Reading Beside the Lines: Using Indentation to Rank Revisions by Complexity

Abram Hindle, Michael W. Godfrey, Richard C. Holt

Software Architecture Group
University of Waterloo
Waterloo, Ontario
Canada

Abstract

Maintainers often face the daunting task of wading through a collection of both new and old revisions, trying to ferret out those that warrant detailed inspection. Perhaps the most obvious way to rank revisions is by lines of code (LOC); this technique has the advantage of being both simple and fast. However, most revisions are quite small, and so we would like a way of distinguishing between simple and complex changes of equal size. Classical complexity metrics,such as Halstead's and McCabe's, could be used but they are hard to apply to code fragments of different programming languages. We propose a language-independent approach to ranking revisions based on the indentation of their code fragments. We use the statistical moments of indentation as a lightweight and revision/diff friendly metric to proxy classical complexity metrics. We found that ranking revisions by the variance and summation of indentation was very similar to ranking revisions by traditional complexity measures since these measures correlate with both Halstead and McCabe complexity; this was evaluated against the CVS histories of 278 active and popular SourceForge projects. Thus, we conclude that measuring indentation alone can serve as a cheap and accurate proxy for computing the code complexity of revisions.

Key words: indentation; complexity; McCabe; Halstead; metrics

1. Introduction

Assessing the complexity and maintainability of changes to large evolving software projects presents many technical challenges. Such systems are often heterogeneous in that they contain sub-components written in multiple languages, and are stored using a variety of repository mechanisms. However, maintainability metrics are commonly language dependent, and computing them requires tools that typically assume access to the full definitions of the software entities, access that we might not have if we are evaluating newly submitted patches.

This paper focuses on characterizing the complexity of revisions (and by assumption their maintainability) by measuring the indentation of the changed code. The patches and revisions are code fragments that represent the difference between old and new versions of software entities. Consequently, measuring indentation is relatively language neutral and does not require complete compilable sources.

We have focussed on revisions because they are the currency of ongoing development. Developers and managers typically want to understand what has happened to the code base since the last revision. They want assurances that new code will implement the desired enhancement or bug fix, and will not break the existing system. Current analysis techniques and tools commonly assume access to compilable source-code.

Email addresses: ahindle@cs.uwaterloo.ca (Abram Hindle), (Abram Hindle), migod@cs.uwaterloo.ca (Michael W. Godfrey), (Michael W. Godfrey), holt@cs.uwaterloo.ca (Richard C. Holt), (Richard C. Holt)

By contrast we seek ways of reliably and efficiently analyzing arbitrary code fragments — not necessarily compilable — representing revisions to source code. If we can measure or estimate the complexity of source code changes we can better rank the maintainability of these changed code fragments, such as revisions in a source control repository. In turn, we can help maintainers identify complex and error prone patches; this can be particularly valuable when analyzing and merging branches.

Maintaining software is costly, yet the costs are hard to predict. It is generally believed that the complexity and size of code dominates most other factors with respect to maintenance cost; consequently, most metrics for maintainability combine traditional code complexity metrics, such as McCabe's Cyclomatic Complexity (MCC) [1] and Halstead's Complexity Metrics(HCM) [2], with size measures such as Lines of Code (LOC) (see, for example, the Maintainability Index [3]). However, measuring the likely maintainability of source code revisions presents some challenges. First, traditional complexity metrics are typically language dependent and cannot easily be applied to code fragments, such as those found in source revisions. Also, while using LOC alone is feasible for measuring code fragments, it provides only a coarse solution, as LOC does not distinguish between revisions of the same size, and the vast majority of revisions comprise only a few lines of code. By contrast, the statistical moments of indentation are relatively language agnostic, and are easy to calculate as they do not require grammar-level semantic knowledge of the languages of the source code being analyzed.

Variance and standard deviation of indentation and the summation of indentation should be good indicators of the complexity of source code. Varying indentation across a code fragment can indicate there are changes at multiple levels of scope, code with different levels of nesting. Differences in how code is nested is likely to indicate changes in branching which implies changing complexity. In the case of procedural code, such as C, indentation can indicate functions and control structures such as conditionals and loops. For Object-oriented (OO) languages such as C++ and Java, indentation can indicate the depth of encapsulation via classes, subclasses, and methods. For functional languages such as OCaml, Scheme and Lisp, indentation indicates new scope, new closures, new functions, and new expressions.

Branches in source code often imply a nested block structure, resulting in a larger variance or standard deviation of indentation [4]; thus statistical moments of indentation (the set of summary statistics about the indentation of a change) serves as a proxy to McCabe's Cyclomatic Complexity as MCC counts branching paths in code. The summation of indentation proxies LOC and complexity as it grows with both line count and indentation depth. Our studies suggest that most code is shallow, with no more than two levels of indentation (LIL); very deep code is rare (see also Section 3.2).

To evaluate indentation metrics we must first see if indentation is indeed regular. In Section 3.2, we show that indentation is very regular across all the languages we evaluated. In some languages, such as Python, indentation has semantic value to the program. However, this is not true for most programming languages in common use today: C, C++, Java, Perl, C#, PHP. Instead, indentation is used as a best practice to aid in the readability of the source code [5, 6].

This work extends our previous investigations on this topic [7] by more thoroughly examining the effectiveness of ranking-by-indentation metrics to ranking-by-complexity. In Section 6 we evaluated how well ranking-by-indentation emulated proxied ranking-by-complexity on thousands of sample cases. We also compared indentation metrics against other revision-based metrics such as code-characters per line and line length. We investigated how the indentation metrics on old code and new code, and their differences were related to both MCC and HCM and the difference in MCC and HCM.

In order to meet classical definition of statistical moments as well as provide more useful summary statistics we extended our measurements by adding skewness (SKEW), kurtosis (KURT), minimum depth (MIN), maximum depth (MAX) and the geometric mean (GEO) of indentation to our indentation statistics. Geometric mean of indentation, minimum depth, and maximum depth were added as mean nesting and maximum nesting have been used previously in complexity metrics [8, 9], which is what indentation is often meant to indicate).

In this paper, we show that revisions can be ranked by their statistical moments of indentation as a proxy for ranking by complexity; this is because indentation correlates strongly with MCC and HCM. We also show linear and non-linear combinations of these measures which allow us to rank by complexity even better.

The contributions in this paper include:

- Evidence that ranking revisions by statistical moments of indentation correlates strongly with rankings by traditional complexity and maintainability measurements.
- New metrics for measuring changes to source code.
- An empirical survey of indentation of popular open-source software projects found on Source-Forge.
- Evidence which shows that measuring indentation is computationally cheaper than applying classical complexity or maintainability metrics.
- A comparison of the effectiveness of Indentation versus LOC, Code Characters and Line Length for ranking chunks of code by complexity.
- An investigation of the relationships between the old and new code and the resulting change in complexity induced by the change.

1.1. Previous Work

Indentation is often promoted for helping program readability [6] and defining structure [10] in code. It can be used to trace a program's flow [11] and has been shown to be generally helpful for program comprehension [12].

Indentation can be used for software visualization, in order to provide an overview of a program's structure [13]. Gorla et al. [14] uses inappropriate indentation as a code quality metric. Some have compared indentation characters to the non-indentation characters [15] and others have measured the horizontal spacing (indented/non-indented) of source code [16]. Other uses of measuring indentation include plagiarism detection [17]. In some cases nesting depth, related to indentation, has been used to characterize complexity [8, 9].

Many software complexity metrics have been proposed by the research community [18]. Two of the best known and most often used in practice are McCabe's Cyclomatic Complexity (MCC) [1] and Halstead's complexity metrics(HCM) [2]. We are interested in these two complexity metrics in particular because many studies, such as that of Oman et al. [3], use these metrics in calculations of maintainability metrics.

McCabe Cyclomatic Complexity (MCC) counts branching and control flow in a program, which means counting control flow structures and tokens. The accepted styles of many programming languages dictate that code within block structures such as if blocks, branches, loops and exceptions should be indented; this suggests that indentation indicates branching, which in turn suggests that there is a potential correlation with MCC. Other complexity metrics, such as HCM, measure the number of unique operators and operands. Although each metric measures something different they all seem to be correlated with LOC [19]. We applied MCC and HCM to source code revisions, which relates to modification-aware change metrics as discussed by German et al. [20]. This work is heavily based our work presented at ICPC 2008 [7].

McCabe's Cyclomatic Complexity (MCC) is a measurement of code that indicates how much branching or how many paths there are through a block of code. In this paper we will use the definition of MCC as one plus the number of possible branch points in a block of code. This differs from the classical definition of McCabe complexity because their count is for a modular unit, like a node, function, procedure, file; instead we are calculating MCC for a continuous block of changed code in a revision. There could be cases where a diff-chunk crosses modular boundaries (entire functions) thus making our MCC slightly different than classical MCC. The branch points we consider are heads of loops, if statements, any other decision statement, such as switches, and function definitions.

Halstead Complexity Metrics (HCM) take an almost information-theoretic approach to measuring software. They suggest that the information content of the source code, the number of tokens (operators or operands), and the number of unique tokens is related to how much effort it takes to write that source code and how difficult it was to create. HCM seems particularly intuitive as a complexity measure for object-oriented (OO) code that employs many identifiers (operands) and operators (including dispatch or overloaded operators); such code is often inherently more complex (and shorter) than semantically similar procedural code which may have fewer operands and operators.

