	n.s.	0.269	−0.279	−0.386
(3) Median	0.188	0.213	0.206	n.s.	0.239
(4) SD	−0.290	−0.128	0.388	n.s.	−0.401
(5) COV	n.s.	0.345	**0.605**	**0.608**	−0.181
(6) Gini	0.022	0.305	**0.646**	**0.609**	−0.082
(7) Hoover	0.195	**0.505**	**0.737**	**0.729**	n.s.
(8) Atkinson	0.105	0.388	**0.767**	**0.778**	−0.104
(9) Shannon	n.s.	n.s.	**−0.584**	**−0.481**	−0.295
(10) Entropy	0.104	0.388	**0.767**	**0.778**	−0.104
(11) Theil	n.s.	0.370	0.469	0.458	−0.143

*(**bold** font indicates a large difference, and n.s. denotes a lack of statistical significance).*

Fig. 2 presents our approach to examine the impact that aggregations have on correlation among software metrics. To understand the correlation among metrics before aggregation, for each system, we calculate ρ between each pair of metrics at the method-level. Assessing if two metrics are strongly correlated is often applied to determine their redundancy in the scope of defect prediction [67], [68]. Similar to prior work [67], [68], [69], we consider that a pair of metrics are too highly correlated to include in the same model if $|\rho| \geq 0.8$ (we call it a "strong" correlation). Hence, we report the percentage of metrics that have $|\rho| < 0.8$ across all of the studied systems (see Table 4).

To study the impact that aggregation schemes have on these correlation values, we use Sloc as our base metric, and for each system, we compute ρ between Sloc and the other metrics at both method- and file-levels. We denote the correlation between Sloc and metric m as $cor.method(\textsc{Sloc}, m)$ at the method-level, and as $cor.file(\textsc{Sloc}, AG(m))$ at the file-level after applying an aggregation scheme AG. We test the null hypothesis below for each scheme:

$H0_1$: *There is no difference between the method-level correlation* $cor.method(\textsc{Sloc}, m)$ *and the file-level correlation* $cor.file(\textsc{Sloc}, AG(m))$.

To test $H0_1$, we use two-sided Mann-Whitney U tests [64] with $\alpha = 0.05$ (i.e., 95 percent confidence level). The Mann-Whitney U test checks if equally large values exist in two input samples. As a non-parametric statistical method, the Mann-Whitney U test makes no assumptions about the distributions that the input samples are drawn from. If there is a statistically significant difference between the input samples, we can reject $H0_1$ and conclude that the corresponding aggregation scheme yields statistically significantly different correlation values at the method- and file-levels. To control family-wise errors, we apply Bonferroni correction and adjust α by dividing by the number of tests.

We also calculate Cliff's δ [70] to quantify the size of the difference in correlation values at the method- and file-levels (see Table 5). We opt to use Cliff's δ instead of Cohen's d [71] because Cliff's δ is widely considered to be a more robust and reliable effect size measure than Cohen's d [72]. Moreover, Cliff's δ does not make any assumptions about the distributions of the input samples.

Cliff's δ ranges from -1 to +1, where a zero value indicates two identical distributions. A negative value indicates

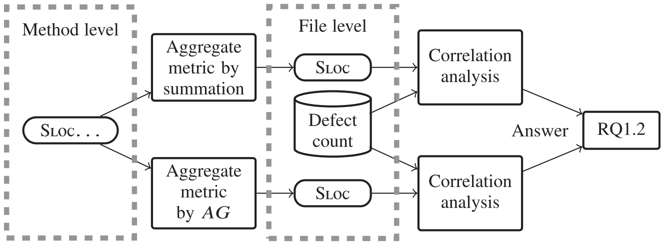


Fig. 3. Our approach to analyze the impact of aggregations on correlations between software metrics and defect count (RQ1.2).

that values in the first sample tend to be smaller than those in the second sample, while a positive value indicates the opposite. To ease interpretation of the effect size results, we use the mapping of Cliff's δ values to Cohen's d significance levels as proposed by prior work [72]:

Negligible:	$0 \leq \|\delta\| < 0.147$
Small:	$0.147 \leq \|\delta\| < 0.330$
Medium:	$0.330 \leq \|\delta\| < 0.474$
Large:	$0.474 \leq \|\delta\| \leq 1$

2) *Correlation Between Metrics and Defect Count*. To further understand the impact of aggregations, we investigate the correlation between defect count and metrics aggregated by each of the studied schemes. Fig. 3 provides an overview of our approach. As defect count is used in models for predicting defect rank, defect count, and in effort-aware models, correlation analysis between defect count and metrics may provide insights for metric selection in the three types of defect prediction models. Software metrics having a substantial correlation with defect count are usually considered to be good candidate predictors for defect prediction models [7], [15]. Similar to prior work [7], [15], we consider that a metric shares a promising correlation with the number of defects if the corresponding $|\rho| \geq 0.4$ (we call it a "substantial" correlation). Hence, we report the percentage of studied systems that have $|\rho| \geq 0.4$ for the defect count and any given metric after applying any of the studied aggregation scheme (see Table 6).

5.1.3 Case Study Results

Aggregation can Significantly Alter the Correlation Among Metrics Under Study. Table 4 shows that many method-level metrics do not have strong correlation values with one another (i.e., $|\rho| < 0.8$). For example, FANIN is not strongly correlated with the other metrics in any of the studied systems. Moreover, SLOC is not strongly correlated with CC in 58 percent of the studied systems.

On the other hand, some method-level metrics also have consistently strong correlation values. For example, CC is strongly correlated with NPATH in all of the studied systems. However, we find that selecting some aggregation schemes can help to reduce the strong correlation values that we observe at the method-level. For example, CC and NPATH are strongly correlated in all of the studied systems at the method-level. But when aggregated to the file-level using the summation, mean, median, standard deviation, coefficient of variation, Gini index, Hoover index, Atkinson index, Shannon's entropy, generalized entropy, and Theil index, they do not share a strong correlation with one another in 1-14 percent of the studied systems. This weaker correlation between CC and NPATH would allow one to safely use both metrics in a defect prediction model. One possible reason for the weak correlation is that aggregation does not only consider the metric values, but also the distribution of metric values. Two semantically correlated metrics may experience different distributions at method level. Thus, the aggregated metrics could significantly differ. As a result, the correlations between aggregated CC and NPATH can become either stronger or weaker.

TABLE 6
The Percentage of Studied Systems Where the Defect Count Shares a Substantial Correlation ($|\rho| \geq 0.4$) with the Metrics

Scheme	SLOC	CC	NPATH	FANIN	FANOUT	EVG
(1) Sum	20%	22%	15%	16%	20%	16%
(2) Mean	2%	1%	3%	2%	1%	3%
(3) Median	1%	2%	2%	1%	1%	0
(4) SD	4%	2%	5%	4%	1%	4%
(5) COV	3%	3%	7%	1%	1%	4%
(6) Gini	3%	2%	5%	1%	1%	3%
(7) Hoover	1%	2%	5%	1%	1%	3%
(8) Atkinson	1%	2%	6%	1%	1%	4%
(9) Shannon	9%	7%	6%	9%	9%	2%
(10) Entropy	1%	2%	6%	1%	1%	4%
(11) Theil	2%	3%	6%	3%	1%	4%

Different Aggregation Schemes have Various Impacts on the Correlation Between SLOC and Other Metrics. To illustrate the effect of the various aggregation schemes, we compute the *gain ratios* of the correlation values between a metric and SLOC when aggregated to the file-level. Below, we define the gain ratio for a metric m when aggregated using a particular scheme AG

$$cor.gain.ratio(m, AG) = \frac{cor.file(\text{SLOC}, AG(m))}{cor.method(\text{SLOC}, m)} \qquad (1)$$

While we find that aggregation schemes do impact correlation values, most aggregation schemes do not have a consistent impact on all of the studied metrics. On the one hand, the gain ratios of Fig. 4 show that summation tends to increases the correlation between SLOC and all of the other metrics. On the other hand, for the NPATH, FANIN, and FANOUT metrics, Fig. 4 shows that the median gain ratios are often below 1, indicating that most aggregation schemes decrease the correlation values between these metrics and SLOC in half of the studied systems.

