Volatility Metric to Detect Anomalies in Source Code Repositories

Yegor Bugayenko
yegor256@gmail.com
Huawei
Moscow, Russia

Abstract

A new metric was introduced to calculate the distance between actively modified files in a source code repository and the files, which are rarely modified and may be considered abandoned or even dead. It was empirically demonstrated that larger repositories have larger values of the introduced metric. The metric may be used for earlier detection of code maintenance anomalies and helping software developers make the decision of splitting the repository into smaller ones in order to prevent maintainability issues.

CCS Concepts: • **Software and its engineering → Maintaining software**.

Keywords: Metrics, Software Size, Software Maintainability

ACM Reference Format:
Yegor Bugayenko. 2021. Volatility Metric to Detect Anomalies in Source Code Repositories. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Beyond Code: No Code (BCNC '21), October 17, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3486949.3486961

1 Introduction

Most software development projects keep their source code in Git [10], which is the de-facto standard in the industry at the moment, or similar version control systems. Every system, including open source products like Git, Subversion, and Mercurial, and commercical tools like StarTeam™ or ClearCase™ have the same feature: keeping track of the changes made to the source code files, also known as "logs."

Git logs provide information about every single change made by every software developer during the entire course of the project. Using this information it's possible to measure which files are being modified more frequently than others.

BCNC '21, October 17, 2021, Chicago, IL, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9125-2/21/10…$15.00
https://doi.org/10.1145/3486949.3486961

On the other hand, it's also possible to spot files that are rarely modified and may be considered either as *abandoned* or *stable* code. Fontana et al. [6] claims that the gap between actively modified code and the code that is stable should be as narrow as possible in a properly maintained repository.

Indeed, abandoned code may be considered as a threat to the maintainability of the entire project, because its lifecycle is not aligned with the lifecycle of active code, which is modified more intensively. Later, when a modification is required to the abandonded code, a programmer may have difficulties finding out the principles of design of the code and making sure new modifications don't break it.

We introduce a Volatility metric to measure the relationship between the amount of actively modified files and files which stay in the repository for a long time without modifications. The hipothesis is that the Volatility is related to the size of the repository: the larger the repository the higher the volatility. Larger repositories are an obvious threat to code maintainability. Thus, if the relationship between size and Volatility exists, the new metric can be used to identify code anomalies and detect badly maintained repositories.

In order to validate the hipothesis and demonstrate how the metric works we applied it to 240 public Java repositories from GitHub and analyzed its impact on two other characteristics of each repository: the number of files in a repository and the size of it in bytes.

Larger repositories shown larger values of the metric.

The paper is organized as follows. Sec. 2 defines various terms used in the paper. Sec. 3 covers related work in the areas of code change metrics and empirical analysis of source code repository size. Sec. 4 covers the source code volatility metric. Sec. 5 covers our empirical case study of 240 Java source code repositories. Sec. 6 covers limitations of both the metric and the study. Finally, we summarize our study in Sec. 7.

2 Background

In this paper, we use several terms regarding version control systems and software metrics. The type of metric we are proposing belongs to the family of *source code change metrics*, which analyze the history of changes in a source code repository [3, 5]. A *change* in a version control system, such as Git, is an atomic modification to the source code file made by a software developer locally on his/her machine and then *committed* to the repository.

Each version control system, including Git, provides an instrument for retrieving the entire history of changes from a repository. We use `git log` for Git.

GitHub™ (currently owned by Microsoft™) is one of the largest platforms for *open source* projects. GitHub provides free access to all public Git repositories, which makes it possible to analyze Git histories.

3 Related Work

"Code change metrics mined from source control repositories have proven to be the most reliable predictors of bugs in contemporary software engineering research," says Muthukumaran et al. [14].

Code churn is one of the simplest metric in the family, which counts all lines being modified (added, edited, and deleted) during the entire lifetime of a project [13]. It was demonstrated by Shin et al. [15] how code churn, together with complexity and other developer activity metrics, is related to software vulnerabilities. There are modifications of code churn, for example taking into accout socio-technical aspect of the metric, as suggested by Meneely and Williams [11].

