Estimating Commit Sizes Efficiently

Philipp Hofmann and Dirk Riehle

SAP Research, SAP Labs LLC
3412 Hillview Ave, Palo Alto, CA, 94304
phil@branch14.org, dirk@riehle.org

Abstract. The quantitative analysis of software projects can provide insights that let us better understand open source and other software development projects. An important variable used in the analysis of software projects is the amount of work being contributed, the commit size. Unfortunately, post-facto, the commit size can only be estimated, not measured. This paper presents several algorithms for estimating the commit size. Our performance evaluation shows that simple, straightforward heuristics are superior to the more complex text-analysis-based algorithms. Not only are the heuristics significantly faster to compute, they also deliver more accurate results when estimating commit sizes. Based on this experience, we design and present an algorithm that improves on the heuristics, can be computed equally fast, and is more accurate than any of the prior approaches.

1 Introduction

The quantitative analysis of source code, in particular of open source software code, is becoming increasingly important. Such analysis can provide us with relevant and empirically validated insights into how and why software development works and subsequently, how we can improve open source and corporate software development further.

An important independent variable and the input of many models is the amount of work that went into a code contribution, typically measured by the source code lines affected by a code contribution, also known as the commit size. The commit size is defined as the sum of the number of source code lines added, removed, or changed in a given commit.

There are at least three problems with using the commit size in the analyses of software projects:

1. The commit size can only be estimated, not measured. Once a developer contributed a piece of code, all we can reliably count is the number of lines added and removed. Of these, some lines may be changed lines, but we have no way of knowing for certain, short of asking the original developer.
2. Using diff algorithms, we can make informed guesses about changed lines of code, but such algorithms need to go back to the original source code; this makes large-scale aggregate analyses infeasible, as we would have to work through the potentially large revision histories of many projects.

3. Large-scale aggregate analyses of open source projects are typically based on project information repositories like FLOSSmole [8] [14], FLOSSmetrics [7] or Ohloh [16]. These repositories provide only partial information, if any. Typically, they provide the number of lines added or removed.

The best reliable information that today's tools can generate is the number of source code lines added and removed. Project information repositories like Ohloh provide this minimal information. However, using this information to determine commit sizes in a naïve way, for example by adding the number of lines added and removed, leads to inaccurate results. Thus, we need to take additional steps to more precisely estimate commit sizes.

This paper presents a simple statistical algorithm (based on linear regression) to determine first diff and then commit sizes of code contributions to software projects. The contributions of the paper are the following:

- It makes prior informally used approaches of estimating commit sizes explicit and discusses their strengths and weaknesses;
- It presents a new and more accurate approach to estimating commit sizes than has been available before and applies it to open source software projects.

The paper is organized as follows. Section 2 discusses the problem situation and prepares the ground with some basic definitions. Section 3 compares existing approaches with each other and discusses their limitations. Section 4 presents a new algorithm and evaluates its correctness and efficiency. Section 5 discusses the limitations of the algorithm and Section 6 presents some final conclusions.

2 Commits and Diffs

2.1 Commit size definition

A ***commit*** is the atomic contribution of source code to a code repository. Source code typically contains program code lines, comment lines, and empty lines. The size of a commit, or ***commit size***, is defined as the sum of the number of source code lines (or source lines of code, SLoC) added, removed, or changed. The commit size is a good approximation of the amount of work that went into the commit.

There are many uses of the commit size in modeling software development processes, practices, and metrics. Typically, it is used to compute models using the revision history of the projects' code repository.

For example, Godfrey et al. have used the commit size as a measure for determining the change speed of software systems [9]. Weißgerber et al. correlate the commit size of patches with their likelihood of getting accepted into an open source project's code base and find that small patches have the highest chances of getting in [17]. Neither Godfrey et al. nor Weißgerber et al. explain how they estimate commit sizes. Hindle et al. provide a classification of large commits and define a large commit as a commit that "includes a large number of files" [13].

