Software Code Quality Measurement: Implications from Metric Distributions

Siyuan Jin[1,2], Ziyuan Li[3,1,*], Bichao Chen[1,*], Bing Zhu[1,*], and Yong Xia[1,*]
[1] HSBC Laboratory, Guangzhou, China
[2] Department of Information Systems, Business Statistics and Operations Management,
Hong Kong University of Science and Technology, Hong Kong, China
[3] School of Physics, Sun Yat-sen University, Guangzhou, China
siyuan.jin@connect.ust.hk, liziyuan3@mail.sysu.edu.cn, bichao.chen@hsbc.com, bing1.zhu@hsbc.com, yong.xia@hsbc.com
*Corresponding Author

Abstract—Software code quality is a construct with three dimensions: maintainability, reliability, and functionality. Although many firms have incorporated code quality metrics in their operations, evaluating these metrics still lacks consistent standards. We categorized distinct metrics into two types: 1) monotonic metrics that consistently influence code quality; and 2) non-monotonic metrics that lack a consistent relationship with code quality. To consistently evaluate them, we proposed a distribution-based method to get metric scores. Our empirical analysis includes 36,460 high-quality open-source software (OSS) repositories and their raw metrics from SonarQube[1] and CK[2]. The evaluated scores demonstrate great explainability on software adoption. Our work contributes to the multi-dimensional construct of code quality and its metric measurements, which provides practical implications for consistent measurements on both monotonic and non-monotonic metrics.

Keywords–open source software, code quality, construct measurement, non-monotonic metric

1. Introduction

Code quality refers to the extent to which code is well-written and meets given needs [1]. Precise code quality measurement can improve software products, increase user satisfaction, and save costs of IT systems [2], which influences the software adoption [3, 4]. Therefore, numerous firms have incorporated measurements to evaluate code quality. However, these methods display a wide range of diversity and lack consistent standards.

Figure 1 shows that code quality is a multi-dimensional construct that includes dimensions: maintainability, reliability, and functionality [1]. Based on the literature on code quality dimension measurements, we identified 20 distinct metrics and divided them into monotonic and non-monotonic metrics. Monotonic metrics consistently impact code quality, while non-monotonic metrics lack a consistent relationship with code quality (Figure 2). Most monotonic metrics exhibit a monotonically decreasing relationship with code quality. A case in point is the number of code smells, which, when it rises, usually denotes a corresponding decline in the overall code quality.

[1]https://www.sonarsource.com
[2]https://github.com/mauricioaniche/ck

The literature remains a gap in the methodologies for consistently evaluating both types of code quality metrics, especially non-monotonic metrics. Therefore, the most prevalent method for assessing code quality within firms continues to be peer code review [5]. To consistently evaluate both types of code quality metrics, we propose a distribution-based method, which shows great explainability on software adoption.

> **Research Question 1**
>
> How to consistently evaluate both monotonic and non-monotonic metrics for software code quality?

We evaluated metric scores by analyzing their probability distributions among high-star OSS. For monotonic metrics, we fit an exponential distribution and use the weighted distance from threshold parameters in their cumulative distribution functions (CDFs) as their scores. For non-monotonic metrics, we fit an asymmetric Gaussian distribution and use the weighted distance away from the central point in their CDFs as their scores. The evaluated scores range from $0 \sim 100$ for each metric.

We conducted our empirical analysis on 36,460 GitHub OSS repositories. The selection of repositories with a high number of stars results in a more rigorous evaluation as those higher-quality repositories are used as reference points. The repositories that are slightly worse than our selected ones typically receive extremely low scores due to our sharper distributions from high-quality repositories.

> **Research Question 2**
>
> What are the implications of the evaluated scores on software adoption?

We investigated the explainability of our code quality metric scores on OSS stars. The number of stars reflects OSS quality and adoption [6]. With standard machine learning approaches, we use R-squared (R2) and accuracy as measures to assess their explanatory power. The results show our code quality scores can explain the number of OSS stars well. Our method-

2693-9177/23/$31.00 ©2023 IEEE
DOI 10.1109/QRS60937.2023.00054

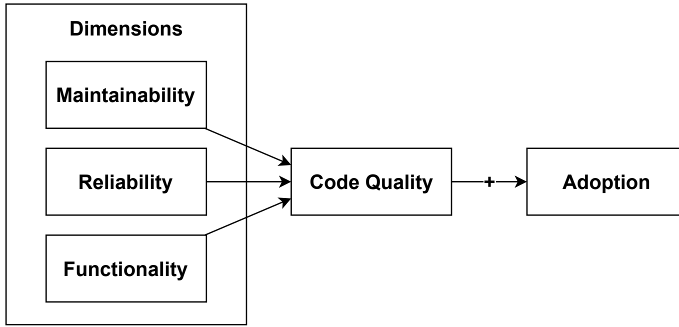

Figure 1: Multi-Dimensional Construct

ology can be applied to different target variables, providing a flexible strategy in various contexts.

