Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods

Davy Landman*, Alexander Serebrenik*†, Jurgen Vinju*‡
* Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
*{Davy.Landman, Jurgen.Vinju}@cwi.nl
† Eindhoven University of Technology, Eindhoven, The Netherlands
a.serebrenik@tue.nl
‡ INRIA Lille Nord Europe, Lille, France

Abstract—Measuring the internal quality of source code is one of the traditional goals of making software development into an engineering discipline. Cyclomatic Complexity (CC) is an often used source code quality metric, next to Source Lines of Code (SLOC). However, the use of the CC metric is challenged by the repeated claim that CC is redundant with respect to SLOC due to strong linear correlation.

We test this claim by studying a corpus of 17.8M methods in 13K open-source Java projects. Our results show that direct linear correlation between SLOC and CC is only moderate, as caused by high variance. We observe that aggregating CC and SLOC over larger units of code improves the correlation, which explains reported results of strong linear correlation in literature. We suggest that the primary cause of correlation is the aggregation.

Our conclusion is that there is no strong linear correlation between CC and SLOC of Java methods, so we do not conclude that CC is redundant with SLOC. This conclusion contradicts earlier claims from literature, but concurs with the widely accepted practice of measuring of CC next to SLOC.

I. INTRODUCTION

In previous work [1] one of us analyzed the potential problems of using the Cyclomatic Complexity (CC) metric to indicate or even measure source code complexity per Java method. Still, since understanding code is known to be a major factor in providing effective software maintenance [2], measuring the complexity aspect of internal source code quality remains an elusive goal of the software engineering community. In practice the CC metric is used on a daily basis for this purpose precisely, next to another metric, namely Source Lines of Code (SLOC).

There exists a large body of literature on the relation between the CC metric and SLOC. The general conclusion from experimental studies [3]–[6] is that there exists a strong linear correlation between these two metrics for arbitrary software systems. The results are often interpreted as an incentive to discard the CC metric for any purpose that SLOC could be used for as well, or as an incentive to normalize the CC metric for SLOC (see below).

At the same time, the CC metric appears in every available commercial and open-source source code metrics tool† and is used in the daily practice of software assessment [7] and fault/effort prediction [8]. This avid use of the metric directly contradicts the evidence of strong linear correlation. Why go through the trouble of measuring CC?

Based on the related work on the correlation between CC and SLOC we have the following working hypothesis.

Hypothesis 1. There is strong linear (Pearson) correlation between the CC and SLOC metrics for Java methods.

Our results in investigating this are negative, challenging the external validity of the experimental results in literature as well as their interpretation. The results of analyzing a linear correlation are not the same for our (much larger) corpus of modern Java code that we derived from Sourcerer [9]. Based on these new results we will conclude that CC cannot be discarded based on experimental evidence of a linear correlation. Supporting the continued use of CC in industry next to SLOC to gain insight in the internal quality of software systems.

The interpretation of experimental results of the past is hampered by confusing differences in definitions of the concepts and metrics. In the following, Section II, we therefore focus on definitions and discuss the interpretation in related work of the evidence of correlation between SLOC and CC. We also identify four more hypotheses. In Section III we explain our experimental setup. After this, in Section IV, we report our results and in Section V we interpret them before concluding in Section VI.

II. BACKGROUND THEORY

A. Defining SLOC and CC

Although defining the actual metrics for lines of code and cyclomatic complexity used in this paper can be easily done, it is hard to define the concepts that they actually measure. This lack of precisely defined dimensions is an often lamented, classical problem in software metrics [10], [11]. The current paper does not solve this problem, but we do need to discuss it in order to position our contributions in the context of related work.

First we define the two metrics used in this paper.

Definition 1 (Source Lines of Code (SLOC)). A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines

†E.g. http://www.sonarqube.org/
containing program headers, declarations, and executable and non-executable statements [12, p. 33].

**Definition 2** (Cyclomatic Complexity (CC)). The cyclomatic complexity of a program is the maximum number of linearly independent circuits in the control flow graph of said program, where each exit point is connected with an additional edge to the entry point [13].

As explained by McCabe [13], the CC number can be computed by counting forks in a control flow graph and adding 1, or equivalently counting the number of language constructs used in the Abstract Syntax Tree (AST) which generate forks (“if”, “while”, etc.) and adding 1.