Table 5 presents the results of the Mann-Whitney U tests and Cliff's δ. We find that summation has a consistently large impact (i.e., p-value is below α and the absolute value of Cliff's δ is greater than 0.474) on the correlation between SLOC and the other metrics in software systems developed in C, C++, C#, and Java. This observation is consistent with Landman et al.'s work [18], which found that summation tends to inflate the correlation between SLOC and CC when aggregated from the method- to the file-level in Java projects.

Not all metrics are sensitive to aggregation schemes. Indeed, only the FANIN and FANOUT metrics are significantly sensitive to aggregation schemes. Furthermore, contrary to the CC results, these aggregations tend to weaken their correlation with SLOC.

When compared to the other aggregation schemes, summation has the largest impact on the correlation between the studied metrics and defect count. Table 6 shows the percentage of the studied systems that have a substantial correlation (i.e., $|\rho| \geq 0.4$)

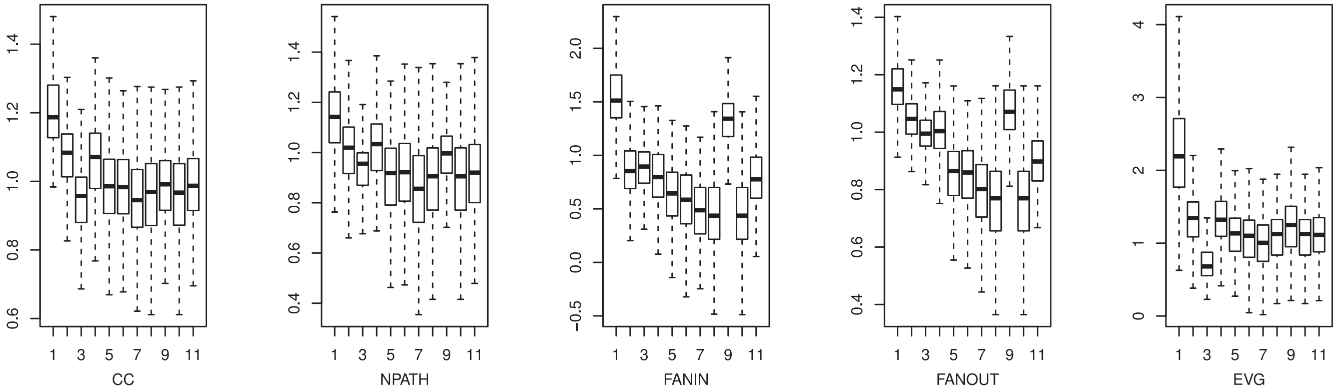


Fig. 4. Boxplots of the gain ratios in correlations between SLOC and other metrics at file-level. The order of the 11 aggregation schemes are the same as shown in Table 5.

between defect count and a given metric when aggregated using the studied schemes. File-level metrics that are aggregated by summation share a substantial correlation with defect count in 15 to 22 percent of the studied systems. The other aggregation schemes show potential to make file-level metrics substantially correlate with defect count, with 1-9 percent of the studied systems yielding substantial correlation values. We further investigate how likely it is that the observed improvements could have happened by chance, i.e., whether an aggregation scheme has the identical effect (i.e., the improvement in the correlation values) on different metrics. We perform a non-parametric test, namely the Cochran's Q test, using the 95 percent confidence level (i.e., $\alpha = 0.05$). The p-values of the Cochran's Q test on the mean, median, and standard deviation schemes are greater than 0.05, indicating that we cannot reject the null hypothesis that these three aggregation schemes have similar impact on the correlation values between defect count and all six studied metrics. On the other hand, the p-values of the Cochran's Q test on other aggregation schemes are less than 0.05, indicating that these aggregation schemes have significantly different effects on different metrics. We observe that these aggregation schemes tend to yield substantial correlation values between defect count and the metric NPATH in more subject systems than that of defect count and other metrics. In addition to summation, applying other aggregation schemes may provide useful new features for defect prediction models.

Aggregation can significantly alter the correlation among the studied metrics and the correlation between the studied metrics and defect count. Experimenting with aggregation schemes may produce useful new metrics for defect prediction models.

5.2 Defect Prediction Models

Our analysis in the prior section shows that aggregation schemes can significantly alter the correlation among metrics and the correlation between defect count and metrics. These results suggest that using additional aggregation schemes may generate new metrics that capture unique characteristics of the studied data, and that may be useful for defect prediction. In this section, we investigate the impact that aggregation schemes have on four types of defect prediction models. While we use the same metrics in each type of defect prediction model, the dependent variable varies as described below:

- *Defect proneness*: A binary variable indicating if a file is defective or not.
- *Defect rank*: A ranked list of files according to the number of defects that they will contain.
- *Defect count*: The exact number of defects in a file.
- *Effort-aware*: A cost-effective list of files ranked in order to locate the most number of defects while inspecting the least number of lines.

5.2.1 Research Questions

To investigate the impact that aggregation schemes have on our four types of defect prediction models, we formulate the following four research questions:

RQ2.1 Do aggregation schemes impact the performance of defect proneness models?

RQ2.2 Do aggregation schemes impact the performance of defect rank models?

RQ2.3 Do aggregation schemes impact the performance of defect count models?

RQ2.4 Do aggregation schemes impact the performance of effort-aware models?

5.2.2 Experimental Design

In this section, we present the design of our experiments, including the evaluation method, the modelling techniques, the performance measures, the model training approach, and the null hypotheses. Fig. 5 gives an overview of our approach to address RQs 2.1-2.4.

1) *Evaluation Method*. In our experiment, we use the out-of-sample bootstrap model validation technique [73]. The out-of-sample bootstrap is a robust model validation technique that has been shown to provide stable results for unseen data [73]. The process is made up of two steps.

First, a bootstrap sample is selected. From an original dataset with N instances, N instances are selected with replacement to create a bootstrap sample. The probability of an instance not being selected after N times is $(1-\frac{1}{N})^N$, and $\lim_{N->+\infty}(1-\frac{1}{N})^N = e^{-1} = 0.368$. Thus, on average, approximately 63.2 percent (i.e., 1-e^{-1}) of unique instances would be selected from the original dataset.

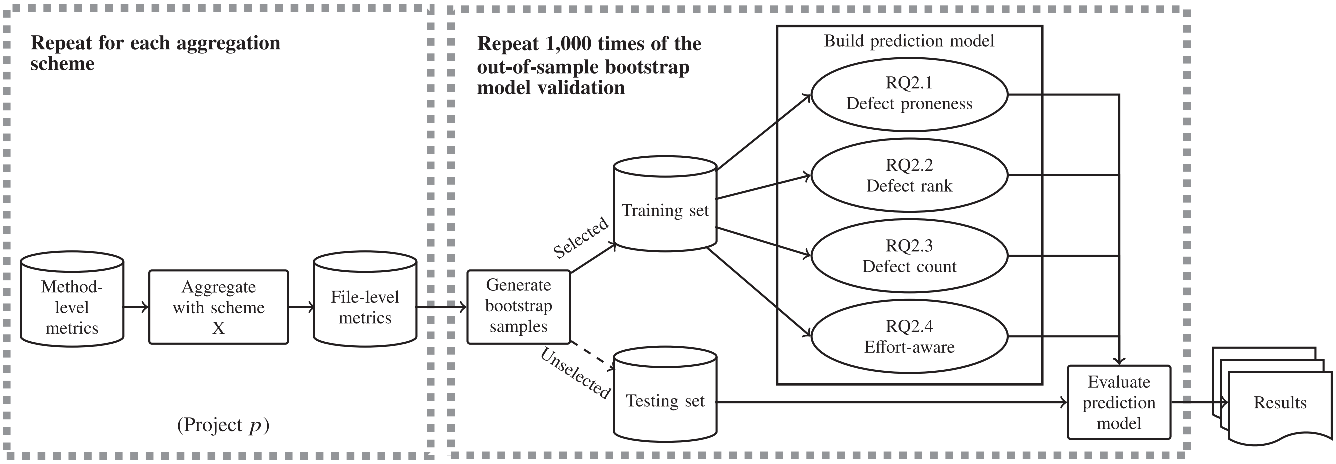


Fig. 5. Our approach to build and evaluate defect prediction models on each of the studied 255 projects, using file-level metrics aggregated from method-level metrics (RQs 2.1 to 2.4).