The Halstead Complexity Metrics are: HLEN, HVOCAB, HVOL, HDIFF, and HEFFORT. Halstead Length (HLEN) is the total number of operators and operands in the code, including repeated ones. Halstead Vocabulary (HVOCAB) is the total number of unique operators and operands. Halstead Volume (HVOL) is HLEN multiplied by the log_2 of HVOCAB; HVOL is similar to the magnitude of information in the system, thus a block with lot of repetition but relatively few unique operators and operands will have a lower HVOL than one of the same size (i.e., HLEN) with more unique operators and operands. Halstead Difficulty (HDIFF) models the difficulty of writing or changing a piece of code; HDIFF is defined as the total number of unique operators multiplied by total operands divided by total unique operands divided by 2. Halstead effort (HEFFORT) estimates the effort or time required to re-implement a block of code from scratch. HEFFORT is calculated as HDIFF multiplied by HVOL.

1.2. Motivation

Both developers and project managers need effective metrics for ranking revisions. Given a set of changes, they need to be able to quickly ascertain which ones are the most complex and therefore the most risky. Most traditional code complexity metrics require access to the full sources to be computed; however, such access is not always available easily to developers who are scanning through the recent revisions. Consequently, metrics are needed that operate on patches, revisions to source code, diffs and diff-chunks because that may be all the information that is available.

Informally, we can define the semantic awareness of a metric to be the amount and depth of information it requires about the system it is measuring. Semantic awareness may include simple facts about the source code such as the number of files / lines / characters, or it may require a deeper understanding of the programming language at the lexical, syntactic, or semantic level. For example, LOC has relatively low semantic awareness of the code, since it needs only to be able to detect line breaks in the code and be able to detect (and ignore) comments. Metrics such as MCC and HCM have stronger semantic awareness since they require lexical and semantic processing. By contrast, computing the statistical moments of indentation requires very low semantic awareness, since all that is needed is the ability to measure beginning whitespace on each line of code.

Metrics vary in their difficulty of implementation and their computational performance. For example LOC can be implemented with a simple character search. Indentation measurements can be implemented and indentation can be measured using a simple scanner, whereas token-based metrics, such as HCM, require a tokenizer for each particular language studied. Using our tool we found that tokenizing took about 2 to 4 times more time than just counting indentation.

The rest of this paper has the following structure: in Section 2, we introduce our methodology, in Section 3, we provide an overview of the indentation we encountered, in Section, 4 we show how the indentation of diffs relate to complexity metrics of the revisions, in Section 5, we extend our previous analysis to other line based measurements, in Section 6, we experiment with ranking revisions, in Section 7, we discuss our results, in Section 8, suggest threats to validity, and finally in Section 9, we discuss our conclusions.

2. Methodology

Our methodology can be summarized as:

1. Choose projects to study and download them. We downloaded the CVS repositories of the top 250 Most Active Source Forge projects and top 250 Most Popular (downloaded) SourceForge projects (as indicated by SourceForge on their website). This resulted in a total of 278 projects since the two groups overlap and not all projects had CVS repositories available at the time.
2. Process the revisions. For each file, we extract each individual revision and we analyze the indentation of the new code.
3. Calculate the complexity metrics (MCC and HCM) for each revision.

```
1 > void square(int * arr, int n) {
2 >     int i = 0;
3 >     for ( i = 0 ; i < n ; i++ ) {
4 >         arr[i] *= arr[i];
5 >     }
6 > }
```

Metric	Raw	Logical
LOC	6	6
AVG	3.33	0.833
MED	4	1
STD	2.75	0.687
VAR	9.07	0.567
SUM	20	5
SKEW	0.23	0.23
KURT	-0.89	-0.89
MCC	2	2
HVOL	142	142
HDIFF	15	15
HEFFORT	2127	2127

Figure 1: An example diff-chunk with corresponding indentation and complexity measures. This example depicts a function being added. The first set of metrics are calculated from measuring the indentation of the code example (see Section 1). MCC is McCabe's Cyclomatic complexity of the code example (1 loop, 1 function). HVOL, HDIFF, HEFFORT are Halstead Complexity metrics.

4. Correlate the indentation measurements and the complexity metrics. We then analyze the results and extract correlations between complexity and the indentation metrics.

The 278 projects spanned a variety of application domains. The studied projected included: Swig (a foreign function interface library); GNU-Darwin (the open-sourced OSX kernel), Amanda (a backup program); PHP-Nuke (a popular CMS); Clam-AV (an open-source anti-virus package); GNU-Cash (a personal finances application); Battle for Wesnoth (a strategy game); and Enlightenment (a window manager).

2.1. Extraction and Measurement

For each revision to C, C++, Java, Perl, PHP, and Python files, we analyzed both the new and revised code. In case a revision was not contiguous we analyzed each separate contiguous code blocks from the change, which we call "diff chunks"; Figure 1 shows an example. We extracted about 13 million diff-chunks, evaluating only the changed-to code (i.e., the new code). Diff-chunks are parts of diffs, not the full diff, unless the full diff was contiguous. We did not measure the initial commits because they would skew the results as these are often full files that are imported, and there were no previous revisions to revise. We measured raw indentation and then calculated the logical indentation as described in Section 3.1.

Our data consisted of 240, 473 files and 14, 466, 882 diff-chunks: 46, 658 C files with 4, 699, 493 diff-chunks, 28, 698 C++ files with 3, 164, 565 diff-chunks, 62, 067 Java files with 2, 559, 896 diff-chunks, 65, 261 .h files with 1, 767, 274 diff-chunks, 26, 664 PHP Files with 1, 551, 746 diff-chunks, 8, 125 Python files with 492, 476 diff-chunks, and 3, 000 Perl files with 231, 432 diff-chunks. The average length of a diff-chunk was 4.8 lines, while the median length was 1, with a standard deviation of 72.2 (quite high). While average diff-chunk length was relatively consistent across languages, the standard deviation of diff-chunk length was not. Diff-chunks of C, C++, Perl and .h files had a high standard deviation above 60, while Java, PHP and Python had standard deviations less than 55. The high standard deviations are partially due to the power-law-like distributions of diff-chunk lengths, which are not Gaussian/normal distributions.

Diff-chunks are not always structurally complete. In fact since most diff-chunks are quite short, they often consist of one line of code. In some cases the line will not be a full statement or expression. Often one will have a change that will not contain the full structure, for instance a change to an if statement could change the whole statement, the head of it, the tail, the middle, just about any possible continuous permutation. A diff-chunk could also represent change to comments or data which never executes. It could be difficult to tell if some of the structure is missing. Thus running these changes through a source-code metric like MCC could be problematic. This difficulty alone suggests the inherent value of studying lightweight metrics which can tolerate this difficulty

We consider raw indentation to be the physical leading white space on each line. Logical indentation is the depth of indentation that the programmers meant to portray. In most cases 4 spaces, 8 spaces, or a tab could all be equivalent to one unit of logical indentation. Logical indentation is the unit in which the depth of indentation is measured, where as raw indentation composes logical indentation. For example, if a line consisted of “␢ ␢ `def sqr`”, where ␢ was a leading space, we would say it has 2 units of raw indentation but had 1 unit of logical indentation if the files used that model of indentation.

Statistical moments are a set of statistics which include the mean (the expected value), the variance (the dispersion or spread), the standard deviation (distribution or spread around the mean), the skewness (the lopsidedness of a distribution) and kurtosis of a distribution (the peakedness of a distribution). While we measure the distribution of indentation in a diff-chunk, we also calculate and measure their statistical moments and summary statistics. The measures we use include: mean or average (AVG), standard deviation (STD), variance (standard deviation squared) (VAR), skewness (SKEW), kurtosis (KURT), median (MED, the middle value), minimum indentation depth (MIN), maximum indentation depth (MAX), sum of indented lines (SUM), lines of code (LOC), and the geometric mean (GEO, a kind of logarithmic mean).

We measured the LOC of each diff-chunk, and then we measured the statistics, the statistical moments, of raw and logical indentation of the diff-chunk. The measures of raw indentation are: STD, AVG, MED, VAR, SUM, MIN, MAX, GEO, SKEW, and KURT. The measures of logical indentation are: LSTD, LAVG, LMED, LVAR, LSUM, LMIN, LMAX, LGEO, LSKEW, and LKURT. Also, we counted the frequency of indentation depth to produce histograms. Figure 1 provides an example of our measurement of a diff-chunk. In this paper we added measures which our previous work did not include: SKEW, KURT, MIN, MAX and GEO.

We also calculated MCC and HCM for each diff-chunk. Each metric used a tokenizing strategy so running the metrics on syntactically incomplete code was straightforward. We used the full population of each data-set of diff-chunks from each repository, minus values that were removed because they produced values such as Infinity or NaN (not a number). Figure 1 shows the application of MCC and HCM to a diff-chunk.