This last method is the easiest and therefore preferred method of computing CC. Unfortunately, which AST nodes generate decision points in control flow for a specific programming language is not so clear since this depends on the intrinsic details of programming language semantics. The unclarity leads to metric tools generating different values for the CC metric, because they count different kinds of AST nodes [14]. Also, derived definitions of the metric exist, such as “extended cyclomatic complexity” [15] to account for a different way of computing cyclomatic complexity. Still, the original definition by McCabe is sufficiently general. If we interpret it based on a control flow graph it is applicable to any programming language which has units to encapsulate a list of imperative control flow statements. So, for Java we count all AST nodes in a method which lead to one extra conditionally executed block of code in its control flow graph and add 1 (see Figure 1).

Note that we included the Boolean && and || operators because they have short-circuit semantics in Java rendering the execution of their right-hand sides conditional. Still, this is not the case for all related work. For completeness sake we therefore put the following hypothesis up for testing as well:

**Hypothesis 2.** The strength of linear correlation between CC and SLOC of Java methods is not significantly influenced by including or excluding the Boolean operators && and ||.

We expect that exclusion of && and || does not meaningfully affect correlations between CC and SLOC, because we expect Boolean operators are not used often enough and not in enough quantities within a single method to make a difference.

**B. Literature on the correlation between CC and SLOC**

We have searched for related work that experimentally investigates a correlation between CC and SLOC. Starting out with the table in Shepperd’s paper from 1988 [10], which also investigates Hypothesis 1, we included all the cited papers and extracted extra information. We excluded papers [27]–[29] comparing CC to Halstead’s token count or instruction count. Then, we used Google Scholar and the following process to find related work (±600 papers in the search results):

1) Filter to peer-reviewed work only.
2) Scan all titles citing Shepperd’s paper [10] and Shepper & Ince paper [30].
3) Scan the titles, citing McCabe’s original paper [13], of the 200 most relevant papers matching the “Empirical” search query.
4) Scan the titles, citing CK metrics [31], of the 200 most relevant papers matching the “Empirical” search query.

After this we filtered the papers down to direct comparisons between SLOC and CC or comparisons between WMC and SLOC where CC was used as the weight. Also, we ignored studies relating system size to the sum of all CC over an entire system (see below). Resulting in a total of 15 papers. If a paper reported R instead of $R^2$ as strength of the Pearson correlation we computed the corresponding $R^2$ by squaring R (see Section III-C3). The result is summarized in Table I.

**C. Aggregating CC over larger units of code**

CC applies to control flow graphs. As such CC is defined when applied to code units which have a control flow graph. This has not stopped researchers and tool vendors to sum the metric over larger units, such as classes, programs, files and even whole systems. We think that the underlying assumption is that indicated “effort of understanding” per unit would add up to indicate total effort. However, we do not clearly understand what such sums mean when interpreted back as an attribute of control flow graphs, since the compositions of control flow graphs they represent do not actually exist.

Perhaps not surprisingly, in 2013 Yu et al. [32] found a Pearson correlation of nearly 1 between whole system SLOC and the sum of all CC. They conclude the evolution of either metric can represent the other because of this. One should keep in mind, however, that choosing the appropriate level of aggregation is vital for validity of an empirical study: failure to do so can lead to an ecological fallacy [33]. Similarly, the choice of an aggregation technique can greatly affect the correlation results [34]–[36].

Curtis, Carleton [11] and Shepherd [10] were the first to state that without a clear definition of what source code complexity is, it is to be expected that metrics of complexity are bound to measure (aspects of) code size. Any metric that counts arbitrary elements of source code sentences, actually measures the code’s size or a part of it. Both Curtis & Carleton and Shepherd conclude that this should be the reason for the strong correlation between SLOC and CC. However, even though CC is a size metric; it still measures a different part of the code. SLOC measures all the source code, while CC measures only a part of the statements which govern control flow. Even if the same dimension is measured by two metrics that fact alone does not fully explain a strong correlation between them. We recommend the work of Abran [37], for an in-depth discussion of the semantics of CC.

Table I lists which studies use which level of aggregation. Note that the method of aggregation is sum in all described

---

3In this context a “program” means a unit of code like a procedure in Pascal, function in C, method in Java, sub-routine in Fortran, program in COBOL.

4Google Scholar determines relevancy based on the text of the paper, its publication source, the authors, and the amount of citations.