Second, a model is trained using the bootstrap sample and tested using the 36.8 percent of the data from the original dataset that does not appear in the bootstrap sample.

The two-step process is repeated K times, drawing a new bootstrap sample with replacement for training a model and testing it on the unselected data. The performance estimate is the average of the performance of each of these bootstrap-trained models. For each studied system, we perform 1,000 bootstrap iterations (i.e., $K = 1{,}000$) in order to derive a stable performance estimate.

2) *Modelling Techniques and Performance Measures*. *Defect Proneness.* Random forest is a robust classification technique [74] and is quite robust to parameter choices in defect proneness models [75]. Similar to much prior work [76], [77], we apply the random forest algorithm [78] to train our defect proneness models. We use the R package *randomForest* [79] with default parameters except for the number of trees that is set to 200 (sensitivity analysis with settings of 100 or 300 trees reach the same conclusion for this research question). Common performance measures for defect proneness models include precision, recall, accuracy, and F-measure. These measures are calculated using a confusion matrix that is obtained using a threshold value. The threshold value is applied on the predicted probability of defect proneness to differentiate between defective and clean entities. Since the aforementioned performance measures are sensitive to the selected threshold, we opt to use the Area Under the receiver operating characteristic Curve (AUC)—a threshold-independent performance measure. AUC is computed as the area under the Receiver Operating Characteristics (ROC) curve, which plots the true positive rate against the false positive rate while varying the threshold value from 0 to 1. AUC values range between 0 (worst performance) and 1 (best performance). A model with an AUC of 0.5 or less performs no better than random guessing.

Defect Rank. To train our defect rank models, we apply linear regression, which has been successfully used in several prior studies of defect density models [28] and defect count models [7], [15]. The regression model is applied to all files in the system and the files are ranked according to their estimated defect count. As suggested by prior work [15], [28], we use Spearman's ρ to measure the performance of our defect rank models. We compute ρ between the ranked list produced by the model and the correct ranking that is observed in the historical data. Larger ρ values indicate a more accurate defect rank model.

Defect Count. Similar to our defect rank models, we apply linear regression to train our defect count models. We use the Mean Squared Error (MSE) to measure the performance of our linear models, which is defined as follows:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y_i} - Y_i)^2, \tag{2}$$

where Y_i and $\hat{Y_i}$ are the actual and predicted value of the ith file, and n is the total number of files. The lower the MSE, the better the performance of the defect count model.

Effort-Aware. We also apply linear regression to train our effort-aware models. We use the P_{opt} measure proposed by Mende and Koschke [80] to measure the performance of our effort-aware models. The P_{opt} measure is calculated by drawing two curves that plot the accumulated lines of analyzed code on the x-axis, and the percentage of addressed bugs on the y-axis. First, the optimal curve is drawn using an ordering of files according to their actual defect densities. Second, the model performance curve is drawn by ordering files according to their predicted defect density. The area between the two curves is represented as Δ_{opt}, and $P_{opt} = 1 - \Delta_{opt}$. The higher the P_{opt} value, the closer the curve of the predicted model is to the optimal model, i.e., the higher the P_{opt} value, the better.

Table 7 summarizes the modelling techniques and performance measures for each type of defect prediction models.

3) *Prediction Model Training*. In each bootstrap iteration, we build 48 models—one model for each combination of the four types of defect prediction models and 12

TABLE 7
The Modelling Techniques and Performance Measures Used in This Study

Prediction type	Modelling technique	Performance measure
Defect proneness	Random forest	AUC
Defect rank	Linear regression	Spearman's ρ
Defect count	Linear regression	MSE
Effort-aware defect count	Linear regression	P_{opt}

configurations of the studied aggregation schemes. We use one configuration to study each of the 11 aggregation schemes individually, and a 12th configuration to study the combination of all of the aggregation schemes.

In each configuration, we use the six software metrics aggregated by the corresponding scheme as predictors for our defect prediction models. Thus, for 11 configurations that only involve one aggregation scheme, we use six predictors, and for the configuration that involves all schemes, we use 66 (i.e., 6×11) predictors.

Since the predictors may be highly correlated with one another, they may introduce multicollinearity, which can threaten the fitness and stability of the models [81]. To address this concern, a common practice is to apply Principal Component Analysis (PCA) to the input set of predictors [82], [83]. PCA permits us to leverage all the signals in our metrics whereas correlation analysis is binary in nature (i.e., we have to include or exclude a metric). Although principal components are difficult to interpret [84], the analysis of the impact of particular metrics is out of the scope of this paper. Hence, we adopt this technique to simplify the process of building defect prediction models in this study. We order the principal components by their amount of explained variance, and select the first N principal components that can explain at least 95 percent [83] of variance for inclusion in our defect prediction models. In total, we train over 12 million (i.e., $48 \times 1{,}000 \times 255$) models in our defect prediction experiment.

4) *Null Hypotheses*. As the four types of prediction models are similar, we formulate two general hypotheses to structure our investigation of the impact that aggregation schemes have on defect prediction models. To enable the comparison, we create an ideal model that achieves the best performance of models that are obtained using any of the 12 studied configurations. For each type of defect prediction model, the best performance is determined by the corresponding performance measure (see Table 7) for each iteration. We test the following two null hypotheses for each studied system:

$H0_{2a}$: *There is no difference in the performance of the best model and models that are trained using metrics that are aggregated by scheme AG.*

$H0_{2b}$: *There is no difference in the performance of the best model and models that are trained using metrics that are aggregated using all 11 schemes.*

To test our hypotheses, we conduct two-sided and paired Mann-Whitney U tests [64] with $\alpha = 0.05$. As we have 255 systems in total, we apply Bonferroni correction to control family-wise errors, and then adjust α by dividing by the number of tests. We use Cliff's δ to quantify the size of the impact.

We consider that a model fails to yield the best performance if the p-value of Mann-Whitney U test is less than α and Cliff's $|\delta| \geq 0.474$ (i.e., large effect).

5.2.3 Case Study Results

In this section, we present our findings from an overall perspective and a programming language-specific perspective.

1) *General Findings. The summation scheme (i.e., the most commonly applied aggregation scheme in the literature) can significantly underestimate the predictive power of defect prediction models that are built with the studied metrics.* Table 8 shows that solely using summation achieves the best performance when predicting defect proneness in only 11 percent of projects. When predicting defect rank or performing effort-aware prediction, solely using summation yields the best performance in 56 percent and 31 percent of projects, respectively. Such findings suggest that the predictive power of defect prediction models can be hindered by solely relying on summation for aggregating metrics.