Infinity or NaN values often occurred when calculating a metric such as Halstead Effort, if there are no operators or operands in a diff-chunk Halstead Effort (HEFFORT) and Halstead Volume (HVOL) will be undefined due to a $log(0)$ ($log(0)$ is negative infinity but once it is multiplied in IEEE floating point or in some interpretations of reals it is considered a NaN or undefined). Halstead Volume is $V = N * log_2(n)$ where N is sum of the number of operands and operators and n is the sum of the unique number of operands and the unique number of operators. There are cases where a diff-chunk has no operators or operands (e.g., whitespace, comments and formatting). Skewness and Kurtosis will also be undefined when the standard deviation is 0 because they both use standard deviation as a divisor.

Since we were using multiple languages and fragments of source code we implemented our own HCM and MCC metrics for C, C++, Java, Perl, Python and PHP. This helped to maintain consistency across the measurements between languages, and allowed us to measure revisions. We had 51GB of CVS repositories and it took about 3 days of runtime to measure each revision of every repository on an Intel Pentium IV; this produced 14 million diff chunks.

2.2. Analysis

Our initial analysis sought to determine if indentation was regular; that is, was it used consistently by code in various languages, and were consistent depths used. We also wanted to compare the indentation styles of languages, based upon the distribution of indentation depth.

To infer consistency in indentation we will look at the distribution of indentation and see how consistent the indentation is. We will look for regularity of indentation at different depths (base 2, base 4, base 8). For raw indentation we will look for peaks at these depths and depressions around non-indentation-base depths.

To analyze the results we extracted, we used various statistical tools, such as the Kolmogorov Smirnov test for comparing distributions of indentation depth, and Pearson, Spearman and Kendall correlations coefficients for calculating correlations. Our data distributions were usually discrete and positive. We like to match our distributions with classical statistical distributions to allow us to further reason about the data itself as well provide a description of the possible properties of the data. If our distributions are related to distributions used to model processes, repeated tests or population growth we can perhaps create analogous hypotheses. The matching distributions [21] often included the Pareto distribution (a power-law distribution), the Poisson distribution (related to Poisson processes which model the probability of arrival of events), the Binomial distribution (a peaked distribution related to the Poisson distribution), and the Exponential distribution (steep decreasing distribution used to model some Poisson processes). We also use summary statistics on the count data [21].

To show a similarity between indentation styles (the kind of indentation used) we compare the distributions of indentation of sets of revisions (indentation per revision per language). We expect that similar indentation distributions suggest similar styles of indentation, coding, indicating scope and similar semantics. For instance C and C++ should be similar since C++ and its syntax was derived from C. Similar distributions should indicate similar counts of lines at certain depths, we hope these counts faithfully represent the block structures and styles of formatting these block structures that would occur in the code.

In order to say that two languages have a similar indentation we need to compare their distributions. We use the Kolmogorov Smirnov test for this comparison; it is a non-parametric test that can handle data that has troublesome non-normal, non-Gaussian distributions like the exponential distribution. It does so by measuring the maximum distance between two cumulative distribution functions. To characterize the indentation of a language we used the distribution of indentation depth. We then used the distance function of Kolmogorov Smirnov test to compare these distributions, the lower the distance the closer the distributions are to each other. If one measurement is similar or related to another measurement, if it can replace the other, we need to show there is a relationship between them. The easiest way to show a relationship between two variables is to see how well they correlate. We use correlation in this paper to show a relationship between indentation metrics and code complexity metrics, thereby showing that one could potentially replace the other.

To determine correlations between variables we use two kinds of correlations: linear correlation and rank-based correlation. The difference is great: a linear correlation attempts to show the strength of a linear relationship between two or more variables. A rank-based correlation does not rely on a linear relationship; instead, it orders the variables, ranking them from smallest to largest and then correlates those ranks with the rank of the other variable. Thus if the high ranked values for the first variable occur often with low ranked values of the second variable, the rank-based correlation will be negative; if a high rank of one variable frequently corresponds to a high rank of the second variable the correlation will be positive. Our linear correlation is the Pearson Correlation Coefficient, our rank based correlations are the Spearman-Rho Correlation Coefficient and the Kendall-Tau Correlation Coefficient [21]. All three of these correlations produce values between -1 and 1 where 0.1 to 0.25 indicates a weak positive correlation (0 indicates no correlation), 0.25 to 0.75 indicates a medium positive correlation and 0.75 to 1.0 indicates a strong positive correlation (and vice versa for negative correlations). To demonstrate the value of rank based correlation look at Figure 2. Figure 2 depicts two exponential variables that have a linear correlation of 0.591 but have a rank based correlation of 1.0; this figure demonstrates how linear correlations often have a difficult time correlating variables whose relations are not linear. This rank-based correlation is especially important for ordering sets to get “top ten” lists, in our case we want the “top ten” most complex revisions.

If there is truly a linear relationship, as suggested by a linear correlation, one should be able to build a linear model of complexity using indentation. The linear model of indentation should be able to do better then a model composed of only LOC. Thus to further support assertions of linear correlation, Least Squares Linear Regression was used to produce a best fit of coefficients of a linear combination of the statistical moments of indentation to both MCC and HCM. This method uses an R^2 measure, which indicates the percent of the variation between the model and the data that is accounted for by the model. Larger values of R^2 (0.6 or greater) indicate a good linear fit.

To calculate these correlations we developed our own tool-set, in OCaml, which parallelized the correlation calculations for Kendall-tau because Kendall-tau correlation has an algorithmic complexity of $O(N^2)$, while Spearman correlation has a complexity of $O(Nlog(N))$. This was a problem as there were 14 million diff-chunks to correlate. The largest correlation run was on the C language revisions, which consisted

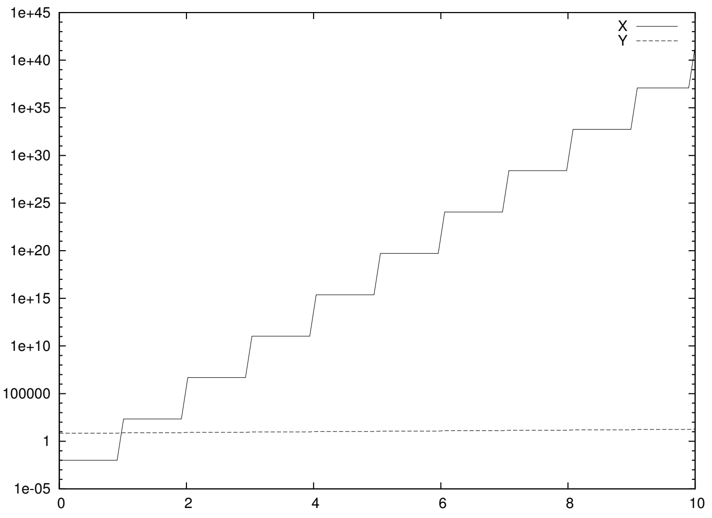

Figure 2: Correlation Example: Linear Correlation between X and Y of 0.591, yet a rank correlation of 1.0

of about 4 million diff-chunks. Our correlations consumed 8 CPU years to calculate, but took only a few weeks of calendar time when computed using a large cluster.

In summary, to test the correlation between our statistical moments of indentation and our complexity metrics we use the three correlation coefficients (Pearson, Spearman and Kendall). We linearly model complexity using Least Squares Linear Regression and the R^2 measure. The results extracted from this methodology and analysis, which took over 8 CPU years to compute, are discussed in Section 3 and Section 4.

3. Indentation of Revisions

In this section we provide an overview of the data we are analyzing, which comprises the source code repositories of 278 open source projects written in six programming languages (C, C++, Java, PHP, Perl, Python). We characterized the indentation depth distributions of the languages and projects; we related the languages with each other via their indentation depth distributions.

3.1. Distributions of Indentation Depth

In general for all projects and languages we found that the actual indentation follows a base 4 rule (raw indentation depth is usually divisible by 4, a single logical unit of indentation was 4 spaces). A logical unit of indentation is the depth of nesting a programmer wanted to convey; for example, inside of an if block a programmer probably often means to indent the conditional code 1 more unit of logical indentation, regardless if they use tabs or spaces to achieve that. If tabs are used, they act as a single unit of logical indentation. Tabs are often used to represent an even number of spaces of indentation. One must note, this is not the indentation of a released product, this is the indentation per diff-chunk in a revision, like those in CVS repositories.

Figure 3 shows us the frequency of raw indentation depth per line found in each language's dataset of diff-chunks. In Figure 3 we can see spikes (large number of lines with the corresponding indentation depth) appearing at line numbers that are divisible by 4. Tabs were considered to be 8 characters in depth mainly because most terminals display tabs that way and many editors use 8 as the default tab-stop size. The spikes in the plots seem to indicate that the data is composed of 2 distributions, the distribution of the lines that form the peaks, and the distribution of lines between the peaks.

Figure 4 shows us the frequency of raw indentation depth per line found in each language's dataset of diff-chunks. In Figure 4 we can see a more smooth slope reminiscent of a Power Law or an Exponential distribution [21] (exponential and power-law-like distributions follow the 80/20 rule: 80% or more are shallow, 20% or less are deep). What is important here is that we can see that base 4 and base 8 levels of raw indentation are very common, more common than base 2, it also shows that this indentation is very regular. We tried to find closest matching parametric distributions per language, we found they looked similar , although the similarities were not non-statistically significant, to Exponential, Binomial, Pareto and Poisson distributions.