We have used the commit size as a dependent variable in the analysis of the adoption of the agile methods' practice of continuous integration in open source projects [6]. We have also interpreted the commit size as an independent variable in the correlation analysis of metrics of interest, for example, comment density in open source software code [2]. Finally, we have used it to analyze the overall commit size distribution of open source and the total code growth of open source [1].

2.2 Estimating the commit size

It is impossible, post-facto, to determine the size of a commit with certainty, simply because without further knowledge we cannot know whether a changed line was really just changed, counting as one line for the commit size, or whether it was removed and then independently added, counting as two lines of work.

A commit consists of several diffs, one for each file (compilation unit). A ***diff*** captures the changes made between two consecutive versions of the same file [15] [12]. The ***diff size*** is the sum of the lines added, removed, or changed in the diff. The ***commit size*** is defined as the sum of the sizes of the constituting diffs.

The computation of a diff is typically performed by a utility of the same name. All a diff utility tool can reliably measure is the number of lines added and removed in a file. An equal number of added and removed lines may actually be changed lines; however, the diff utility cannot determine this with certainty.

We call this measure a ***diff data pair*** (a, r) of source code lines added (a) and removed (r). A diff data pair represents one or more different diff events, all with their own diff size. A ***diff event*** is a triple (a, r, c) of actual source code lines added, removed, or changed. However, given a diff data pair (a, r) we do not know which diff event actually happened.

Table 1 shows an example of a diff data pair, where the diff utility reports that five source code lines were added and three were removed. As can be seen in Table 1, the pair (5, 3) can signify any one of the four different diff events (5, 3, 0), (4, 2, 1), (3, 1, 2), and (2, 0, 3).

Table 1: Example diff data pair and its interpretation

(5, 3)	Number of SLoC added	Number of SLoC removed	Number of SLoC changed	**Diff Size**
Event 1	5	3	0	**8**
Event 2	4	2	1	**7**
Event 3	3	1	2	**6**
Event 4	2	0	3	**5**

As stated, the ***diff size*** is the sum of the lines added, removed, or changed in the diff.

$$diff\_size(a, r, c) = a + r + c \quad \text{// size of diff event} \quad (1)$$

In Table 1, each of the diff events has a unique size of 8, 7, 6, or 5 source code lines.

Let (a, r) be a diff data pair, that is, let a be the number of source code lines added in a given commit, and r be the number of lines removed under the assumption that no lines where changed. Then:

```
lower_bound(a, r) = max(a, r)                                   // lower bound of diff size    (2)
upper_bound(a, r) = a + r                                       // upper bound of diff size    (3)
range(a, r) = [lower_bound(a, r), upper_bound(a, r)] // diff size range                        (4)
cardinality(a, r) = upper_bound(a, r) – lower_bound(a, r) + 1 // # events                      (5)
```

Since a diff data pair (a, r) can stand for several possible diff events, each with their own different size, we don't know the size of the underlying diff event. We can only estimate. The next Section discusses current approaches to estimating the diff size.

A diff data pair can be computed from a given diff event by adding the changed lines back to both the added and removed lines. Effectively, one changed line is a line added and a line removed and thus needs to be added to both. DDP stands for diff data pair, and DE stands for diff event.

$$a_{DDP} = a_{DE} + c_{DE} \quad (6)$$

$$r_{DDP} = r_{DE} + c_{DE} \quad (7)$$

The commit size, again, is the sum of the sizes of the constituting diffs. Once we estimated the diff sizes, computing the commit size becomes a simple summation.

