

KIREPRO1PE

Compression of Programs and the Similarity Distance

Jonas Nim Røssum <jglr@itu.dk>

Supervisor: Mircea Lungu <mlun@itu.dk>

May 15, 2025

*They took away the old timbers from time to time, and put new and sound ones in their places, so that the vessel became a standing illustration for the philosophers in the mooted question of growth, some declaring that it **remained the same, others that it was not the same vessel.***

— Plutarch, Life of Theseus 23.1

I would like to thank Christian Gram Kalhauge <chrg@dtu.dk> for proposing the idea for this project and for his valuable external supervision and guidance.

Contents

1. Introduction	4
2. Introduction to Similarity Distance Metrics	5
2.1. Information distance	5
2.2. Lines of Code Changed as a measure of information distance	5
3. Shortcomings of LoCC	6
3.1. First problem: Ambiguous LoCC definitions	6
3.2. Second problem: Rename-detection pitfalls	6
3.3. Third problem: Automation-driven LoCC spikes	6
3.4. Fourth problem: Time bias	7
4. Mitigating the problems of LoCC using Compression Distance	8
4.1. Compression Distance	8
4.1.1. Mitigating survivorship bias in compression distance	9
5. Present method	10
6. Present implementation	11
7. Present experiments	12
7.1. Compression Distance vs. manual subjective classification	12
7.2. Compression Distance vs. semi automatic objective classification	12
7.3. Compression Distance aggregated over authors	12
7.3.1. Observations	12
7.4. Limitations of compression distance	12
7.5. Git Truck	12
Bibliography	13
Index of Figures	14
Index of Listings	14

1. Introduction

Software engineering has consistently explored metrics to measure and compare the complexity and evolution of software systems. Among these, Lines of Code Changed (LoCC) has become a commonly used measure, providing insights into developer productivity and team velocity. However, as this paper will show, LoCC has limitations that need to be considered when interpreting its results. In this paper, we will explore the shortcomings of LoCC and propose an alternative metric: Compression Distance (CD), derived from lossless compression algorithms with large search windows. By measuring the compressibility of changes between software revisions, CD offers a novel perspective on software evolution, addressing many of the shortcomings inherent in LoCC.

This work proposes an alternative approach: Compression Distance (CD), a metric derived from lossless compression algorithms such as ZStandard (zstd) with large search windows. By measuring the compressibility of changes between software revisions, CD offers a novel perspective on software evolution, addressing many of the shortcomings inherent in LoCC. This paper explores the theoretical underpinnings of CD, its practical implementation, and its potential to complement or even replace traditional metrics in certain contexts.

Through a series of experiments, we evaluate the effectiveness of CD in quantifying code complexity, distinguishing between commit types, and mitigating biases present in LoCC.

RQ1 Is the compression distance using zstd a more representative metric for quantifying the complexity than lines of code changed?

RQ2 To what extent does compression distance using zstd discriminate between manual or semi-automatic commit types (e.g. such as bugfix, feature, refactoring, documentation and style)?

RQ3 Does the compression distance using zstd suffer from the same limitations as lines of code changed in quantifying the contributions of developers?

2. Introduction to Similarity Distance Metrics

2.1. Information distance

In the field of information theory, the concept of *information distance* is used to quantify the similarity between two objects (TODO:REF). This is done by measuring the amount of information needed to transform one object into another. The most common way to measure this distance is by using a *distance metric*, which is a function that quantifies the difference between two objects.

While there exists no perfect such metric, we can approximate the information distance between two objects using various practical techniques.

2.2. Lines of Code Changed as a measure of information distance

Lines of code (LoC) [1] is one of the most widely used distance metrics in the software industry [2]. It is often used as a measure of software complexity, maintainability, and productivity.

Since many projects employ version control systems, such as Git (TODO: Ref to Git), for keeping track of changes, we can track the *Lines of code changed* (LoCC) over time using diffing algorithms. The LoCC metric is typically defined as the number of lines added and removed in a commit. (TODO: source?). This provides a measure of the information distance between revisions of a software system. Git includes this functionality by default (TODO: Ref numstat).