3.2. Language Analysis

Figure 5 compares the logical indentation depth distributions using the distance function from the Kolmogorov Smirnov test. The distance function uses the maximum difference between the two CDFs (Cumulative Distribution Functions, created by plotting the cumulative count of bins up to a certain bin). Each logical indentation depth distribution is converted to a CDF and then compared with this distance function, values less than 0.9 indicate less similarity, values closer to 1.0 indicate increasingly significant similarity.

Java was notable because it seemed to have more noise between the base 4 indentation levels. Some Java projects used an initial indentation of one space. Since all methods must be within a class, some Java programmers apparently try to save screen space by indenting in only one space for the initial classes' scope. Java's logical indentation distribution was most similar to a Binomial distribution [21] (a peaked and positive distribution) with a p value of 0.017 (the p-value indicates it was statistically significant within the 99% confidence interval), this is because of the tall peak at Logical Indentation Level (LIL) 2.

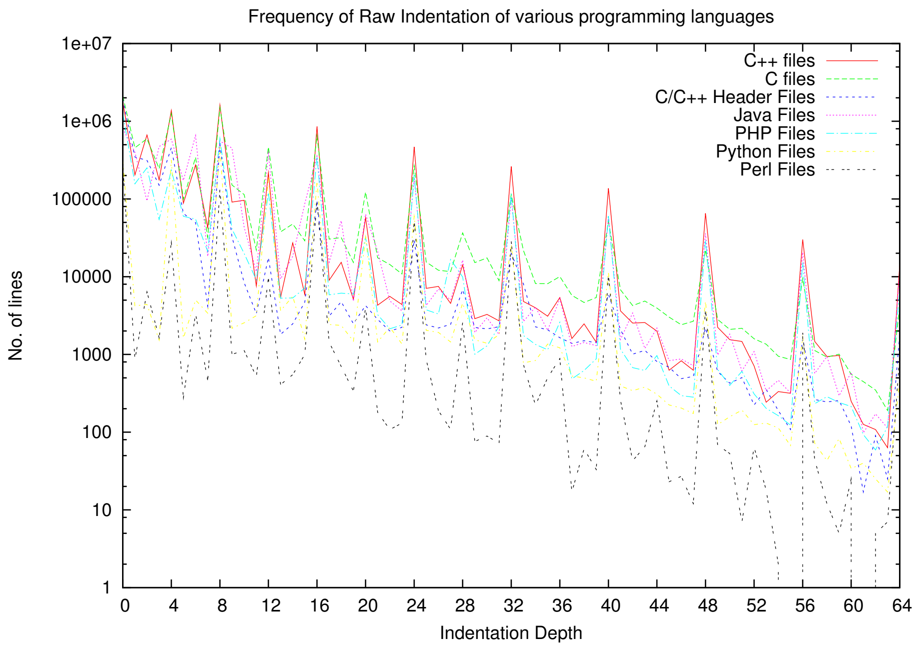

Figure 3: Frequency of Physical Raw Indentation of various languages (Log Scale), note the peaks at depths divisible by 4 and 8. These peaks illustrate a regularity in the use of base 4 and base 8 spaces/tabs for indentation.

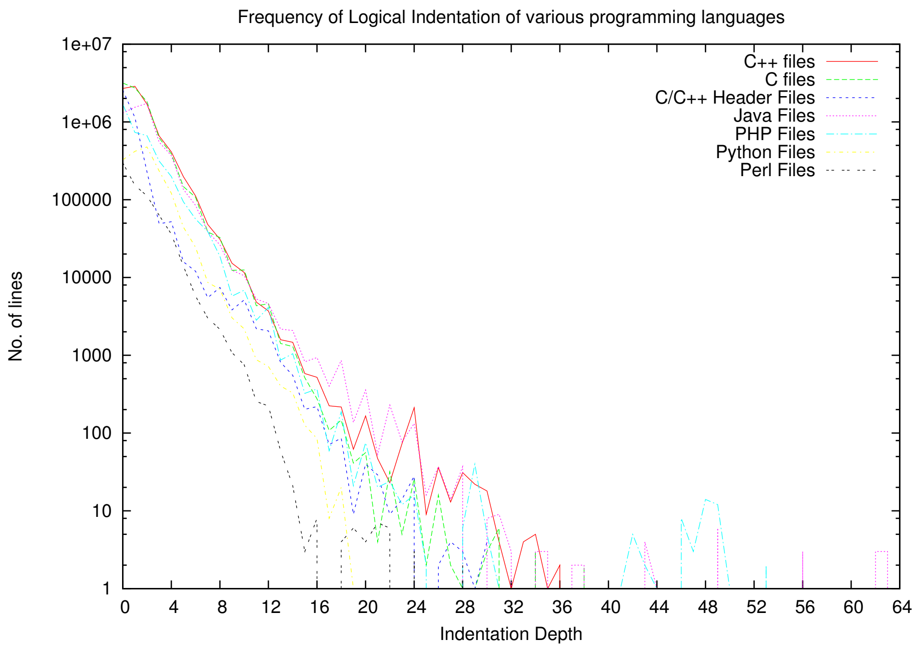

Figure 4: Frequency of Physical Logical Indentation of various languages (Log Scale). Note the smooth descent of the distribution, it follows an Exponential/Power Law like distribution. This distribution demonstrates that deep indentation is less common than shallow indentation.

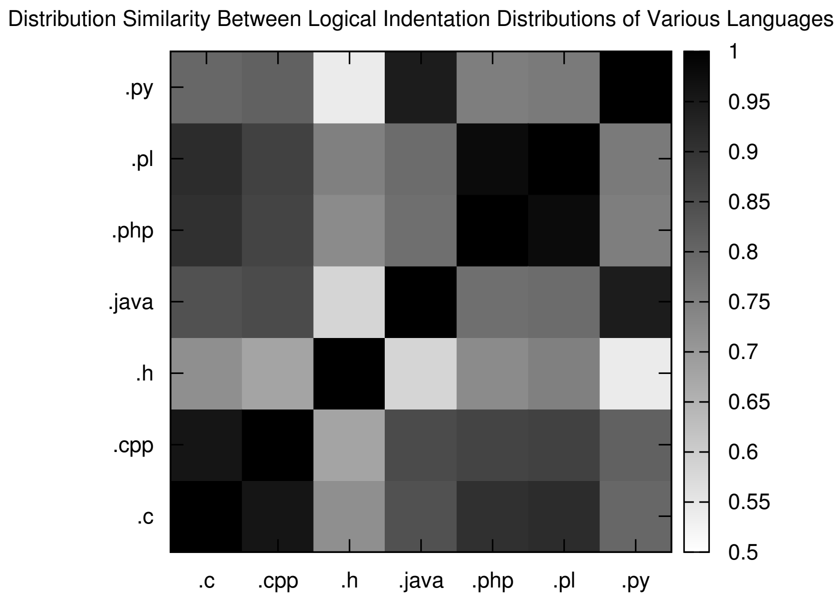

Figure 5: Logical Indentation Distribution Similarity between Languages (1.0 indicates they are very similar)

Header files (.h files) for C and C++ were predictably indented very little. LIL 0 was the most popular followed by LIL 1. LIL 1 was composed of 4 spaces or 1 tab. There were many lines (4th most frequent raw indentation depth) indented by 1 or 2 spaces but there were more lines of LIL 1. According to Figure 5, header files have the least similar logical indentation distribution compared to other languages, which is understandable because they probably contain the least code since they are mostly for definitions.

We found that Perl's indentation distribution is the closest to C, PHP and C++. This might be because classes in Perl do not require further indentation since they are denoted by a `package` keyword. Often, Perl code uses 4 spaces and tabs, although sometimes 2 spaces are used. All of the Perl indentation distributions follow an exponential distribution. This makes sense since Perl is often used for scripting; scripts are often quite linear and lack much nesting.

Python's logical indentation distribution is the most similar to Java's. Python is unique among the languages we studied as it uses indentation to indicate scope, that is, indentation has semantics. We observed that Python's logical units of indentation were very consistent, either 4 spaces or 1 tab. More lines were indented at LIL 1 or LIL 2 times than at LIL 0. Notably, Python's logical indentation distribution matched closest with a Poisson distribution, which is a peaked distribution that models the arrival of events. The peak in the Python distribution suggested that a lot of Python code is contained within definitions (the implementation) of classes, methods and functions.

PHP's indentation was the most similar to Perl and C. PHP stood out because it had some common deep indentations with logical units of 1 tab and 4 spaces. It appears that due to the mixing of HTML and PHP code that the logical indentation units of PHP ends up being mixed between spaces and tabs.

C++ files (.cpp files) were the most similar with C files and were somewhat similar with Perl files. Perl and C++ define methods similarly so this might have been the reason. C++ files had definite pronounced non-base-4 heights, 2 spaces was quite common although most files followed a 4 spaces or tabbed indentation. 0 to 2 LILs were common with C++.

C files (.c files) were very similar to C++ files in distribution and style. 2 spaces were common units, although 4 spaces and tabs dominated. C's indentation was more similar to C++'s than with the indentation of Perl or PHP (which were the next closest distributions).