This work has the following contributions. Prior literature has discussed diverse code quality metrics [7, 8, 9] without consistent metric evaluations. We extended them by dividing code quality metrics into two types and evaluated them with a novel distribution-based method. We conducted our empirical analysis on 36,460 GitHub OSS repositories. We used our evaluated scores to explain the OSS adoption [1], generating implications into how code quality may influence the OSS adoption. Our study advances the understanding of code quality with two different types of code quality metrics and contributes to better quality control standards and practices.

2. Literature Review

Open Source Software (OSS) uses a communal approach to software development, which significantly increases their code quality [10, 11] and fuels innovation [12, 13, 14]. Many firms and developers actively contribute to and utilize OSS [15]. These contributions serve to incrementally enhance the overall OSS code quality. The reuse of OSS has been widely adopted because many OSS have high code quality [16] which can effectively reduce the search costs for developers [17, 18]. Therefore, we use OSS as the benchmark to evaluate software code quality.

Many research studied performance evaluations for software [19]. Inappropriate performance measurements have been identified as a major cause of IT systems failing [2, 20]. As new technologies and techniques emerge, such as blockchain systems [21], and AI agent systems [22], more precise measurements of software code quality are needed [23]. Our approach sets itself apart from past studies by considering the distribution of high-quality software and delivering accurate scores for each software.

Code quality encompasses various dimensions [24]. The IEEE standard defines code quality as the collective features and characteristics of software that meet given needs [25]. Later on, user-friendliness and useful functionalities are included in the definition of code quality [1], echoing the three dimensions in the ISO/IEC 25010 standard [26]: maintainability, reliability, and functionality [27]. Similarly, other studies have similar dimensions: maintainability [28], readability [29], and functionality [30]. We base on the literature to define the construct and dimensions in Table I.

Code quality has metrics, including the size of components [8], code complexity [7, 9], and so on. However, most existing metric identifications have focused on monotonic metrics rather than non-monotonic metrics, because monotonic metrics have a consistent relationship with software code quality. Our paper considers both types and proposes a uniform solution for evaluating them.

Reflective measurements, such as the number of stars [6], can indicate the overall level of software code quality. Although OSS adoption activities are determined by many factors, such as commitment [32], transparency [33], and leader resources [34], OSS adoption decision can reflect good OSS code quality. Lee et al. [1] highlight the impact of code quality on user satisfaction and adoption. The OSS repositories that see the highest adoption rate are often those that maintain exceptional code quality. Therefore, we use GitHub stars as a reflective measure of code quality.

3. Methodologies

Our study employs the number of stars as a reflective measurement for identifying good-quality repositories. We divide code quality metrics into two distinct groups. We then analyze various code quality metric distributions and introduce a consistent distribution-based approach to evaluate all metrics within these categories.

We first map out the distribution of each metric in high-star OSS repositories and then score them according to their corresponding metric CDFs.

Table III presents two different types of metric distributions: monotonic and non-monotonic metrics. We fit exponential distributions to monotonic metrics, the probability distribution function (PDF) of which reads as:

$$f_1(x;c,\lambda) = \begin{cases} 0 & \text{if } x \le c \\ \lambda \exp\left[-\lambda(x-c)\right] & \text{if } x > c \end{cases} \tag{1}$$

where λ and c are the fitting parameters. The corresponding score function based on the CDF of Eq. (1) reads as

$$M_1(x;c,\lambda) = 100 \times \begin{cases} 1 & \text{if } x \le c \\ \exp\left[-\lambda(x-c)\right] & \text{if } x > c \end{cases} \tag{2}$$

The score falls into the range of $0 \sim 100$ and it peaks at c and decays exponentially for $x > c$.

The non-monotonic metrics follow an asymmetric Gaussian distribution (see the left of Fig. 2), the PDF of which reads as

$$f_2(x;\mu,\sigma_1,\sigma_2) = \begin{cases} \frac{1}{\sqrt{2\pi}} \frac{2}{\sigma_1+\sigma_2} \exp\left(-\frac{(x-\mu)^2}{2\sigma_1^2}\right) & \text{if } 0 \le x < \mu \\ \frac{1}{\sqrt{2\pi}} \frac{2}{\sigma_1+\sigma_2} \exp\left(-\frac{(x-\mu)^2}{2\sigma_2^2}\right) & \text{if } x \ge \mu \end{cases} \tag{3}$$

TABLE I: Construct Definition

Construct	Definition	Dimensions	Definition
Code Quality	The extent to which code is well-written and meets given needs. [1]	Maintainability	The code is easy to understand, enhance, or correct. [31]
		Reliability	The code is user-friendly and stable. [1]
		Functionality	The code has useful functions. [1]