3. Efficacy of Current Approaches

3.1 Current approaches to estimating diff sizes

We know of the following six approaches having been used in estimating diff sizes:

- ***Lower Bound.*** Starting with a diff data pair (a, r), the lower bound (Equation (2)) is used to estimate the diff size;
- ***Upper Bound.*** Starting with a diff data pair (a, r), the upper bound (Equation (3)) is used to estimate the diff size;
- ***Bounds Mean.*** Starting with a diff data pair (a, r), the diff size is calculated as the mean value of the lower and upper bound;
- ***GNU diff.*** Starting with the original source code, the GNU diff utility is used to estimate changed lines and then diff size (based on solving the common longest subsequence problem) [10];

- ***GNU diff -d.*** Like GNU diff, with the option –d set, which is frequently used but poorly defined. The man page explains the –d option by stating: "Try hard to find a smaller set of changes" [11].
- ***Ldiff.*** Starting with the original source code, the ldiff utility is used to estimate changed lines and then diff size (ldiff is based on calculating the Levenshtein edit distance between two text blocks) [3].

The diff data pair (a, r) used in the Lower Bound, Upper Bound, and Bounds Mean approaches represents the number of lines added and removed under the assumption that no lines were changed.

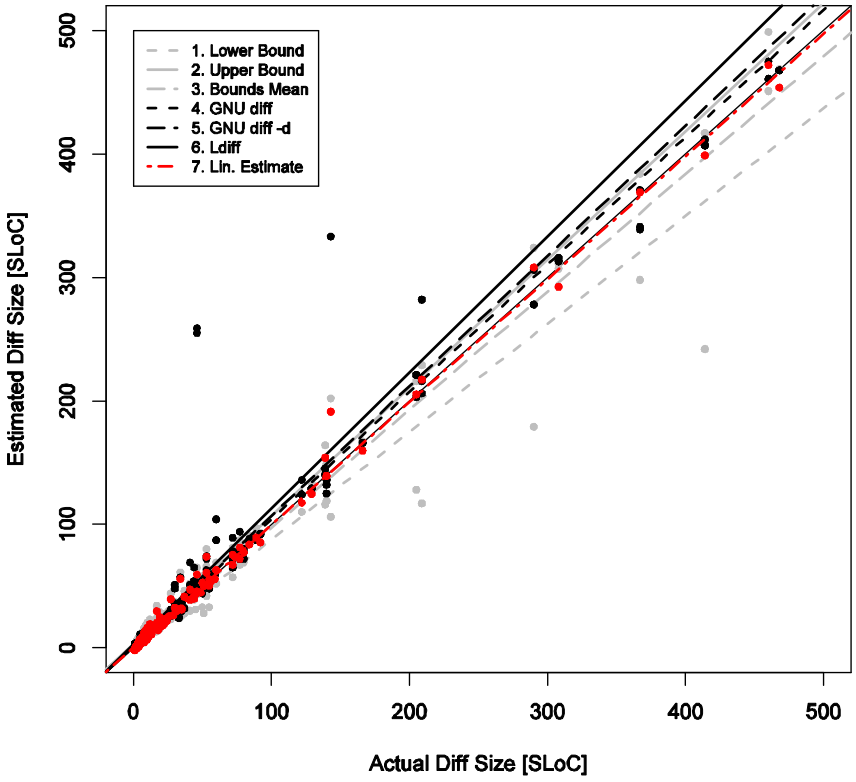

Figure 1: Comparison of diff size estimation approaches

GNU diff and ldiff both provide a diff event triple (a, r, c) which they consider to be the most likely event to have happened. Using Equation (1), the diff size is computed by summing up a, r, and c. We chose GNU diff because it is the most widespread implementation of diff and ldiff because of its superior performance in recognizing changed source code lines.

Figure 1 illustrates how well approaches 1 - 6 do when measured against a ground truth [4]. It also shows the performance of a new algorithm called "linear estimation" described in the next section. The ground truth consists of 229 diffs that have been validated by hand for accurate accounting of changed lines.

