Analysis and Improvement of the Criticality Score Algorithm

Blaise Swartwood[†] and Z. Berkay Celik[‡]

[†]Rose-Hulman Institute of Technology
[‡]Purdue University

July 19, 2023

Abstract

Open source software (OSS) projects are third-party resources increasingly targeted by software supply chain attacks. To counteract this, a criticality score algorithm identifies the most important OSS projects to prioritize their security. The algorithm uses 10 features extracted from GitHub projects in a deterministic calculation to measure their criticality. However, the current criticality score algorithm suffers from inconsistent and redundant features. In this work, we examine the weaknesses of these features through statistical analysis and identify the most important features to create a more consistent, precise criticality score. First, we randomly selected 100K from 900K GitHub projects to visualize the high-dimensional data using t-SNE. Then, we calculate each feature's correlation and display the results in a heat map. Furthermore, we model the data using Linear, Lasso, and ElasticNet regression to help with feature selection. The best regression had a test R^2 value of 0.71. Lastly, we use Recursive Feature Extraction, SelectFromModel, Sequential Forward Floating Selection, and Sequential Backward Floating Selection to determine the 5 most essential signals. The resulting regression from these 5 features has a test R^2 value of 0.70, indicating that only these features are necessary to calculate the criticality score. The criticality algorithm using these consistent features paves the way for better identifying important OSS projects. Future work will create a non-deterministic criticality score with machine learning algorithms or Graphs for Understanding Artifact Composition (GUAC).

Keywords— criticality score, open-source software, security prioritization, statistical analysis

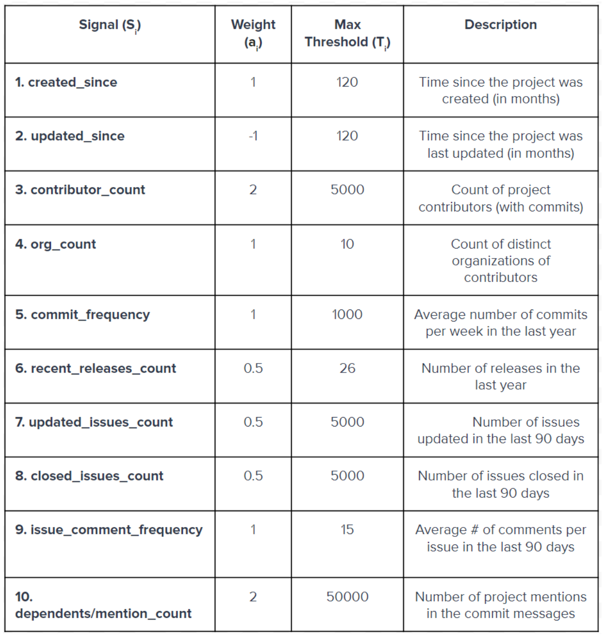
1 Introduction

Many companies have IT projects that rely on open-source software projects [1]. These projects are prone to software supply chain attacks. In 2020, SolarWinds, a major software company that provides system management tools for networks, was a victim of such an attack [2]. Consequently, information from the US Government (military, state, treasury) and major companies like Microsoft, Intel, and Cisco was left vulnerable [2].

Thus, it is vital that all projects, including open-source projects, are properly secured. A system has been created to prioritize the most "critical projects," which are projects with high centrality and transitive dependents. This system, invented by Robert Pyke, was the criticality score – a number ranging from 0 to 1 to identify the most important projects on GitHub [1]. The algorithm to compute the score follows:

$$C_{project} = \frac{1}{\sum_i a_i} \sum_i a_i \frac{\log(1 + S_i)}{\log(1 + \max(S_i, T_i))}$$

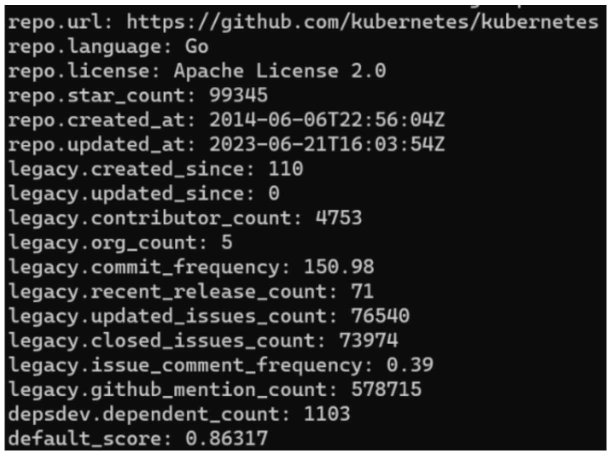
There are 10 parameters – called signals, S_i – used to calculate the score (which will be referenced according to their number later) shown in Table 1. Each signal's weight (a_i) is its relative importance to the calculation. The max threshold, (T_i), is the signal value where the feature is deemed to be "critical" and given a max weighting [1]. In other words, it is a cap/maximum value a signal can have. This prevents the issue of having one feature skew the criticality score if it is a disproportionately large value. However, in the article about the criticality score, little reasoning is provided for the choice of signals, weights, and max threshold values [1]. There is no evidence of data analysis helping to determine these values.