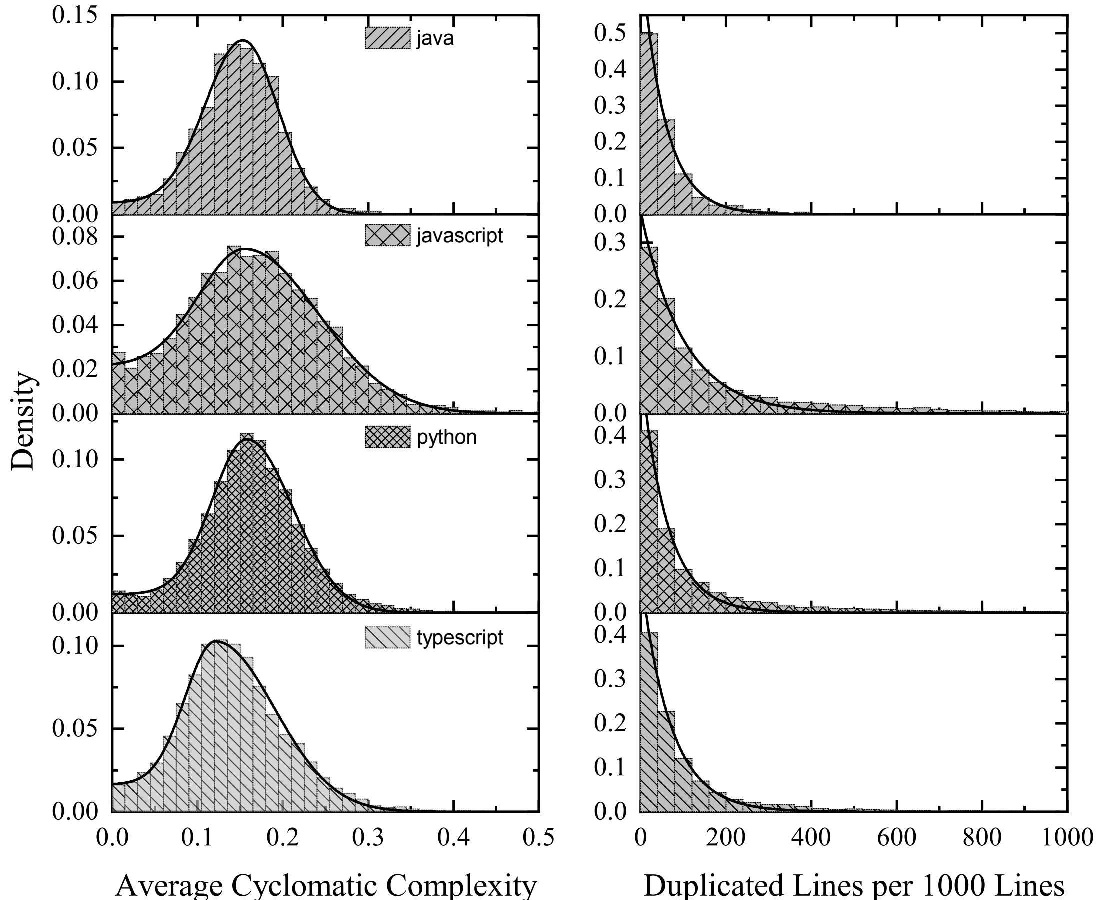

Figure 2: Examples of Non-Monotonic Metric Distribution and Monotonic Metric Distribution

where $\mu, c, \sigma_1, \sigma_2$ are fitting parameters representing the peak position, peak height on the right, and peak widths on each side, respectively. The corresponding score function is

$$M_2(x, \mu, \sigma_1, \sigma_2) = 100 \times \begin{cases} 1 - \mathrm{erf}\left(\frac{x-\mu}{\sigma_1\sqrt{2}}\right) & \text{if } 0 \leq x < \mu \\ 1 - \mathrm{erf}\left(\frac{x-\mu}{\sigma_2\sqrt{2}}\right) & \text{if } x \geq \mu \end{cases} \tag{4}$$

where the score falls into the range of $0 \sim 100$, peaks at μ, and decays according to the Z-score of the Gaussian function on each side.

To obtain an overall score, we assign weights to individual scores. The overall score for a given repository, denoted by k, can be computed as follows:

$$Q_k^{overall} = \sum_i \omega_i \cdot Q_{i,k}^{metric}, \text{subject to:} \sum_i \omega_i = 1. \tag{5}$$

The weights ω_i are derived from the importance values from supervised learning models for metric scores to a target variable such as repository stars.

4. Empirical Analysis

4.1 Data Sources

GitHub is the largest OSS management platform that has more than 39 million public repositories (As of June 2023). We selected a subset of repositories with Java, Python, JavaScript, and TypeScript as the main programming languages and sorted them by the number of stars. We collected code from the top $\sim 20,000$ repositories for each programming language. The number of GitHub stars is a measure of OSS adoption [6]. We removed non-engineering repositories by pattern matching, such as a guide for Java interviews in JavaGuide.

We used code scanners to obtain metrics. Scripting language

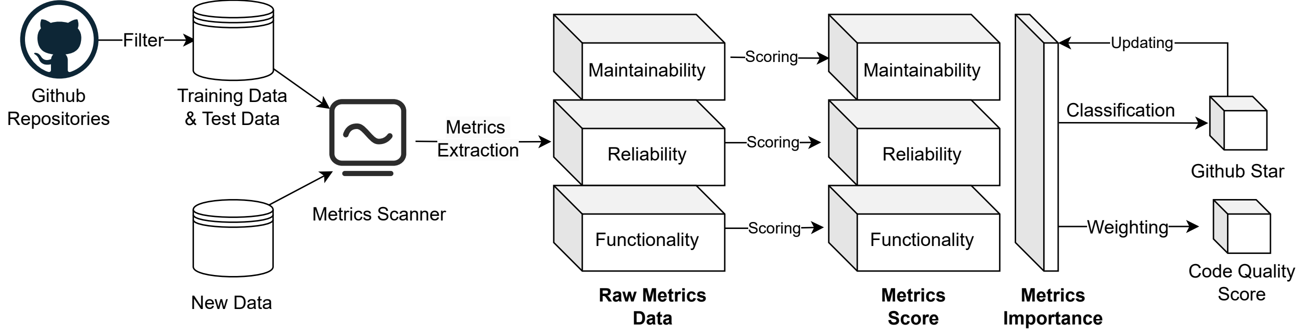

Figure 3: Workflow for Code Quality Scoring with GitHub Stars as the Target Variable

