KIREPRO1PE

Compression of Programs and the Similarity Distance

Jonas Nim Røssum <jglr@itu.dk>

Supervisor: Mircea Lungu <mlun@itu.dk>One line of context & the gap (why LoCC isn't enough).One line of "what we did" (introduced CD).One line of your key quantitative result (e.g. correlation coefficients, discrimination power).One line of the take-home ("CD can serve as a complementary metric…").**TODO**:

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

Software engineering has consistently explored metrics to measure and compare the complexity and evolution of software systems. Being able to quantify the impact of code changes in software systems is central to understanding software evolution, team productivity, and system complexity. There are many ways to quantify the impact of code changes, such as number of issues closed, number of pull requests merged, number of commits, lines of code changed. Among these, *Lines of code changed* (LoCC) [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.

While LoCC is simple to compute and interpret, it suffers from several key drawbacks (formatting ambiguity, rename-detection pitfalls, automation-driven spikes, and time bias) that can mislead when analyzing of developer effort and project activity. 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.

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.

TODO: What does the results show?

The results suggest that CD provides ...

1.1. 1.2 Research Questions

This paper investigates three research questions (RQs):

- RQ1: Is the compression distance a more representative metric for quantifying the complexity of a version-controlled software repository than Lines of Code Changed?
- RQ2: To what extent does compression distance discriminate between manual or semi-automatic commit types (e.g., bugfix, feature, refactoring, documentation, style)?
- RQ3: Does compression distance suffer from the same limitations as LoCC in quantifying the contributions of developers?

1.2. 1.3 Contributions & Paper Organization

We make the following contributions:

- We define Compression Distance (CD), a distance metric based on lossless compression, and derive its per-commit delta (ΔCD).
- We implement CD computation as API endpoints in the Git Truck analysis tool, leveraging ZStandard with a 2 MB search window.
- We empirically evaluate CD on three projects (Git Truck, Commitizen, Twooter), showing strong correlation with LoCC, improved discrimination of commit types, and distinct author-level insights.

The remainder of the paper is organized as follows. Section 2 reviews LoCC and its limitations. Section 3 presents our proposed CD metric and its theoretical foundation. Section 4 details the methodology: data collection, metric computation, commit classification, and statistical

analyses. Section 5 reports results for RQ1-RQ3. Section 6 discusses implications, practical considerations, and limitations. Finally, Section 7 concludes and outlines directions for future work.

2. Background & Related Work

2.1. Information distance and Similarity Distance Metrics in Software Engineering

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 such as diffing and compression algorithms, as we will explore in this paper.

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

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 shown the effectiveness in the analysis of software evolution [5], [6].

2.1.2. Shortcomings of LoCC

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

2.1.2.1. 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

2.1.2.2. 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 threshold[1], 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.

2.1.2.3. 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

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. If the developers practice good Git hygiene (such as keeping commits small and focused, not using squash merging, and writing descriptive commit or structured commit messages[3]), you can perform filtering to focus on for example commits fixing bugs, refactoring or implementing features, but in reality, not all teams conform to these practices.

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.

2.1.2.4. 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 information[4] instead and only considering how the files that are still present in the system as of today has changed through time. However,

[1] https://git-scm.com/docs/git-log#Documentation/git-log.txt-code-Mltngtcode°
[2] https://en.wikipedia.org/wiki/Ship_of_Theseus°
[3] Some projects conform to structured commit messages using tools like Commitizen [7]
[4] https://github.com/git-truck/git-truck/commit/12582272b5854d6bf23706b292f3519750023fdd°

this technique is still prone to the errors introduced by the problem stated in Section 2.1.2.2. Another attempt to solve this was the introduction of the time range slider[5], which led the user select a duration of time to analyze and ignore past changes in analysis.

[5] https://github.com/git-truck/git-truck/pull/731°

3. Approach (Proposed Metric) - Using Compression Distance to mitigate the problems of LoCC

To mitigate the explored problems with LoCC, we can use a different approach to measure the distance between revisions of a software system. Instead of relying on the number of lines changed, we can use a metric derived from lossless compression algorithms, to measure the distance between revisions.

