Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

# When do Software Complexity Metrics Mean Nothing? – When Examined out of Context

Joseph (Yossi) Gil[a] Gal Lalouche[a]

a. Department of Computer Science, Technion—Israel Institute of Technology, Technion City, Haifa 32000, Israel

Abstract This paper places its attention on a familiar phenomena: that code metrics such as lines of code are extremely context dependent and their distribution differs from project to project. We apply visual inspection, as well as statistical reasoning and testing, to show that such metric values are so sensitive to context, that their measurement in one project offers little prediction regarding their measurement in another project.

On the positive side, we show that context bias can be neutralized, at least for the majority of metrics that we considered, by what we call ***Log Normal Standardization*** (LNS). Concretely, the LNS transformation is obtained by shifting (by subtracting the mean) and scaling (by dividing by the standard deviation) of the log of a metric value.

Thus, we conclude that the LNS-transformed-, are to be preferred over the plain-, values of metrics, especially in comparing modules from different projects. Conversely, the LNS-transformation suggests that the "context bias" of a software project with respect to a specific metric can be summarized with two numbers: the mean of the logarithm of the metric value, and its standard deviation.

## 1 Introduction

*Internal software metrics* or "code metrics", such as *Lines of Code* (henceforth, **LOC**), are concrete, well-defined, and easy to compute. In contrast, *external software metrics*—quality traits such as *maintainability*, *correctness*, and *robustness*, are more illusive, yet much more important [30, pp. 3–10]. A holy grail of the software engineering community is finding ties between the readily available and the useful, e.g., computing a predictor of maintainability from **LOC**.

This work does *not* make an attempt to find this legendary grail. Our less ambitious objective is to show that the search may have been pursuing a somewhat false track.

Allusion to the famous "When do changes induce fixes?" [38] is intentional.

Our line of reasoning is as follows: to correlate internal metrics with external metrics (or with anything else for that matter), researchers cannot use only one software project, because of (at least) two reasons:

- Results obtained in a single project must be substantiated against others.
- Rarely does a single project provides enough data for useful statistical inference.

However, we show that software code metrics measured in one such project are seldom relevant as-is to other projects: Mixing measurements drawn from different contexts may confound statistical results.

This work is thus dedicated to the study of *context bias*—the bias of metric values due to the project in which these were measured—and finding means for eliminating this bias. To this end, we set granularity at the software module level. Specifically, the study focus is on JAVA [4] software, where modules are `.java` files, each of which contains at least one class, and, typically, no more than that.

## 1.1 Background: Code Metrics as Indicators of Software Defects

There is a long history of trying to use code metrics to predict software defects. Recent research along these lines includes that of Kim, Whitehead, and Zhang [27] who applied machine-learning classification techniques to detect software defects. They trained separate *Support Vector Machines* (SVM) on the file-change histories of a dozen separate open-source projects and achieved a mean prediction accuracy of $78\%$. Hata, Mizuno and Kikuno [23] applied a Bayesian spam-filtering approach to classifying program modules as either fault prone or not and achieved $F_1$ rates of $71\%$ and $58\%$ in their two experiments. A comparison of machine learning techniques can be found in the work of Challagulla, Bastani, Yen, and Paul [13].

Unfortunately, the fruits of this research direction largely contradict the notion that external qualities are related to internal code metrics: For example, Moser, Pedrycz, and Succi [31] used two different kinds of metrics to analyze JAVA sources from the Eclipse project by applying logistic regression, *Naïve Bayes Classifiers* (NBC), and *classifying decision trees*. They found more than $75\%$ classification accuracy and concluded that *software engineering process* metrics predict defects much better than code metrics for the Eclipse project. Recently, Shivanji, Whitehead, Akella, and Kim [37] described their efforts to locate a small set of features out of tens of thousands that would strongly predict software defects in an SVM or an NBC model; NBC generally won out. More importantly, they found that their "top 100" lists of statistically significant features did *not* include any conventional code-complexity metrics.

Still, an extensive statistical benchmark of a large variety of different bug-finding metrics, due to D'Ambros, Lanza, and Robbes [14], did manage to correlate internal metrics with bug prediction with some statistical significance but had substantial problems generalizing their metrics to predict bugs out-of-sample. This previous work left bug prediction as open a problem as ever.

## 1.2 Possible Causes of Poor Results

A possible explanation for this failure could be that code metrics tend to exhibit *context bias*, sometimes phrased as the cliche: "every software project is different". In "*How does Context Affect the Distribution of Software Maintainability Metrics?*", Zhang, Mockus, Zou, Khomh, and Hassan [40] ask whether code metrics are affected by context of the containing software artifact, including diverse factors such as programming language, age, and lifespan. The thrust of the work of Zhang et al. is that the majority of metrics are correlated with what the authors call "contextual factors".

Applying a suite of 20 file-level code metrics to dataset of 26 open-source JAVA software artifacts with a similar profile, we find that context bias is the norm, rather than the exception. More importantly, we show that this bias can be (almost) eliminated by a process which we will call "*rank-transformation*". (As explained below in Sect. 3, our work can be seen as complementing that of Zhang et al.)

Further, we show that context bias can be neutralized, by the LNS (***L****og* ***N****ormal* ***S****tandardization* transformation, defined by shifting (by subtracting the mean) and scaling (by dividing by the standard deviation) of the $\log$ of a metric value.

The implication is not only that LNS-transformed-, are to be preferred over the plain-, values of metrics, especially in comparing modules from different projects. The discovery of the LNS-transformation suggests that the "context bias" of a software project with respect to a specific metric can be summarized by two numbers: the mean of the logarithm of the metric value, and its standard deviation.

Our thesis is then that if a tie between internal and external metrics is ever found, it is unlikely that it would solely employ the traditional, "context independent" manner of computing metrics. We support this claim by bringing statistical evidence that internal metric values of a software module are more likely to be meaningful when "normalized" against their context, i.e., the containing global software artifact from which this module was drawn.

Thus, we demonstrate that the plain positive integer value of (say) **LOC** of a given software module $m$ is inferior to a transformed value of **LOC**, where the transformation takes into account the distribution of the **LOC** metric in other modules in the software artifact from which $m$ was drawn, when comparing between metric values of different artifacts in different projects. With this conclusion, it remains to redo, e.g., the work of Shivanji et al. [37] to check whether normalized metric values are better defect predictors than their non-normalized versions. We leave such a study for continued research.

The direction taken by this study is different than some of the other work on metrics. Consider, e.g., the definition of QMOOD [8,35], in which the set of metrics are computed for the *project as a whole*. Unlike our work, individual module-level metrics and their distributions play no role in the study of *project level metrics*, whose purpose is aiding software managers evaluate *overall* project quality. Module-level metric values are used in project level metrics only in an aggregated, e.g., sum of average, form.

Concrete contributions made by the analysis of this empirical data include:

1. Introduction of the similarity measure.
2. Demonstration that context bias exists for all code metrics in our suite.
3. Definition of rank-normalization and showing that it eliminates context bias in the majority of metrics.
4. Showing that context bias is largely about shifting (with the negative of the mean) and scaling (by the standard deviation), though not of the metric itself, but its logarithm.
5. Finding that the common theme of those metrics that we could normalize is a small variety of metric values, as measured by its information-theoretic entropy.

Readers may also find the method that we use for statistical analysis to be of value: Recall that a major limitation of any statistical test is that it can be used for rejecting a hypothesis (the null-hypothesis in the respective lingo) but it never yields a result which *confirms* a hypothesis. In this work, we overcome this predicament by applying a test not to just one data instance but rather to an ensemble of these. Suppose that the test fails to reject the null hypothesis in the majority of the ensemble. Then, we may be convinced that the null hypothesis is true. Such inference requires caution, e.g., it is required that each of the instances is sufficiently large that the null hypothesis is unlikely to be rejected due to statistical noise.

**Outline.** The remainder of this document is organized as follows: Destined for readers interested in furthering their understanding of the background of this work Sect. 3 is concerned with delicate issues of statistical inference raised in comparing the current work with that of Zhang et al. The body of the paper begins with Sect. 4 which describes the dataset and the twenty-six projects constituting it. Sect. 5 enumerates the code metrics employed in this study. Some visual evidence to the context bias of metrics is presented in Sect. 6. In Sect. 7, the main section of this article, we apply statistical tests and reasoning to substantiate the claims made above. This section also presents a novel technique for statistical analysis, the *similarity measure*, for estimating the number of distinct distributions that a metric follows.

Threats to the validity of our results are discussed in Sect. 8. Sect. 9 concludes.

## 2 Related Work

This paper is concerned with the question of extrapolation from one software project to another. As might be expected, this question intrigued many others in the research community. This section visits a number (but naturally not all) of research works revolving around this topic.

Alves, Ypma and Visser, in a publication entitled "*Deriving metric thresholds from benchmark data*" [7], propose thresholds to metrics; values that exceed these thresholds trigger an alarm for more careful, manual, inspection. The authors acknowledge that the distributions of the same metric in distinct project might be vastly different, and meet this challenge with a sophisticated transformation of the numerical values. However, unlike the word presented here, the validity of this transformation is demonstrated merely by visual inspection of plots (which might be misleading as can be seen below), instead of rigorous statistical tests.

Marinescu [28] take another perspective at setting threshold values, by correlating these with human judgment of design flaws such as "God Class", "Feature Envy" and "Shotgun Surgery". The methodology involved relies also on examining what might be called survival rate (between consecutive versions of the project), in attempt to asses the quality of judgment. Unfortunately, this work fails to recognize that the inferred thresholds might be distinct in different projects; in particular, this study revolves around a single project.

Another work that recognizes the need for setting threshold values is that of Capiluppi, Faria and Ramil [11], where thresholds are based on rank. The use of rank is justified (rather implicitly) by the skewness of the distribution of metrics. This work however is concerned solely with one project.

A subsequent publication entitled "*A model to predict anti-regressive effort in Open Source Software*" by Capiluppi and Ramil [12] pays tribute to the stark variations between projects on which we dwell: The said work proposes a model for inferring anti-regressive work in the course of evolution of projects. The Capiluppi and Ramil "MGM" model uses intra-project rank of metric values as the main predictor. Further, it is implicitly argued that project are idiosyncratic in that the best predicting metric is project dependent. (We discuss the idiosyncraticity of projects in detail below in Sect. 3.)