The x-axis of Figure 1 provides the true commit size as provided by the ground truth by summing up a, r, and c. The y-axis provides the commit size as estimated by Algorithms 1 - 6 as defined above. The closer an algorithm's curve is to the diagonal curve, the better it performs. The color-coded dots are the individual estimates provided by an algorithm, and the correspondingly colored curve represents a linear approximation of the algorithm.

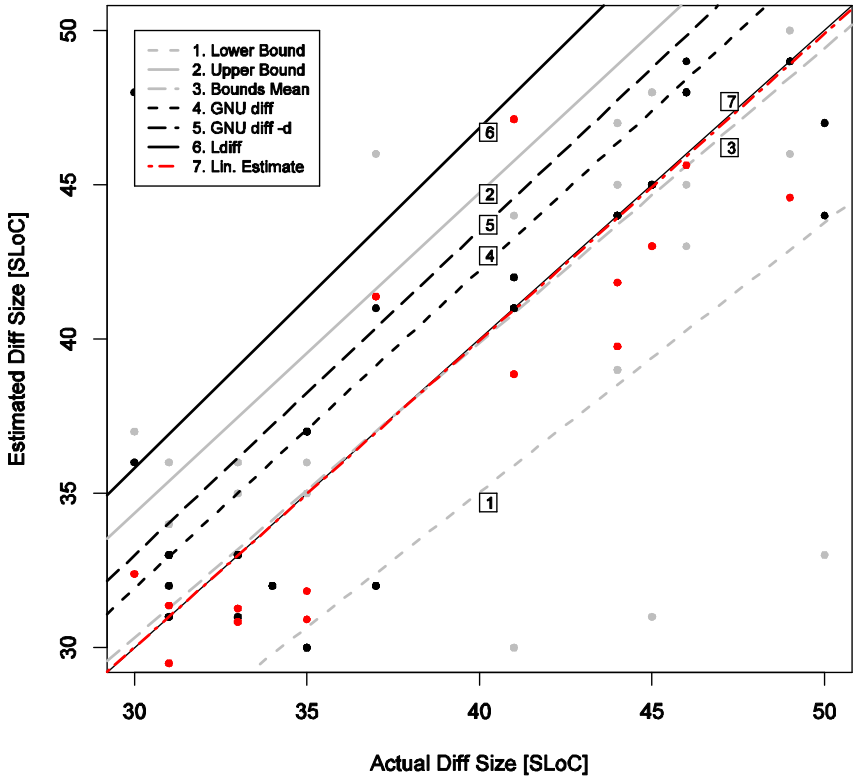

Figure 2: Actual diff size range 30 - 50 for the illustrated algorithms

Curve 1 represents the lower bound and curve 2 represents the upper bound. Together they define a corridor of possible commit sizes. Curve 3 represents the mean of the upper and lower bound. Curve 4 represents the algorithm based on GNU diff, curve 5 represents the algorithm based on GNU diff –d, and curve 6 represents the algorithm based on ldiff. Despite its superior performance in recognizing changed

code lines (but not added or removed lines) [3] [4], ldiff provides impossible values and performs poorly in this task. We have not been able to further determine a reason for this behavior.

The curves used to represent these algorithms are based on linear regressions for the given data set. (There is no simple curve, because multiple different diff events can have the same commit size.) Curve 7 is our own solution discussed below.

Figure 2 zooms in on the diff size range of 30 - 50 SLoC to illustrate how the different approaches diverge as diff sizes get larger.

From the visual illustration in Figures 1 and 2, and statistically supported in the next subsection, we can see that all text-analysis-based algorithms (GNU diff, ldiff) perform worse than the simple Bounds Mean algorithm. (This is not a statement about the performance of the diff tools themselves; rather, it is a statement about their suitability for estimating diff sizes.) Thus, the best performing simple heuristic is the Bounds Mean algorithm.

3.2 Evaluation of current approaches

The error of an estimate provided by any of the algorithms is the difference between the true value, as provided by the ground truth, and the estimated value. The occurrence of diff sizes can be considered a random variable. Hence, the error of any of the algorithms is a random variable as well.