Yet another simple metric from the family is the number of changes being made to a particular file, class, method or line of code, which is used for example by Demeyer et al. [4] in order to analyze the effect of refactorings.

Moser et al. [12] made a comparative analysis of 17 change metrics to understand their efficiency for defect prediction: number of distinct authors per file, total modifications per file, total additions per file, maximum number of files committed together, and others.

Biazzini and Baudry [2] introduced a number of metrics to analyze Git history (specifically related to GitHub), such as unique-count, unique-ratio, VIP-count, VIP-ratio, scattered-count, pervasive-count and a few others. Some of the metrics are supposed to be calculated for a repository together with its forks, while others may be calculated for a single repository history.

Batista et al. [1] provided a detailed analysis of existing metrics and introduced a new one to measure, by looking at Git/GitHub project commit history, how "close" developers stay to each other and form pairs.

Fontana et al. [6] suggested to measure "File Volatility" as the ratio between the maximum number of changes happened to a file line and the number of all changes to the file since its creation. Going further, they also suggested to calculate "Repository Stability" metric as a relationship between active files (with high File Volatility) and closed files. The also concluded that repositories where most files change over time can suffer from organizational or design issues.

To our knowledge, the metric introduced in this research, in order to detect anomalies in source code repository maintenance, has never been suggested.

4 Metric

First, by looking at the Git history, it is observed how many times every source code file out of N was touched during the lifetime of the repository (excluding the files that don't exist in the repository anymore):

$$T = [t_1, t_2, \dots, t_N] \tag{1}$$

Here, t_i is the number of commits found in the repository, which modified the i-th file. Then, the entire interval between $\check{T}$ (the maximum value) and $\hat{T}$ (the minimum value) is divided to Z (the input parameter of the method) equivalent groups:

$$\delta = (\check{T} - \hat{T})/Z \tag{2}$$

$$G = [g_1, g_2, \dots, g_Z] \tag{3}$$

$$g_j = \sum_{i=1}^{N} [j \times (\delta - 1) < t_i < j \times \delta] \tag{4}$$

Here, g_j is the total number of t_i, which are larger than $j \times (\delta - 1)$ and smaller than $j \times \delta$. In other words, T, the entire set of measurements, is sorted and then split into Z sectors, where g_j is the counter of measurements that belong to the sector j.

Then, the mean μ is calculated as:

$$\mu = \frac{1}{Z} \sum_{j=1}^{Z} g_j \tag{5}$$

Finally, the variance is calculated as:

$$\text{Var}(g) = \frac{1}{Z} \sum_{j=1}^{Z} |g_j - \mu|^2 \tag{6}$$

The variance $\text{Var}(g)$ is the Volatility of the source code. The smaller the Volatility the more cohesive is the repository and the smaller the amount of the abandonded code inside it. The metric demonstrates how large is the gap between intensively modified files and the abandoned ones: the larger the variance, the wider the graph, and the larger the gap. A repository with a thousand files all being modified at the same pace would have a Volatility of zero. On the other hand, a repository with just a few files, where one of them is modified a thousand times, while others are modified just once, will have the Volatility of a miillion.

5 Empirical Results

A list of Java repositories were retrieved from GitHub via their public API. The first 240 repositories were taken, which satisfied the selection criteria: 1) more than 1,000 GitHub stars, 2) more than 200 Kb of data, 3) not archived, and 4) public. The list included popular Java open source products, such as Spring, RxJava, Guava, MyBatis, Clojure, JUnit, Lombok, Graal, Selenium, Spark, Mockito, Neo4j, Jenkins, Netty, and others.

The Volatility metric was calculated for each repository, using the formula explained above (the value of Z was set to 64). Then, a few other metrics were collected for each repository and their values were compared with the Volatility.

Fig. 1 demonstrates the relationship between the number of files in the repository (M_1) and its Volatility (V). Both axixes of the graph have logarithmic scales, for the sake of visual understandability: the difference between the minimum and the maximum values of the Volatility is logarithmically large. It is visually obvious that repositories with larger number of files tend to have higher values of the Volatility metric.