Blondeau et al. " [10] take the following approach to variations between projects: A relatively large number of metrics is subjected into a ***P****rinciple* ***C****omponent* ***A****nalysis* (PCA). PCA makes it possible to identify outliers and for finding an "adequate" way of weighing the projects. However, other than acknowledgment of this variance, their work is not relevant to ours for two reasons. Firstly, the metrics of Blondeau et al. are product-, not code-, metrics. Secondly, their metrics are not of individual modules, but of the entire project.

Nagappan, Ball and Zeller [32] used PCA to predict faults. The authors note that there is no subset of metrics that can be applied to all projects. Further, when trying to extrapolate PCA predictions from one project to the next, they had "mixed" results, admitting such extrapolation makes sense only for "similar" projects, but leaving open the question of how this similarity might be measured.

Later [41], a larger scale experiment on the use of cross-project data for predicting post-release defects suggested means for estimating such similarity. As before, one conclusion of the experiment was that raw values of code metrics are poor predictors in the cross-project setting. (The current research, in highlighting how the distributions of such metrics vary between projects, might be viewed as an explanation.) Similarity between projects could only be detected when using "compound" metrics. However, even when using these new set of metrics, similarity was found only in less than 4% of the cases.

## 3 The Idiosyncratic Context Presumption

The only previous publication that takes interest in the impact of "context" on metrics, is that of Zhang, Mockus, Zou, Khomh, and Hassan [40]. The authors report on a large scale empirical study, involving 320 open source software systems, whose aim was "*to understand if the distributions of metrics do, in fact, vary with contexts.*". Towards this end, the study first classifies software systems according to seven "context factors": application domain, programming language, age, lifespan, number of changes, and number of downloads, and then employs statistical analysis to evaluate the impact of these factors on the distribution of 39 code metrics.

There are several differences in terminology between this work and that of Zhang et al.: First, the term "code metric" is used by Zhang at al. to refer to three kinds of metrics: **(i)** project level metrics, such as "total number of files" which assign a single number to the entire project; **(ii)** method level metrics, such as "number of parameters", by which each method in each class is ranked; and **(iii)** class level metrics, such as "response for a class", applicable to each class in the project. Our study is concerned solely with class metrics, and the term "metric" is henceforth restricted to class level metrics.

Second, we use the term "context" or "project's context" here to include individual and idiosyncratic properties of the project. These individual properties are not necessarily related to the context factors studied by Zhang et alii.

We argue that the Zhang et al.'s study tacitly makes the assumption that distributions of the same metric in distinct projects should be similar, all other factors being equal.

To see this, we follow the same analysis of Zhang et al.'s study, but this time employing a nonsensical context factor:

**Definition.** *Red herring context factor. The* red herring context factor *of a software system named $n$ is an integer $h \in \{0, 1, 2\}$, computed by $h = s \bmod 3$, where $s$ is the sum of the ASCII values of the characters comprising $n$.*

Specifically, as in Zhang et al.'s study, we use the $h$ value to partition our set of software projects into three groups, and then we employ the Kruskal-Wallis statistical test, for each metric $\mu$ drawn from our suite of 20 metrics, to check whether the distributions of $\mu$ in the three groups are distinct. In accordance with the original empirical study, the input to the test are three distributions: $D_0(\mu)$, $D_1(\mu)$, and $D_2(\mu)$, where $D_i(\mu)$ is simply the union of the $\mu$ values in all projects whose $h$ value is $i$.

The largest $p$ value returned by the test is $0.0005$[1]. In fact, in all but three metrics, the $p$ value returned was lower than the smallest non-zero positive number represented by the system. Following the Zhang et al.'s line of reasoning leads to conclusion that the red herring context factor has its impact on all metrics in our suite.

The fault in the deduction can be explained by the *idiosyncratic context presumption*.

**Definition.** *The idiosyncratic context presumption. The distribution of a code metric within a specific software system may be unique to that software system and may be unrelated to traditional "context factors".*

By this presumption, two projects may be inherently distinct even if they appear to be the same being judged by the above seven "context factors" or other objective factors. Stated differently, the presumption is that differences between projects may be linked to non-objective and hard to measure factors such as dynamics of personal interactions among members of the development team.

If this presumption is true, then the collection of values in each $D_i(\mu)$ depends on idiosyncratic properties of the projects whose $h$ value happens to be $i$. Worse, since it is well known that the size of projects is long-tailed, it is likely that the majority of values in $D_i(\mu)$ are drawn a small number of dominant projects[2]. Thus, the Kruskal-Wallis test applied to distributions $D_0(\mu)$, $D_1(\mu)$, and $D_2(\mu)$ estimates the impact of idiosyncratic variation between dominant projects rather than the desired impact of the context factor.

We show that the distributions of values of code metric value are quite distinct, even for projects of similar profile. Thus, our results support the idiosyncratic context conjecture, and in this sense complement the work of Zhang et al. This support, together with the fallacy we demonstrated with the red herring conjecture, indeed casts doubt on some of the analysis carried out in Zhang et al.'s work, but stops short of invalidating them. The impact of contextual factors could be still demonstrated by subjecting the gathered empirical data to an analysis that accounts for variation in size and to the idiosyncratic context presumption.

Moreover, our results may contribute to future research of contextual factors by showing that the idiosyncratic properties of a project for any given metric can be reduced to the two parameters that define the LNS transformation. If this is done, then the method of analysis Zhang et al. employ for project level metrics may be applicable to class level code metrics, and perhaps even method level code metrics.

## 4 The Dataset

We shall be using the term "*project*" to denote a software artifact augmented with its revision history. Our dataset comprises twenty-six such projects, which are either full blown applications, libraries, or frameworks. Each project comprises a substantial number of software modules and is managed as a distinct project in a version control system. In the interest of brevity of presentation, we denote the projects by the capital letters of the Latin alphabet: 'A', 'B',..., 'Z'.

The process by which the projects were selected tried to eliminate niche projects and identify the profiles common to projects that make the "top lists" of open source repositories.

[1]Note that this value indicates high statistical significant even after multiplying it by Bonferroni correction (20 in our case).

[2]Note that this set of dominating projects does not depend on $\mu$.

Specifically, the following criteria were applied in the making of the dataset: public availability, programming language uniformity, longevity, community of developers, non-meager development effort, and reasonable recency.

- *Public Availability.* For the sake of reproducibility, among other reasons, all projects are open-source. Moreover, we required that both the source and its *history* be available through a publicly accessible version management repository; specifically all projects were drawn from two well-known repositories: GitHub[3] and Google Code[4].
- *Programming Language Uniformity.* The primary programming language of all projects

[3]http://github.com/
[4]https://code.google.com/

Table 1 – Software projects constituting the dataset (in descending number of commits). We trace over a century of development and follow the behavior of almost 2,000 software engineers. The last column is added for reproducibility.

| **Id** | **Project** | **First Version** | **Last Version** | **#Days** | **#Authors** | **Last Commit ID** |
|---|---|---|---|---|---|---|
| A | wildfly | '10-06-08 | '14-04-22 | 1,413 | 194 | `5a29c860` |
| B | hibernate-orm | '09-07-07 | '14-07-02 | 1,821 | 150 | `b0a2ae9d` |
| C | hadoop-common | '11-08-25 | '14-08-21 | 1,092 | 69 | `0c648ba0` |
| D | jclouds | '09-04-28 | '14-04-25 | 1,823 | 100 | `f1a0370b` |
| E | elasticsearch | '11-10-31 | '14-06-20 | 963 | 129 | `812972ab` |
| F | hazelcast | '09-07-21 | '14-07-05 | 1,809 | 65 | `3c4bc794` |
| G | spring-framework | '10-10-25 | '14-01-28 | 1,190 | 53 | `c5f908b1` |
| H | hbase | '12-05-26 | '14-01-30 | 613 | 25 | `c67682c8` |
| I | netty | '11-12-28 | '14-01-28 | 762 | 72 | `3061b154` |
| J | voldemort | '01-01-01 | '14-04-28 | 4,865 | 56 | `fb3203f3` |
| K | guava | '11-04-15 | '14-02-25 | 1,047 | 12 | `6fdaf506` |
| L | openmrs-core | '10-08-16 | '14-06-18 | 1,401 | 119 | `05292d98` |
| M | CraftBukkit | '11-01-01 | '14-04-23 | 1,208 | 156 | `62ca8158` |
| N | Essentials | '11-03-19 | '14-04-27 | 1,134 | 67 | `229ff9f0` |
| O | docx4j | '12-05-12 | '14-07-04 | 783 | 19 | `8edaddfa` |
| P | atmosphere | '10-04-30 | '14-04-28 | 1,459 | 62 | `557e1044` |
| Q | k-9 | '08-10-28 | '14-05-04 | 2,014 | 81 | `95f33c38` |
| R | mongo-java-driver | '09-01-08 | '14-06-16 | 1,984 | 75 | `5565c46e` |
| S | lombok | '09-10-14 | '14-07-01 | 1,721 | 22 | `3c4f6841` |
| T | RxJava | '13-01-23 | '14-04-25 | 456 | 47 | `12723ef0` |
| U | titan | '13-01-04 | '14-04-17 | 468 | 17 | `55de01a3` |
| V | hector | '10-12-05 | '14-05-28 | 1,270 | 95 | `f2fc542c` |
| W | junit | '07-12-07 | '14-05-03 | 2,338 | 91 | `56a03468` |
| X | cucumber-jvm | '11-06-27 | '14-07-22 | 1,120 | 93 | `fd764318` |
| Y | guice | '07-12-19 | '14-07-01 | 2,386 | 17 | `76be88e8` |
| Z | jna | '11-06-22 | '14-07-07 | 1,110 | 46 | `a5942aaf` |
| | | | **Total** | 38,250 | 1,932 | |
| | | | **Average** | 1,471 | 74 | |
| | | | **Median** | 1,239 | 68 | |

is JAVA[5].