Table 2 shows the mean of the error and its standard deviation under the assumption that the error random variable follows a normal distribution.

Table 2. Statistical evaluation of Algorithms 1 - 6 and the new approach

	Approach	Error Mean	Error Standard Deviation
1	Lower Bound	3.86	16.64
2	Upper Bound	-4.41	6.39
3	Bounds Mean	-0.27	7.68
4	GNU diff	-1.96	19.55
5	GNU diff –d	-3.06	30.87
6	Ldiff	-5.95	40.35
7	Linear Estimation	0	5.44

The perfect algorithm would not only have an error mean of zero SLoC but also an error standard deviation of zero SLoC. As can be seen, the best performing algorithm is Algorithm (7), based on linear regression of the ground truth, which will be discussed in the next section.

3.2 Performance of computation

Algorithms 1 - 3 above are fast if the diff data pair is known. The main computational expense is determining the diff; to this the algorithms add only constant time.

Algorithms 4 - 6 require scanning (though not necessarily parsing) the source code, which already puts the computation of metrics based on a large number of projects out of reach, in particular if the complete revision history of a project is involved. The original ldiff implementation from [5] performs much worse than GNU diff in terms of runtime; we measured a 1,000 times slower execution. (It should be noted that this is mostly due to the tool chain, not the actual algorithmic complexity.)

Many of the metrics and models based on the commit sizes of software projects work on large data sets. Computing the average commit size for a given successful open source project may mean estimating the sizes of tens of thousands of commits. Our database contains over 8 million commits in total [1].

Also, software development firms like SAP or Google or well-known services like SourceForge experience tens of commits per second. Tracking the sizes of such commits requires well-performing implementations. Thus, we developed an algorithm that can efficiently estimate commit sizes and handle large data sets fast.

4 An Improved Solution Based on Linear Regression

4.1 The general solution

We are looking for an algorithm that given a diff data pair (a, r) estimates a single scalar value, the diff size:

$$\text{diff\_size} \leftarrow (a, r) \tag{8}$$

Assuming linear behavior, such an algorithm needs to take the following form:

$$\text{diff\_size}(a, r) = c_a \times a + c_r \times r + b \tag{9}$$

In Equation (9), a and r represent the diff data pair of code lines added and removed, coefficients c_a and c_r represent the percentage that a respectively r contribute to the diff size, and b represents the y-axis (diff size) intercept.

The coefficients c_a, c_r, and b can be calculated using linear regression of a sample population (ground truth) that is representative for the projects of which we intend to estimate commit sizes.

Using (9) to estimate diff sizes is fast: The only input needed is a diff data pair (a, r) and each diff size can be computed in constant time, equally fast to the heuristics discussed in the previous Section. Thus, the algorithm works well when estimating diff sizes and then commit sizes in large data sets.

The algorithm for estimating commit sizes sums up the diff sizes in each of the diffs that belong to the commit.

$$\text{commit\_size}(\text{diffs}) = \sum_{\forall d \in diffs} \left[c_a \times a_d + c_r \times r_d + b \right] \quad (10)$$

In Equation (10), a_d refers to the code lines added in the given diff data pair and r_d refers to the code lines removed.

Unfortunately, these coefficients might depend on the software under investigation. The most coarse-grain distinction is between open and closed source software. We cannot assume that open source programmers exhibit the same commit behavior as closed source programmers. But even within open source one might think that not all projects are alike and that other factors like project size and age have an influence on commit behavior.

4.2 A solution for open source

The ground truth introduced earlier provides us with sufficient data to perform a linear regression that is representative of the underlying project, PostgreSQL, and as we suggest below, also representative of most of open source.