5In some cases CC was used as the weight for the WMC metric.
<table>
<thead>
<tr>
<th>Year</th>
<th>Level</th>
<th>Correlation</th>
<th>Language</th>
<th>Corpus</th>
<th>$R^2$</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>[16]</td>
<td>1979 Subroutine</td>
<td>SLOC vs CC</td>
<td>Fortran</td>
<td>27 programs with SLOC ranging from 25 to 225</td>
<td>0.65</td>
<td>The first result is for a CC correlation on subroutine level, and the second result is on a program level.</td>
</tr>
<tr>
<td>[3]</td>
<td>1979 Program</td>
<td>SLOC vs CC</td>
<td>Fortran</td>
<td>27 programs with SLOC ranging from 36 to 57</td>
<td>0.41</td>
<td></td>
</tr>
<tr>
<td>[4]</td>
<td>1979 Program</td>
<td>log(Clause Count) vs log(CC)</td>
<td>PL/I</td>
<td>197 programs with an median of 54 statements.</td>
<td>0.90</td>
<td>Clause count is similar to Logical Lines of Code (LLOC).</td>
</tr>
<tr>
<td>[17]</td>
<td>1979 Subroutine</td>
<td>LOC vs CC</td>
<td>Fortran</td>
<td>26 subroutines</td>
<td>0.90</td>
<td></td>
</tr>
<tr>
<td>[18]</td>
<td>1980 Module</td>
<td>LOC vs CC</td>
<td>Fortran</td>
<td>10 modules, 339 SLOC</td>
<td>0.90</td>
<td></td>
</tr>
<tr>
<td>[19]</td>
<td>1981 Module</td>
<td>SLOC vs CC</td>
<td>Fortran</td>
<td>25.5K SLOC over 137 modules</td>
<td>0.65</td>
<td>One of the larger studies, referenced in Shepperd and Ince’s paper, however without the correlation.</td>
</tr>
<tr>
<td>[5]</td>
<td>1984 Module</td>
<td>SLOC vs CC</td>
<td>Fortran</td>
<td>517 code segments of one system</td>
<td>0.94</td>
<td>No correlation between module SLOC and module CC. Then the authors grouped modules into 5 buckets (by size), and calculated the average CC per bucket. Over these 5 data-points they reported the high correlation.</td>
</tr>
<tr>
<td>[20]</td>
<td>1987 Program</td>
<td>SLOC vs CC</td>
<td>Fortran</td>
<td>255 student assignments, range of 10 to 120 SLOC</td>
<td>0.82</td>
<td>Study comparing 31 metrics, showing histogram of the corpus, and scatter-plots of selected correlation. Not included in either of Shepperd’s papers.</td>
</tr>
<tr>
<td>[21]</td>
<td>1989 Routine</td>
<td>SLOC vs CC</td>
<td>Pascal &amp; Fortran</td>
<td>1 system, 4500 routines, 2372K SLOC, 112K SLOC</td>
<td>0.72</td>
<td>The first result was for Pascal, the second Fortran.</td>
</tr>
<tr>
<td>[22]</td>
<td>1991 Module</td>
<td>SLOC vs CC</td>
<td>Pascal &amp; COBOL</td>
<td>19 systems, 824 modules, 150K SLOC</td>
<td>0.90</td>
<td>The paper also compared different variants of CC.</td>
</tr>
<tr>
<td>[23]</td>
<td>1993 Program</td>
<td>LOC vs CC</td>
<td>COBOL</td>
<td>3K programs</td>
<td>0.76</td>
<td>A very interesting study discussing the confounding factor of size in the Weighted Methods per Class (WMC) metric. For their study they used CC as weight, and whilst certainly not the main point of their study, they also reported the correlation between SLOC and WMC.</td>
</tr>
<tr>
<td>[24]</td>
<td>2001 Class</td>
<td>SLOC vs CC</td>
<td>C++</td>
<td>174 classes</td>
<td>0.77</td>
<td></td>
</tr>
<tr>
<td>[25]</td>
<td>2009 File</td>
<td>log(LOC) vs log(CC)</td>
<td>Java, C, C++</td>
<td>2200 Projects from SourceForge</td>
<td>0.78</td>
<td>After discussing the distribution of both LOC and CC and their wide variance, they calculate a repeated median regression and recalculate $R^2$: 0.87, 0.93, and 0.97.</td>
</tr>
<tr>
<td>[26]</td>
<td>2010 File</td>
<td>log(SLOC) vs log(CC)</td>
<td>C</td>
<td>ArchLinux packages, 300K files, of which 200K non header files.</td>
<td>0.59</td>
<td>Initially they observed a low correlation between CC and SLOC, further analysis revealed header files as the cause. The second correlation is after removing these. The authors show the influence of looking at ranges of SLOC on the correlation.</td>
</tr>
<tr>
<td>[6]</td>
<td>2013 Method</td>
<td>LOC vs CC</td>
<td>C</td>
<td>Linux kernel</td>
<td>0.77</td>
<td>The authors show the scatter-plot of LOC vs CC, and report on a high correlation. Hereafter they limit to methods with a CC higher than 100, for these 108 methods they find a much lower correlation to SLOC.</td>
</tr>
</tbody>
</table>
experiments. The table suggests that higher levels of aggregation correspond to higher correlations. This brings us to our third hypothesis:

**Hypothesis 3.** The higher the level of aggregation—the more units we add up the CC for—the more this aggregated sum correlates with aggregated SLOC.

If this is true this would explain the high correlation coefficients found in literature when aggregated over larger units of code; it would be computing the sum over code units that causes it rather than the metric itself.

**D. Data transformations**

Hypothesis 1 is motivated by the earlier results from the literature in Table I. Some newer results of strong correlation are only acquired after a power transform (log) on both variables [4], [25], [26]; indeed, power transform can help to normalize distributions that have a positive skew [38] (which is the case both for SLOC and for CC). A strong correlation which is acquired after power transform does not directly warrant dismissal of one of the metrics, since any minor inaccuracy of the linear regression is amplified by the reverse power transform back to the original data. Nevertheless, a fourth hypothesis to confirm or deny results from literature is:

**Hypothesis 4.** After a power transform on both the SLOC and CC metrics of a Java method, the Pearson correlation is higher than the Pearson correlation on the untransformed data.

The power transform is motivated by observing skewed long tail distributions of SLOC and CC [25], [26], [39]. This puts all related work on smaller data sets which do not interpret the shape of the distributions in a different light. How to interpret these older results? Such distributions make relatively “uninteresting” smaller Java methods dominate any further statistical observations. Therefore we should also investigate what happens if we ignore the smallest methods. Conversely, the enormously large methods in the long tails of both distributions may have a confounding effect.

**Hypothesis 5.** The strength of the linear correlation between SLOC and CC is improved by zooming in on quantiles of the SLOC data.

This hypothesis was inspired by Herraiz and Hassan’s observation on the correlation [26]. They reported an increasing correlation for the higher ranges of SLOC. The question is interesting given the corpus which has a significant amount of data in the tails of the two distributions.

We compare our results with the results of Table I in Section IV and we discuss the interpretation of the cited results when we interpret our own in Section V.

**III. EXPERIMENTAL SETUP**

In this section we discuss how the study has been set up. To perform empirical evaluation of the relation between SLOC and CC for methods in object-oriented programs we needed a large corpus of such methods. To construct such a corpus we have processed Sourcerer [9], a collection of 19 K open source Java projects (Section III-A). Then SLOC and CC have been computed for each method in the corpus (Section III-B), and performed a statistical analysis of the data (Sections III-C and IV).

**A. Preparing the Corpus**

Sourcerer [9] is a large corpus of open source Java software. It was constructed by fully downloading the source code of 19 K projects, of which 6 K turned out to be empty.

**Remove non-Java files** While Sourcerer contains a full copy of each project’s Source Code Management (SCM), because of our focus on Java, we excluded all non-Java files.

**Remove SCM branches** When Sourcerer was compiled the whole SCM history was cloned. In particular, this means that multiple versions of the same system are present. However, inclusion of multiple similar versions of the same method would influence our statistics. Therefore, we removed all directories named /tags/, /branches/, and /nightly/.

**Remove duplicate projects** Sourcerer projects have been collected from multiple sources including Apache, Java.net, Google Code and SourceForge. Based on Sourcerer’s metadata we detected 172 projects which were extracted from multiple sources—for example from both SourceForge and Google Code. Similarly to removal of SCM branches we have kept only one version of each project, in this case we chose the largest version in bytes.

**Manually reviewed duplicate files** We calculated the MD5 hash per file. The 278 projects containing more than 300 duplicate files (equal hash) were manually reviewed and fixed in case the duplication could be explained. Common reasons were non-standard SCM structure and committed libraries. A list of the duplicate projects and manually removed directories is available online.

**Remove test code** Finally, we have decided to remove test code as it is a priori not clear whether test code exhibits the same relation between SLOC and CC as non-test code. Hence, we excluded all directories named /test/.