TABLE II: Statistical Summary

Programming Language	Max Number of Stars	Min Number of Stars	Number of Filtered Repositories
Java	50k	100	1,645
Python	228k	260	16,096
JavaScript	107k	270	7,722
TypeScript	202k	60	10,997

repositories (Python, Javascript, TypeScript) can be directly imported, while non-scripting Java repositories need to be compiled first. Compiling Java repositories is challenging due to their different JDK, maven, or Gradle versions. Therefore, we only chose repositories with GitHub releases for compilation, which led to 36,460 repositories and over 600 million lines of code. Table II reports the statistics of the cloned repositories. The minimum number of repository stars is above 50, which demonstrates good code quality compared to overall OSS repositories.

4.2 Metrics Overview

We used SonarQube and CK to extract metrics from OSS repositories. For Java repositories, we generated over 100 metrics and selected 20 based on the ISO/IEC 25010 international standard [26]. For scripting language repositories, we only extracted 12 metrics. Table III shows the 20 metrics with their corresponding ISO/IEC 25010 characteristics.

We normalized metrics to ensure score fairness. Cyclomatic complexity, cognitive complexity, code smells, line to cover, and violations-related metrics are normalized by non-comment lines of code, duplicated lines are normalized by lines of code, and comment lines are normalized by the sum of non-comment lines and comment lines, to account for repository size. File complexity and duplicated files are normalized by the number of files, and duplicated blocks are normalized by the number of statements to adjust for differences across repositories. This normalization process results in a more unbiased score for the metrics across different OSS.

4.3 Importance Weights

We use standard machine-learning approaches to derive weights for different metric scores and calculate a repository's overall code quality score. Our model can explain OSS adoption (Github stars) using evaluated scores.

Figure 3 illustrates the entire process from data collection to final scores. We use custom data filters to ensure genuine engineering repositories are retained. We extract code quality metrics using a metric scanner and generate metric scores using the distribution-based method in Section 3, with each programming language having its distribution for each metric. We implement a Gradient Boosting Classifier (GBC) model with 0-1 labels as dependent variables based on the number of GitHub stars. We label the top and bottom quintiles (20%) of the OSS repository stars as 1 and 0, respectively. The model generates importance values as weights for each metric. Finally, we obtain a weighted average code quality score according to Eq. (5).

Algorithm 1: GBC in Our Context

Input: Training dataset $\mathcal{D} = \{(\mathbf{m}_i, c_i)\}_{i=1}^{N}$, number of iterations T
Output: Ensemble model $F(\mathbf{m})$
Initialize model $F_0(\mathbf{m}) = 0$;
for $t = 1$ to T **do**
  Compute the negative gradient:
  $r_{it} = -\frac{\partial L(c_i, F(\mathbf{m}_i))}{\partial F(\mathbf{m}_i)}\Big|_{F(\mathbf{m})=F_{t-1}(\mathbf{m})}$;
  Fit a base learner $h_t(\mathbf{m})$ to the negative gradient:
  $h_t(\mathbf{m}) = \arg\min_h \sum_{i=1}^{N} L(c_i, F_{t-1}(\mathbf{m}_i) + h(\mathbf{m}_i))$;
  Update the ensemble model: $F_t(\mathbf{m}) = F_{t-1}(\mathbf{m}) + \eta h_t(\mathbf{m})$, where η is the learning rate;
end

The GBC algorithm is presented in Algorithm 1, where each data point contains a metric score $\mathbf{m}_i$ and its corresponding classification c_i according to its GitHub star. We divide the

TABLE III: Definition of 20 Code Quality Metrics

Dimension	Metric	Definition
Maintainability	Cyclomatic Complexity[a]	Number of independent paths through code.
	File Complexity[b]	Cyclomatic complexity averaged by files.
	Cognitive Complexity[a]	Combination of cyclomatic complexity and human assessment.
	Code Smells[a]	Number of code smell issues.
	Coupling Between Objects	Number of classes coupled to a particular class.
	Fan-in	Number of input dependencies a class has.
	Fan-out	Number of output dependencies a class has.
	Depth Inheritance Tree	Number of ”fathers” a class has.
	Number of Children	Number of immediate subclasses that a particular class has.
	Lack of Cohesion of Methods	Degree to which class methods are coupled.
	Tight Class Cohesion	Ratio of the number of pairs of directly related methods in a class to the maximum number of possible methods in the class.
	Loose Class Cohesion	Ratio of the number of directly or indirectly related method pairs in a class to the maximum number of possible method pairs.
Reliability	Total Violations[a]	Number of issues including all severity levels.
	Critical Violations[a]	Number of issues of the critical severity.
	Info Violations[a]	Number of issues of the info severity.
Functionality	Line to Cover[a]	Lines to be covered by unit tests.
	Comment Lines[b]	Number of comment lines.
	Duplicated Blocks[e]	Number of duplicated blocks of line.
	Duplicated Files[d]	Number of files involved in duplicated blocks.
	Duplicated Lines[c]	Number of lines involved in duplicated blocks.