- *Longevity.* The duration of recorded project evolution is at least a year.
- *Community of Developers.* The software development involved at least ten authors.
- *Non-Meager Development Effort.* At least 100 files were added during the history of each project, and at least 300 commit operations were performed. (Observe that even the smallest projects in the table are in the same order of magnitude as that of the entire data set used in seminal work of Abreu, Goulão, and Esteves [16] on the "MOOD" metric set.)
- *Reasonable Recency.* Project is in active development (most recent change was no longer than a year ago[6]).

In selecting projects for our dataset, we scanned—in no particular order or priority—the JAVA projects in *GitHub's Trending repositories*[7] and the list provided by the *GitHub* JAVA *Project* due to Charles and Allamanis [6][8], selecting projects that match the above criteria.

Tab. 1 supplies the essential characteristics of the dataset. The first table's column is the project's *ID* (in the range 'A'–'Z'), while the second is the name by which it is known in its respective public repository. The dataset constitutes *all* publicly available versions of each of the projects, as specified in the next three table columns: date of first and last commit operations, and the number of days between these two operations. We see that overall, our dataset traces more than a century of software development. The penultimate column of Table 1 teaches that we follow the behavior of circa 2,000 software engineers.

Each **git** commit includes a SHA-1 [17] hash value to identify it. The identifier of the last commit in the repository is listed in the last column for reproducibility.

The second column of Tab. 2 shows the number of *repository commits*. A *file commit* is an addition of a new file or a modification of an existing file, and is distinguished from a *repository commit* (of GIT, SVN, etc.). A single repository commit may include commits of more than one file or none at all (e.g., file deletions).

Our study concerned only `.java` files embodying JAVA **class**es; source files offering a JAVA **interface** as well `package-info.java` files were ignored.

The third column of Tab. 2 notes the total number of inspected files in each project, while the next three columns show that, while the public repositories not always keep trace of the full development history the project, they still record the bulk of the development [9]. The penultimate column shows the median number of commits per file. This number indicates that most files are changed multiple times in their respective repositories. The last column notes the maximum number of commits for a given file. As evident, this number can be very large, even in the smaller projects.

Recall that a single file may change often during the lifetime of a project; in fact, it is not uncommon for files to be modified hundreds of times. To avoid introducing a bias for files which have been modified several times, while still reflecting the evolution, the metric value for a given file is computed by averaging across all its values across all its versions in the project.

[5]It is possible for a project to contain non-JAVA code (e.g., shell scripts) but we only analysed JAVA code.
[6]Our dataset was assembled in the course of 2014.
[7]https://github.com/trending?l=java&since=monthly
[8]http://groups.inf.ed.ac.uk/cup/javaGithub/
[9]To see this bulk, compare "#Files added" column with the "#Files in first version" column.

Table 2 – Essential statistics of commit operations in the dataset. We see that the bulk of the development is recorded in the repository's history. Also, most files are changed more than once, and some are changed even hundreds of times.

| Id | #Repo. commits | #Files inspected | #Files in first Version | #Files in last version | #Files added | Median commits per file | Max commits per file |
|---|---|---|---|---|---|---|---|
| A | 7,705 | 36,045 | 1 | 8,374 | 8,373 | 2 | 182 |
| B | 2,355 | 27,445 | 8 | 7,615 | 7,607 | 4 | 116 |
| C | 2,545 | 25,281 | 9 | 4,655 | 4,646 | 5 | 146 |
| D | 4,830 | 26,626 | 1,836 | 5,282 | 3,446 | 2 | 458 |
| E | 3,663 | 26,406 | 3 | 3,764 | 3,761 | 5 | 93 |
| F | 4,353 | 20,417 | 1 | 2,430 | 2,429 | 5 | 338 |
| G | 1,773 | 20,223 | 1 | 5,405 | 5,404 | 4 | 39 |
| H | 2,177 | 12,081 | 672 | 2,074 | 1,402 | 3 | 214 |
| I | 2,080 | 9,924 | 286 | 1,062 | 776 | 6 | 177 |
| J | 2,446 | 9,413 | 3 | 954 | 951 | 5 | 249 |
| K | 1,849 | 8,803 | 337 | 1,665 | 1,328 | 4 | 86 |
| L | 2,406 | 7,589 | 972 | 1,495 | 523 | 3 | 197 |
| M | 1,979 | 7,618 | 7 | 541 | 534 | 6 | 255 |
| N | 2,702 | 6,714 | 99 | 367 | 268 | 11 | 292 |
| O | 618 | 5,493 | 2,348 | 2,776 | 428 | 2 | 31 |
| P | 2,554 | 5,210 | 58 | 335 | 277 | 8 | 329 |
| Q | 2,638 | 5,492 | 44 | 347 | 303 | 3 | 492 |
| R | 1,514 | 4,153 | 42 | 359 | 317 | 5 | 194 |
| S | 832 | 3,323 | 2 | 702 | 700 | 2 | 93 |
| T | 1,001 | 3,524 | 1 | 450 | 449 | 4 | 575 |
| U | 637 | 2,833 | 202 | 534 | 332 | 4 | 69 |
| V | 745 | 2,670 | 182 | 459 | 277 | 4 | 74 |
| W | 866 | 2,566 | 205 | 386 | 181 | 3 | 89 |
| X | 745 | 2,492 | 19 | 462 | 443 | 2 | 116 |
| Y | 316 | 1,793 | 10 | 511 | 501 | 3 | 41 |
| Z | 525 | 1,788 | 188 | 303 | 115 | 3 | 90 |
| **Total** | 55,854 | 285,922 | 7,536 | 53,307 | 45,771 | 108 | 5,035 |
| **Average** | 2,148 | 10,997 | 289 | 2,050 | 1,760 | 4 | 193 |
| **Median** | 2,029 | 7,151 | 43 | 828 | 528 | 4 | 161 |

## 5 The Metrics Suite

Hundreds of code metrics are discussed in the literature [3, 19, 26]. We restrict ourselves to module level metrics, i.e., metrics pertaining to individual classes (rather than to an entire project, packages or individual methods). Our suite includes a sample of the most common module metrics, some variants of these, and a number of newly introduced metrics:

*The Chidamber & Kemerer metrics* [2] **CBO**, **DIT**, **NOC**, **RFC** and the **WMC** metric. The variant used for weighting methods for **WMC** is a count of the number of the tokens present in a methods. Metric **NOM** (**N**umber **o**f **M**ethods) represents another variant of **WMC**, in which all methods receive the weight of 1.

*Class metrics:* In addition to **NOM**, we employ **NIV** (**N**umber of **I**nstance **V**ariables), **MTE** (**Mut**ability (**E**xplicit)–number of non-`final` instance fields), **CON** (**Co**nstructor **C**ount). The **CHAM** (**Cham**eleonicity) metric we introduce here also belongs in this group; it is defined as the maximum, over all methods of the number of polymorphic variables accessed by the method.[10]

[10]The rationale is that **CHAM** is the best approximation (within the limits of the depth of our code analysis) of the "Chameleonicity" [18] metric, which was shown to be linked to the algorithmic complexity of analyzing virtual function calls.

*Size metrics:* **LOC** (**L**ines **o**f **C**ode), as well as **NOT** (**N**umber **o**f **T**okens) and **NOS** (**N**umber **o**f **S**tatements)[11].

*Comment metrics:* Comment lines are counted in **LOC**, but the suite include two more specific metrics: **ABC** (**A**lpha **b**etical **C**omment **C**haracter **C**ount) and **CCC** (**C**omment **C**haracters **C**ount). Both are computed on all line-, block-, and JAVAdoc- comments, with the exclusion of comments occurring before the `package` declaration, as these are typically boilerplate. The reason for excluding of non-alphabetical characters (punctuation and the such) in **ABC** is the practice, witnessed in our dataset, of using these for embellishing comments.

*Regularity metrics:* It was suggested (e.g., by Jbara and Feitelson [25]) that regularity is a better predictor of maintainability than size. The suite therefore includes metric **LZW** (Regularity (**L**empel-**Z**iv-**W**elch compression)), defines as the number of compressed tokens after applying the LZW compression algorithm [42] to the tokens of a class. Another regularity metric is **GZIP** (Regularity (**GZIP** compression)), the number of bytes in the GZip [15] compressed byte-stream of the class's source code.

*Flow complexity metrics:* First of these is **MCC**, the famous McCabe [29] Cyclomatic Complexity. We also use a simpler variant, **MCCS**, which ignores lazy circuit evaluation.s A newly introduced metrics is also present in this group: **HOR** (**Hor**izontal complexity), which is similar to **NOS** except that in the statements count, each statement is weighted by the number of control structures (`if`, `while`, etc.) that surround it.[12]

For easy reference the metrics comprising our suite are summarized in Tab. 3. (The "Classic" column is somewhat subjective, expressing our personal perspective on how entrenched the metric is.)

Comparing our method suite with that of Zhang et al. [40], ours is limited to file/class-level metrics, while Zhang et al. also investigate project-level and method-level metrics. In contrast, the suite employed in this work introduces a number of new metrics, including *chameleonicity*—the number of polymorphic variables in the code—and *code regularity*, first proposed by Jbara and Feitelson [25].

## 6 Comparing Distributions of Metric Values

In this section, we make a (partial) visual comparison of the distribution of the same metric in different projects. The plots here are meant to demonstrate that, although judgement of the visual data presented in the plots might be subjective, we believe that the data indicate context bias. For the sake of comparability, the values of all metrics were normalized such that the range of different metrics is somewhat similar. Firstly, that the similarity between the distributions is not great, and secondly, that precise statistical tests should be employed.

Fig. 1 depicts the probability distributions of metrics in our suite. For each metric, the probability density functions of the distribution of its values in each of the projects are plotted together under a log-log scale of the axis. Evidently the PDFs are noisy, although it is hard to argue that any two such distributions of values are due to the same common PDFs.

Fig. 2 depicts the *Complementary Cumulative Distribution Function* (CCDF) of the metrics in our suite. Given a random continuous variable $x \in \mathbb{R}$ with its probability density function PDF$(x)$, function CCDF is defined by CCDF$(y) = 1 - \int_{-\infty}^{y}$ PDF$(x)dx$. It holds that CCDF$(y) = P(x \geq y)$.