TABLE 8
The Percentage of the Studied Systems on Which the Model Built with the Corresponding Configuration of Aggregations Achieves the Best Performance

Scheme	Defect proneness	Defect rank	Defect count	Effort-aware
All schemes	**102 (40%)**	**153 (60%)**	248 (97%)	42 (16%)
Sum	28 (11%)	143 (56%)	**253 (99%)**	79 (31%)
Mean	19 (7%)	33 (13%)	222 (87%)	176 (69%)
Median	21 (8%)	28 (11%)	210 (82%)	**180 (71%)**
SD	17 (7%)	37 (15%)	230 (90%)	124 (49%)
COV	24 (9%)	40 (16%)	238 (93%)	58 (23%)
Gini	21 (8%)	31 (12%)	231 (91%)	69 (27%)
Hoover	20 (8%)	28 (11%)	227 (89%)	83 (33%)
Atkinson	21 (8%)	37 (15%)	230 (90%)	106 (42%)
Shannon	36 (14%)	92 (36%)	246 (96%)	51 (20%)
Entropy	25 (10%)	39 (15%)	229 (90%)	103 (40%)
Theil	19 (7%)	42 (16%)	232 (91%)	77 (30%)

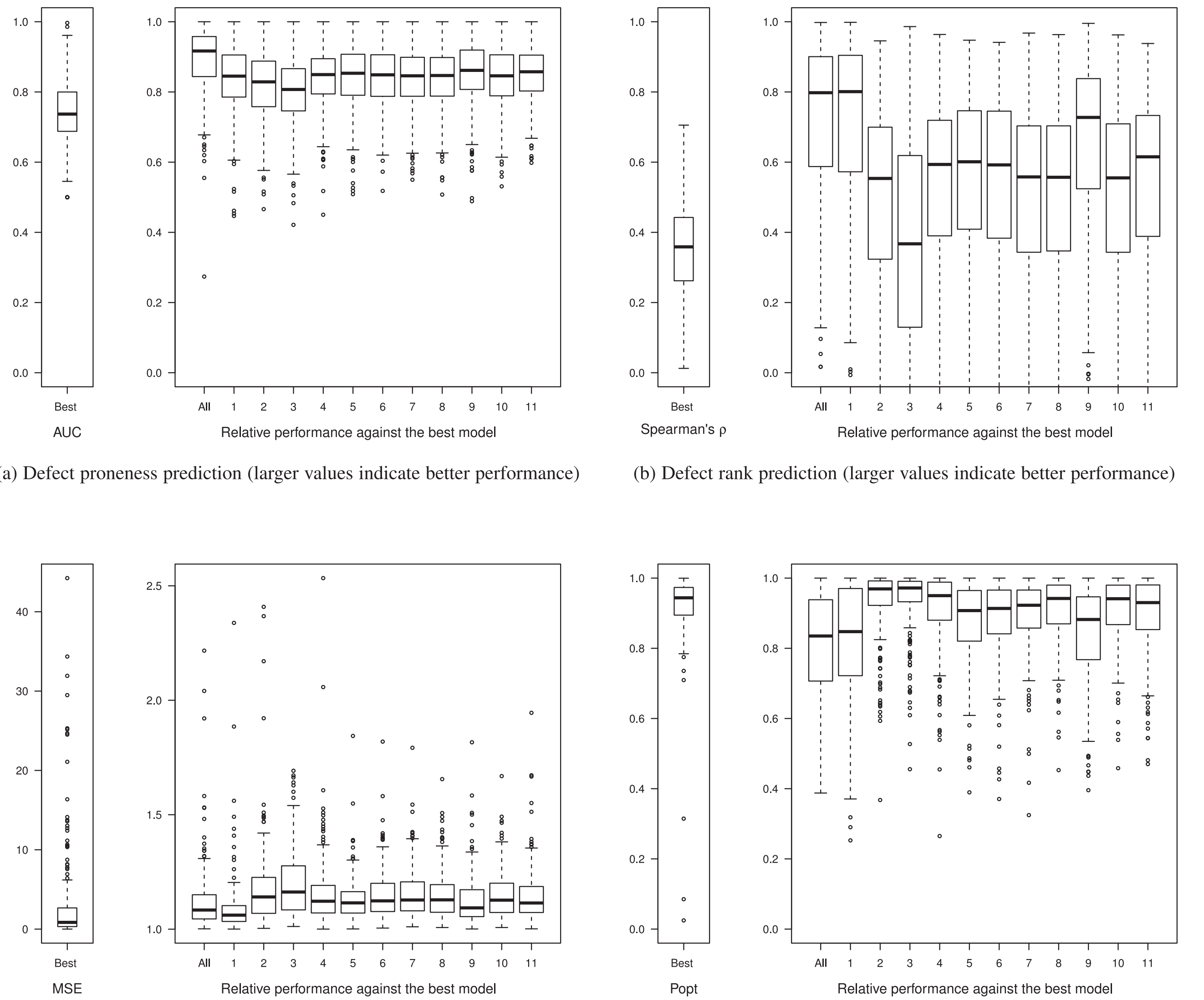
*The **bold** font highlights the best configuration.*

On the other hand, using all of the studied aggregation schemes is significantly better than solely using summation in models that predict defect proneness (Fig. 6a). Specifically, using all schemes achieves the best performance in 40 percent of projects. This finding indicates that exploring various aggregation schemes can yield fruitful results when building models to predict defect proneness.

In models that predict defect rank (Fig. 6b) and defect count (Fig. 6c), the difference between using all schemes and solely using summation is marginal, and both are closer to the best performance than any other aggregation scheme. In models that predict defect rank, using all schemes is slightly better than solely using summation (i.e., 60 versus 56 percent). When predicting defect count, solely using summation is slightly better than using all schemes (i.e., 99 versus 97 percent). Given the higher percentage of studied systems where the defect count shares a substantial correlation with the summed metrics (Table 6), it is understandable that summation would be a good aggregation scheme for predicting defect count.

When fitting effort-aware models (Fig. 6d), the situation changes, i.e., neither using all schemes nor solely using summation is advisable. The median scheme provides the best performance in 71 percent of projects. Using the mean scheme is a viable alternative, as it achieves the best performance in 69 percent of projects. Both mean and median aggregation schemes are much better than using any other configuration of aggregators.

We suspect that using either the median or the mean scheme performs the best for effort-aware models because files with the same number of predicted defects may be still distinguishable when using these two schemes. For example, let's consider two files having the same number of predicted defects. In a model that is built using only LOC (such a model may still achieve good predictive power [13]), these two files may have the same sum_SLOC (when the model is built using



(a) Defect proneness prediction (larger values indicate better performance)

(b) Defect rank prediction (larger values indicate better performance)

(c) Defect count prediction (smaller values indicate better performance)

(d) Effort-aware defect prediction (larger values indicate better performance)

Fig. 6. In each sub figure, the left boxplot shows the best performance, and the right boxplots present the performance by models built with each aggregation scheme *relative to the best performance*. The order of aggregation schemes: all schemes, summation, mean, median, SD, COV, Gini, Hoover, Atkinson, Shannon's entropy, generalized entropy, and Theil.

the summation scheme) or the same *avg*_SLOC (when the model is built using the mean scheme). Then these two files have the same density of defects (when using the summation scheme) or their density of defects is further determined by the number of methods (when using the mean scheme). In the latter case, these two files can be distinguished from one another. The file with fewer methods (thus smaller sum_SLOC and less effort for code inspection) has a higher density of predicted defects and is ranked before the other one. This is in agreement with the concept of effort-aware defect prediction, i.e., finding the same number of defects with less effort.

Fig. 6 provides boxplots of the best performance of our various model configurations, together with the performance of models built using each configuration relative to the best model. As described above, Fig. 6 shows that when using all schemes together, the performances of defect proneness models are generally greater than using a single scheme. Furthermore, when predicting defect rank and count, solely using summation or using all schemes achieve very similar amounts of predictive power, and both are generally better than using any other aggregation scheme. Hence, applying all schemes together is beneficial for defect proneness models, while using summation is likely sufficient for models that predict defect rank and count. Moreover, when building effort-aware models, either using mean or median generally achieves better performance than using any other configuration. Hence, the median or mean schemes are advisable for building effort-aware models.