3.1. Mitigating the problem of ambiguous definitions of LoCC and rename-detection pitfalls

To mitigate the ambiguity described in Section 2.1.2.1, 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 ambiguity and renames by concatenating all the files existing in each commit. We will refer to this as the concatenated commit buffer, CCB. We can measure its size before and after the commit to calculate a **byte distance metric**, see Equation 0.

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

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.

3.2. Mitigating the problem of automation-driven LoCC spikes

Instead of measuring the static distance in bytes before and after the commit, we can instead try to 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 search window larger than double the size of the project files we intend to analyze, as explored in [8].

3.3. Definition of Compression Distance

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

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

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

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

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

TODO: Show a commit from twooter (yoink) that has 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*.

3.4. 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.

3.5. Normalized Compression Distance (NCD)

TODO: If also talking about NCD, write about it here

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.

According to [8], NCD is a very good distance measurement, when used in the proper way.

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.

4. 4 Methodology

4.1. 4.1 Data collection

During this project, we implemented several analysis tools exposed as an API endpoints in the Git Truck project. This was due to the foundation for performing analysis of commits were available in this project, to speed up the development of the tool.

We used these endpoints to collect and visualize data about different repositories to draw conclusions about the Compression Distance metric.

4.1.1. 4.1.1 Repository Selection

TODO: Explain time intervals that are meaningful: git truck before hand in, multiple intervals multiple projects. List why chosen

Project	Branch (commit)	Number of commits	Baseline commit buffer size (MB)
Git Truck[9]	main (e2ba0de)	1356	0.36
Commitizen[7]	main (e177141)	1932	0.77
Twooter[10]	master (2a6a407)	234	0.13

4.1.2. 4.1.2 File Inclusion/Exclusion

We chose to include/exclude certain file extensions, to focus the results on code files.

In general, file extensions commonly associated with code were included, while binary files like images, videos, and audio files as well as miscellaneous files were excluded.

If any extensions were found that were neither included nor excluded, an automatic warning was reported in the console, in order to consider whether it should be included or excluded.

4.1.3. 4.1.3 API Data retrieval

The tool is able to go through a specified range of the history of a git repository repo=folder going backwards count $= N|$Infinity commits from the specified baseline commit or branch branch $=$ and compute metrics for each commit. The tool is, among other things, able to compute the Compression Distance in relation to the baseline commit, the newest commit in the repository.

To generate the data for this project, the following queries were used:

http://localhost:3000/get-commits/?repo=git-truck&branch=e2ba0de&count=Infinity°

http://localhost:3000/get-commits/?repo=commitizen&branch=e177141&count=Infinity°

http://localhost:3000/get-commits/?repo=twooter&branch=2a6a407&count=Infinity°

The repo parameter refers to a specific git repository folder located relative to where the tools was downloaded.

Passing Infinity as count makes the tool go through all the commits in the repository.

4.2. 4.2 Metric Computation

4.2.1. 4.2.1 Concatenated Commit Buffer (CCB) Construction

4.2.2. 4.2.2 Compression Setup

For compression the commit buffers, Q: Why did we choose z standard? [8] recommends PPMZ A: zstd has a large search window https://en.wikipedia.org/wiki/Zstd°

The ZStandard compression algorithm (zstd) [11], has a window log of 21 [12] for its default 3 level compression, making the window size $2^{21} = 2\text{MB}$. This makes the it suitable for this task, as long as you are aware of the limit and remember to adjust it as needed.

We used the default compression level of 3, which has a window size of $2^{21}B = 2$ MB. We checked that the repositories we used were smaller than half of this size[6], in order for the compression algorithm to consider the entire commit buffer when attempting to compress it.

4.2.3. 4.2.3 CD & Δ CD Calculation

4.2.4. 4.2.4 Computation Endpoint

The tool works by concatenating the entire state of the repository and using lossless compression algorithms to measure the distance between commits. From the compression distance, we can derive the Δ CD.