This is a commonly used technique used to detect activity in software systems over time [3]. It can be used to assess team velocity, developer productivity and more. These metrics can be automatically obtained via version control systems using tools like Git Truck[4]. LoCC is a useful metric for quantifying contributions or regions of interest in software systems over time and tools like Git Truck have proven the effectiveness in the analysis of software evolution [5], [6].

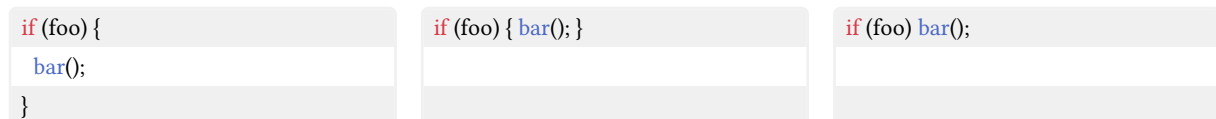
3. Shortcomings of LoCC

There are multiple problems when relying on LoCC as a sole metric of productivity.

3.1. First problem: Ambiguous LoCC definitions

Firstly, the term itself is ambiguous and subjective to the formatting of the code. One could say that line breaks are just an arbitrary formatting character. In reality, the program could have been written in a single line of code.

See the following ambiguous examples in Listing 1. A few questions arise: should these snippets be counted as three separate lines of code or collapsed into a single line? In terms of actual work, a developer who writes either version is equally productive, yet if we simply count physical lines changed, the author of the first listing would be credited with three times the contribution of the second. This discrepancy shows how formatting alone can skew LoCC-based distance measures, as trivial style differences inflate the perceived distance between revisions and undermine the metric's reliability.



```
if (foo) {  
  bar();  
}  
  
if (foo) { bar(); }  
  
if (foo) bar();
```

Listing 1: Ambiguous line counts

3.2. Second problem: Rename-detection pitfalls

The LoCC metric can be distorted also by file renames, which may artificially inflate contribution counts, since renaming a file requires little effort but appears as significant line changes. Git supports rename detection using a similarity threshold¹, comparing deleted and added files to identify likely renames, utilized by tools such as Git Truck. While Git doesn't exclude renames by default, this tracking can be used to filter them out manually. However, rename detection is inherently ambiguous — at what point should we stop treating a change as a rename and instead count it as having deleted the old file and added a new one?²

The issue is further exacerbated when developers squash commits, potentially losing rename information. Ultimately, it's a trade-off between overcounting trivial renames and missing substantial, legitimate contributions.

3.3. Third problem: Automation-driven LoCC spikes

Another case where the LoCC metric falls short is when performing automated actions that affects a vast amount of files and leads to spikes in line changes. Examples of this include running formatting and linting scripts or automated lock file updates when installing packages, all of which can lead to astronomical amounts of line changes.

Figure of git truck showing file with high activity / many automated commits

¹<https://git-scm.com/docs/git-log#Documentation/git-log.txt-code-Mltngtcode>

²https://en.wikipedia.org/wiki/Ship_of_Theseus

These kinds of changes do not directly reflect genuine development effort, but they still result in high LoCC values from which one often draw this conclusion. This shows why this metric cannot stand alone as a measure of productivity. This may cause the results to be misinterpreted, overstating the activity levels of certain developers or areas of the system.

3.4. Fourth problem: Time bias

Tools like Git Truck track the LoCC through the entire history of a project by default, weighing ancient changes as much as recent changes. In the development of Git Truck, this was attempted to be mitigated by looking at blame information³ instead and only considering how the files that are still present in the system as of today has changed through time. However, this technique is still prone to the errors introduced by the problem stated in Section 3.2. Another attempt to solve this was the introduction of the time range slider⁴, which led the user select a duration of time to analyze and “forget the past”.