Signal (S_i)	Weight (a_i)	Max Threshold (T_i)	Description
1. created_since	1	120	Time since the project was created (in months)
2. updated_since	-1	120	Time since the project was last updated (in months)
3. contributor_count	2	5000	Count of project contributors (with commits)
4. org_count	1	10	Count of distinct organizations of contributors
5. commit_frequency	1	1000	Average number of commits per week in the last year
6. recent_releases_count	0.5	26	Number of releases in the last year
7. updated_issues_count	0.5	5000	Number of issues updated in the last 90 days
8. closed_issues_count	0.5	5000	Number of issues closed in the last 90 days
9. issue_comment_frequency	1	15	Average # of comments per issue in the last 90 days
10. dependents/mention_count	2	50000	Number of project mentions in the commit messages

Table 1: The 10 signals being used to calculate criticality score with each of the weights, max thresholds, and descriptions.

We demonstrate the current implementation of the criticality score in Figure 1. The program will automatically scrape the 10 signals from a GitHub repository and return the calculated criticality score. Its ease of use and speed make it a good organizer of open-source projects. Currently, OSSF (Open Source Security Foundation) uses the criticality score as one of the factors to create a list of the most essential open-source projects [3].



```
repo.url: https://github.com/kubernetes/kubernetes
repo.language: Go
repo.license: Apache License 2.0
repo.star_count: 99345
repo.created_at: 2014-06-06T22:56:04Z
repo.updated_at: 2023-06-21T16:03:54Z
legacy.created_since: 110
legacy.updated_since: 0
legacy.contributor_count: 4753
legacy.org_count: 5
legacy.commit_frequency: 150.98
legacy.recent_release_count: 71
legacy.updated_issues_count: 76540
legacy.closed_issues_count: 73974
legacy.issue_comment_frequency: 0.39
legacy.github_mention_count: 578715
depsdev.dependent_count: 1103
default_score: 0.86317
```

Figure 1: Example using the current criticality score implementation on the Kubernetes GitHub repository, which has the 5th highest score in the data set used in this study.

While the criticality score is useful to obtain a baseline idea of how often a package is used, it is far too reliant on detecting the activity of a project rather than its importance [4]. This is a major issue, as one of the driving factors behind the creation of the score was to identify old, vulnerable packages that are still used [4]. These are perfect candidates for attackers to breach and gain access to critical company systems. Without securing and updating these packages, open-source projects will remain at widespread risk [5].

There have been some attempts to solve this issue. In a GitHub issue post, Caleb Brown describes an alternative algorithm to calculate the criticality score. He describes several guidelines for improving the criticality score. Additionally, he details a risk-based approach to calculating the score [4]. The algorithm for his alternative risk-based uses a multiplicative method, which is given as the following:

$$score = 1 - (\prod_{i=1}^{n} 1 - x_i)^{\frac{1}{n}}$$

where:

$$x_i = (\frac{w_i}{W})\frac{f(\min(\max(s_i - l_i, 0), u_i - l_i))}{f(u_i - l_i)}$$

- s_i is the signal value
- l_i is the signal's lower bound
- u_i is the signal's upper bound
- w_i is the signal's weight
- W is the total weight of all the signals
- $f(x)$ is a function (in this case $log(1 + x)$)
- n is the total number of signals

Unfortunately, little reasoning is provided for the development of this algorithm in the article, and the experiments using this score are not publicly available.

Furthermore, in a different article, a different methodology is taken to determine the criticality of a project [6]. Pfeiffer leaves the criticality score altogether due to its reliance on activity and its inconsistent features, favoring PageRank and Truck factor to determine a project's importance [6].