Thus we have established that indentation is regularly used within the multiple languages we studied, as base 4 indentation depths were the most common indentation depths (see Figures 3 and 4). We attempted to show how different languages had similar indentation styles, by comparing the logical indentation distributions of diff-chunks from different languages. We found sets of languages that shared similar indentation styles (see Figure 5): C and C++, Perl and PHP, and Python and Java).

4. Indentation and Complexity

In this section we examine the correlation we found in our study between complexity metrics, such as Halstead's and McCabe's, and statistical moments of indentation.

For McCabe's Cyclomatic Complexity we measure the MCC, defined as one plus the number of branch points in a block of code, and the number of return statements. McCabe's Cyclomatic Complexity (MCC) measures the number of branch points in a block of code. The Halstead Complexity metrics (HCM) are a set of measurements based on tokens. These measurements include length measured by count of tokens (HLEN), vocabulary size as number of distinct tokens (HVOCAB), volume as length times the log of vocabulary size (HVOL), difficulty as half of operator vocabulary times number of operands divided by operand vocabulary (HDIFF), and effort (HEFFORT) as difficulty times volume.

Our indentation metrics were the statistical moments of raw and logical indentation: LOC, AVG and LAVG, MED and LMED, STD and LSTD, VAR and LVAR, SUM and LSUM, SKEW and LSKEW, KURT and LKURT, GEO and LGEO, MIN and LMIN, MAX and LMAX.

4.1. Measures and Correlation

Our observation was that the AVG, MED, SKEW, KURT, MIN, MAX, and the GEO of indentation did not correlate well with any of the MCC or HCM complexity metrics for both linear correlation (Pearson) and

Indentation to	AVG Spearman	AVG Pearson
LOC	0.407	0.620
AVG	0.290	0.038
MED	0.282	0.033
STD	0.462	0.133
VAR	0.462	0.059
SUM	0.425	0.672
KURT	-0.224	0.040
SKEW	0.083	0.060
MIN	0.114	-0.015
MAX	0.352	0.124
GEO	0.140	-0.007

Table 1: Average correlation coefficients of Indentation Metrics across languages with MCC

rank-based correlation (Spearman and Kendall). The average Spearman and Pearson correlation coefficients are shown in Table 1. We correlated MCC and HCM metrics against the indentation metrics for raw indentation and logical indentation. Rank and Linear Correlation coefficients between .25 and .75 are considered medium strength where as those of .75 and above are considered strong correlations.

We expected that MED and AVG would not correlate well with MCC because a block of code with no change in indentation could have a high average indentation if it was relatively deep. Similarity AVG, MED and GEO do not seem to give any good indication of branching, so we did not expect a high correlation with MCC.

Skewness (SKEW) and Kurtosis (KURT) did not correlate well with MCC or HCM linearly because they are inherently non-linear measurements. SKEW and KURT are computed based on the a power of the AVG divided by a power of the STD. The poor correlation comes from STD being used as a denominator, STD is positively correlated, the inverse is not, thus it will probably cancel out whatever positive correlation we gained from the AVG. Although we observe that KURT rank-correlates much better with MCC than SKEW does.

Minimum (MIN) and Maximum (MAX) were measured because they indicate the range of nesting in a diff-chunk and MAX nesting has been used a complexity metric in the past [8, 9]. MIN probably did not correlate with MCC and HCM well because the value of MIN was often small would not reliably indicate if any branching had occurred. MAX correlated better but still did not do well. In the case of diff-chunks with low variance, MAX would be similar to the AVG and thus would be a poor indicator of branching.

The indentation metrics LOC, SUM, STD, and VAR had medium strength (0.4 to 0.6) rank based correlations and small linear correlations (0.2 to 0.4) with complexity metrics such as Halstead Difficulty (HDIFF) and MCC. For MCC, LOC had a linear correlation of 0.75 and a rank-based correlation of 0.41 to 0.45. For HDIFF, LOC had rank and linear based correlation of 0.49 to 0.55.

HCM has count-based metrics such as Halstead length, Halstead vocabulary and Halstead volume, and we found that these linearly correlated well with LOC and SUM. Halstead Difficulty and Halstead Effort estimate complexity based on the number of unique operands and operators versus the total number of operands and operators. Halstead Effort is intended to model the time it took to write the source code, which correlates well with LOC in most cases.

4.2. Complexity and Language

In general, rank based correlations showed that SUM and STD correlated better with McCabe and Halstead complexity than LOC did. For linear correlations LOC usually fared better than SUM. Figures 6 and 7 depict the correlation coefficients of MCC with SUM and STD. In both figures, per each language, we first compare the Pearson correlation between MCC and LOC and SUM, then we compare the Spearman correlation between MCC and LOC and SUM. The Halstead length metrics all correlated best with LOC, both with linear and rank-based correlations.

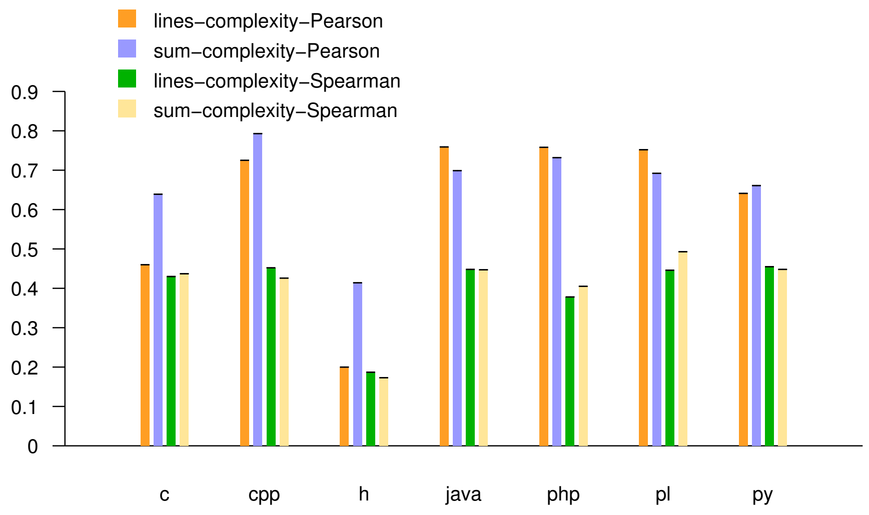

Figure 6: Correlation of McCabe's Complexity to Summation of Indentation and McCabe's Complexity to LOC across seven languages

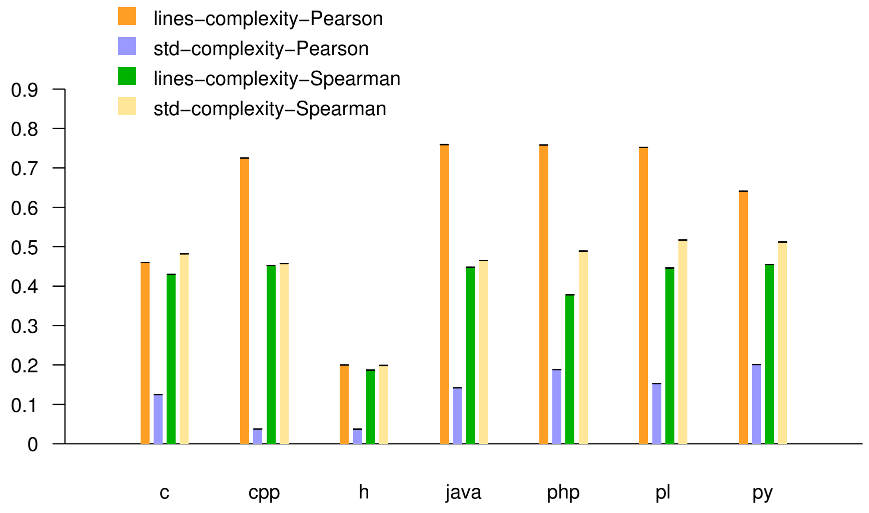

Figure 7: Correlation of Complexity to Standard Deviation of Indentation and McCabe's Complexity to LOC across seven languages

For changes to C files, summation of indentation (SUM) was often a better predictor of MCC than LOC. In general C had files medium strength rank based and linear correlations between SUM, STD and VAR of indentation and MCC. The C files had low scores for linear Pearson correlation, with MCC correlating better with SUM than LOC. Rank based correlations confirmed that LOC was correlated with complexity measures but also that STD and VAR were important. Kendall correlation coefficients were lower than Spearman coefficients (both are rank-based correlations). Both Spearman and Kendall correlation of STD (Spearman 0.48, Kendall 0.44) were more correlated with MCC than LOC (Spearman 0.43, Kendall 0.39). C was interesting in that it had much lower linear correlation between LOC and MCC than the other languages, even lower than SUM and MCC.

C++ had strong and medium strength correlation between SUM, STD, VAR and MCC. For C++, SUM exhibited a strong linear correlation with MCC (0.79), which was better than LOC's correlation (0.73). LOC rank correlated with MCC slightly better than SUM did. Although with rank based correlation STD, VAR and LOC of indentation were equally correlated with MCC (0.45).