[a] Normalized by Non-comment Line of Codes.
[b] Normalized by Sum of Non-comment Line of Codes and Comment Lines.
[c] Normalized by Line of Codes.
[d] Normalized by Number of Files.
[e] Normalized by Number of Statements.

whole dataset into a training ($\mathcal{D}$) and a validation set by a ratio of 4:1. The GBC algorithm works with an ensemble model $F_0(\mathbf{m})$ and we fine-tune it by fitting base learners $h_t(\mathbf{m})$ to the loss function's negative gradient. The learning rate η determines the base learners' contribution, resulting in the final ensemble model $F(\mathbf{m})$ providing the aggregate prediction.

5. Results

5.1 Metric Distributions

We conducted our empirical analysis on 36,460 GitHub OSS repositories. The selection of high-star repositories provides a more critical evaluation, because they generally have better performance, resulting in sharper distributions.

Table IV and Table V present the fitted parameters for the asymmetric Gaussian [Eq. (3)] and Exponential [Eq. (1)] distributions, respectively. Java repositories have 8 more maintainability metrics describing cohesion and coupling in the codes, which are absent for other programming languages due to a lack of proper metric scanners.

Monotonic metrics, such as 'Code Smells', exhibit an exponential distribution pattern, as represented in Fig. 2 and Table V. This distribution aligns with our understanding that superior code quality is associated with fewer bugs, verifying the effectiveness of our method. Furthermore, the threshold parameter c reflects the tolerance value for full scores. In the probability density function (Eq. (1)) except 'Code Smells', 'Depth Inheritance Tree', and 'Total Violations' where c approximates 1.

The fitted exponential decay parameter, λ, reflects the sensitivity of metrics to scores. Particularly, a $\lambda \lesssim 1$ is observed for metrics such as 'File Complexity', 'Depth Inheritance Tree', 'Number of Children', 'Duplicated Blocks', and 'Duplicated Files', which implies a low sensitivity to metric variations of the order of 1. Conversely, the λ value for total violation is high, which reflects the high sensitivity of the number of violations.

Non-monotonic metrics, such as the 'Cyclomatic Complexity', follow an asymmetric Gaussian distribution. According to Eq. (4), repositories with metric values close to the Gaussian center get higher scores since they fall into the range where high-quality OSS are mostly located. In Table IV, the Gaussian centers μ are large ($\gg 1$) for the metrics of 'Cyclomatic Complexity', 'Cognitive Complexity', and 'Comment Lines' in most cases except for the 'Comment Lines' of the Javascript and Typescript languages. The latter two distributions are almost monotonic ($\mu = 0$), potentially because these two

TABLE IV: Parameters of the Fitted Asymmetric Gaussian Distributions (μ, σ_1, σ_2)

Metric	Java(μ,σ_1,σ_2)	JavaScript(μ,σ_1,σ_2)	Python(μ,σ_1,σ_2)	TypeScript(μ,σ_1,σ_2)
Cyclomatic Complexity	(155.228,50.947,40.902)	(166.692,88.415,78.289)	(162.321,53.497,52.789)	(127.273,51.616,66.733)
Cognitive Complexity	(50.870,40.120,75.664)	(33.238,32.586,121.541)	(170.042,33.546,0.000)	(29.619,22.964,81.617)
Comment Lines	(15.841,11.451,137.269)	(0.007,6.575,96.312)	(91.730,64.805,148.192)	(0.002,9.300,72.443)
Fan-in	(1.101,0.463,1.217)	/	/	/
Fan-out	(5.181,2.043,4.639)	/	/	/
Loose Class Cohesion	(0.329,0.149,0.176)	/	/	/
Tight Class Cohesion	(0.228,0.100,0.128)	/	/	/
Coupling Between Objects	(7.055,2.580,5.086)	/	/	/

TABLE V: Parameters of the Fitted Exponential Distributions (c, λ)

Metric	Java(c,λ)	JavaScript(c,λ)	Python(c,λ)	TypeScript(c,λ)
File Complexity	(0,0.485)	(0,0.884)	(0,0.917)	(0,0.492)
Code Smells	(1.123,50.731)	(0.036,60.260)	(0.004,37.177)	(0.017,16.530)
Depth Inheritance Tree	(1.003,0.502)	/	/	/
Number of Children	(0.002,0.137)	/	/	/
Lack of Cohesion of Methods	(0.053,80.004)	/	/	/
Total Violations	(1.160,54.376)	(0.054,63.313)	(0.004,387.551177)	(0.021,18.168)
Critical Violations	(0.019,9.872)	(0.020,48.811)	(0.007,9.443)	(0.005,5.497)
Info Violations	(0.019,1.934)	(0.001,1.436)	(0.002,1.401)	(0.003,1.535)
Line to Cover	(0,0.000)	(0,0.000)	(0,0.000)	(0,0.000)
Duplicated Blocks	(0,0.015)	(0.001,0.021)	(0,0.010)	(0,0.021)
Duplicated Files	(0.003,0.135)	(0.001,0.203)	(0,0.222)	(0,0.116)
Duplicated Lines	(0.439,63.284)	(0.145,163.258)	(0.081,124.342)	(0.085, 102.796)

languages are generally easy to understand and do not require additional command lines.