[11]both are based on a syntactical code analysis: tokens are the language lexical tokens while the term "statements" refers to the respective production in the language EBNF, except that bracketed blocks are ignored

[12]Similar metrics are mentioned in the literature, e.g., Harrison and Magel [22] suggested "a complexity measure based on nesting level". See also the work of Piwowarsky [34] and that of Hindle, Godfrey and Holt [24]

| Acronym | Full name | Object-Oriented | Classic | Citation |
|---|---|---|---|---|
| **ABC** | **A**lpha **b**etical **C**omment **C**haracter **C**ount | ✗ | ✗ | |
| **CBO** | **C**oupling **B**etween **C**lass **O**bjects | ✓ | ✓ | [2] |
| **CCC** | **C**omment **C**haracters **C**ount | ✗ | ✗ | |
| **CHAM** | **Cham**eleonicity | ✓ | ✗ | [18] |
| **CON** | **Con**structor **C**ount | ✓ | ✗ | |
| **DIT** | **D**epth of **I**nheri**t**ance | ✓ | ✓ | [2] |
| **GZIP** | Regularity (**GZIP** compression) | ✗ | ✗ | [25] |
| **HOR** | **Hor**izontal complexity | ✗ | ✗ | [22, 24, 34] |
| **LOC** | **L**ines **o**f **C**ode | ✗ | ✓ | |
| **LZW** | Regularity (**L**empel-**Z**iv-**W**elch compression) | ✗ | ✗ | [25] |
| **MCC** | **M**cCabe **C**yclomatic **C**omplexity | ✗ | ✓ | [29] |
| **MCCS** | **M**cCabe **C**yclomatic **C**omplexity (simple) | ✗ | ✓ | [29] |
| **MTE** | **M**u**t**ability (**E**xplicit) | ✓ | ✗ | |
| **NIV** | **N**umber of **I**nstance **V**ariables | ✓ | ✓ | |
| **NOC** | **N**umber **o**f **C**hildren | ✓ | ✓ | [2] |
| **NOM** | **N**umber **o**f **M**ethods | ✓ | ✓ | [2] |
| **NOS** | **N**umber **o**f **S**tatements | ✗ | ✓ | |
| **NOT** | **N**umber **o**f **T**okens | ✗ | ✗ | |
| **RFC** | **R**esponse **f**or a **C**lass | ✓ | ✓ | [2] |
| **WMC** | **W**eighted **M**ethods per **C**lass | ✓ | ✓ | [2] |

Table 3 – Metrics used in this study. We use a mixture of object-oriented and classical metrics.

Comparing Fig. 1 with Fig. 2, we see that (as expected) the mathematical integration carried out by the CCDF significantly reduces noise. This smoothing also highlights the fact that although the distributions are close, they are distinct, and it is difficult to believe that they are all due to a single, omnipresent distribution inherent to this metric, which does not depend on the actual project.

The CCDF plots in each of the $X/Y$ graphs of Fig. 2 may look "quite similar" to some readers and "quite distinct" to others. To set a baseline for comparison, we introduce yet another "metric", which uses the value of the cryptographic SHA-1 [17] function of the source code. Specifically, metric SHA-1 is defined as the lower 8 bits of the hash code. Obviously, SHA-1 demonstrates an almost uniform distribution in the range of $[0, 255]$. Deviations between the distributions of SHA-1 in distinct projects, is due only to statistical noise.

To visually compare the closeness of distributions of a "real" metric such as **NOT** with that of SHA-1, we need to draw these on the same scale. To this end, we employ a linear, *same-scale transformation*, defined by the following property:

> The median value of the metric is 0.5, while its decile value of the metric is $0.1$, i.e., $10\%$ of the transformed metric value are in the range $(0, 0.1)$, $40\%$ are in the range $(0.1, 0.5)$ and the remaining $50\%$ in the range $(0.5, \infty)$.

Fig. 3 shows the CCDF distributions of the thus transformed SHA-1 metric in all projects. As expected, these distributions are very close to each other.

Fig. 2 also uses the same transformation, where the median and the decile used in computing the parameters of same-scale transformation were computed from the entire set of metric values for all projects.

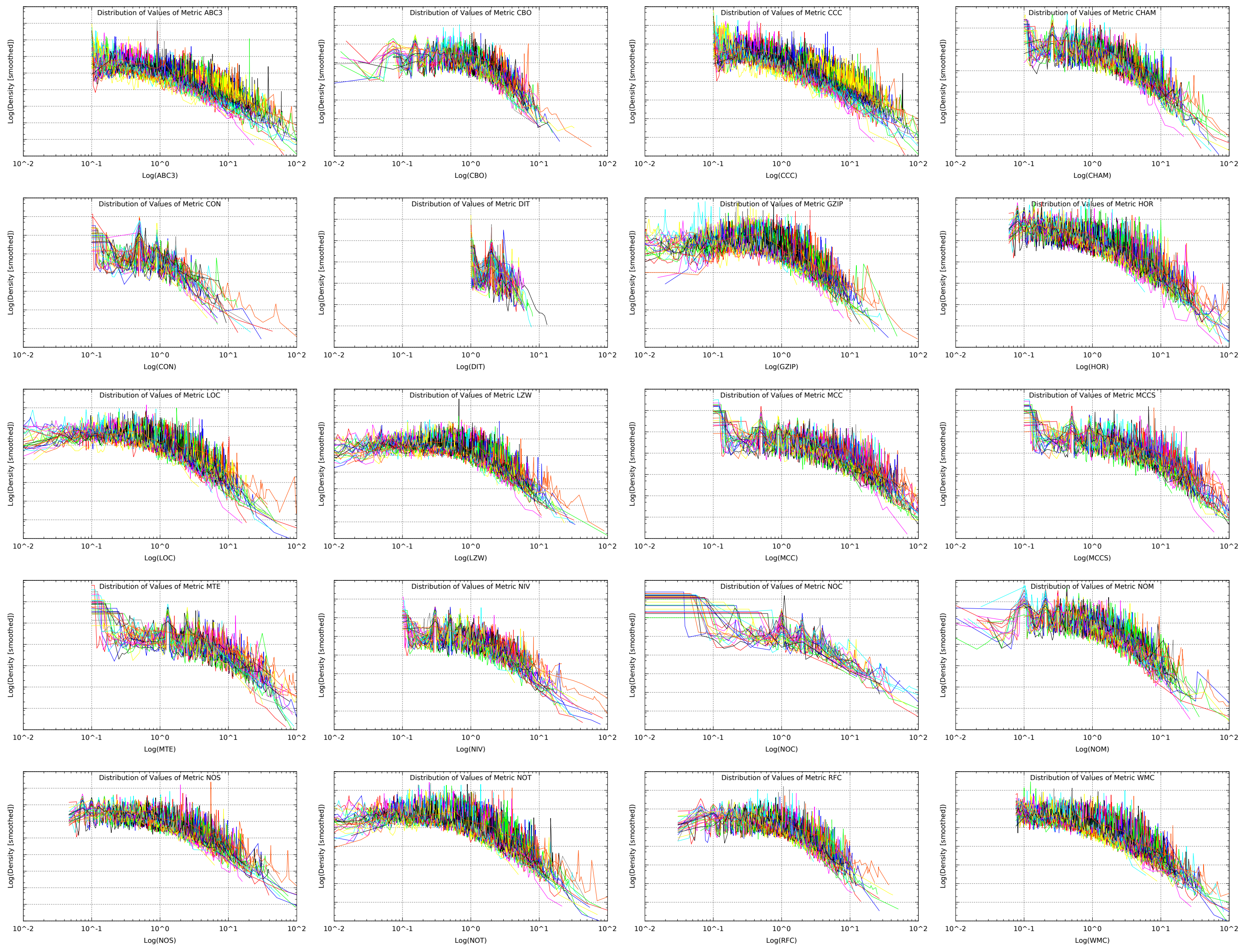

Figure 1 – Probability Density Functions (PDF) of all metrics in our suite. Although the plots are noisy, it is hard to argue that any two distributions of values are due to the same common PDFs.

Comparing the closeness of the curves in each of the $X/Y$ graphs in Fig. 2 with that of Fig. 3 (while observing that the $X$-range and the $Y$-range are identical in all of these), we can conclude that the deviation of the variety of distributions of the same metric in distinct projects cannot be explained by statistical noise.

## 7 Statistical-Tests

This section applies statistical reasoning to verify the suggestion raised by the visual inspection of sample of figures sample of the previous section, i.e., that: *"the same metric tends*