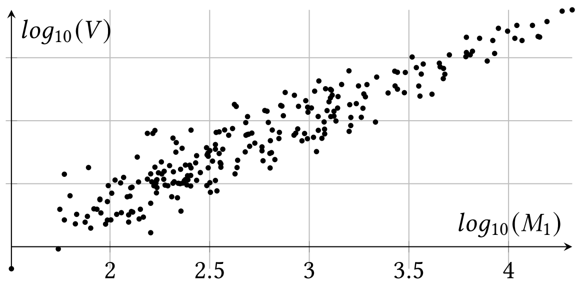

Figure 1. The relationship between the number of files in the repository and its Volatility (Z is set to 64)

Fig. 2 demonstrates the relationship between the logarithm of the size of the Git repository in bytes (M_2) and the logarithm of its Volatility (V). It is visually obvious that binary-size larger repositories tend to have higher values of the Volatility metric.

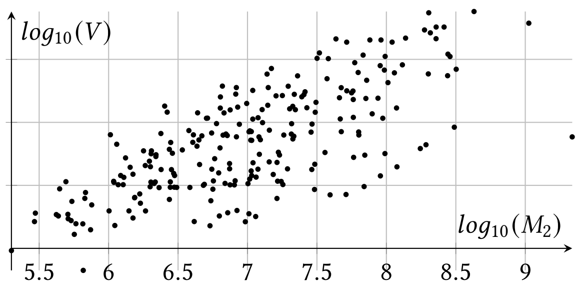

Figure 2. The relationship between the size of the Git repository in bytes and its Volatility (Z is set to 64)

Fig. 1 demonstrates the same relationship as Fig. 1, but the parameter of the Volatility formula is set to 32 instead of 64. The trend of the graph didn't change much.

Fig. 1 demonstrates the same relationship as Fig. 1, but this time the parameter of the Volatility formula is increased

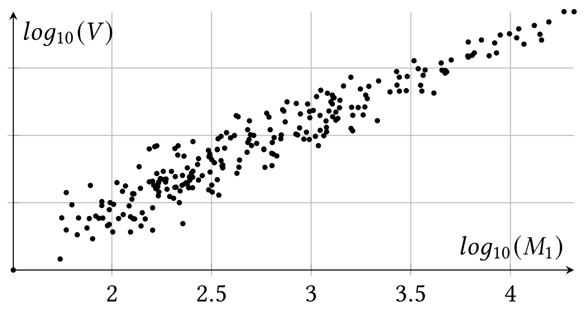

Figure 3. The relationship between the number of files in the repository and its Volatility (Z is set to 32)

to 128 instead of the original value of 64. The trend of the graph didn't change much.

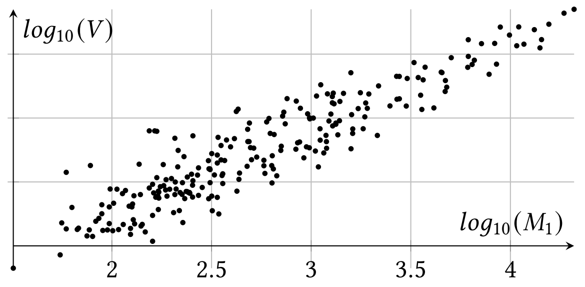

Figure 4. The relationship between the number of files in the repository and its Volatility (Z is set to 128)

6 Discussion

It was demonstrated that the size of the source code is an important factor negatively affecting maintainability and other quality characteristicts. For example, Heitlager et al. [7] suggested to consider "volume" (the number of lines per file, the number of methods in a class, and so on) as a primary contributor to an aggregate maintainability index of a software because "it is fairly intuitive that the total size of a system should feature heavily in any measure of maintainability."

However, the discussion of whether larger repositories are a better practice than smaller ones is still open. Google, for example, advocates for larger ones, known as "monolithic," which tend to keep many modules and components in one Git space, simplifying dependencies management [9]. As was demonstrated in our research, larger repositories will tend to have maintenance anomalies: some parts of source code will be changed much less frequently than other parts. Moreover, the distance between "popular" parts and "abandoned" ones will grow when the size of the repository grows.