The fitted widths $\sigma_{1,2}$ are large and have asymmetric sensitivity; i.e. relatively long tails are observed on the right of the asymmetric Gaussian distributions. For "command line" in Python, increasing command lines before the center point has high sensitivity, while it becomes less sensitive after the center point.

After obtaining metric distributions, we score the metrics of each OSS repository based on their respective locations in the distributions.

5.2 Importance Weights

Table VI shows the feature importance values from the GBC model in Section 4, which we use as metric score weights in Eq. (5) within the three dimensions: maintainability, reliability, and functionality. The relative importance values are listed in Table VI. We normalized the importance values for each dimension to get relative weights within dimensions.

In the maintainability dimension, 'File Complexity' has the largest weight across four programming languages, followed by 'Cognitive Complexity' 'Cyclomatic Complexity', and 'Code Smells'. These metrics contribute more to the maintainability scores. For Java repositories, all the coupling and cohesion metrics show similar contributions $\lesssim 0.1$, reflecting their weak contribution to OSS adoption.

In the reliability dimension, 'Total Violations' contributes mostly to Java, while 'Critical Violations' contributes mostly to the other three languages, which suggests varying priorities of solving violations for different languages.

In the functionality dimension, the 'Comment Lines' metric contributes more to Java, potentially because Java is less intuitive to understand, which requires code comments for better understanding. The 'Comment Lines' metric also contributes significantly to the other three scripting languages. We note that zero 'Line to Cover' metric values were obtained in our raw data, either caused by problems in obtaining this metric or because codes in OSS repositories are rarely tested. This gap can be closed when applying our methodology in specific companies where values of 'Line to Cover' are obtained for their close-source repositories.

5.3 Software Adoption

We present the overall scores of included OSS repositories in Fig. 4 and assess the explanatory power of our metric scores on the OSS stars using Table VII. We observe that Java code metric scores show higher explanatory power for the OSS repository's stars compared to the other languages, which suggests that code quality can better determine the success of

TABLE VI: Importance Values for Metric Scores

Dimension	Metric	Importance			
		Java	JavaScript	Python	TypeScript
Maintainability	Cyclomatic Complexity	0.110 (0.083)	0.190 (0.082)	0.250 (0.120)	0.223 (0.081)
	File Complexity	0.220 (0.165)	0.396 (0.171)	0.449 (0.215)	0.402 (0.146)
	Cognitive Complexity	0.086 (0.065)	0.289 (0.125)	0.119 (0.057)	0.215 (0.078)
	Code Smells	0.066 (0.049)	0.125 (0.054)	0.182 (0.087)	0.160 (0.058)
	Coupling Between Objects	0.096 (0.072)	/	/	/
	Fan-in	0.108 (0.081)	/	/	/
	Fan-out	0.057 (0.043)	/	/	/
	Depth Inheritance Tree	0.075 (0.057)	/	/	/
	Number of Children	0.026 (0.020)	/	/	/
	Lack of Cohesion of Methods	0.078 (0.058)	/	/	/
	Tight Class Cohesion	0.010 (0.008)	/	/	/
	Loose Class Cohesion	0.068 (0.051)	/	/	/
	Sum	1 (0.752)	1 (0.432)	1 (0.479)	1 (0.363)
Reliability	Total Violations	0.474 (0.056)	0.288 (0.070)	0.293 (0.068)	0.228 (0.065)
	Critical Violations	0.272 (0.032)	0.420 (0.102)	0.410 (0.095)	0.414(0.118)
	Info Violations	0.254 (0.030)	0.292 (0.071)	0.297 (0.069)	0.358 (0.102)
	Sum	1 (0.118)	1 (0.243)	1 (0.232)	1 (0.285)
Functionality	Line to Cover	0.000 (0.000)	0.000 (0.000)	0.000 (0.000)	0.000 (0.000)
	Comment Lines	0.454 (0.059)	0.317 (0.103)	0.370 (0.107)	0.318 (0.112)
	Duplicated Blocks	0.162 (0.021)	0.286 (0.093)	0.197 (0.057)	0.148 (0.052)
	Duplicated Files	0.190 (0.025)	0.120 (0.039)	0.166 (0.048)	0.179 (0.063)
	Duplicated Lines	0.194 (0.025)	0.277 (0.090)	0.267 (0.077)	0.355 (0.125)
	Sum	1 (0.130)	1 (0.325)	1 (0.289)	1 (0.352)

The parenthesis values are original importance values, while the values outside parenthesis are normalized in the dimension level.