2) *Programming Language-Specific Findings*. The distributions of software metric tend to vary based on the programming language in which the system is implemented [57]. This varying distribution may interfere with our analysis of aggregation schemes. To investigate the role that programming language plays, we partition the results of Table 8 according to programming languages, and present the results in Table 9.

Irrespective of the programming language, the impact that aggregation schemes have on defect prediction models that are built with the studied metrics remains largely consistent. For instance, using all schemes is generally beneficial to most of the studied

TABLE 9
The Percentage of the Studied Systems per Programming Language, on Which the Model Built with the Corresponding Aggregation Scheme Achieves Similar Predictive Power as the Best Model

(a) Defect proneness

Scheme	Programming language			
	C	C++	C#	Java
All schemes	**9 (26%)**	**28 (33%)**	**3 (20%)**	**62 (51%)**
Sum	2 (6%)	11 (13%)	0 (0%)	15 (12%)
Mean	2 (6%)	5 (6%)	**3 (20%)**	9 (7%)
Median	3 (9%)	9 (11%)	0 (0%)	9 (7%)
SD	3 (9%)	5 (6%)	2 (13%)	7 (6%)
COV	5 (15%)	5 (6%)	1 (7%)	13 (11%)
Gini	2 (6%)	9 (11%)	2 (13%)	8 (7%)
Hoover	6 (18%)	4 (5%)	1 (7%)	9 (7%)
Atkinson	2 (6%)	7 (8%)	1 (7%)	11 (9%)
Shannon	4 (12%)	15 (18%)	1 (7%)	16 (13%)
Entropy	4 (12%)	10 (12%)	1 (7%)	10 (8%)
Theil	3 (9%)	4 (5%)	2 (13%)	10 (8%)

(b) Defect rank

Scheme	Programming language			
	C	C++	C#	Java
All schemes	**19 (56%)**	**54 (64%)**	**6 (40%)**	74 (61%)
Sum	16 (47%)	46 (54%)	**6 (40%)**	**75 (62%)**
Mean	5 (15%)	10 (12%)	2 (13%)	16 (13%)
Median	9 (26%)	2 (2%)	1 (7%)	16 (13%)
SD	6 (18%)	10 (12%)	4 (27%)	17 (14%)
COV	6 (18%)	15 (18%)	3 (20%)	16 (13%)
Gini	6 (18%)	10 (12%)	1 (7%)	14 (12%)
Hoover	5 (15%)	7 (8%)	2 (13%)	14 (12%)
Atkinson	4 (12%)	15 (18%)	2 (13%)	16 (13%)
Shannon	13 (38%)	36 (42%)	3 (20%)	40 (33%)
Entropy	4 (12%)	17 (20%)	2 (13%)	16 (13%)
Theil	5 (15%)	15 (18%)	2 (13%)	20 (17%)

(c) Defect count

Scheme	Programming language			
	C	C++	C#	Java
All schemes	**34 (100%)**	**84 (99%)**	13 (87%)	117 (97%)
Sum	**34 (100%)**	**84 (99%)**	**15 (100%)**	**120 (99%)**
Mean	31 (91%)	76 (89%)	14 (93%)	101 (83%)
Median	29 (85%)	71 (84%)	13 (87%)	97 (80%)
SD	32 (94%)	76 (89%)	14 (93%)	108 (89%)
COV	33 (97%)	80 (94%)	14 (93%)	111 (92%)
Gini	32 (94%)	76 (89%)	14 (93%)	109 (90%)
Hoover	32 (94%)	76 (89%)	14 (93%)	105 (87%)
Atkinson	33 (97%)	76 (89%)	14 (93%)	107 (88%)
Shannon	34 (100%)	83 (98%)	14 (93%)	115 (95%)
Entropy	33 (97%)	76 (89%)	14 (93%)	106 (88%)
Theil	33 (97%)	79 (93%)	14 (93%)	106 (88%)

(d) Effort-aware defect count

Scheme	Programming language			
	C	C++	C#	Java
All schemes	8 (24%)	18 (21%)	1 (7%)	15 (12%)
Sum	12 (35%)	28 (33%)	8 (53%)	31 (26%)
Mean	26 (76%)	**62 (73%)**	11 (73%)	77 (64%)
Median	**27 (79%)**	54 (64%)	**12 (80%)**	**87 (72%)**
SD	19 (56%)	51 (60%)	6 (40%)	48 (40%)
COV	12 (35%)	20 (24%)	3 (20%)	23 (19%)
Gini	12 (35%)	26 (31%)	6 (40%)	25 (21%)
Hoover	15 (44%)	30 (35%)	5 (33%)	33 (27%)
Atkinson	17 (50%)	35 (41%)	8 (53%)	46 (38%)
Shannon	10 (29%)	16 (19%)	1 (7%)	24 (20%)
Entropy	17 (50%)	32 (38%)	8 (53%)	46 (38%)
Theil	12 (35%)	30 (35%)	4 (27%)	31 (26%)

*The **bold** font highlights the best performing scheme.*

systems when predicting defect proneness (Table 9a), no matter what programming language the system is written in. When predicting defect rank (Table 9b), using all schemes achieves results that are the closest to the performance of the best model for projects developed in C and C++, while using summation is slightly better than using all schemes for only one project developed in Java. For projects developed in C# or Java, solely using summation in models that predict defect count (Table 9c) is slightly better than using all schemes with a two and three projects difference, respectively. When building effort-aware models (Table 9d), using the median scheme is beneficial to most of the systems written in C, C#, and Java. For systems written in C++, using the mean scheme achieves results that are slightly closer to the best performance than using median. Hence, we conclude that the impact of aggregation schemes is largely consistent across systems developed in any of the four studied programming languages.

Solely using summation rarely leads to the best performance in models that predict defect proneness or effort-aware models, where using all schemes and using mean/median are recommended, respectively. Moreover, using all schemes is still beneficial to defect rank models, especially for projects written in C and C++. In models that predict defect count, solely using the summation is probably sufficient. Indeed, applying all schemes is a low-cost option that is worth experimenting with.

5.3 Guidelines for Future Defect Prediction Studies

In this section, we discuss the broader implications of our results by providing guidelines for future defect prediction studies. Note that these guidelines are valid for studies using exactly the same metrics as this study. If different metrics are used, researchers and practitioners can follow our approach to derive the most appropriate guidelines for their studies. All the needed information to perform our analysis can be obtained from the training data (e.g., data from a previous release) that is used to build a model as in all prior defect prediction studies.

1) *Regardless of the programming language, using all studied aggregation schemes is recommended when building models for predicting defect proneness and rank.* With the initial set of predictors that are aggregated using all of the available schemes, feature reduction (e.g., PCA) could then be applied to mitigate redundancies before fitting a prediction model. In particular, defect proneness models that use all aggregation schemes achieve the best performance in 40 percent of the studied systems, while solely using the summation scheme achieves the best performance in only 11 percent of projects. Furthermore, for models that rank files according to their defect density, using all schemes is better than solely using summation for projects developed that are in C and C++.
2) *Using summation is recommended for defect count models.* Solely using summation is better than using all schemes for projects that are developed in C# or Java,

and leads to the same predictive power as using all schemes for projects that are developed in C and C++.

3) *Either the mean or the median aggregation scheme should be used in effort-aware defect prediction models.* In particular, the median aggregation scheme should be used for projects developed in C, C#, or Java. The mean aggregation scheme is suggested when building effort-aware defect prediction models for C++ projects. In general, using median achieves the best performance for 71 percent of the studied systems.

6 THREATS TO VALIDITY

In this section, we discuss the threats to the validity of our study with respect to Yin's guidelines for case study research [85].