Performing these steps we have reduced the 390 GB corpus to 14.3 GB. The resulting corpus has been made publicly available [40].

**B. Measuring SLOC and CC**

While numerous tools are available to measure SLOC and CC on a file level, to perform our study we require to calculate SLOC and CC per method and to precisely control the definition of both metrics. We use the M3 framework [41], which is based on the Eclipse JDT, to parse the full Java source code and identify the methods in the corpus. This also generates full ASTs for each method for further analysis. As explained in the introduction, Figure 1 depicts the source code of computing the CC from the AST of a method. The code recursively traverses the AST and matches the enumerated nodes, adding 1 for each node that would generate a fork in the Java control flow graph.

---

6 All code and data is available at: http://www.wi.wi/~landman/csmc2014/

7 e.g., http://cloc.sourceforge.net/, http://www.sonarqube.org/

8 http://www.eclipse.org/jdt
int calcCC(Statement impl) {
    int result = 1;
    visit (impl) {
        case \text{\texttt{\textbf{if}(...) : result += 1;}}
        case \text{\texttt{\textbf{if}(..., ...) : result += 1;}}
        case \text{\texttt{\textbf{case}(...) : result += 1;}}
        case \text{\texttt{\textbf{do}(...) : result += 1;}}
        case \text{\texttt{\textbf{while}(...) : result += 1;}}
        case \text{\texttt{\textbf{for}(..., ...) : result += 1;}}
        case \text{\texttt{\textbf{foreach}(..., ...) : result += 1;}}
        case \text{\texttt{\textbf{catch}(..., ...) : result += 1;}}
        case \text{\texttt{\textbf{conditional}(..., ...) : result += 1;}}
        case \text{\texttt{\textbf{infix}("\&\&", ...) : result += 1;}}
        case \text{\texttt{\textbf{infix}("|\|", ...) : result += 1;}}
    } return result;
}

Fig. 1. RASCAL source code to calculate the CC of a given method. The visit statement is a combination of a regular switch and the visitor pattern. The cases pattern match on elements of the AST.

For SLOC we decided not to depend on the information in the Eclipse ASTs (ASTs are not designed for precisely recording the lexical syntax of source code). Instead we use the ASTs only to locate the source code of each separate method. To compute its SLOC we defined a grammar in RASCAL [42] to tokenize Java input into newlines, whitespace, comments and other words. The parser produces a list of these tokens which we filter to find the lines of code that contain anything else but whitespace or comments. We tested and compared our SLOC metric with other tools measuring full Java files to validate its correctness.

C. Visualization & Statistics

1) Distributions: Before comparing SLOC and CC, we describe the distributions in our data using histograms and descriptive statistics (median, mean, min and max). The shape of distributions does have an impact on the correlation measures used, as explained above. All further results should be interpreted with these distributions in mind.

2) Scatter plots: Scatter plots with SLOC on the x-axis and CC on the y-axis represent the data in a raw form. Due to the long tail distributions of both CC and SLOC, the data is concentrated in the lower left quadrant of the plots and many of the dots are placed on top of each other. Therefore, we also show log-log scatter plots and use a 256 step gray scale gradient, also on a log scale, to depict how many dots are placed on top of each other. Nevertheless, it should be noted that this gradient has a limited resolution and as such can still hide the true impact of the skewness of the distribution.

3) Correlation: Most related work, if reported, uses Pearson product-moment correlation coefficient (hereafter Pearson correlation), measuring the degree of linear relationship between two variables. The square of Pearson correlation is called the coefficient of determination ($R^2$). $R^2$ estimates the variance in the power of one variable to predict the other using a linear regression. Hereafter we report the $R^2$ to describe a correlation.

It should be noted that Pearson correlation works best when the distribution of the independent variable is close to normal since it depends on a measure of the distance of all points to a best fit linear model. To compensate for the skewness of the distribution, one can apply a power transform and then compute the Pearson correlation [4], [25], [26]. The important matter of interpreting the results after a power transform back to the original data is discussed in Section V.

Other researchers have transformed the data using more advanced methods in order to improve the chances for linear correlation. For example, using Box-Cox transformation [26] or performing the Repeated Median Regression (RMR) method on a random sample [25]. Box-Cox is a power transform similar to the basic power transform using log. We have chosen to stick with the simpler method. RMR is useful to find a better linear model, but it entails a lossy transformation. The median regression method reduces the effect of random measurement errors in the data by computing a running median. We do not have random errors in the CC or SLOC measurements, so a running median would hide interesting data. Therefore, RMR is outside the scope of this paper.