While both this alternate criticality score and PageRank method are insightful, we will focus our study on the original criticality score, touching on the alternative algorithm later. We believe that the main reason why the criticality score measures activity over importance is the choice of specific signals. In the article detailing the criticality score creation, the reasoning behind weight/max threshold values was not provided [1]. We hypothesize that signal meaning changes depending on the project type. For example, a high commit_frequency could mean a project is active and crucial. However, a low commit_frequency could mean that an essential package is so widely used only careful and thoroughly reviewed commits are allowed. Additionally, we think that multiple signal variables are strongly correlated with each other, inflating the criticality score when it should only be measuring one of these signals. Closed_issues_count, updated_issues_count, and comment_frequency are candidates to have a strong relationship with each other based on how they are acquired. Furthermore, we reason that dependents_counts is too inconsistent – it queries GitHub API for names through commit messages, which inflates this signal's value for projects named with commonly used terms.

2 Methodology

Almost 900,000 OSS GitHub projects with their signal and criticality score data are openly stored on the criticality score web page from June $7th$, 2021 [1]. The data is stored in an Excel spreadsheet and sorted by criticality score. In this study, we use this data to examine the weaknesses of signals through statistical analysis and identify essential features to create a more consistent, precise criticality score.

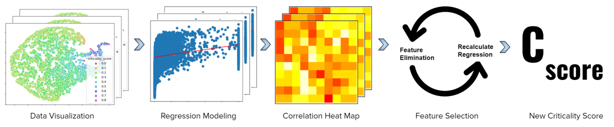


Figure 2: A brief overview of the 5 main steps executed in this study.

Using Python and packages such as Scikit-Learn, we implement t-distributed stochastic neighbor embedding (t-SNE) to reduce the 10-dimensional data into 2 dimensions for visualization [7]. Next, we model the relationship between signals and criticality score using multiple linear, Lasso, and ElasticNet regression [8]. We use these models to reduce potential overfitting through regularization. A seeded 70-30 training test split verified the performance of the regression. The R^2 value of each regression was the chosen measure of accuracy. These regressions would serve as the models for feature selection.

We then perform a correlational analysis to check for inter-dependencies among variables, and we visualize the results in a heat map [9]. This would aid in consolidating signals that are strongly correlated with each other. Spearman's correlation was used because of the monotonic relationship between feature and score.

After this, we select critical features using 4 types of feature selection: Recursive Feature Elimination

(RFE), SelectFromModel (SFM), Sequential Forward Floating Selection (SFFS), and Sequential Backward Floating Selection (SBFS) [10]. RFE works by recursively considering smaller and smaller sets of features, while SelectFromModel is a meta-transformer that is selecting features based on the features_importance_ attribute [10]. SFFS and SBFS are similar; SFFS is a greedy procedure that finds the best new feature to add to the set of selected features and SBFS works similarly but starts with all the features and removes the least vital feature [10]. Multiple methods confirmed that the most important signals we report were not specific to a certain feature selection technique. Using the holistic results from these methods and using our best judgment, we pick the most vital features to create the criticality score, thus eliminating any redundant features.

Lastly, we recreate the criticality score with these features weighted heavily. After calculating the percent change between the scores, we performed a visual test to verify the resulting project rankings. Furthermore, we rebuild the alternative criticality score algorithm mentioned earlier [4] and also test it using these critical features.

3 Results & Discussion

The t-SNE visualization of 10,000 randomly selected projects is shown in Fig. 3. Each criticality score was rounded for color classification, so, for example, any criticality score from 0.15 – 0.25 is rounded to 0.2 and colored orange. As expected, clusters with a higher criticality score appear at the end of the axes, indicating all-around higher signal values. However, there is a stronger correlation between higher scores and the x-axis over the y-axis. While the 2D representation has no direct meaning, it seems to indicate that certain features are more indicative of higher criticality scores than others.

We plot the linear regression with each signal for visual analysis, only showing two here for simplicity. As seen in Fig. 4, the regression line for contributor_count is effective at modeling the data, while the dependents_count regression has little modeling power. This indicates how dependents_count could

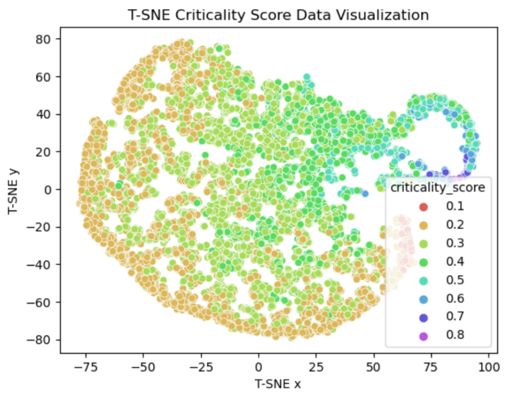


Figure 3: t-SNE visualization of 10,000 criticality score samples.

be a feature that provides little meaning to the criticality score calculation. Another signal whose regression line did not match the shape of the data was closed_issues_count, also showing its potential irrelevance. All other signals regression fit the data relatively well.