Threats to conclusion validity are concerned with the relationship between the treatment and the outcome. The threat to our treatments mainly arises from our choice of metrics (i.e., only six method-level metrics, and no class-level metrics), our choice of modelling techniques (i.e., random forest for defect proneness models and linear regression for the other three types of defect prediction models), and our choice of model parameters (i.e., 200 trees in random forest). The choice of parameters has been explored in prior work [75], [86], [87]. However, the primary goal of our study is not to train the most effective defect prediction models, but instead to measure relative improvements by exploring different aggregation schemes.

Threats to internal validity are concerned with our selection of subject systems and analysis methods. As the majority of systems that are hosted on SourceForge and GoogleCode are immature, we carefully filter out systems that have not accumulated sufficient history to train defect prediction models. To obtain a stable picture of the performance of our defect prediction models, we perform 1,000 iterations of out-of-sample bootstrap validation. In addition, we apply non-parametric statistical methods (i.e., Mann-Whitney U test and Cliff's δ) to draw our conclusions.

Threats to external validity are concerned with the generalizability of our results. We investigate 11 schemes that can capture five aspects (summation, central tendency, dispersion, inequality index, and entropy) of the distribution of software metrics. Moreover, we study 255 open source systems that are drawn from a broad range of domains. Hence, we believe that our conclusions may apply to other defect prediction contexts. Nonetheless, replication studies may prove fruitful.

Another threat is our choice of PCA for removing multicollinearity, as we lose interpretability of the produced models. If the goal of a study is interpretability, a more careful choice of aggregations might be needed. Since the focus of this work is on model performance, our exploration approach is useful. But future work may want to explore other methods for preserving interpretability.

Threats to reliability validity are concerned with the possibility of replicating this study. Our subject projects are all open source systems, and the tool for computing software metrics is publicly accessible. Furthermore, we provide all of the necessary details of our experiments in a replication package that we have posted online.[2]

2. http://www.feng-zhang.com/replications/TSEaggregation.html

7 CONCLUSION

Aggregation is an unavoidable step in training defect prediction models at the file-level. This is because defect data is often collected at file-level, but many software metrics are computed at the method- and class-levels. One of the widely used schemes for metric aggregation is summation [5], [6], [7], [8], [9], [10], [11], [15], [17]). However, recent work [18] suggests that summation can inflate the correlation between SLOC and CC in Java projects. Fortunately, there are many other aggregation schemes that capture other dimensions of a low-level software metric (e.g., dispersion, central tendency, inequality, and entropy). Yet, the impact that these additional aggregation schemes have on defect prediction models remains largely unexplored.

To that end, we perform experiments using 255 open source systems to explore how aggregation schemes impact the performance of defect prediction models. First, we investigate the impact that aggregation schemes have on the correlation among metrics and the correlation between metrics and defect count. We find that aggregation can increase or decrease both types of correlation. Second, we examine the impact that aggregation schemes have on defect proneness, defect rank, defect count, and effort-aware defect prediction models. Broadly speaking, we find that summation tends to underestimate the performance of defect proneness and effort-aware models. Hence, it is worth applying multiple aggregation schemes for defect prediction purposes. For instance, applying all 11 schemes achieves the best performance in predicting defect proneness in 40 percent of the studied projects.

From our results, we provide the following guidelines for future defect prediction studies. When building models for predicting defect proneness and rank, our recommendation is to use all of the available aggregation schemes to generate the initial set of predictors (i.e., aggregated metrics, such as the summation, median, and standard deviation of lines of code), and then perform feature reduction (e.g., PCA) to mitigate redundancies. For models that predict defect count, solely using summation is likely sufficient. For effort-aware defect prediction models, surprisingly, using all 11 schemes to generate the initial set of predictors does not outperform using a single scheme (i.e., the median or the mean scheme); instead, the median scheme is advised for projects developed in C, C#, or Java, and the mean scheme is suggested for projects written in C++.

If a researcher or a practitioner has a reason for selecting a particular aggregation scheme, that should indeed trump our approach. But, in many cases, selecting an aggregation scheme is not straightforward. Our results show that naïvely selecting the summation may not yield the best results. Instead, in such cases, our approach would be better. The improvement in model performance is substantial enough to outweigh the analysis cost on these additional aggregation schemes. Therefore, we suggest that researchers and practitioners experiment with many aggregation schemes when building defect prediction models.

REFERENCES

[1] P. Bhattacharya and I. Neamtiu, "Assessing programming language impact on development and maintenance: A study on C and C++," in *Proc. 33rd ACM Int. Conf. Softw. Eng.*, 2011, pp. 171–180.

[2] A. Tosun, A. Bener, B. Turhan, and T. Menzies, “Practical considerations in deploying statistical methods for defect prediction: A case study within the Turkish telecommunications industry,” *Inform. Softw. Technol.*, vol. 52, no. 11, pp. 1242–1257, Nov. 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584910001163

[3] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, “The design space of bug fixes and how developers navigate it,” *IEEE Trans. Softw. Eng.*, vol. 41, no. 1, pp. 65–81, Jan. 2015.

[4] T. Menzies, R. Krishna, and D. Pryor, “The promise repository of empirical software engineering data,” North Carolina State University, Department of Computer Science, (2015). [Online]. Available: http://openscience.us/repo

[5] 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. 30th ACM Int. Conf. Softw. Eng.*, May 2008, pp. 181–190.

[6] 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.

[7] T. Zimmermann and N. Nagappan, “Predicting defects using network analysis on dependency graphs,” in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 531–540.

[8] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In, “Micro interaction metrics for defect prediction,” in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng.*, 2011, pp. 311–321.

[9] J. Nam, S. J. Pan, and S. Kim, “Transfer defect learning,” in *Proc. 2013 Int. Conf. Softw. Eng.*, 2013, pp. 382–391. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486839

[10] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, “Towards building a universal defect prediction model,” in *Proc. 11th Work. Conf. Mining Softw. Repositories*, 2014, pp. 41–50.

[11] T. Nguyen, B. Adams, and A. Hassan, “Studying the impact of dependency network measures on software quality,” in *Proc. 26th IEEE Int. Conf. Softw. Maintenance*, Sep. 2010, pp. 1–10.

[12] D. Posnett, V. Filkov, and P. Devanbu, “Ecological inference in empirical software engineering,” in *Proc. 26th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2011, pp. 362–371.

[13] M. D’Ambros, M. Lanza, and R. Robbes, “Evaluating defect prediction approaches: A benchmark and an extensive comparison,” *Empirical Softw. Eng.*, vol. 17, no. 4/5, pp. 531–577, Aug. 2012. [Online]. Available: http://dx.doi.org/10.1007/s10664–011-9173-9

[14] A. G. Koru and H. Liu, “Building defect prediction models in practice,” *IEEE Softw.*, vol. 22, no. 6, pp. 23–29, Nov. 2005. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1524911

[15] T. Zimmermann, R. Premraj, and A. Zeller, “Predicting defects for eclipse,” in *Proc. Int. Workshop Predictor Models Softw. Eng.*, May 2007, Art. no. 9.

[16] T. Holschuh, M. Pauser, K. Herzig, T. Zimmermann, R. Premraj, and A. Zeller, “Predicting defects in SAP Java code: An experience report,” in *Proc. 31st IEEE Int. Conf. Softw. Eng.-Companion Vol.*, 2009, pp. 172–181. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5070975

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

[18] D. Landman, A. Serebrenik, and J. Vinju, “Empirical analysis of the relationship between CC and SLOC in a large corpus of java methods,” in *Proc. 30th IEEE Int. Conf. Softw. Maintenance Evolution*, Sep. 2014, pp. 221–230.

[19] N. Fenton and N. Ohlsson, “Quantitative analysis of faults and failures in a complex software system,” *IEEE Trans. Softw. Eng.*, vol. 26, no. 8, pp. 797–814, Aug. 2000. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=879815

[20] T. Graves, A. Karr, J. Marron, and H. Siy, “Predicting fault incidence using software change history,” *IEEE Trans. Softw. Eng.*, vol. 26, no. 7, pp. 653–661, Jul.2000. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=859533

[21] T. McCabe, “A complexity measure,” *IEEE Trans. Softw. Eng.*, vol. TSE-2, no. 4, pp. 308–320, Dec. 1976.