TABLE VII: Metric Scores Explanatory Power

Language	Java	JavaScript	Python	TypeScript
Accuracy	0.947	0.826	0.808	0.817
Precision	0.971	0.838	0.831	0.834
Recall	0.917	0.803	0.771	0.784
F1	0.943	0.820	0.800	0.808
AUC_ROC	0.946	0.826	0.815	0.817
R2	0.787	0.274	0.186	0.247

Java-based OSS repositories in terms of stars received, which may be attributed to the greater availability of metrics for Java or the nature of repositories developed using Java for large-scale platforms and systems.

In contrast, JavaScript, Python, and TypeScript exhibit relatively lower explanatory power of metric scores, indicating their code quality might be less critical in determining their OSS adoption, possibly because of their primary use in data analytics or other domains where their adoption is less influenced by code quality.

6. Conclusion

Our research focuses on code quality with three dimensions: maintainability, reliability, and functionality. We evaluate metrics based on their distributions. Our study advances the understanding of code quality and contributes to better quality control standards and practices, ultimately supporting the success and sustainability of software.

Although our study provides valuable implications, it has some limitations that need to be acknowledged. We have not yet systematically validated the effectiveness of the method. Moving forward, it would be beneficial to incorporate validation techniques, such as sensitivity tests, to ensure the accuracy and reliability of the distribution fitting. Additionally, the parameters of the fitted distribution are sensitive to data distribution, making it necessary to incorporate more data for determining them.

Acknowledgment

Y. Xia is partly supported by the ”Pioneering Innovator” award from the Guangzhou Tianhe District government. Z. Li is partly supported by the Guangdong Basic and Applied Basic Research Foundation (2021A1515012039). We would like to acknowledge useful discussions and support from Mianmian Zhang and other colleagues at the HSBC Lab.

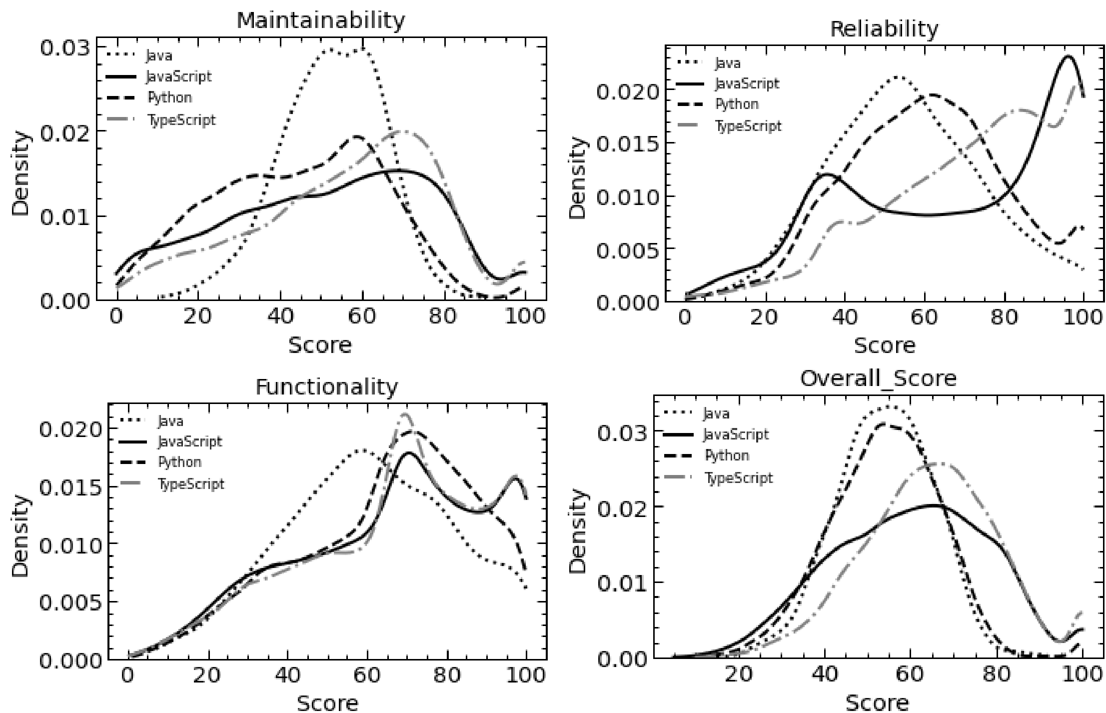

Figure 4: Overall Scores for Four Languages

REFERENCES

[1] Sang-Yong Tom Lee, Hee-Woong Kim, and Sumeet Gupta. Measuring open source software success. Omega, 37(2):426–438, 2009.

[2] Sunder Kekre, Mayuram S Krishnan, and Kannan Srinivasan. Drivers of customer satisfaction for software products: implications for design and service support. Management Science, 41(9):1456–1470, 1995.

[3] Kevin Crowston, Hala Annabi, and James Howison. Defining open source software project success. In 2003 International Conference on Information Systems (ICIS) Proceedings, 2003.

[4] David I Levine and Michael W Toffel. Quality management and job quality: How the ISO 9001 standard for quality management systems affects employees and employers. Management Science, 56(6):978–996, 2010.

[5] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: a case study at google. In Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, pages 181–190, Gothenburg Sweden, 2018. ACM.

