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 to Similarity Distance Metrics

1.1. 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 techniques.

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

Lines of code (LoC) [1] is one of the most widely used metrics in the software industry [2]. The LoCC metric is typically defined as the number of lines added and removed in a commit. (TODO: source?). Many projects also utilize version control systems, such as Git (TODO: Ref to Git). By utilizing the historic data, we can we can track the *Lines of code changed*, or LoCC over time using diffing algorithms. 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].

2. Shortcomings of LoCC

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

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

2.2. 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.

2.3. 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 scripts or installing packages, which often leads to a lot of line changes in tracked lock files [ref].

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°
[2] https://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.

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[3] 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 2.2. Another attempt to solve this was the introduction of the time range slider[4], 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°
[4] https://github.com/git-truck/git-truck/pull/731°

3. Compression Distance

"Definition" covers **ambiguity** in what counts as a line of code. "Automated bulk edits" addresses **non-development actions** (formatting, installs) that spike LoCC. "Renames and metadata limits" deals with how Git's **rename heuristics distort** counts.

To mitigate the ambiguity described in Section 2.1, we can instead consider the number of bytes instead of lines of code.

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, measuring its size of the project before and after the commit to calculate a **byte distance metric**.

However, we are not homefree yet. We need to attempt to address the problem with automatic actions overstating impact.

This is where the Compression Distance comes in.

Instead of measuring the static difference in bytes before and after the commit, we can instead try to see how well the changes we added compress with the previous version of the repository. This way, in theory, large, repetitive actions would have a lower impact in the productivity score.

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

This also has the side effect of introducing intentional survivorship bias into the system. By measuring the compressed distance to the final revision of the project, we value changes that are more akin to the final version higher, in order to tell a story about how we got to the final version and which commits were the most influential in getting there. This might not be what you want, but for some purposes this makes very good sense, but you still need to keep the bias in mind. For example, a detour in the project that didn't make it in the final version is weighted by a negative distance, telling us this brought the project further away from the final 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*.

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

- Correlation between CD delta and complexity

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.

4. Present method

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

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

5. Present implementation

6. Present experiments

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

6.1. Compression Distance vs. manual subjective classification

6.2. Compression Distance vs. semi automatic objective classification

6.3. Compression Distance aggregated over authors

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

6.3.1. Observations

“Reasons for skews:

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

Intentional survivorship bias

6.4. Limitations of compression distance

1. it’s much slower to compute than <- is it though???
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

6.5. 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. Af ter 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°

[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.

Index of Figures

Index of Listings