[22] C. Gini, “Measurement of inequality of incomes,” *Econ. J.*, vol. 31, no. 121, pp. 124–126, Mar. 1921. [Online]. Available: http://dx.doi.org/10.1007/s12546–014-9138-0

[23] A. B. Atkinson, “On the measurement of inequality,” *J. Econ. Theory*, vol. 2, no. 3, pp. 244–263, 1970. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022053170900396

[24] J. Hoover and M. Edgar, “The measurement of industrial localization,” (in English), *Rev. Econ. Statist.*, vol. 18, no. 4, pp. 162–171, 1936. [Online]. Available: http://www.jstor.org/stable/1927875

[25] C. Shannon, “A mathematical theory of communication,” *Bell Syst. Tech. J.*, vol. 27, pp. 379–423/623–656, 1948.

[26] F. A. Cowell, “Generalized entropy and the measurement of distributional change,” *Eur. Econ. Rev.*, vol. 13, no. 1, pp. 147–159, 1980. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0014292180900513

[27] H. Theil, *Economics and Information Theory*. Amsterdam, The Netherlands: North-Holland Pub. Co., 1967.

[28] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” in *Proc. 27th ACM Int. Conf. Softw. Eng.*, 2005, pp. 284–292.

[29] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, “Predicting faults from cached history,” in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 489–498.

[30] A. Hassan, “Predicting faults using the complexity of code changes,” in *Proc. 31st IEEE Int. Conf. Softw. Eng.*, 2009, pp. 78 –88.

[31] M. D’Ambros, M. Lanza, and R. Robbes, “An extensive comparison of bug prediction approaches,” in *Proc. 7th IEEE Work. Conf. Mining Softw. Repositories*, May 2010, pp. 31–41. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5463279

[32] H. Hata, O. Mizuno, and T. Kikuno, “Bug prediction based on fine-grained module histories,” in *Proc. Int. Conf. Softw. Eng.*, 2012, pp. 200–210. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337247

[33] 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.

[34] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, “Cross-project defect prediction: A large scale experiment on data versus domain versus process,” in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2009, pp. 91–100. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595713

[35] T. Mende and R. Koschke, “Effort-aware defect prediction models,” in *Proc. 14th Eur. Conf. Softw. Maintenance Reengineering*, 2010, pp. 107–116. [Online]. Available: http://dx.doi.org/10.1109/CSMR.2010.18

[36] Y. Kamei, S. Matsumoto, A. Monden, K.-I. Matsumoto, B. Adams, and A. Hassan, “Revisiting common bug prediction findings using effort-aware models,” in *Proc. 26th IEEE Int. Conf. Softw. Maintenance*, Sep. 2010, pp. 1–10.

[37] T. Gyimothy, R. Ferenc, and I. Siket, “Empirical validation of object-oriented metrics on open source software for fault prediction,” *IEEE Trans. Softw. Eng.*, vol. 31, no. 10, pp. 897–910, Oct. 2005.

[38] E. Giger, M. D’Ambros, M. Pinzger, and H. C. Gall, “Method-level bug prediction,” in *Proc. ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas.*, 2012, pp. 171–180. [Online]. Available: http://doi.acm.org/10.1145/2372251.2372285

[39] X. Meng, B. Miller, W. Williams, and A. Bernat, “Mining software repositories for accurate authorship,” in *Proc. 29th IEEE Int. Conf. Softw. Maintenance*, Sep. 2013, pp. 250–259.

[40] B. Vasilescu, A. Serebrenik, and M. Van den Brand, “By no means: A study on aggregating software metrics,” in *Proc. 2nd ACM Int. Workshop Emerging Trends Softw. Metrics*, 2011, pp. 23–26. [Online]. Available: http://doi.acm.org/10.1145/1985374.1985381

[41] R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz, “Comparative analysis of evolving software systems using the Gini coefficient,” in *Proc. 25th IEEE Int. Conf. Softw. Maintenance*, Sep. 2009, pp. 179–188.

[42] A. Serebrenik and M. Van Den Brand, “Theil index for aggregation of software metrics values,” in *Proc. 26th IEEE Int. Conf. Softw. Maintenance*, Sep. 2010, pp. 1–9.

[43] B. Vasilescu, “Analysis of advanced aggregation techniques for software metrics,” Master’s thesis, Dept. Math. Comput. Sci., Eindhoven Univ. Technol., Eindhoven, The Netherlands, 2011.

[44] E. Giger, M. Pinzger, and H. Gall, “Using the Gini coefficient for bug prediction in eclipse,” in *Proc. 12th Int. Workshop Principles Softw. Evolution 7th ACM Annu. ERCIM Workshop Softw. Evolution*, 2011, pp. 51–55. [Online]. Available: http://doi.acm.org/10.1145/2024445.2024455

[45] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang, "An investigation on the feasibility of cross-project defect prediction," *Automated Softw. Eng.*, vol. 19, no. 2, pp. 167–199, Jun. 2012. [Online]. Available: http://dx.doi.org/10.1007/s10515–011-0090-3

[46] O. Goloshchapova and M. Lumpe, "On the application of inequality indices in comparative software analysis," in *Proc. 22nd Australian Softw. Eng. Conf.*, Jun. 2013, pp. 117–126.

[47] S.-C. Kolm, "Unequal inequalities. I," *J. Econ. Theory*, vol. 12, no. 3, pp. 416–442, 1976. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022053176900375

[48] J. E. Gaffney, "Estimating the number of faults in code," *IEEE Trans. Softw. Eng.*, vol. 10, no. 4, pp. 459–464, Jul. 1984.

[49] B. Kitchenham, L. Pickard, and S. Linkman, "An evaluation of some design metrics," *Softw. Eng. J.*, vol. 5, no. 1, pp. 50–58, Jan. 1990.

[50] A. Mockus, "Amassing and indexing a large sample of version control systems: Towards the census of public source code history," in *Proc. 6th IEEE Int. Work. Conf. Mining Softw. Repositories*, May 2009, pp. 11–20.

[51] SciTools, "Understand 3.1 build 726," 2015. [Online]. Available: https://scitools.com, Accessed on: Jun. 15, 2015.

[52] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced?: Bias in bug-fix datasets," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2009, pp. 121–130. (2009). [Online]. Available: http://doi.acm.org/10.1145/1595696.1595716

[53] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *Proc. 33rd ACM Int. Conf. Softw. Eng.*, 2011, pp. 481–490. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985859

[54] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: How misclassification impacts bug prediction," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 392–401. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486840

[55] F. Rahman, D. Posnett, I. Herraiz, and P. Devanbu, "Sample size versus bias in defect prediction," in *Proc. 21th ACM SIGSOFT Symp. 15th Eur. Conf. Foundations Softw. Eng.*, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2491411.2491418

[56] F. Peters, T. Menzies, and A. Marcus, "Better cross company defect prediction," in *Proc. 10th Work. Conf. Mining Softw. Repositories*, 2013, pp. 409–418. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487161

[57] F. Zhang, A. Mockus, Y. Zou, F. Khomh, and A. E. Hassan, "How does context affect the distribution of software maintainability metrics?" in *Proc. 29th IEEE Int. Conf. Softw. Maintainability*, 2013, pp. 350–359.

[58] A. Mockus and L. Votta, "Identifying reasons for software changes using historic databases," in *Proc. 16th Int. Conf. Softw. Maintenance*, 2000, pp. 120–130.

[59] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proc. 2nd Int. Workshop Mining Softw. Repositories*, 2005, pp. 1–5.