[6] Poonacha K Medappa and Shirish C Srivastava. Does superposition influence the success of FLOSS projects? An examination of open-source software development by organizations and individuals. Information Systems Research, 30(3):764–786, 2019.

[7] Thomas J McCabe. A complexity measure. IEEE Transactions on software Engineering, 0(4):308–320, 1976.

[8] Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, and Georgios L. Bleris. Code quality analysis in open source software development. Information Systems Journal, 12(1):43–60, 2002.

[9] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Transactions on Software Engineering, 37(6):772–787, 2010.

[10] Jan Ljungberg. Open source movements as a model for organising. European Journal of Information Systems, 9(4):208–216, 2000.

[11] Georg Von Krogh and Eric Von Hippel. The promise of research on open source software. Management Science, 52(7):975–983, 2006.

[12] Sheen S Levine and Michael J Prietula. Open collaboration for innovation: Principles and performance. Organization Science, 25(5):1414–1433, 2014.

[13] Padmal Vitharana, Julie King, and Helena Shih Chapman. Impact of internal open source development on reuse: Participatory reuse in action. Journal of Management Information Systems, 27(2):277–304, 2010.

[14] Vineet Kumar, Brett R Gordon, and Kannan Srinivasan. Competitive strategy for open source software. Marketing Science, 30(6):1066–1078, 2011.

[15] Amit Mehra, Rajiv Dewan, and Marshall Freimer. Firms as incubators of open-source software. Information Systems Research, 22(1):22–38, 2011.

[16] Donald E Harter, Mayuram S Krishnan, and Sandra A

Slaughter. Effects of process maturity on quality, cycle time, and effort in software product development. Management Science, 46(4):451–466, 2000.

[17] Stefan Haefliger, Georg Von Krogh, and Sebastian Spaeth. Code reuse in open source software. Management Science, 54(1):180–193, 2008.

[18] Manuel Sojer and Joachim Henkel. Code reuse in open source software development: Quantitative evidence, drivers, and impediments. Journal of the Association for Information Systems, 11(12):868–901, 2010.

[19] Jehad Al Dallal and Anas Abdin. Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. IEEE Transactions on Software Engineering, 44(1):44–69, 2017.

[20] David Fitoussi and Vijay Gurbaxani. IT outsourcing contracts and performance measurement. Information Systems Research, 23(1):129–143, 2012.

[21] Si Yuan Jin and Yong Xia. CEV Framework: A central bank digital currency evaluation and verification framework with a focus on consensus algorithms and operating architectures. IEEE Access, 10:63698–63714, 2022.

[22] Stephan Diederich, Alfred Benedikt Brendel, Stefan Morana, and Lutz Kolbe. On the design of and interaction with conversational agents: An organizing and assessing review of human-computer interaction research. Journal of the Association for Information Systems, 23(1):96–138, 2022.

[23] Shannon W Anderson and Amanda Kimball. Evidence for the feedback role of performance measurement systems. Management Science, 65(9):4385–4406, 2019.

[24] Greta L Polites, Nicholas Roberts, and Jason Thatcher. Conceptualizing models using multidimensional constructs: a review and guidelines for their use. European Journal of Information Systems, 21:22–48, 2012.

[25] Ronan Fitzpatrick. Software quality: definitions and strategic issues. Report, 1, 1996.

[26] Matej Klima, Miroslav Bures, Karel Frajtak, Vaclav Rechtberger, Michal Trnka, Xavier Bellekens, Tomas Cerny, and Bestoun S Ahmed. Selected code-quality characteristics and metrics for internet of things systems. IEEE Access, 10:46144–46161, 2022.

[27] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. Test code quality and its relation to issue handling performance. IEEE Transactions on Software Engineering, 40(11):1100–1125, 2014.

[28] Simona Motogna, Andreea Vescan, and Camelia Şerban. Empirical investigation in embedded systems: Quality attributes in general, maintainability in particular. Journal of Systems and Software, 201:111678, 2023.

[29] Ángel González-Prieto, Jorge Perez, Jessica Diaz, and Daniel López-Fernández. Reliability in software engineering qualitative research through inter-coder agreement. Journal of Systems and Software, 202:111707, 2023.

[30] Qi Shen, Shijun Wu, Yanzhen Zou, Zixiao Zhu, and Bing Xie. From api to nli: A new interface for library reuse. Journal of Systems and Software, 169:110728, 2020.

[31] Ignatios Deligiannis, Martin Shepperd, Manos Roumeliotis, and Ioannis Stamelos. An empirical investigation of an object-oriented design heuristic for maintainability. Journal of Systems and Software, 65(2):127–139, 2003.

[32] Likoebe M Maruping, Sherae L Daniel, and Marcelo Cataldo. Developer centrality and the impact of value congruence and incongruence on commitment and code contribution activity in open source software communities. MIS Quarterly, 43(3):951–976, 2019.

[33] Maha Shaikh and Emmanuelle Vaast. Folding and unfolding: Balancing openness and transparency in open source communities. Information Systems Research, 27(4):813–833, 2016.

[34] John Qi Dong and Sebastian Johannes Götz. Project leaders as boundary spanners in open source software development: A resource dependence perspective. Information Systems Journal, 31(5):672–694, 2021.