The linear regression over the ground truth provides us with the coefficients and the intercept value. Table 3 shows the results of the regression:

Table 3. Linear regression over ground truth for estimating diff sizes

	c_a [SLoC]	c_r [SLoC]	b [SLoC]
Coefficients and Intercept	0.9497	0.9744	-2.9965
Standard Error for the Coefficients	0.0065	0.0082	0.3954
R-square:	0.9951	Standard Error:	5.4677

Thus, part of the algorithm for estimating diff sizes of projects represented by the ground truth consists of computing the following equation for a given diff data pair:

$$\text{diff\_size}(a, r) = 0.9497 \times a + 0.9744 \times r - 2.9965 \quad (11)$$

Curve 7 in Figures 1 and 2 were computed using this equation. The high R-square value in Table 3 suggests a high goodness of fit. Hence, the superior performance of the algorithm as shown in Table 2 comes as no surprise.

For the diff event pair (1, 1), Equation (11) estimates a diff size below zero. To better cope with this situation, Table 4 presents a linear regression over the ground truth where we enforce b = 0, that is, fixed the y-axis intercept at zero.

Table 4. Linear regression over ground truth fixed at y-axis intercept zero

	c_a [SLoC]	c_r [SLoC]	b [SLoC]
Coefficients and Intercept	0.9370	0.9590	0
Standard Error for the Coefficients	0.0089	0.0067	0.3954
R-square:	0.9947	Standard Error:	6.1099

For small commits, Table 4 provides a superior regression. Hence, we use Equation (12), resulting from the regression in Table 4, for small diffs.

$$diff\_size(a, r) = 0.9370 \times a + 0.9590 \times r \quad (12)$$

By equating the diff size of Equation (11) with the one of Equation (12) we can determine the switch-over condition where, with increasing diff sizes, we switch from using Equation (12) to Equation (11).

$$0.01269 \times a + 0.01540 \times r > 2.9965 \quad (13)$$

Thus, the complete algorithm is a conditional computation (13) of either Equation (11) or (12), provided as pseudo code (14) below.

```
function real diff_size(int a, int r)                          (14)
    if (0.01269 × a + 0.01540 × r > 2.9965)
        return 0.9497 × a + 0.9744 × r – 2.9965
    else
        return 0.9370 × a + 0.9590 × r
    end
end
```

In prior work [1], we used a different and substantially more complex algorithm. We computed a probability distribution for diff sizes mapped into the range [0, 1] by applying the heuristics of Section 3 to a set of open source projects representative of all of open source. The projects were of different programming languages, ages, sizes, etc.

The distribution provides us with the probabilities of each possible diff event for a diff data pair. This probability distribution, which was derived using a completely different data source, provides the same results as Equation (11) above. For this reason, we consider (11) as representative of open source, not just PostgreSQL.

5 Limitations of Approach

As described, the algorithm is more accurate than prior algorithms and equally fast to the heuristics of Section 3. Here we discuss limitations and possible threats to validity of the presented work:

- ***Linearity assumption.*** One may argue that diff sizes are not a linear function of a diff data pair but follow some other function. We have not found any indication of what other function this might be. Any solution has to fit into the linear-growth lower/upper bounds corridor; this constraint excludes most other simple solutions like polynomial functions. Hence we have stuck with the simplest assumption, a linear function. The high R-square value of 0.99 from the linear regression supports this assumption.
- ***Sample size.*** Naturally, a larger sample could lead to a more accurate regression and hence algorithm for estimating diff sizes. We feel that 229 diffs can be improved upon but are already sizeable enough to deliver a useful result, as our evaluation in Section 3 showed.
- ***Sample bias.*** We did not determine the sample ourselves but rather received it from a different research group [4]. Cerulo confirmed that the sample was randomly picked from the set of available diffs in the PostgreSQL revision history. Thus, the main bias may be that PostgreSQL is not representative of all software development projects. As discussed above, we believe that it is representative of open source software projects. However, it is unclear whether it is representative of closed source software projects. We cannot tell and intend to improve our ground truth in future work to more comprehensively represent software development projects. Also, we are currently undertaking a large-scale investigation of the comparability of open source with closed source.
- ***Granularity error.*** The granularity of a diff is on the level of files. However, programmers change files in chunks rather than randomly. Thus, we could improve accuracy by breaking files into sections. However, this argument applies to the diff algorithms and not to the commit size estimation algorithms. Thus, we don't think this applies to our work.