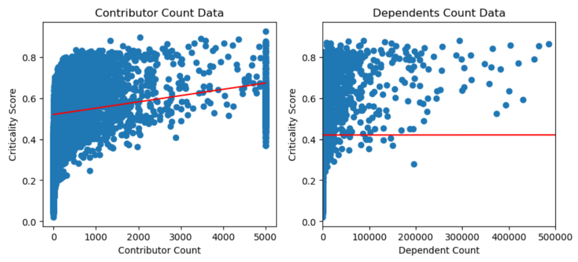


Figure 4: Contributor_count (left) and dependent/mention_count are plotted with their respective regression lines.

After performing the other regressions, we compare each method in Table 2. Both training and test performance was similar for all regressions, which validates the accuracy of our models. Lasso performed slightly worse on both the training and test data set (0.688 for testing), while linear and ElasticNet had a similar level of performance (both 0.715 for testing). This indicates that the linear regression model does not suffer from much overfitting.

We hypothesize that the R^2 peaked at 0.73 be-

Regression	Training R^2	Test R^2
Linear	0.734	0.715
Lasso	0.693	0.688
ElasticNet	0.731	0.715

Table 2: Each R^2 value from the regression during training and testing is shown.

cause once a value passed the max threshold, T_i, the increased value no longer led to a higher criticality score. Instead, it only misled the regression, keeping the accuracy capped. Several outliers, such as Linux having a dependents_count of over 50 million (when the max threshold for this feature is 50K), also lowered the modeling performance.

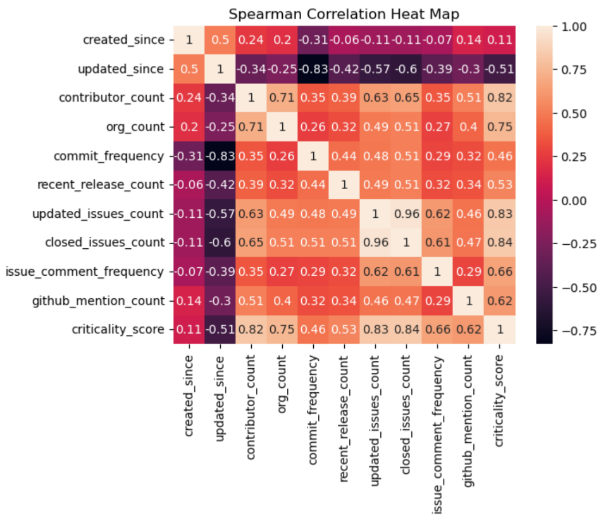


Figure 5: The Spearman correlation analysis displays how the variables relate to each other.

Our correlation analysis, visualized in the heat map in Fig. 5, displays the correlation between each signal. As we hypothesized, there is a significant correlation between certain variables, with 3 main pairs found:

- commit_frequency, updated_since (-0.83)
- contributor_count, org_count (0.71)
- updated_issues, closed_issues (0.96)

The strong relationships between these variable pairs indicate that they measure similar activity in a project and are skewing the criticality score. These parameters are logically connected: a higher commit frequency means a project has been updated recently since commits count as updates, more contributors increase the chances of these contributors coming from different organizations, and more updates on issues imply there are more issues that will be closed. Thus, having both pairs of these features double-counts these similar measurements in the calculation.

For our next step, we select the most pivotal signals for the criticality score algorithm. An example showing how regression R^2 increases more features using SFFS is shown in Figure 6. Around 5 - 7, adding an additional feature hardly impacts the criticality score. Thus, this was the optimal number of features determined for most models.

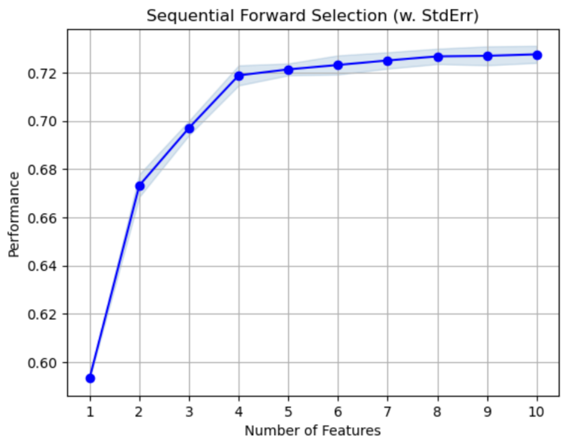


Figure 6: The Spearman correlation analysis displays how the variables relate to each other.

For RFE, SFFS, and SBFS, the models stop selecting after the best 5 features are found. The SFM model trains until it can no longer increase accuracy with an additional feature.