Header files (.h files) generally did not correlate well MCC, except for SUM. For .h files, LOC, SUM, then STD, in descending order, correlated well with HDIFF and MCC. Surprisingly SUM linearly correlated well with MCC, with the number of returns. This means that SUM was correlating with the complexity of functions and methods implemented in .h files.

For Java, LSUM, not SUM, linearly correlated with complexity better than LOC (LSUM Pearson correlation of 0.77 versus LOC Pearson correlation of 0.76 with MCC); SUM did worse. SUM and LOC were had equivalent strength rank-based correlations. For rank based measures STD and VAR had medium correlations with MCC and HDIFF ($0.43 - 0.45$).

For PHP, STD and SUM had better rank-based correlations with MCC than LOC did. Both SUM and LOC were strongly linearly-correlated with complexity (MCC).

Python files were interesting as their correlation coefficients between LOC and complexity were lower than the other languages except C, and .h files (linear correlation of 0.64 and rank based correlation of 0.49). STD had a medium linear correlation with HDIFF (0.39).

For Perl, STD was more linearly-correlated with HDIFF than with LOC (0.47 versus 0.42), although LOC strongly linearly correlated with MCC (0.75). For rank based correlations STD is correlated best for MCC (Spearman 0.52, and Kendall 0.47) and SUM correlated best with HDIFF (Spearman 0.47 and Kendall 0.44).

Thus for all the languages we have shown, there were medium to strong linear correlations between MCC and HDIFF with LOC and SUM. We have also shown for all languages studied, there were medium strength linear and rank-based correlations between complexity metrics MCC and HDIFF and each of STD and VAR.

4.3. Complexity Correlation per Project

For most projects, LSUM and SUM had a greater linear correlation for MCC than LOC, which means that the summation of indentation generally correlated better with MCC than LOC did. LOC correlated better with HDIFF than SUM. For rank based correlations, STD and SUM were better correlated than LOC for complexity, but LOC was better correlated for HDIFF.

If we evaluate correlations between MCC and STD we found that some projects had relatively strong linear (0.55 to 0.67) correlations between complexity and STD, such projects included: Sodipodi, Bittorrent, Dynapi, Aureal, PHPnuke. Some projects which did not exhibit medium to strong linear correlations (0.01 to 0.07) between MCC and STD were: CrystalSpace, sed, jedit, BOOST. For rank based correlations, Bastille-Linux, Unikey, Sodipodi and OpenCV were above 0.67 (Bittorrent, Dynapi, Aureal, and PHPNuke were all above 0.5).

4.4. Linear Combinations

While correlation is useful for showing the relationship between two variables, what we actually want is to model complexity metrics (MCC and HCM) with the statistical moments of indentation. A model would be an equation consisting of our indentation metrics as input and MCC or HCM metrics as output.

α_0	AVG	MED	STD	VAR	SUM	LMED	LSTD	LVAR	LSUM
1.21	-0.09	0.09	-0.05	-3.0e-04	5.6e-04	-0.12	-0.01	-0.08	0.08

Table 2: Coefficients for the linear model of complexity (MCC). This model had a R^2 of 0.385

The model we seek is essentially an equation based on our indentation metrics that produces an accurate MCC or HCM metric output. Such a model would provide us with a function which when we would use to evaluate the complexity of a diff-chunk, the parameters to the function would be the statistical moments of indentation we measured.

Since we used Pearson correlation, which indicates a linear relationship between two variables we decided to try linear models of complexity (MCC or HCM) using the statistical moments of indentation. In this case a linear model consists of a constant plus the weighted sum of our indentation metrics, each multiplied by a constant coefficient. Our model is:

$$c = \alpha_0 + \alpha_1\beta_1 + \alpha_2\beta_2 + ... + \alpha_{n-1}\beta_{n-1} + \alpha_n\beta_n$$

where c is MCC or HDIFF (or other HCM metrics) and α_1 through α_n are the coefficients of the indentation metrics, that are enumerated as β_1 through β_n, and where n is the number of indentation metrics.

We used linear regression to determine the coefficients ($alpha_0$ to $alpha_n$) of each indentation metric (such as SUM, STD, etc.). Linear regression applies a search heuristic to determine the appropriate coefficients (α_i) for each variable (β_i). The best coefficients are those that minimize the variance and difference between the value being modelled (either MCC or HCM) and the inputs (β_1 to β_n), our statistical moments of indentation). α_0 is the initial intercept constant. The linear regression we ran produced the coefficients seen in Table 2, variables such as skew and kurtosis had coefficients of 0, indicating that they did not help linearly model MCC. Linear regression returns a goodness of fit measure called R^2 is between 0 and 1, where 1 is an exact best fit. The difference between 1 and R^2 $(1 - R^2)$ indicates the percentage of unexplained variance. This is basically how much variance in the model is unexplained. Thus an R^2 value of 0.2 has 80% unexplained variance, this is often considered to be too much, even 0.4 for an R^2 is considered low in some fields. A low R^2 does not mean the variables are not related, rather it can mean their relationship is not linear.

In the model shown in Table 2, we do not use LOC as we want to see if the linear relationship still holds without LOC. For MCC, LOC does not improve the R^2 much, it increases from 0.385 to 0.388; this implies that our indentation metrics provide most of the information that LOC provides. Without SUM and without LOC most of the R^2 values are very low.

Linear models of Halstead difficulty had worse results (R^2 of $0.20 - 0.22$) than models of McCabe's Cyclomatic complexity. Halstead Effort had an R^2 below 0.041, which is very low. These values are so low that they imply they can not be linearly modelled very well. This makes sense since Halstead Effort is not linear, rather it is Halstead Volume multiplied by Halstead Difficultly, thus not of the form $y = a * x + b$. Halstead Volume and Length had the highest R^2 values of 0.6 and 0.5. This suggests that the Halstead metrics such as Halstead Difficulty and Halstead Effort were harder to model linearly than MCC.

We can see there is some linear relationship between statistical moments of indentation and complexity (MCC and HCM), although there is much variation unaccounted for in these models. LOC on its own does not fare well against most of measures. The best models built on LOC alone were for: Halstead volume (R^2 of 0.59), Halstead length (R^2 of 0.51), number of returns (R^2 of 0.38) and MCC (R^2 of 0.29). We can see that by including indentation metrics in our model we do gain information and accuracy from our linear models. We have shown that there is value in measuring indentation as well as LOC as we can model complexity more accurately with indentation and LOC combined.

5. Other Measurements and their Correlations

In order to be thorough, to make sure we were not missing any obvious correlations, and to demonstrate that measuring indentation is worthwhile, we felt it was necessary to measure other diff-chunk and line-based values, and their statistical moments. We measured other line-based metrics such as Line Length (length of each line), Code-char length (number of code characters per line), raw and logical indentation *before* a change, and the difference between raw and logical indentation statistical moments *before* and *after* a change.

Since Code-chars did not represent nesting and line-length only had nesting as a component, our initial expectations were that Line Length and Code-chars would not work well with McCabe's Cyclomatic complexity. Since MCC seemed inherently intertwined with nested code (code that branches) and code characters do not provide any indication of nesting we expect the correlation between MCC and the statistical moments of Code-Chars to be low.

We found that Indentation Metrics correlated better with MCC than either Line Length or Code-chars, although Line Length and Code-chars did better with HCM. The summary of these correlations can be seen in Table 3.

5.1. Code-Chars

Indentation counts non-code characters, but are we throwing valuable information away? Code-chars are the characters on a line that are not indentation characters. Code-chars fared much better with HCM than with McCabe metrics. This is somewhat intuitive since Code-chars are probably heavily correlated with the number of tokens per line thus making it score high with most of the HCM, which primarily count tokens and the number of unique tokens. Code characters were extracted the same way indentation was extracted except we skipped the initial whitespace, the indentation, and counted the characters immediately after the last indentation character.

Summation of Code-chars did not correlate as well with MCC as summation of indentation did: Pearson correlation coefficients were 0.57 versus 0.66, while Spearman correlation coefficients were 0.35 versus 0.42. For the standard deviation of Code-chars and indentation, indentation still fared better with a Spearman correlation coefficient of 0.46 compared to the Code-chars Spearman correlation coefficient of 0.38.

Code-chars might not have fared well with MCC but they fared far better in comparison to indentation with the HCM. For rank correlation with HCM, we compared sum of Code-chars to the sum of indentation: the average correlation coefficient for the 6 Halstead metrics was 0.53 for summation of Indentation and 0.74 for summation of Code-chars. We found that Code-chars correlated far better than indentation metrics in this case.

5.2. Line Length

In order to be thorough we felt it was necessary to also investigate line length since it is about as easy as indentation is to measure. Thus we extended our previous work by investigating how Line Length correlates with MCC and HCM.

Line Length is the number of characters on a line, including indentation. While Line Length is an intuitive measure, it easily computed, but it also lacks the contextual information that is stored within Code-chars or indentation. Thus, we expected that it would not perform as well for branch-based metrics such as MCC. Some Line Length metrics correlated quite well with HCM, but for MCC Code-chars and indentation did better.

Line Length was extracted measuring the raw number of characters on each changed-to line in the diff-chunk, much like how we measured indentation, except we counted all the characters on the line (except the line-break). Thus for every file and every revision and each diff-chunk we measured the Line Length of the changed code.