6 Conclusions

In this paper we present several algorithms for estimating the size of code contributions (commits) to software projects. Our performance evaluation shows that simple straightforward heuristics are superior to the more complex text-analysis-based algorithms. Not only are the heuristics significantly faster to compute, they also deliver more accurate results when estimating commit sizes. Based on this experience, we design and present an algorithm that improves on the heuristics, can be computed in constant time given some diff input, and is the most accurate algorithm known today.

Acknowledgements

We are indebted to Luigi Cerulo and his colleagues Gerardo Canfora and Massimiliano Di Penta for providing us with the ground truth that this paper builds on. Their generosity saved us time and effort that we could spent on the actual algorithms and their evaluation.

References

[1] Oliver Arafat, Dirk Riehle. "The Commit Size Distribution of Open Source Software." In *Proceedings of the 42nd Hawaiian International Conference on System Sciences* (HICSS 42). IEEE Press, 2009. Forthcoming.

[2] Oliver Arafat, Dirk Riehle. "The Comment Density of Open Source Software Code." In *Companion to the Proceedings of the 31st International Conference on Software Engineering* (ICSE 2009). 4 pages.

[3] Gerardo Canfora, Luigi Cerulo, Massimiliano Di Penta. "Identifying Changed Source Code Lines from Version Repositories." In *Proceedings of the Fourth International Workshop on Mining Software Repositories*. IEEE Press, 2007. Pages 14pp.

[4] Luigi Cerulo. Private communication, 2008.

[5] Carlo Daffara. "How Many Stable and Active Libre Software Projects?" Retrieved on Sept 13, 2007, from http://flossmetrics.org/news/11

[6] Amit Deshpande, Dirk Riehle. "Continuous Integration in Open Source Software Development." In *Proceedings of the Fourth Conference on Open Source Systems* (OSS 2008). Springer Verlag, 2008. Page 273-280.

[7] FLOSSmetrics. See http://flossmetrics.org/

[8] FLOSSmole. See http://ossmole.sourceforge.net/

[9] Michael Godfrey, Xinyi Dong, Cory Kapser, Lijie Zou. "Four Interesting Ways in Which History Can Teach Us About Software" In *Proceedings of the First International Workshop on Mining Software Repositories*. IEEE Press, 2004. Pages 58pp.

[10] GNU diff. See http://www.gnu.org/software/diffutils/diffutils.html

[11] GNU diff –d. See man page to [10]

[12] Paul Heckel. "A Technique for Isolating Differences Between Files." *Communications of the ACM*, Volume 21, Number 4 (April 1978). Pages 264-268.

[13] Abram Hindle, Daniel M. German, Ric Holt. "What Do Large Commits Tell Us? A taxonomical study of large commits." In *Proceedings of the*

Fifth International Workshop on Mining Software Repositories. IEEE Press, 2008. Pages 99pp.

[14] James Howison, Megan Conklin, Kevin Crowston. "FLOSSmole: A Collaborative Repository for FLOSS Research Data and Analyses." *International Journal of Information Technology and Web Engineering*, Vol. 1, Issue 3.

[15] J. W. Hunt and M. Douglas McIlroy. *An Algorithm for Differential File Comparison*. Bell Telephone Laboratories CSTR #41, 1976.

[16] Ohloh.net. See http://ohloh.net/

[17] Peter Weißgerber, Daniel Neu, Stephan Diehl. "Small Patches Get In!" In *Proceedings of the Fifth International Workshop on Mining Software Repositories*. IEEE Press, 2008. Pages 67pp.