³<https://github.com/git-truck/git-truck/commit/12582272b5854d6bf23706b292f3519750023fdd>^o

⁴<https://github.com/git-truck/git-truck/pull/731>^o

4. Mitigating the problems of LoCC using Compression Distance

To mitigate the ambiguity described in Section 3.1, we can instead consider the change in the number of bytes instead of LoCC as a quantifier of developer activity.

However, this means we can no longer rely on line-based diffing algorithms, akin to those used in `TODO:GIT_DIFF_REF` and must use an alternative method to measure the distance between revisions.

We can mitigate the problem with renames by concatenate all the files existing in each commit, the concatenated commit buffer, measuring its size of the project before and after the commit to calculate a **byte distance metric**.

$$\Delta|R| = |x| - |x - 1|$$

where $\Delta|R|$ is the distance in bytes, $|x|$ is the size of the given commit buffer and $x - 1$ is the size of the previous commit buffer.

This method does not address the problem with automatic actions overstating developer impact.

This leads us to the Compression Distance.

4.1. Compression Distance

Instead of measuring the static distance in bytes before and after the commit, we can instead measure the compression of the changes we added. Using a lossless compression algorithm, we compress the concatenation of the given commit buffer x with the newest revision of the project y and compare this value with the one for the previous commit $x - 1$. The hypothesis being that this would make it such that large repetitive actions would have a lower impact, since they would compress better than smaller, but more complex changes. This assumes that the changes added in a commit will compress better in the presence of similar code. This requires the compression algorithm to have large search window, as explored in [7]. This is the case for the ZStandard compression algorithm (zstd) [8], which has for the default level 3 compression has a window log of 21 [9] making the window size $2^{21} = 2\text{MB}$, making it suitable for this task, as long as you are aware of the limit and remember to adjust it as needed.

We define the Compression Distance CD metric as a measure of how much a given concatenated commit buffer compresses with the baseline commit buffer:

$$\text{CD}(x, y) = |Z(x)| - |Z(x \cup y)| \quad (1)$$

where CD is the compression distance, x is the given concatenated commit buffer and Z is a lossless compression algorithm.

Now we can define the impact of the commit ΔCD , compared to the previous commit buffer, giving us

$$\Delta \text{CD} = \text{CD}(x) - \text{CD}(x - 1) \quad (2)$$

Unlike the normalized compression distance, this metric is not bounded and is dependent on the absolute sizes of the compressed files. This is fine in this scenario, as we want to compare commits in the same project and the magnitude of the compression distance tells us an interesting story about the impact of the commit.

Then, we can compute the change in the size in bytes before and after the commit.

If we attempt to compress the commit buffers in presence of the newest commit buffer of the repository, we get the side effect of introducing survivorship bias into the system. By measuring the compressed distance to the newest commit buffer of the project, we value changes that are more akin to the newest version higher, in order to tell a story about how we got to the newest version and which commits were the most influential in getting there. This might not be what you want, but for some purposes this makes a lot of sense, as long as you keep the survivorship bias in mind. For example, a detour in the project that didn't make it in the newest version is weighted by a negative distance, telling us this brought the project further away from the newest version, but it might still have been a valuable journey to take. Making "mistakes" is what move projects forward, since you might realize what *not* to do, in order to find out what *to do*.

4.1.1. Mitigating survivorship bias in compression distance

If we intend to value these types of detours better, we can use the magnitude of the distance instead, which then gives us the impact of the commit, no matter if it moved the project closer or further from the project

To mitigate these problems, we can use an alternative approach where we concatenate the entire state of the repository, makes the analysis resistant to renames, as we don't care about file names, we only care about the bytes contained in the commit. However, in this case, we can no longer rely on diffing the concatenated string. We need to use another approach to derive the information distance from before and after a commit.

Normalized Compression Distance (NCD) is a way of measuring the similarity between two compressible objects, using lossless compression algorithms such as GZIP and ZStandard. It is a way of approximating the Normalized Information Distance and has widespread uses such as cluster analysis [<https://arxiv.org/abs/cs/0312044>], and it has even been used to train sentiment analysis.