[60] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software fault prediction metrics: A systematic literature review," *Inform. Softw. Technol.*, vol. 55, no. 8, pp. 1397–1418, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584913000426

[61] S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Trans. Softw. Eng.*, vol. 7, no. 5, pp. 510–518, Sep. 1981.

[62] R. Subramanyam and M. Krishnan, "Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects," *IEEE Trans. Softw. Eng.*, vol. 29, no. 4, pp. 297–310, Apr. 2003.

[63] M. Jureczko and D. D. Spinellis, "Using object-oriented design metrics to predict software defects," *Proc. 5th Int. Conf. Dependability Comput. Syst.*, pp. 69–81, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0378375808000207

[64] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*, 4 ed. London, U.K.: Chapman & Hall/CRC, Jan. 2007.

[65] M. Triola, *Elementary Statistics*. Upper Saddle River, NJ, USA: Pearson/Addison-Wesley, 2004. [Online]. Available: https://books.google.ca/books?id=MX5GAAAAYAAJ

[66] M.-T. Puth, M. Neuhäuser, and G. D. Ruxton, "Effective use of Spearman's and Kendall's correlation coefficients for association between two measured traits," *Animal Behaviour*, vol. 102, no. 0, pp. 77–84, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0003347215000196

[67] G. Selim, L. Barbour, W. Shang, B. Adams, A. Hassan, and Y. Zou, "Studying the impact of clones on software defects," in *Proc. 17th Work. Conf. Reverse Eng.*, Oct. 2010, pp. 13–21.

[68] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi, "An empirical study of just-in-time defect prediction using cross-project models," in *Proc. 11th ACM Work. Conf. Mining Softw. Repositories*, 2014, pp. 172–181. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597075

[69] G. Succi, W. Pedrycz, S. Djokic, P. Zuliani, and B. Russo, "An empirical exploration of the distributions of the Chidamber and Kemerer object-oriented metrics suite," (in English), *Empirical Softw. Eng.*, vol. 10, no. 1, pp. 81–104, 2005. [Online]. Available: http://dx.doi.org/10.1023/B

[70] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions," *Psych. Bull.*, vol. 114, no. 3, pp. 494–509, Nov. 1993.

[71] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences: Jacob Cohen*, 2nd ed. Washington, DC, USA: Lawrence Erlbaum, Jan. 1988.

[72] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys?" in *Proc. Annu. Meet. Florida Assoc. Institutional Res.*, Feb. 2006, pp. 1–33.

[73] B. Efron and R. Tibshirani, *An Introduction to the Bootstrap*, Abingdon, U. K.: Taylor & Francis, 1994. [Online]. Available: http://books.google.ca/books?id=gLlpIUxRntoC

[74] M. Robnik-Šikonja, "Improving random forests," in *Proc. 15th Eur. Conf. Mach. Learning*, 2004, pp. 359–370. [Online]. Available: http://dx.doi.org/10.1007/978–3-540-30115-8_34

[75] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "Automated parameter optimization of classification techniques for defect prediction models," in *Proc. 38th ACM Int. Conf. Softw. Eng.*, 2016, pp. 321–332. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884857

[76] G. Gousios, M. Pinzger, and A. V. Deursen, "An exploratory study of the pull-based software development model," in *Proc. 36th ACM Int. Conf. Softw. Eng.*, 2014, pp. 345–355. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568260

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

[78] L. Breiman, "Random forests," (in English), *Mach. Learning*, vol. 45, no. 1, pp. 5–32, 2001. [Online]. Available: http://dx.doi.org/10.1023/A

[79] A. Liaw and M. Wiener, "Randomforest: Breiman and Cutler's random forests for classification and regression," 2015. [Online]. Available: http://CRAN.R-project.org/package=randomForest, Accessed on: April 18, 2016.

[80] T. Mende and R. Koschke, "Revisiting the evaluation of defect prediction models," in *Proc. 5th ACM Int. Conf. Predictor Models Softw. Eng.*, 2009, pp. 7:1–7:10. [Online]. Available: http://doi.acm.org/10.1145/1540438.1540448

[81] C. van Koten and A. Gray, "An application of Bayesian network for predicting object-oriented software maintainability," *Inf. Softw. Technol.*, vol. 48, no. 1, pp. 59–67, 2006.

[82] N. E. Fenton and M. Neil, "Software metrics: Roadmap," in *Proc. ACM Conf. Future Softw. Eng.*, 2000, pp. 357–370. [Online]. Available: http://doi.acm.org/10.1145/336512.336588

[83] G. Denaro and M. Pezzè, "An empirical evaluation of fault-proneness models," in *Proc. 24rd Int. Conf. Softw. Eng.*, May 2002, pp. 241–251.

[84] E. Shihab, "Practical software quality prediction," in *Proc. IEEE Int. Conf. Softw. Maintenance Evolution*, Sep. 2014, pp. 639–644.

[85] R. K. Yin, *Case Study Research: Design and Methods*, 3rd ed. Newbury Park, CA, USA: SAGE, 2002.

[86] A. L. I. Oliveira, P. L. Braga, R. M. F. Lima, and M. L. Cornélio, "GA-based method for feature selection and parameters optimization for machine learning regression applied to software effort estimation," *Inform. Softw. Technol.*, vol. 52, no. 11, pp. 1155–1166, Nov. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2010.05.009

[87] L. Song, L. L. Minku, and X. Yao, "The impact of parameter tuning on software effort estimation using learning machines," in *Proc. 9th ACM Int. Conf. Predictive Models Softw. Eng.*, 2013, pp. 9:1–9:10. [Online]. Available: http://doi.acm.org/10.1145/2499393.2499394

Feng Zhang received the bachelor's and master's degrees from Nanjing University of Science and Technology (China), in 2004 and 2006, respectively. He received the PhD degree in computer science from Queen's University, in 2016. He is currently a postdoctoral research fellow in the Department of Electrical and Computer Engineering, Queen's University, Canada. His research interests include empirical software engineering, software re-engineering, mining software repositories, source code analysis, and defect prediction. His research has been published at several top-tier software engineering venues, such as the International Conference on Software Engineering (ICSE), and the *Springer Journal of Empirical Software Engineering (EMSE)*. More about him and his work is available online at http://www.feng-zhang.com

Ahmed E. Hassan is the Canada Research Chair (CRC) in Software Analytics, and the NSERC/BlackBerry Software Engineering Chair at the School of Computing at Queen's University, Canada. His research interests include mining software repositories, empirical software engineering, load testing, and log mining. Hassan received a PhD in Computer Science from the University of Waterloo. He spearheaded the creation of the Mining Software Repositories (MSR) conference and its research community. Hassan also serves on the editorial boards of *IEEE Transactions on Software Engineering*, Springer Journal of Empirical Software Engineering, Springer Journal of Computing, and PeerJ Computer Science. Contact ahmed@cs.queensu.ca. More information at: http://sail.cs.queensu.ca/

Shane McIntosh received the bachelor's degree in applied computing from the University of Guelph and the MSc and PhD degrees in computer science from Queen's University. He is an assistant professor in the Department of Electrical and Computer Engineering, McGill University. In his research, he uses empirical software engineering techniques to study software build systems, release engineering, and software quality. His research has been published at several top-tier software engineering venues, such as the International Conference on Software Engineering (ICSE), the International Symposium on the Foundations of Software Engineering (FSE), and the *Springer Journal of Empirical Software Engineering (EMSE)*. More about him and his work is available online at http://shanemcintosh.org. He is a member of the IEEE.

Ying Zou is the Canada research chair in Software Evolution. She is an associate professor in the Department of Electrical and Computer Engineering, and cross-appointed to the School of Computing, Queen's University, Canada. She is a visiting scientist of IBM Centers for Advanced Studies, IBM Canada. Her research interests include software engineering, software reengineering, software reverse engineering, software maintenance, and service-oriented architecture. More about her and her work is available online at http://post.queensu.ca/~zouy. She is a member of the IEEE.