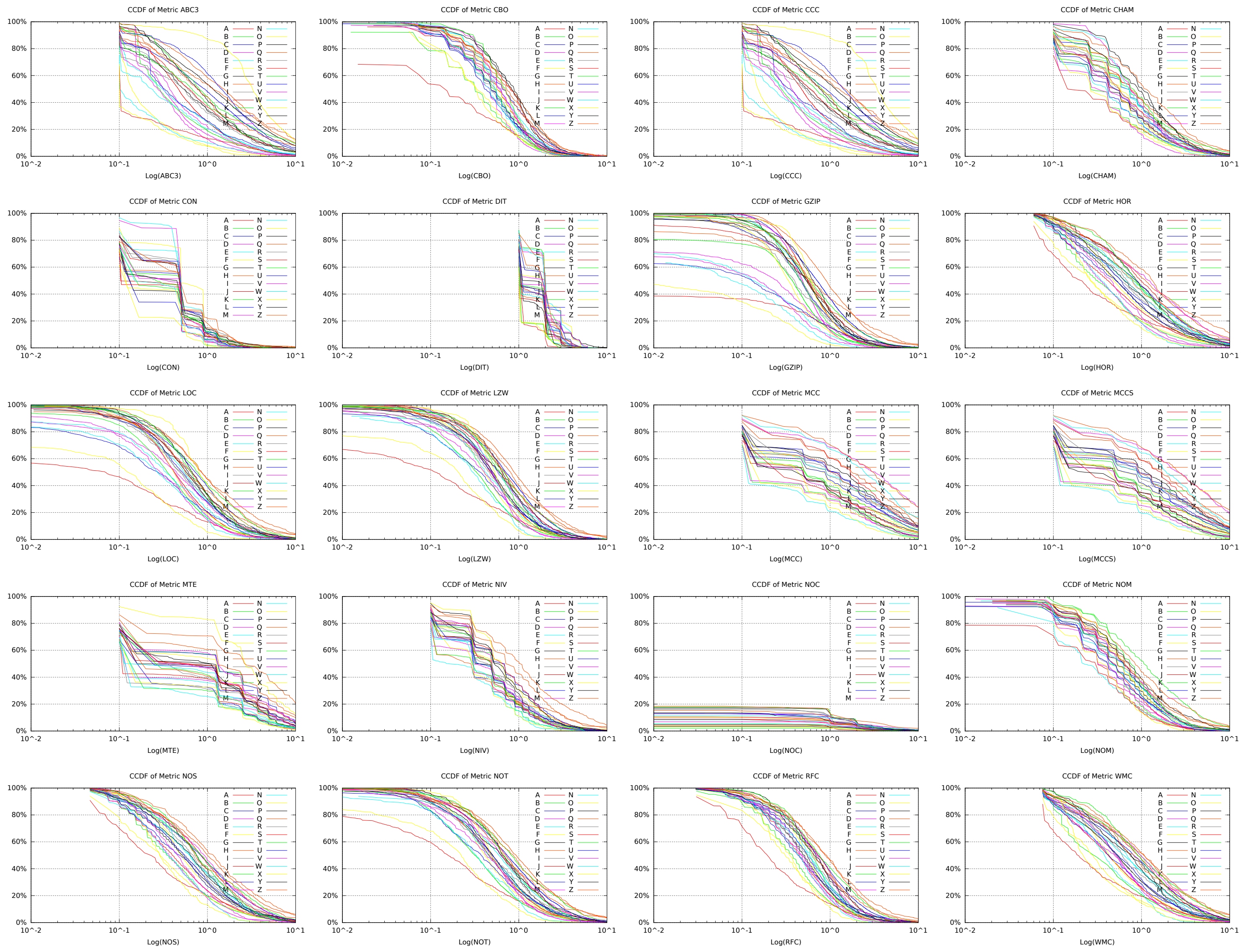

Figure 2 – CCDF of the distributions of selected metrics in all projects. The mathematical integration significantly reduced the noise levels of the PDF metrics. There is clear difference between metrics in different projects.

*to distribute in distinct manners in different projects"*. Concretely, we shall observe that for each metric $\mu$, the multi-sets of values that $\mu$ assumes in the projects, $\mu_A$, $\mu_B$,..., $\mu_Z$, appear as if they are drawn from distinct distributions, and search for metrics' transformation to eliminate this predicament.

Sect. 7.1 provides a brief reminder of the $U$-test, which we employ for statistically measuring the similarity of *two* multi-sets. Sect. 7.2 then discusses our "similarity measure", designed for dealing with more than two multi-sets and confirms the suggestion that the distributions of the plain metrics are indeed distinct in distinct projects. In Sect. 7.3 we define "rank transformation" and show that it is "normalizing" in the sense that after applying it to

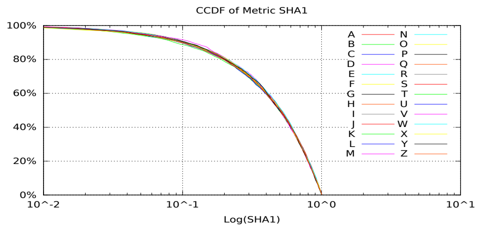

Figure 3 – CCDF of the **SHA** metric. This plot is an example of a metric that behaves exactly the same in each project

(most) metrics, their distributions in distinct projects become similar. Other candidates for normalizing transformations are the subject of Sect. 7.4. Finally, Sect. 7.6 reflects on the search for normalizing transformations arguing that it is more difficult when the entropy of metric distribution is small.

## 7.1 The $U$-Test

As evident from the figures of the previous section, the distribution of metrics does not make a simple Gaussian, which suggests we should employ a non-parametric test. We will use the $U$-test in our analysis[13].

Recall that the input to a $U$-test is a pair of two multi-sets of numbers: $\mathcal{D}_A$ and $\mathcal{D}_B$. The test examines the null hypothesis: $\mathcal{D}_A$ and $\mathcal{D}_B$ are drawn from some common (yet, unknown) distribution, conjugated with the claim that values in any of $\{\mathcal{D}_A, \mathcal{D}_B\}$ are not significantly greater than those values found in the other such multi-set. Such judgment is made by computing the $U$ *statistics*, defined by the summation carried over each $d_1 \in \mathcal{D}_A$ of its *rank in* $\mathcal{D}_B$, i.e., the number of values $d_2 \in \mathcal{D}_B$ such that $d_2 < d_1$. As usual with statistical tests, values of $U$ exceeding certain critical points, reject the null hypothesis for the desired statistical significance level.

Just like the Kolmogorov-Smirnov (K-S) test, the $U$-test is *non-parametric* in the sense that it ignores the magnitude of values. Also, just like the K-S test, and unlike the $\chi^2$-test, the $U$-test is *ordinal*: even though magnitude is ignored, the test uses the relative ordering of the values. Consequently, statistical preserving transformations applied to both $\mathcal{D}_A$ and $\mathcal{D}_B$ are transparent to the test.

The $U$-test is preferred over K-S in the case of software metrics, because it is better suited to discrete distributions where repetitions are the norm, not the exception.

## 7.2 Comparing Multiple Sets

How should we quantify statistical similarity between more than *two* multi-sets? A fundamental difficulty is that statistical tests can never prove similarity; all they can do is disprove it. Another issue is that the generalization of the $U$-test for more than two samples, i.e., the *Kruskal-Wallis test* invalidates the null hypothesis even if only *one* multi-set is drawn from a distribution distinct to that of the others.

Instead, we propose to apply a $U$-test to each pair of multi-sets. Specifically, for a given metric $\mu$, we apply a $U$-test to each of the $325$ unordered pairs: $\mu_i, \mu_j \in \{\mu_A, \mu_B, \ldots, \mu_Z\}$ for all integers: $1 \leq i < j \leq 26$. Let $s$ be the percentage of pairs in which the null hypothesis

[13]The test is known mainly as the *Mann-Whitney-Wilcoxon U test*, but also as the *Wilcoxon rank-sum test*

*is not rejected*, i.e.,

$$s = s(\mu) = \frac{\#\{\langle \mu_i, \mu_j \rangle | i, j \in \mathcal{C}, i < j, U(\mu_i, \mu_j) \geq 0.05\}}{\#\{\langle \mu_i, \mu_j \rangle | i, j \in \mathcal{C}, i < j\}} \cdot 100\% \qquad (1)$$

where $\mathcal{C}$ is the collection of all projects and $\mu_i \in \mathcal{C}$ is the distribution of metric $\mu$ in the $i^{th}$ project. We shall use $s(\mu)$ as a measure of the similarity of the distributions of $\mu$ in the different projects: If $s(\mu) = 100\%$, then the $U$ test is unable to distinguish between any of the distributions of $\mu$ in the constituents of $\mathcal{C}$.

Note that although we perform multiple statistical tests, the Bonferroni correction is not applicable here. We do not seek the few individual cases in which the null hypothesis is rejected. Actually, confidence in the desired conclusion is increased by the opposite case in which the null-hypothesis is not rejected by the majority of comparisons. However, the confidence behind the Bonferroni correction is still applicable in the following sense: if significance level is set at $0.05$, then confidence in the conclusion is not reduced if the null hypothesis is rejected in as many as $5\%$ of the cases.

Values of $s(\mu)$ for all $\mu$ in our suite, are shown in Tab. 4. For now, concentrate only on the first and the third table's columns—showing metric's name and the value of this measure, respectively. Subsequent columns show the values of $s(\Phi(\mu))$ for a variety of *transformation* functions $\Phi$; these are discussed below in Sect. 7.3 and Sect. 7.4.

To appreciate the numbers in the table, if it is the case that a metric is *distributed in the same manner in all projects*, then the similarity-measure is expected to be in the range of $95\%$ to $100\%$, depending on the conservativeness of the $U$-test.

In case there are *two*, equally popular, distinct distributions, this expectation drops to about $52.5\%$; in the case there are *ten* such distributions, to about $10\%$.

We set the threshold at $75\%$, considering a set of distributions as "similar" when the measure exceeds this level. This threshold means that the number of different distributions, normalized by their frequency, is somewhere between one and two.

For example, the third column in the **LOC** row shows that $13\%$ of the projects pairs did not reject the null hypothesis. This measure value can be interpreted as:

> *Choose two projects at random. Then, in six out of seven cases, values of the lines-of-code metric in one project are going to be noticeably greater than those found in the other project.*

We see that even the popular **LOC** metric is sensitive enough to what might be called *context bias*, i.e., that metric values found in one project are largely irrelevant to another project.

Examining the third column, we see that **LOC** is not unique among its peers. The low measure value characterizes *all* metrics in our suite. Thus, a grasp on the distribution of values of any metric in one project is highly unlikely to be useful for another project.

## 7.3 Metric Normalization

This section affirms that the context bias can be neutralized for most metrics in our suite. Consider the right-most column of Tab. 4. This column is similar to third column explained above, except that the depicted values are with respect to the "*rank-normalized*" metric. Rank normalization transforms the $i^{th}$ largest metric value into $(i-1)/n$, where $n$ is the total number of distinct values that the metric assumes. Therefore, rank normalization converts any discrete distribution to a discrete approximation of the uniform distribution in the range $[0, 1]$.

The $U$-test should judge any two rank-normalized discrete distributions as being the same, except when the approximation of the uniform distribution is poor, which happens if $n$ is small or if there is a bias in repetition of values.