These maintenance anomalies will lead to maintability and quality problems mostly because it will be difficult for programmers to quickly switch between different contexts inside one repository: one set of files is being modified every day, while another one stays untouched for months. A programmer will try to avoid making changes to the part that is abandonded because it won't look like code that is expecting changes.

Also, maintenance anomalies is a sign of differences in customer demand for the source code components. The module with smaller frequency of changes must be released less frequently than the module that is being edited every day. Not paying attention to the growing Volatility of the repository means ignoring anomalies and, because of that, redundant releasing of the code that doesn't require new releases. Moreover, releasing larger software modules usually means longer build cycles, which are the key failure factor of continuous delivery, as explained by [8]. Getting rid of maintenance anomalies may help reduce build cycles.

It would be beneficial in future researches to analyze the relationship between Volatility and other parameters of source code repositories, such as the programming language being used, the popularity in GitHub (the number of stars and forks), the total number of lines of code, the average number of lines of code per file, the number of Git commit authors, the total age of the repository, the number of issues and pull requests in GitHub, and many others. Would be interesting to empiricially demonstrate which of these parameters (or combinations of them) impact the behavior of Volatility and in which direction.

7 Conclusion

A new source code Volatility metric was introduced and applied to 240 open source Java repositories from GitHub. It was empirically demonstrated that larger repositories have higher values of the Volatility metric.

The source code of Ruby and Python scripts used to do the research is available in GitHub repository: `yegor256/volatility-vs-size`.

References

[1] Natércia A Batista, Guilherme A Sousa, Michele A Brandão, Ana Paula C da Silva, and Mirella Moura Moro. 2018. Tie strength metrics to rank pairs of developers from github. *Journal of Information and Data Management* 9, 1 (2018), 69–69.

[2] Marco Biazzini and Benoit Baudry. 2014. "May the fork be with you": novel metrics to analyze collaboration on GitHub. In *Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics*. 37–43.

[3] Garvit Rajesh Choudhary, Sandeep Kumar, Kuldeep Kumar, Alok Mishra, and Cagatay Catal. 2018. Empirical analysis of change metrics for software fault prediction. *Computers & Electrical Engineering* 67 (2018), 15–24.

[4] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. 2000. Finding refactorings via change metrics. *ACM SIGPLAN Notices* 35, 10 (2000), 166–177.

[5] NE Fenton and SL Pfleeger. 1997. Software Metrics.

[6] Francesca Arcelli Fontana, Matteo Rolla, and Marco Zanoni. 2014. Capturing Software Evolution and Change through Code Repository Smells. In *International Conference on Agile Software Development*. Springer, 148–165.

[7] Ilja Heitlager, Tobias Kuipers, and Joost Visser. 2007. A practical model for measuring maintainability. In *6th international conference on the quality of information and communications technology (QUATIC 2007)*. IEEE, 30–39.

[8] Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education.

[9] Ciera Jaspan, Matthew Jorde, Andrea Knight, Caitlin Sadowski, Edward K Smith, Collin Winter, and Emerson Murphy-Hill. 2018. Advantages and disadvantages of a monolithic repository: a case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. 225–234.

[10] Jon Loeliger and Matthew McCullough. 2012. *Version Control with Git: Powerful tools and techniques for collaborative software development*. O'Reilly Media, Inc.

[11] Andrew Meneely and Oluyinka Williams. 2012. Interactive churn metrics: socio-technical variants of code churn. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–6.

[12] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. 2008. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*. 181–190.

[13] John C Munson and Sebastian G Elbaum. 1998. Code Churn: A measure for estimating the impact of code change. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 24–31.

[14] K Muthukumaran, Abhinav Choudhary, and NL Bhanu Murthy. 2015. Mining GitHub for novel change metrics to predict buggy files in software systems. In *2015 International Conference on Computational Intelligence and Networks*. IEEE, 15–20.

[15] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. 2010. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE transactions on software engineering* 37, 6 (2010), 772–787.