The MCC correlation for SUM and STD of Line Length was not very high compared to indentation or code chars. Line Length SUM and STD had Spearman correlations of 0.36 and 0.39. Line Length is a measure of how many characters, how much information is on a line, although indentation is part of the line length measure. What line length does not do is give a good indication of how deeply nested the code is and how much branching is occurring, which is what a lower MCC correlation shows.

Rank correlation between Line Length and HCM fared well. Summation of Line Length had an average HCM correlation of 0.75, while standard deviation had an average HCM of 0.515. Thus summation of Line

	MCC	Length	Vocab	Volume	Difficulty	Effort	Mental
LOC	0.41	0.64	0.61	0.59	0.59	0.57	0.62
Line Length SUM	0.36	**0.77**	**0.75**	**0.81**	**0.69**	**0.77**	**0.75**
Line Length STD	0.39	0.53	0.50	0.54	0.49	0.53	0.51
indentation SUM	0.43	0.58	0.57	0.45	0.59	0.47	0.59
indentation STD	**0.46**	0.46	0.45	0.46	0.48	0.49	0.48
Code-char SUM	0.35	0.76	0.74	**0.81**	0.68	**0.77**	0.74
Code-char STD	0.38	0.53	0.50	0.54	0.48	0.53	0.51

Table 3: MCC and Halstead Metrics Rank-Correlated with Summation and Standard Deviation of Indentation, Code-chars and Line Length

Length rank correlated better than Code-chars for many of the HCM. This means that if one wants to rank diff-chunks by HCM then Line Length metrics would work quite well.

5.3. Differences

Diff-chunks essentially have two parts, a *before* and *after*. We wanted to ensure we were not missing an important correlation, we are measuring changes so why not measure the difference that is available to us? Thus we measured the difference of metrics between *before* and *after* diff chunks.

We decided to evaluate various metrics such as SUM or STD of indentation correlated with differences between the *before* and *after* MCC and HCM. We also wanted to see if the difference in the *before* and *after* measurements of SUM and STD of indentation correlated with differences in *before* and *after* measurements of MCC and HCM.

Differences are calculated by subtracting the statistical moments of indentation, line-length and Code-chars of the *before* code from the *after* code.

Overall we found that there were better correlations between the *difference* in MCC and the *difference* in indentation metrics, than between the *difference* in MCC and the *after* indentation metrics. We found that the correlations between SUM and STD of indentation, Line Length, and Code-chars rank-correlated with a medium strength (0.30) to the *difference* in MCC and HCM. While the *difference* of SUM and STD measures of indentation, Line Length and Code-chars ranked correlated with MCC and HCM with a correlation coefficient of at least 0.4. Thus to rank revisions by *difference* of MCC one should rank by *difference* of SUM or STD of indentation.

5.4. Analysis of the Difference in MCC

We wanted to look at issues regarding if complex code became more complex or less complex. We wanted to know the relationship between complex code and the changes to complex code.

*Is MCC **before** a change correlated with MCC **after** a change?*

The correlation of MCC *before* and *after* a revision is about 0.41 linearly and 0.61 rank-based. This is intuitive since the code being changed was the *before* code. What this does not answer very well is how much the MCC of the *before* code relates to the difference in MCC (ΔMCC).

*Is MCC **before** a revision correlated with the difference in MCC?*

The MCC of code *before* a revision is negatively correlated with the difference in MCC, the ΔMCC. The strength of the correlation is much less than the correlation between the two measurements but it still exists. This means that the ΔMCC is not independent of initial MCC of the code. Although the correlation coefficients are for -0.46 linearly and -0.26 ranked. Linearly there is a negative correlation, investigating further shows that about 60-80% of changes do not result in a difference in MCC, and the minimum MCC is 1 so a negative linear correlation is expected. We suspect this biases the Spearman correlation as well, but what we observed was that large complex blocks of code often were removed, this resulted in a large drop in complexity but also meant that, rank wise, the correlation would be negative.

*Are the HCM of the code **before** a revision correlated with the HCM of the code **after** a revision?*

We found that there is a positive correlation between HCM *before* and *after* a revision (Pearson 0.46, Spearman 0.60). This implies that code that ranked high with HCM probably ranked similarly *after* a change. This does not say if the HCM went up or down *before* or *after* a release, it just suggests that HCM of code *before* release is not independent of HCM of code *after* a release.

Are the HCM of the code ***before*** *a revision correlated with the difference in HCM?*

There is a negative correlation between the HCM of the **before** code and the change in the HCM caused by the revision (Pearson -0.41, Spearman -0.29). A negative correlation means that smaller HCM values of the **before** code are often correlated with changes that increase the HCM values. Conversely large HCM values of the **before** code were correlated with changes that decrease in HCM values. Many of the HCM can be broken down into 3 distributions with respect to their *before* code measurements and their change in HCM to their new code measures. The first group is where no change occurred, this is a group of positive HCM that have no change. The second group is the group that exhibited a negative change in HCM, likely due to a removal of code. The third group is the group where the HCM increased, either from 0 or more. Thus larger HCM values of *before* code seemed to be associated with more negative decreases in HCM.

6. Precision and Recall of Rank by Indentation

We wanted to see if ranking by indentation metrics was indeed better than ranking by LOC. Furthermore, we wanted to know if we integrated these metrics into a tool, could they be used effectively to rank revisions by complexity? Our previous correlation studies, presented in the paper, have shown that is likely the case.

To demonstrate the applicability of ranking revisions by indentation metrics in order to proxy ranking revisions by complexity, we have set up an experiment where per each language, $10,000$ times, we sampled 100 revisions, then we ranked these revisions by complexity (MCC) and other measures such as LOC, SUM or VAR. Then we took the top ten revisions, ranked them by MCC and asked: if they had been ranked by LOC, SUM or VAR, would they have been in the top ten? That is, we effectively asked: when we rank by LOC, SUM or VAR are the top ranked revisions still the top ranked revisions if we had ranked by MCC?

Precision is the percentage of relevant documents retrieved in a query where as recall is the fraction of the total relevant documents retrieved by a query. If we have 10 documents (top 10 highest ranked documents) in our query results and we are only looking for 10 documents (the top 10 most complex revisions), precision and recall will be the same number since we are always dealing with the same number of relevant documents and retrieved documents. In our case, Precision is essentially accuracy, the percentage of documents that were in our top ten that were ranked in the MCC top ten.

Our initial intuition was that since the rank correlation coefficient between STD and SUM with was between .4 and .5 that we would probably get a precision or recall of a similar value. This would mean 40% to 50% of the ranked results would belong to the top ten most complex diff-chunks as determined by MCC.

Once we ran our experiment many times and averaged the performance of LOC, SUM, STD and VAR for estimating the top ten most complex diff-chunk, we found that LOC and SUM had similar precision and recalls of 0.475 and 0.472 (although LOC did do better), STD and VAR had precisions and recalls of 0.392. This means that on average LOC and SUM returned a top 10 where nearly 1/2 of the suggestions were actually in the top 10, While STD and VAR alone would produce top ten sets where only 2/5 were the most complex. LOC did not perform better than SUM for all languages. In fact for Java, PHP, Perl, and Python, SUM performed better, had greater precision and recall values, than LOC (see Table 4). Thus we have demonstrated that the rank correlations we witnessed paid off when we attempt to rank diff-chunks by complexity using indentation metrics.

We had previous success by combining LOC and other indentation metrics so we tried to see if any combination of metrics together could produce precision and recall values better than either LOC or SUM. On average SUM multiplied by LOC ranked diff-chunks a little better, with an average accuracy of 0.497, beating both SUM and LOC (see Table 4). These results suggest that the combination of LOC and the indentation metrics make for the most accurate estimated ranking of diff-chunks by MCC.

Language	LOC Accuracy	SUM Accuracy	STD Accuracy	LOC * SUM
C	0.5223	0.5140	0.389	**0.5425**
C++	0.5406	0.4996	0.370	**0.5528**
.h files	0.2237	0.2219	0.210	**0.2302**
Java	0.4918	0.5056	0.413	**0.5135**
Python	0.5266	0.5407	0.437	**0.5558**
Perl	0.5533	0.5587	0.475	**0.5890**
PHP	0.4638	0.4652	0.451	**0.4937**

Table 4: Accuracy (precision and recall) of metrics producing a Top 10 list of most complex diff-chunks

7. Discussion

We can see from the results there is some correlation between our indentation measures and traditional complexity measures (MCC and HCM). There is some linear correlation and there is some better rank based correlation but it is not overly strong. This suggests that our statistical moments of indentation can be used as proxies for ranking revisions by complexity. That is, we found that the larger our measurements, the more complex the code was and, in particular, the larger the standard deviation and summation of indentation, the more complex the code was.

Summation and Standard deviation of indentation seems to be a good proxy for complexity because one could argue that the greater the change in indentation depth, the more complex the statements are in that code block. A large standard deviation in indentation could indicate multiple if blocks or expressions within a diff-chunk, which would correlate well with HCM.