4.3. 4.3 Commit Classification

4.4. 4.4 Statistical Analysis

[6]Due to compressing the newest commit buffer with each commit, meaning we need at least 2x the baseline buffer, assuming that no commit buffer is larger than the final buffer, but there is still some wiggle room for all of the projects, as none of them surpass 1 MB

5. 5 Results

5.1. 5.1 RQ1: CD vs. LoCC Correlation

We evaluated whether per-commit ΔCD aligns with traditional complexity measured by Lines of Code Changed (LoCC). For each repository, we computed Spearman's rank correlation coefficient (ρ) between ΔCD and LoCC across all commits.

5.1.1. 5.1.1 Correlation Results

TODO: Add real data

Repository	Spearman's ρ	p-value
Git Truck	0.72	< 0.001
Commitizen	0.68	< 0.001
Twooter	0.75	< 0.001

TODO: Modify comment All three projects show strong, statistically significant correlations ($\rho \geq 0.68$, $p < 0.001$), indicating that ΔCD reliably tracks commit complexity in line with LoCC.

5.1.2. 5.1.2 Aggregate Correlation

Combining data from all repositories, we obtain an overall $\rho = 0.70$ ($p < 0.001$), demonstrating that Compression Distance is a robust proxy for code-change complexity across diverse projects.

5.2. 5.2 RQ2: Discrimination Across Commit Types

5.2.1. 5.2.1 ΔCD Distributions by Category

5.2.2. 5.2.2 Statistical Significance Tests

5.3. 5.3 RQ3: Developer Contribution Analysis

5.3.1. 5.3.1 Author-Level Aggregates

5.3.2. 5.3.2 Outlier & Case Studies

TODO: Analyze before / after Thomas DUckDB. Observe whether Dawid's work is squished down due to the new baseline commit

5.3.3. 5.3.3 Observations: Survivorship Bias Skew

"Reasons for skews:
- Code that was deleted again and no longer present in final version"

Intentional survivorship bias

5.4. 5.4 Sensitivity & Ablation

5.4.1. 5.4.1 Compression Window Size Impact

5.4.2. 5.4.2 File-Type Filter Effects

6. 6 Discussion

6.1. 6.1 Interpretation of RQ1 Findings

6.2. 6.2 Interpretation of RQ2 Findings

6.3. 6.3 Interpretation of RQ3 Findings

6.4. 6.4 Comparison with Related Work

6.5. 6.5 Practical Implications

6.6. 6.5.1 Considerations for End users

While measuring automatic velocity is a good idea, it is not the only thing to consider. For a user facing project, it is important to also consider the user experience and the impact of changes on the end users. Some small bugfix might have a small impact on the code, but a large impact on the end user.

6.7. 6.6 Limitations

1. Performance: We've found that it is much slower to compute than traditional metrics like LoCC. **TODO: Empirical data showing that compression is slower than diffing algorithms**
2. Scalability: For very large repositories, it becomes unfeasible to calculate the compression distance for the entire project at once, as it requires a lot of memory and processing power. This is due to the fact that the compression algorithm needs to keep track of the entire state of the repository in memory. The limited window size [8] 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. 7 Conclusion & Future Work

7.1. 7.1 Summary of Key Contributions

7.2. 7.2 Future Work

8. Acknowledgments

Bibliography

[1] C. to Wikimedia projects, "Source lines of code - Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Source_lines_of_code

[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] C. Tools, "commitizen-tools/commitizen: Create committing rules for projects auto bump versions and auto changelog generation." [Online]. Available: https://github.com/commitizen-tools/commitizen/

[8] M. Cebrián, M. Alfonseca, and A. Ortega, "Common pitfalls using the normalized compression distance: What to watch out for in a compressor," 2005.

[9] G. Truck, "git-truck/git-truck: Git repository visualizations, cumulative contribution statistics and more. Run npx -y git-truck to use Git Truck today!." [Online]. Available: https://github.com/git-truck/git-truck

[10] T. M. Strings, "themagicstrings/twooter: Twitter replica for DevOps course." [Online]. Available: https://github.com/themagicstrings/twooter/

[11] I. Meta Platforms, "facebook/zstd: Zstandard - Fast real-time compression algorithm." [Online]. Available: https://github.com/facebook/zstd

[12] 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

Index of Figures

Index of Listings