Table 4 – Similarity measure of the distributions in the projects of plain and normalized metric values (Measure defined as the percentage of projects pairs that *do not reject* the “same distribution” hypothesis at significance level $\leq 0.95$). Values above 75% are considered “similar”. All metrics initially exhibit low similarity. Using rank normalization or log-normal-standardizationmanages to eliminate context bias in most of the metrics.

| Metric | Entropy | $(\cdot)$ | $\frac{(\cdot)}{\sigma}$ | $(\cdot)-\mu$ | $\frac{(\cdot)-\mu}{\sigma}$ | $\frac{\log(\cdot)}{\sigma}$ | $\log(\cdot)-\mu$ | $\frac{\log(\cdot)-\mu}{\sigma}$ | rank$(\cdot)$ |
|---|---|---|---|---|---|---|---|---|---|
| **ABC** 3 | 10.87 | 12% | 13% | 1% | 11% | 5% | 63% | **75**% | **99**% |
| **CBO** | 7.26 | 21% | 22% | 29% | **76**% | 19% | **92**% | **94**% | **100**% |
| **CCC** | 10.94 | 11% | 13% | 1% | 12% | 6% | 62% | **76**% | **99**% |
| **CHAM** | 6.30 | 26% | 18% | 10% | 19% | 15% | 70% | **83**% | **95**% |
| **CON** | 2.90 | 24% | 14% | 22% | 38% | 8% | 35% | 24% | 37% |
| **DIT** | 2.25 | 20% | 8% | 15% | 20% | 22% | 16% | 17% | 19% |
| **ENT** | 15.12 | 20% | 2% | **95**% | **100**% | 2% | **89**% | **100**% | **100**% |
| **GZIP** | 13.60 | 13% | 11% | 10% | 38% | 3% | **76**% | **92**% | **100**% |
| **HOR** | 9.30 | 17% | 16% | 4% | 10% | 19% | **96**% | **95**% | **100**% |
| **LOC** | 11.36 | 18% | 13% | 8% | 21% | 8% | **96**% | **99**% | **100**% |
| **LPC** | 2.92 | 31% | 38% | 0% | 0% | 0% | 2% | 0% | 0% |
| **LZW** | 12.60 | 19% | 11% | 16% | 38% | 7% | **99**% | **100**% | **100**% |
| **MCC** | 5.78 | 18% | 10% | 3% | 6% | 27% | 64% | 64% | 59% |
| **MCCS** | 5.66 | 20% | 8% | 3% | 8% | 26% | 64% | 64% | 58% |
| **MTE** | 3.95 | 25% | 33% | 2% | 6% | 2% | 18% | 10% | 16% |
| **NIV** | 5.44 | 26% | 22% | 11% | 25% | 18% | 65% | **75**% | **88**% |
| **NOC** | 1.14 | 52% | 53% | 0% | 0% | 0% | 0% | 0% | 0% |
| **NOM** | 6.78 | 32% | 13% | 11% | 26% | 16% | **80**% | **90**% | **99**% |
| **NOS** | 8.91 | 17% | 15% | 5% | 17% | 19% | **96**% | **96**% | **100**% |
| **NOT** | 12.78 | 20% | 13% | 11% | 25% | 9% | **99**% | **100**% | **100**% |
| **RFC** | 8.48 | 24% | 22% | 16% | 55% | 20% | **96**% | **98**% | **100**% |
| **WMC** | 11.21 | 19% | 18% | 6% | 14% | 14% | **89**% | **93**% | **100**% |
| | **Mean** | 22% | 18% | 13% | 26% | 12% | 67% | 70% | **76**% |
| | $\sigma$ | 8% | 10% | 19% | 24% | 8% | 31% | 34% | 35% |
| | **Median** | 20% | 14% | 9% | 20% | 12% | 73% | **86**% | **99**% |
| | **MAD** | 3% | 4% | 6% | 10% | 7% | 20% | 12% | 0% |
| | **Min** | 11% | 2% | 0% | 0% | 0% | 0% | 0% | 0% |
| | **Max** | 52% | 53% | **95**% | **100**% | 27% | **99**% | **100**% | **100**% |

A case in point is the value of 40% in the **MCC** row of this column. The judgment of the $U$ test is that the distribution of the *rank normalized* **MCC** in two projects $i, j$, $i \neg j$ is distinct in 60% of all such pairs. The explanation is that the number of values that **MCC** assumes is small and repetitions of these values shows different bias in distinct projects.

The last column is an approximate upper bound on the score of similarity that can be achieved with any transformation. Eleven out of the fourteen metrics scored more than $80\%$ in the similarity scale; seven metrics even scored a perfect $100\%$.

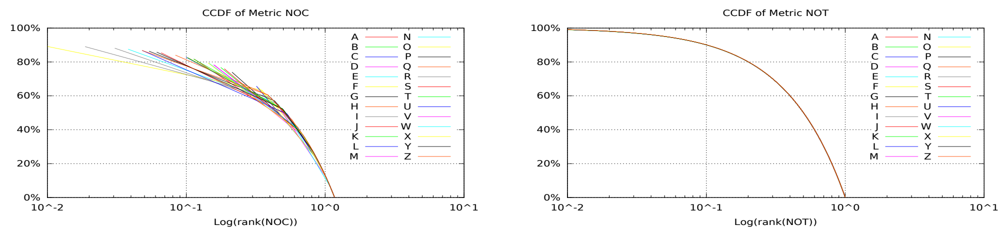

Figure 4 – Cumulative Complementary Density Function of the distribution of selected metrics in our projects, after applying rank normalization. The difference between the metric on the left, which still exhibits context bias, and the metric on the right, which no longer does, is clearly evident.

Fig. 4 depicts the CCDF of the distributions of two selected metrics in all projects. To appreciate the visual information carried by the figure, recall that in the absence of repetitions, rank normalization makes the best discrete approximation of the uniform distribution and that this approximation is not subject to statistical noise. Thus, in metric **NOT** where the set of values is large and repetitions are scarce, the closeness of the curves is even better than that of SHA-1 (Fig. 3). We do expect such a behavior in, e.g., metric **NOC** where the relatively small discrete range forces many repetitions. Even with plot smoothing, the curves in the left hand side of Fig. 4 look much farther away than in Fig. 3.

From the two figures and Tab. 4, we conclude that the rank normalized value of a metric largely eliminates context bias. Rank normalization is a useful pre-processing stage when values of a metric in one project are mixed with values drawn from another project or conclusions drawn based on one project are to be applied to another. Its major drawback is in being *irreversible*, i.e., it is impossible to retrieve the original values after using it.

## 7.4 Investigating Context Bias

We proceed in our attempts to identify the means to eliminate "context bias". Various "normalization" functions are applied to the metric values prior to the computation of the similarity measure. The results are presented in the inner columns of Tab. 4.

Consider for example, the fourth column showing the similarity measure computed by dividing the metric value by the standard deviation of the project from which the value was drawn. This column tests the hypothesis that projects differ only by their standard deviation. The low similarity measure values found in this column indicate that this is not a worthy pre-processing state.

Similarly, the two subsequent columns test the conjecture that subtracting the mean eliminates the context bias and that by subtracting the mean and dividing by the standard deviation also eliminates context bias. Other than isolated the high similarities for metrics **CBO**, the generally low measure values under these two columns lead us to eliminate the possibility that the mean or the standard deviations capture context bias.

#### 7.4.1 log-Normal-Standardization

The next three columns follow the recommendation proposed by Cohen & Gil [39] to consider the logarithm of the metric values. As witnessed by the larger similarity measures in these three columns, this recommendation is a promising direction. In fact, the penultimate column of Tab. 4 rivals its ultimate column. This rivalry suggests that the effects of the context bias are limited to scaling (by the standard deviation) and shifting (by the negative of the mean) in the logarithmic scale. We call the penultimate normalization the log-*normal-standardization* (LNS). The name is not to imply that the metrics follow a log-normal distribution. In fact, the Shapiro-Wilk test rejects the log-normal null hypothesis for the vast majority of the selection of a metric and a project pair. It is rather in suggestion to the *standard score* for normal distributions.

The CCDF plots of some of the metrics, after being subjected to LNS, are depicted in Fig. 5[14].

The actual curves were subjected to same-scale transformation (see Sect. 6) to allow visual comparison with Fig. 3. Obviously, curves in each of the $X/Y$ graphs of Fig. 5 are not overwhelmingly more distant than in Fig. 3. Moreover, each of the $X/Y$ graphs of Fig. 5 when compared with its Fig. 2 counterpart shows that LNS makes close to identical distributions in different projects of the same metric. It can even be argued that closeness of some of the curve bundles may be at the level of statistical fluctuations of the SHA-1 metric (Fig. 3).

Yet, we are compelled to draw attention to the curves associated with the **NOC** metric. As can be seen in Tab. 4, both rank and LNS fail to make the distributions of **NOC** to be as close as curves in other metrics. The raggedness of the curves in Fig. 5 associated to this metric is explained by this failure, which, in turn, might be explained by the relatively small number of values that **NOC** typically assumes.

## 7.5 Median-Based Normalization

Since the $U$ test is non-parametric, better normalization results might be obtained by replacing the mean and standard deviation statistics with the median and *median absolute deviation* (MAD), respectively. The results of these normalizations are aggregated in Tab. 5.

Consider the fifth column in Tab. 5, which is analogue to the standard normalization in Tab. 4 (the sixth column). The results of this column, as summarized in the last 6 rows of each table, are much higher. However, inspecting the last column of Tab. 5, we see that not only was the log transformation less beneficial than it was in Tab. 4, but the results are actually *poorer*. We conclude that this direction is not preferable to either rank or LNS.

## 7.6 The Effects of Low Entropy on Metric Normalization

We turn our attention to the metrics that we could not normalize. One metric in which the similarity measure is low, even after rank normalization, is easy to explain. It was shown [20] that the average number of constructors in a class is slightly greater than one. Subsequently, the variety (measured, e.g., by the entropy of the distribution) of values of **CON** must be small. Consider now a typical case where in one project $60\%$ (say) of all classes have one constructor, while this ratio is $70\%$ in another project. Then, the discrete nature of the rank function makes it impossible to apply an order preserving, non-collapsing, transformation under which this gap can be explained as statistical noise about some common distribution. The above deduction may also be applicable to other metrics, e.g., **NIV** and **NOC**.

[14]The PDF plots are omitted as they are, even after normalization, still too noisy.