We noticed there was little difference between logical and non-logical indentation in the correlations. This suggests two things: that the relationship between logical and non-logical indentation is for the most part linear, they differ by a constant multiplier (for examples 4 spaces are often 1 logical unit), and that indentation is regular enough that logical indentation does not matter. What it also suggests is that the out-lier indentations, non-base 4 indentations, do not affect the results much otherwise there would be significant differences between raw indentation and logical indentation.

Indentation can provide information that a tokenizer cannot, as indentation can show the scope of expressions whereas a tokenizer provides a flattened representation. To get the information that indentation supplies one would have to parse the source code into a tree. Although statistical moments of indentation can proxy complexity metrics, they are potentially their own complexity metrics. HCM do not count scope where as MCC often does, but indentation will capture more scoping semantics than MCC because not every new scope is a new branch in the code.

We did notice that indentation metrics provided different information than say Line Length metrics or Code-char metrics. Indentation metrics correlated better with MCC than with HCM. This suggests that indentation is better for measuring nesting, while line length and Code-chars correlate better with count-based metrics like HCM. Overall there did not seem to be much difference between Code-chars and Line Length metrics.

8. Validity Threats

We see five main threats to the validity of our work presented her: applicability of metrics used, data sampling, data cleansing, programming language choice, and the influence of development tools.

First, our measurement of MCC and HCM was performed on revisions, not on methods, functions, modules, or files, which is the typical use of these metrics. Often these measurements are taken at a higher semantic level of structural granularity (functions, modules) but we only applied them to diff-chunks.

Second, data cleaning is an issue. If a value was NaN it could not be included in calculations; a 0 could not even be used. We had to remove the entire entry for that calculation. This could create a bias where entries with NaN values might be ignored. These entries were not a large proportion of changes so they should not have too much of an effect on the values.

Third, our study examined only open source systems, specifically those stored in the well known SourceForge site that also had accessible CVS repositories. While our study may arguably be authoritative for large open source systems, we have not examined non-open source systems.

Fourth, we chose to examine systems written in the six most popular programming languages in the SourceForge repository: C, C++, Java, PHP, Perl, and Python. We note that all of these languages have some common heritage with C and have many syntactic similarities with C, hence this may skew the results favorably. However, these six are also by far the most used programming languages in industry, so they seem a reasonable proxy for industrial software in general.

Finally we note that potentially the auto-indentation provided to programmers, by linters, pretty printers and IDEs, affects our results. We suspect they would not, since we showed indentation is relatively uniform. A potential problem is how much trust we have in the developers and their tools.

9. Conclusions

We have shown with sufficient confidence that to rank revisions by statistical moments of indentation is analogous to ranking revisions by complexity or maintainability. We have also provided some evidence that measuring statistical moments of indentation is more computationally efficient than applying traditional complexity metrics, such as those of Halstead and McCabe.

We tested and confirmed our assertion that indentation was a meaningful proxy for code complexity. It has been suggested by others [19] that LOC correlated with complexity metrics well enough such that complexity metrics were not needed. We have shown through correlations and linear models that cheap metrics such as the statistical moments of indentation of a change, when combined with LOC or alone, can be used to better model and simulate complexity measures than just LOC alone. We showed that for revisions to source code, there were medium to strong linear and rank based correlations between complexity metrics and the summation and standard deviation of indentation. In many cases summation of indentation and standard deviation of indentation did better than LOC, especially with rank based correlations. Rank based correlations are important because we want to order revision by their complexity thus a rank correlation indicates how successful our proxy will be at ranking revisions according to complexity metrics like MCC or HCM. We found little difference between raw and logical indentation metrics with respect to complexity.

We have provided an overview of indentation with respect to a large body of successful, popular Open Source software, as ranked by SourceForge. We have shown that indentation is quite regular across multiple languages, at least for the projects we sampled. We expected common logical units of indentation of 2 spaces to be frequent, but across all of the languages, 4 spaces of indentation or 1 tab of indentation were the most common logical units. We compared the distributions of indentation per language to each other and found that the indentation of one language was often similar to another. For example, Python and Java had similar indentation styles, while Perl, C and PHP were similar to each other, and C and C++ were very similar to each other.

We have shown that indentation is regular and consistent enough to be worth measuring. We demonstrated the value of measuring indentation along side LOC, it can be used as a proxy for complexity, and it is almost as cheap as LOC to calculate. Thus with the knowledge that indentation metrics are generally language agnostic, language unaware and cheap to calculate, we can use them as cheap proxies for complexity and maintainability of source code revisions. These measurements help to quickly and efficiently rank patches to source code by their complexity and maintainability.

9.1. Future Work

One avenue of future work is to investigate languages other than the six we examined, such as Smalltalk, Ruby, LISP, Scheme, Haskell and Dylan. While these languages are much less common in industry, there are also syntactically unlike C, and so the results may be quite different.

Acknowledgements: This work was partially funded by an NSERC PGS D Scholarship. We would like to thank SHARCNET for use of their infrastructure.

References

[1] T. J. Mccabe, A complexity measure., IEEE Trans. Software Eng. 2 (4) (1976) 308–320.

[2] M. H. Halstead, Elements of Software Science (Operating and programming systems series), Elsevier Science Inc., New York, NY, USA, 1977.

[3] P. W. Oman, J. Hagemeister, Construction and testing of polynomials predicting software maintainability, J. Syst. Softw. 24 (3) (1994) 251–266. http://dx.doi.org/http://dx.doi.org/10.1016/0164-1212(94)90067-1 doi:http://dx.doi.org/10.1016/0164-1212(94)90067-1.

[4] A. Hindle, M. Godfrey, R. Holt, From indentation shapes to code structures, in: 8th IEEE Intl. Working Conference on Source Code Analysis and Manipulation (SCAM 2008), 2008.

[5] D. Conway, Perl best practices, 1st Edition, O'Reilly, Beijing [u.a.], 2005.

[6] P. W. Oman, C. R. Cook, Typographic style is more than cosmetic, Commun. ACM 33 (5) (1990) 506–520. doi:http://doi.acm.org/10.1145/78607.78611.

[7] A. Hindle, M. Godfrey, R. Holt, Reading beside the lines: Indentation as a proxy for complexity metrics, in: Proceedings of ICPC 2008, 2008.

[8] J. Munson, T. Khoshgoftaar, The dimensionality of program complexity, Software Engineering, 1989. 11th International Conference on (1989) 245–253.

[9] H. Gong, M. Schmidt, A complexity measure based on selection and nesting, SIGMETRICS Perform. Eval. Rev. 13 (1) (1985) 14–19. doi:http://doi.acm.org/10.1145/1041838.1041840.

[10] R. Power, D. Scott, N. Bouayad-Agha, Document structure, Comput. Linguist. 29 (2) (2003) 211–260. http://dx.doi.org/http://dx.doi.org/10.1162/089120103322145315 doi:http://dx.doi.org/10.1162/089120103322145315.

[11] R. F. Mathis, Flow trace of a structured program, SIGPLAN Not. 10 (4) (1975) 33–37. doi:http://doi.acm.org/10.1145/987278.987282.

[12] R. J. Miara, J. A. Musselman, J. A. Navarro, B. Shneiderman, Program indentation and comprehensibility, Commun. ACM 26 (11) (1983) 861–867. doi:http://doi.acm.org/10.1145/182.358437.

[13] R. B. Findler, PLT DrScheme: Programming environment manual, Technical Report PLT-TR2007-3-v371, PLT Scheme Inc. (2007).
URL http://download.plt-scheme.org/doc/371/pdf/drscheme.pdf

[14] N. Gorla, A. C. Benander, B. A. Benander, Debugging effort estimation using software metrics, IEEE Trans. Softw. Eng. 16 (2) (1990) 223–231. doi:http://dx.doi.org/10.1109/32.44385.

[15] R. E. Berry, B. A. Meekings, A style analysis of C programs, Commun. ACM 28 (1) (1985) 80–88. doi:http://doi.acm.org/10.1145/2465.2469.

[16] D. Coleman, D. Ash, B. Lowther, P. W. Oman, Using metrics to evaluate software system maintainability, Computer 27 (8) (1994) 44–49. doi:http://dx.doi.org/10.1109/2.303623.

[17] X. Chen, B. Francia, M. Li, B. Mckinnon, A. Seker, Shared information and program plagiarism detection, IEEE Trans. Information Theory 50 (2004) 1545–1551.
URL citeseer.ist.psu.edu/636752.html

[18] R. Harrison, S. Counsell, R. Nithi, An evaluation of the mood set of object-oriented software metrics, Software Engineering, IEEE Transactions on 24 (6) (1998) 491–496. doi:10.1109/32.689404.

[19] I. Herraiz, J. M. Gonzalez-Barahona, G. Robles, Towards a theoretical model for software growth, in: MSR 2007: Proceedings, IEEE Computer Society, Washington, DC, USA, 2007, p. 21. doi:http://dx.doi.org/10.1109/MSR.2007.31.

[20] D. M. German, A. Hindle, Measuring fine-grained change in software: towards modification-aware change metrics, in: Proceedings of 11th International Software Metrics Symposium, 2005.

[21] NIST/SEMATECH e-Handbook of Statistical Methods, http://www.itl.nist.gov/div898/handbook/ (2008).
URL http://www.itl.nist.gov/div898/handbook/