However, nobody has used normalized compression ratio as a distance metric in version controlled systems yet. The hypothesis of this work is that NCD-derived metrics could function as a complement to the more well-known LC and NoC metrics mentioned above, and be resilient to renaming.

5. Present method

Q: Why did we choose z standard? [7] recommends A: zstd has a large search window <https://en.wikipedia.org/wiki/Zstd>^o

According to [7], NCD is a very good distance measurement, when used in the proper way.

6. Present implementation

7. Present experiments

Present test cases

| Project | Baseline commit buffer size | Git Truck | | Commitizen | | Twotter | Size 20

time intervals that are meaningful: git truck before hand in, multiple intervals multiple projects

7.1. Compression Distance vs. manual subjective classification

To answer RQ1

- Correlation between CD delta and complexity

7.2. Compression Distance vs. semi automatic objective classification

To answer RQ2

7.3. Compression Distance aggregated over authors

To answer RQ3

<http://localhost:3000/get-commits/?repo=git-truck&branch=645333e46ceea6abd5ee1d4a0cb2c605bd959725&count=Infinity>° <http://localhost:3000/get-commits/?repo=git-truck&count=Infinity>°

7.3.1. Observations

“Reasons for skews:

- Code that was deleted again and no longer present in final version“

Intentional survivorship bias

7.4. Limitations of compression distance

1. it's much slower to compute than traditional metrics like LoCC.
2. limited window size [7] leading to distorted distance measurements, especially in worst case when comparing identical objects exceeding these size constraints. This means that we are back to just measuring something similar to the byte distance

7.5. Git Truck

If you look beyond our bachelors project, where we all four worked on the project, you see that Dawid joined as a contributor during his master thesis. After that, Thomas did his master thesis on the project as well, which is very

Bibliography

- [1] C. to Wikimedia projects, “Source lines of code - Wikipedia.” [Online]. Available: https://en.wikipedia.org/wiki/Source_lines_of_code^o
- [2] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Boehm, “A SLOC counting standard,” in *Cocomo ii forum*, 2007.
- [3] M. Goeminne and T. Mens, “Analyzing ecosystems for open source software developer communities,” *Software Ecosystems*. Edward Elgar Publishing, pp. 247–275, 2013.
- [4] K. Højelse, T. Kilbak, J. Røssum, E. Jäpelt, L. Merino, and M. Lungu, “Git-truck: Hierarchy-oriented visualization of git repository evolution,” in *2022 Working Conference on Software Visualization (VISSOFT)*, 2022, pp. 131–140.
- [5] M. Lungu, R.-H. Pfeiffer, M. D’Ambros, M. Lanza, and J. Findahl, “Can git repository visualization support educators in assessing group projects?,” in *2022 Working Conference on Software Visualization (VISSOFT)*, 2022, pp. 187–191.
- [6] A. Neyem, J. Carrasco-Aravena, A. Fernandez-Blanco, and J. P. Sandoval Alcocer, “Exploring the Adaptability and Usefulness of Git-Truck for Assessing Software Capstone Project Development,” in *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1*, 2025, pp. 847–853.
- [7] M. Cebrián, M. Alfonseca, and A. Ortega, “Common pitfalls using the normalized compression distance: What to watch out for in a compressor,” 2005.
- [8] I. Meta Platforms, “facebook/zstd: Zstandard - Fast real-time compression algorithm.” [Online]. Available: <https://github.com/facebook/zstd>^o
- [9] I. Meta Platforms, “zstd/lib/compress/clevels.h at f9938c217da17ec3e9dcd2a2d99c5cf39536aeb9 · facebook/zstd.” [Online]. Available: <https://github.com/facebook/zstd/blob/f9938c217da17ec3e9dcd2a2d99c5cf39536aeb9/lib/compress/clevels.h#L31C7-L31C9>^o

Index of Figures

Figure 1	6
----------------	---

Index of Listings

Listing 1 Ambiguous line counts	6
---------------------------------------	---