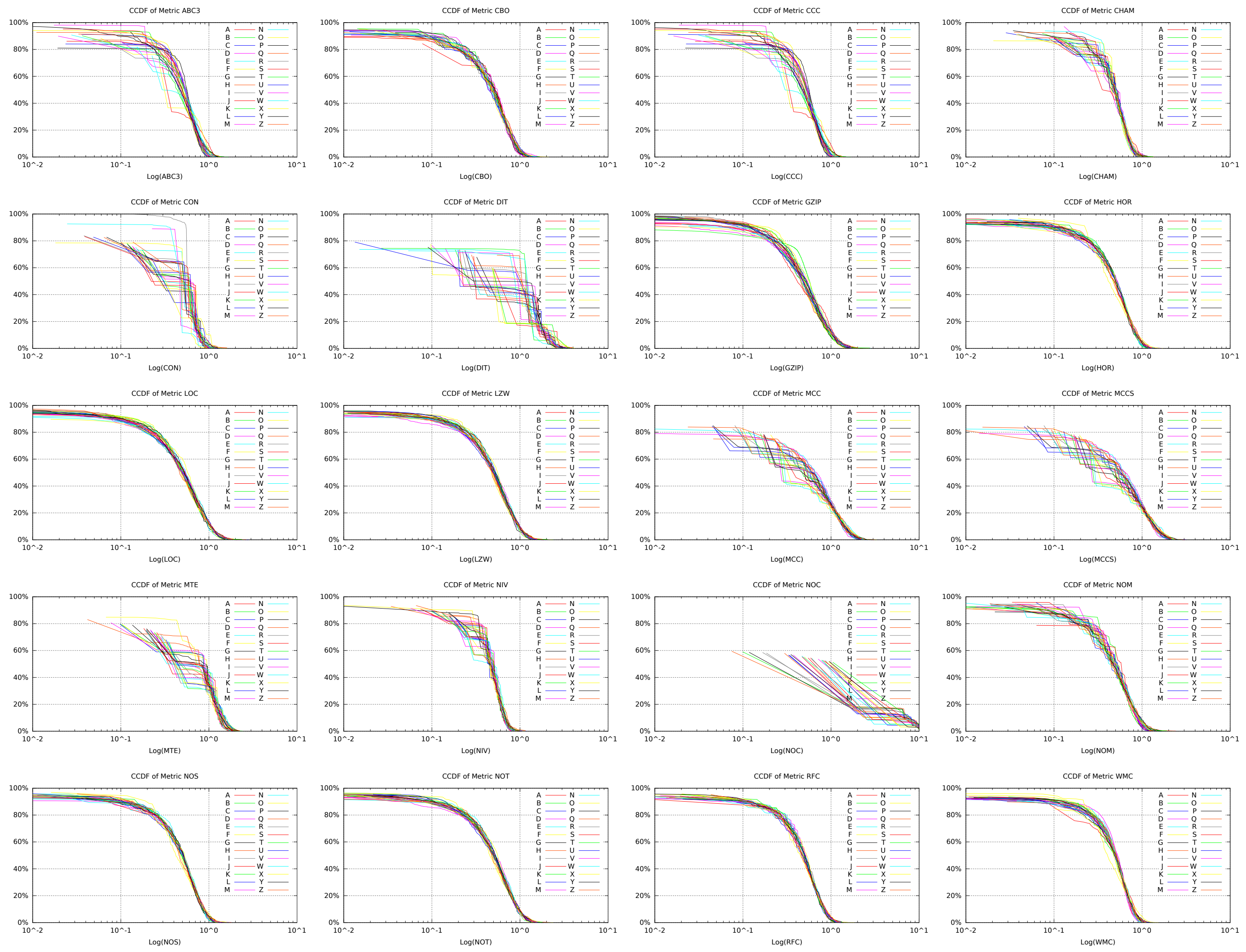

Figure 5 – Cumulative Complementary Density Function of the distribution of selected metrics in our projects, after using the log-normal-standardization. For some metrics, e.g., **NOT**, the context bias has been greatly reduced, while for some, e.g., **DIT**, it didn't.

This points us towards finding a commonality between those metrics. We hypothesize that metrics that exhibit only small variation in values would be more resistant to normalizations. One way of measuring the variation in values is measuring the *information theoretical entropy* [36] of the set of metric values in the unified project[15]. Given a metric value $\mu$, the entropy of $\mu$'s values is defined as: $-\sum_{v\in V} P(v)\cdot\log_2 P(v)$, where $V$ is the set of all values and $P(v)$ is the relative frequency of a given value $v$. A possible interpretation of entropy is

[15]The unified project is made up from all classes in all projects in our dataset

Table 5 – Similarity measure of the distributions in the projects using *median-based* normalizations (Measure defined as the percentage of projects pairs that *do not reject* the "same distribution" hypothesis at significance level $\leq 0.95$). Values above 75% are considered "similar". Compared with the normalizations in Tab. 4, median-based normalizations are actually poorer for the most part.

| Metric | Entropy | $(\cdot)-\mu_{1/2}$ | $\frac{(\cdot)}{\text{MAD}}$ | $\frac{(\cdot)-\mu_{1/2}}{\text{MAD}}$ | $\log(\cdot)-\mu_{1/2}$ | $\frac{\log(\cdot)}{\text{MAD}}$ | $\frac{\log(\cdot)-\mu_{1/2}}{\text{MAD}}$ |
|---|---|---|---|---|---|---|---|
| **ABC** 3 | 10.87 | 32% | 46% | 59% | 59% | 7% | 55% |
| **CBO** | 7.26 | 69% | 29% | **84%** | **88%** | 10% | **90%** |
| **CCC** | 10.94 | 35% | 47% | 59% | 60% | 8% | 56% |
| **CHAM** | 6.30 | 62% | 37% | 72% | 69% | 14% | 63% |
| **CON** | 2.90 | 26% | 15% | 27% | 14% | 11% | 19% |
| **DIT** | 2.25 | 15% | 16% | 17% | 15% | 13% | 16% |
| **ENT** | 15.12 | **96%** | 1% | **93%** | **97%** | 1% | **95%** |
| **GZIP** | 13.60 | 72% | 16% | **78%** | **82%** | 1% | **79%** |
| **HOR** | 9.30 | 44% | 69% | **90%** | **86%** | 15% | **86%** |
| **LOC** | 11.36 | **78%** | 29% | **87%** | **89%** | 6% | **92%** |
| **LPC** | 2.92 | 27% | 27% | 36% | 22% | 0% | 27% |
| **LZW** | 12.60 | **83%** | 38% | **96%** | **96%** | 6% | **97%** |
| **MCC** | 5.78 | 28% | 19% | 67% | 39% | 33% | 48% |
| **MCCS** | 5.66 | 24% | 16% | 60% | 33% | 36% | 36% |
| **MTE** | 3.95 | 22% | 23% | 32% | 14% | 3% | 15% |
| **NIV** | 5.44 | 45% | 44% | 54% | 48% | 17% | 55% |
| **NOC** | 1.14 | 52% | 52% | 52% | 52% | 0% | 52% |
| **NOM** | 6.78 | 58% | 48% | 68% | 65% | 20% | 66% |
| **NOS** | 8.91 | 56% | 58% | **90%** | **84%** | 19% | **84%** |
| **NOT** | 12.78 | 65% | 48% | **91%** | **89%** | 7% | **91%** |
| **RFC** | 8.48 | 68% | 59% | **92%** | **90%** | 17% | **90%** |
| **WMC** | 11.21 | 51% | 64% | **82%** | **86%** | 8% | **85%** |
| | **Mean** | 50% | 36% | 67% | 62% | 11% | 63% |
| | $\sigma$ | 21% | 18% | 23% | 28% | 9% | 26% |
| | **Median** | 51% | 38% | 70% | 67% | 9% | 65% |
| | **MAD** | 18% | 14% | 17% | 21% | 6% | 23% |
| | **Min** | 15% | 1% | 17% | 14% | 0% | 15% |
| | **Max** | **96%** | 69% | **96%** | **97%** | 36% | **97%** |

the expected number of bits needed to specify a value. By observing the second column of Tab. 4, we conclude that all of the metrics that we could not normalize have low entropy.

## 8 Threats to Validity

1. *Dataset Selection.* Sect. 4 outlined our methods for selecting the dataset. As it is invariably the case with empirical research, our selection involves many arbitrary decisions: the specific number of projects, the repositories from which these are drawn, our adequacy criteria and the particular threshold levels for each of these, etc.

   Perturbing any of these decisions, even slightly, shall alter the dataset under inspection. The numbers in the tables above will then change as well. Perturbations here, as in all empirical studies, may invalidate the results. This technical threat to validity is no different than in other works of this sort. Moreover, in targeting mainstream projects from a mainstream programming language, our conclusions should be useful in a wide array of domains at least as baseline for comparison.

Conversely, a justification for this particular selections employs the perspective of a software engineer appreciating the values of the code metrics in his current project: Past projects—carried out in the same store and under controlled development environment [33]—are the ideal baseline for comparison. However, experience indicates [21] that such previous data is rarely available. Our engineer may then turn to using data gathered in "similar" projects. We argue that our selection process is sufficiently similar to that of a "reasonable" engineer, to advise against the plain use of the code metrics, and to recommend the LNS process.

2. *The Metrics Suite.* We cannot claim that our suite is definitive; but, we do suggest that it is sufficiently representative to cast a doubt on un-normalized uses of code metrics. Also, it should be clear that rank-normalization is applicable to any metric with sufficiently large entropy. Partial extrapolation to method-level metrics can be made relying on data found in the literature [9, 39], showing that the distributions of many of these is Pareto or log-normal.

3. *Concentrating on Open Source Software.* Does our decision to concentrate on open-source software raise a threat to validity? Obviously, caution must be exercised before extrapolating to software developed in a closed-source setting. Yet, we argue that this extrapolation makes our conclusions *even more* likely to be true. To see this, consider two projects drawn from *distinct* closed-source stores. The separate and sometimes secretive nature of non-free software development is likely to make these projects even more "different". (Clearly, the threat to validity is in effect when comparing projects of the same factory.)