If no linear model is to be expected, or is found using Pearson’s method, we use Spearman’s rank-order correlation coefficient (hereafter Spearman correlation or $\rho$). Similarly to the Pearson correlation, Spearman’s correlation is a bivariate measure of correlation/association between two variables. However, opposed to the Pearson correlation, Spearman’s correlation is employed with rank-order data, and therefore, measuring the degree of monotone relationship between two variables. We apply this method only for completeness sake, since it does not generate a predictive model which we could use to discard one of the metrics.

IV. RESULTS

In this section we report the raw data resulting from our experiments and the statistics we applied to it.

A. Distributions

Figure 2 shows the histogram of SLOC per project and Table II describes this distribution. The corpus contains 17.8M methods spread out over 2M files.

Figure 3 and Figure 4 show the distribution of SLOC and CC per method. Table III describes their distributions. We observe skewed distributions with a long tail. The third moment skewness of SLOC is 272.78 and 1.05 after the power transform.
This means that the mean values are not at all representative of the corpus, and that the smallest methods dominate the data. For SLOC, 8.9 M of the methods have 3 SLOC or fewer. This is 50% of all data points. There are 1.2 M methods with 1 or 2 SLOC, these are the methods with an empty body, in two different formatting styles or (generated) methods without newlines. The other 7.7 M methods of 3 SLOC contain the basic getters, setters, and throwers pattern frequently seen in Java methods—often called one-liners.

Similarly for CC, 15.4 M methods have a CC of 3 or less. This is 86% of all data points. There are 11.7 M methods without any forks in the control flow (1 CC). We observe that the lion’s share of Java methods adhere to the common CC thresholds of 10 (96.90%) [13] or 15 (98.60%) [43].

B. Scatter plots

Figure 5a shows a zoomed-in (CC ≤ 300 and SLOC ≤ 750) scatter-plot of the methods in our corpus. Due to the skewed-data, this figure still shows 99.99% of all data points. Figure 5b show the same scatter-plot in a log-log space, allowing to show more data. The two gray lines in both figures shows the linear regressions before and after the power transform which will be discussed later. The logarithmic gray scale gradient of the points in the scatter-plot visualizes how many methods have that combination of CC and SLOC: the darker, the more data points. Figure 6 shows an even more zoomed-in range of the scatter-plots, in these box plots we can more clearly see the variance of CC. However such a plot does not scale to larger ranges.

The plots show a widely scattered and noisy field, with a high concentration of points in the left corner. The outline of this concentrations hints at a positive (linear) monotone relation, but that should be taken with a grain of salt since the same outline is bounded by the minimum CC number (1) and the expected maximum CC number (CC is usually not higher than SLOC given a source code layout of one conditional statement on a single line).

We do find some points above the expected maximum CC, which we found out to be generated code and code with dozens of Boolean operators on one single line.

The human eye identifies a number of “lines” in the scatter plots (Figure 5). The 1 CC line shows mostly initialization code that uses no control flow statements at all. The darkest line in the “each conditional on a new line” group, identifying methods which consist almost only of control flow statements. The other two lighter lines—less easy to see—identify common coding idioms (chains of if-then-else statements and longer switch cases of different sizes).

C. Pearson correlation

In Table IV, the first row shows the Pearson correlation over the whole corpus. The $R^2$ of SLOC and CC is 0.43. Figure 5a depicts this linear fit, $CC = 0.88 + 0.16 \cdot SLOC$, as a solid gray line. This $R^2$ is much lower than the related work in Table I, even if we focus on the metric at the subroutine/function/method level.

The Pearson correlation after a power transform (log-log) showed higher numbers, which are more in line with related work that also applies a power transform [4], [25], [26]. The fit, the dashed line in Figure 5a, is $\log_{10}(CC) = \log_{10}(SLOC) - 0.29 + 0.67 \cdot \log_{10}(SLOC) \Rightarrow CC = 10^{-0.29 + SLOC^{0.67}}$. More on the interpretation of this transform and the results in Section V.

As discussed earlier, the data is skewed towards small methods and simple control flow graphs. Since 50% of the data has a SLOC between 1 and 3, these points have a high influence on correlation. Is the relation between SLOC and CC interesting, for methods with fewer than 4 lines of code? Therefore Table IV also shows the Pearson correlations for parts