4. *The Sole Use of* JAVA *Projects.* Would one expect the results to be valid in other programming languages? We cannot offer a definite answer here. On the one hand, none of our research methods or metrics were unique to JAVA. On the other, we know that coding style (e.g., typical method and module length) varies greatly between languages. Consider, for example, the tendency of SMALLTALK [1] methods to be terse, or, the habit of C++ [5] programmers to place several classes in a single compilation unit, in contrast to JAVA's idiom of one class in each compilation unit. It remains to determine whether these style differences affect context bias.

5. *Correlation vs. Causation.* The famous "correlation vs. causation" trap exists in almost every statistical study. In our case though, it takes a different shape, because our conclusion relies on demonstrating "non-correlation" rather than "correlation". An (apparent) fallacy in our line of reasoning can be phrased as:

   > *It is obvious that some software artifacts are more complex, buggy, risky, etc., than others. Therefore, with the belief that code metrics predict complexity (bugginess, risk-factor, etc.) we should expect each artifact to exhibit its own bias of code metrics. Context bias is hence nothing more than a paler reflection of the inherent variation in artifacts' complexity.*

   We suggest that this devil-advocating claim be rejected outright. Suppose to the contrary that correlation between internal and external metrics is indeed so profound to make the above claim true. Now, it would be relatively easy to measure external qualities of software projects (rather than individual modules) using such internal metrics.

Then, it is highly unlikely that such a correlation would never have been detected by the research community.

## 9 Discussion

The suspicion of the existence of *context bias* raised by the PDF and CCDF plots of Sect. 6, and fed by the general belief that "every software project is different", was confirmed in Sect. 7:

> *For all metrics we examined, we found that the distribution of the metric value in one software artifact yields, with high probability, a biased judgement of the values in another such artifact.*

Bluntly, the code metric values, when inspected out of context, mean nothing.

As noted above in Sect. 3, we do not go as much as studying the impact of criteria such as host, size, and recency on projects' similarity. However, we do show that a study of this sort is at risk to lead to fallacies unless caution is exercised to account for the idiosyncratic context presumption.

Our work is applicable to research of ties between internal and external properties: for this research to be of general value, it cannot concentrate on artifacts developed in a single controlled environment. We also argue that LNS (or rank-normalization) should be applied prior to merging data collected in such research.

Sect. 7 also substantiates other claims made in Sect. 1: rank normalization, obtained by re-scaling the ordinal rank into the $[0, 1]$ range, removes the context-bias of the majority of metrics. We also learned that most of the context bias is captured by two factors: the mean and the standard deviation of the logarithm of—rather than the plain version of—the metric values. Using LNS is preferable because it is reversible.

Overall, we have shown that the plain, un-corrected value of a software metric is *unlikely* to predict external qualities of software. The major research question that this research leaves open is whether corrected values can be correlated with such external qualities.

Other directions for further research may include:

- *Impact of Entropy.* We have shown that while we could not neutralize the context bias in all of the metrics, the values of those metrics also exhibit smaller entropy. Further research may provide better insight into the relationship of entropy and software metrics.

- *Use of Average Metric Value.* Software artifacts are never stagnate; so are the modules that constitute the artifact. Previous work on software metrics tends to ignore this fact, measuring the code metrics of modules in a snapshot of the enclosing artifact, usually on the most recently available snapshot.

  Our work is different in that the value assigned to a code metric of a certain module is computed by averaging this metric's value across all versions of this module. We argue that this makes the results more robust.

  Repeating the analysis and considering only the latest snapshot, the context bias was even more apparent, because the entropy of the metric values actually decreases. Consider, for example, the **NOC** metric: by default, its set of values is very limited (as discussed in Sect. 7). However, when averaging these value across several files, they are no longer necessarily integers. Hence, the set of values increases greatly.

- *Using the $U$-Test.* The null hypothesis standing trial by the $U$-test is that values in one distribution are neither greater nor lesser than the other. Therefore, the high-percentage values, reaching $100\%$ in the last column Tab. 4, are a bit misleading. These ratios do not mean that the distributions in different project of a certain metric value are the same (as far as the test can detect). To show that the distribution are the "same", future research may need to employ the Kolmogorov-Smirnoff test.

**Acknowledgements** This research was supported by the ***I**srael **S**cience **F**oundation* (ISF), grant No. 1803/13. We thank Eli Sennesh (Gottlieb) for inspiring discussions and for cooperating in the development of the software base underlying our empirical work.

## References

[1] *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley Publishing Company, Reading, MA, 1984.

[2] A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[3] *Software Metrics*. Prentice-Hall, Englewood Cliffs, New Jersy 07632, 1996.

[4] *The* JAVA *Programming Language*. The Java Series. Addison-Wesley Publishing Company, Reading, MA, 1996.

[5] *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, MA, 1997.

[6] M. Allamanis and S. Charles. Mining source code repositories at massive scale using language modeling. In *The* $10^{th}$ *Working Conference on Mining Software Repositories*, pages 207–216. IEEE, 2013.

[7] T. L. Alves, C. Ypma, and J. Visser. Deriving metric thresholds from benchmark data. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[8] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Softw. Eng.*, 28(1):4–17, Jan. 2002.

[9] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. Understanding the shape of java software. In P. L. Tarr and W. R. Cook, editors, *(OOPSLA'06)*, Portland, Oregon, Oct.22-26 2006. ACM.

[10] V. Blondeau, N. Anquetil, S. Ducasse, S. Cresson, and P. Croisy. Software metrics to predict the health of a project? In *IWST'15*, page 8, 2015.

[11] A. Capiluppi, A. E. Faria, and J. Fernández-Ramil. Exploring the relationship between cumulative change and complexity in an open source system. In *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*, pages 21–29. IEEE, 2005.

[12] A. Capiluppi and J. Fernández-Ramil. A model to predict anti-regressive effort in open source software. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 194–203. IEEE, 2007.

[13] V. U. B. Challagulla, F. B. Bastani, I.-L. Yen, and R. A. Paul. Empirical assessment of machine learning based software defect prediction techniques. *International Journal on Artificial Intelligence Tools*, 17(02):389–400, 2008.

[14] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Mining Soft. Repo. (MSR), 2010 7th IEEE Working Conference on*, pages 31–41. IEEE, 2010.

[15] L. P. Deutsch. Gzip file format specification version 4.3. RFC #1952, 1996.

[16] F. B. e Abreu, M. G. ao, and R. Esteves. Toward the design quality evaluation of object-oriented software. In *(ICSE'95)*, pages 44–57, Seattle, WA, USA, Apr. 23-30 1995.

[17] D. Eastlake and P. Jones. US secure hash algorithm 1 (SHA1), 2001.

[18] The complexity of type analysis of object oriented programs. In E. Jul, editor, *(ECOOP'98)*, volume 1445, pages 601–634, Brussels, Belgium, July 20–24 1998. Springer Verlag.

[19] N. Fenton and J. Bieman. *Software Metrics: A Rigorous and Practical Approach, Third Edition*. CRC Press, Inc., Boca Raton, FL, USA, 3rd edition, 2014.

[20] J. Y. Gil and T. Shragai. Are we ready for a safer construction environment? In J. Vitek, editor, *(ECOOP'08)*, volume 5142, pages 495–519, Cyprus, July7–11 2008. Springer Verlag.

[21] K. P. Grant and J. S. Pennypacker. Project management maturity: an assessment of project management capabilities among and between selected industries. *IEEE Transactions on Software Engineering*, 53(1):59–68, 2006.

[22] W. A. Harrison and K. I. Magel. A complexity measure based on nesting level. *ACM Sigplan Notices*, 16(3):63–74, 1981.

[23] H. Hata, O. Mizuno, and T. Kikuno. An extension of fault-prone filtering using precise training and a dynamic threshold. In *Proc. of the 2008 Int. working Conf. on Mining Software Repositories*, pages 89–98. ACM, 2008.

[24] A. Hindle, M. Godfrey, and R. Holt. Reading beside the lines: Indentation as a proxy for complexity metric. In *The* $16^{th}$ *IEEE Int. Conf. on Program Comprehension (ICPC'08)*, pages 133–142, June10-13 2008.

[25] A. Jbara and D. G. Feitelson. On the effect of code regularity on comprehension. In *ICPC*, pages 189–200, 2014.

[26] J. Kidd. *Object-Oriented Software Metrics: a practical guide*. Prentice-Hall, Englewood Cliffs, New Jersy 07632, 1994.

[27] S. Kim, E. J. Whitehead, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.

[28] R. Marinescu. Detection strategies: metrics-based rules for detecting design flaws. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 350–359, Sept 2004.

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

[30] B. Meyer. *Object-Oriented Software Construction*. International Series in Computer Science. Prentice-Hall, Englewood Cliffs, New Jersy 07632, 1988.

[31] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *(ICSE'08)*, pages 181–190, Leipzig, Germany, May10-18 2008. ACM.

[32] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors, *(ICSE'06)*, pages 452–461, Shanghai, China, May20-28 2006. ACM Press, New York, NY, USA.

[33] M. Paulk. *Capability maturity model for software*. Wiley Online Library, 1993.

[34] P. Piwowarski. A nesting level complexity measure. *ACM Sigplan Notices*, 17(9):44–50, 1982.

[35] P. L. Roden, S. Virani, L. H. Etzkorn, and S. Messimer. An empirical study of the relationship of stability metrics and the qmood quality models over software developed using highly iterative or agile software processes. In *Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on*, pages 171–179. IEEE, 2007.

[36] C. E. Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[37] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering*, 39(4):552–569, 2013.

[38] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? 30(4):1–5, 2005.

[39] Self-calibration of metrics of Java methods. In *(TOOLS'00 Pacific)*, pages 94–106, Sydney, Australia, Nov. 20-23 2000. Prentice-Hall, Englewood Cliffs, New Jersy 07632.

[40] F. Zhang, A. Mockus, Y. Zou, F. Khomh, and A. E. Hassan. How does context affect the distribution of software maintainability metrics? In *(ICSM'13)*, pages 350–359, Eindhoven, The Netherlands, Sept.22–28 2013. IEEE Computer Society.

[41] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100. ACM, 2009.

[42] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5):530–536, 1978.