As can be seen from Table 3, the models all selected similar features that they regarded as essential, regardless of which regression algorithm was used. Therefore, signals 2,3,4,7, and 9 (updated_since, con-

Method	RFE	SFM	SFFS	SBFS
Linear	1,2,4,5,9	4,9	2,3,4,7,9	2,3,4,7,9
Lasso	2,3,4,7,9	2,4,7,9	2,3,4,7,9	2,3,4,7,9
Linear	1,2,4,7,9	2,4,9	2,3,4,7,9	1,2,4,7,9

Table 3: The 4 different feature selection results are shown here using all 3 regression models built earlier. The feature numbers correspond with the labeling in Table 1.

tributor_count, org_count, updated_issues_count, issue_comment_frequency) were deemed to be necessary to calculate an accurate criticality score. Using only these signals, an ElasticNet regression model achieved a 0.70 R2 from the test data, indicating that these features alone are almost as sufficient in predicting criticality score as all 10 features.

We recreate the criticality score using these features with a heavier weight. To avoid the loss of information, we lower the weight of two signals (recent_releases_count, created_since). The remaining signals (commit_frequency, closed_issue_count, and dependents/mention_count) are removed due to their redundancy and inconsistency.

We found an error during the recreation of the criticality score. The negative weight of updated_since was not functioning correctly and made it possible to obtain a negative criticality score, which is not allowed. Thus, we follow a recommendation from the alternative criticality score algorithm [10] and made the updated_since weight a positive value. To adjust for this, we replace the original feature value with:

$$\text{updated\_since} = T_i - \min(T_i, s_i)$$

Using this, a previously 'critical' value of 0 for updated_since will now have the new maximum value of T_i and the worst value of 120 will now be a value of 0. This keeps the same scoring criteria for updated_since but removes the inconsistencies associated with the negative weight. We suggest that any future criticality score algorithm follows this same process with updated_since.

The final criticality score has the following parameters and weights:

Signal (S_i)	Weight (a_i)
updated_since	2
contributor_count	2
org_count	2
updated_issues_count	2
issue_comment_frequency	2
created_since	0.5
recent_releases_count	0.5

Table 4: The list of final weights and features we use in the new criticality score, based on this study.

We analyze this new criticality score with essential features and found an average change of 30.9%. However, the top 500 criticality score projects had an average change of only 5.7%, indicating that the majority of significant projects were still deemed as important. For example, Linux, the top project in the old criticality score project, only dropped 5 places.

Precisely measuring the actual improvement of the criticality score is difficult. We use a comparison-based approach by analyzing the top-ranked projects of the score and other hand-compiled sources. The top projects given by this criticality score (such as Rust, TensorFlow, and Nodejs) are also the most important packages in the Harvard Census II Program [11] and the OSSF list of the top 150 most significant projects [1]. This validates the usefulness of this new criticality score with more consistent signals.

We implement the alternative criticality score algorithm mentioned earlier [4]. With no parameter values given, we use a simple lower bound of 0 for all the signals. The change between the original criticality score and this alternate algorithm is 1.9%, with the top 500 projects only changing 1.2%. Our recreation indicates a very similar list to the original criticality score algorithm, but the added flexibility of a lower bound can make this a versatile option to compute criticality. More testing between the two algorithms is necessary to determine which one is superior.

We also test the alternative criticality score using only the same essential parameters as above and found similar results. The overall average change was 46.7%, while the average change for the top 500 projects was 7.7%. Once again, the results are in line with what is expected.

4 Conclusion

The results of this statistical analysis verify the weaknesses of specific features and identify the most important features to create a more consistent, precise criticality score.

Thus, we improve the criticality score by:

1. Combining/removing certain highly correlated feature pairs commit_frequency and updated_since, updated_issues and closed_issues
2. Removing dependent_count until the query through GitHub's API can filter out using the actual project usage and not just project mentioning
3. Increasing the weights of the top 5 most critical features mentioned above (from 1 to 1.5 or 2), and decrease the other less critical feature weights (from 1 to 0.5)
4. Change the negative weight from updated_since to positive by using $T_i - \min(s_i, T_i)$ to avoid calculation issues

The final criticality score uses 5 heavily weighted signals and 2 less weighted signals as described before. Our study improves the reliability and consistency of the criticality score's ability to determine the importance of an open-source project, ensuring proper protective measures can be placed on the most pivotal works.

5 Future Work

Future work will focus on selecting other signals that are strong indicators of importance rather than activity in the criticality score. We believe a new signal that will accurately measure criticality will involve Graph for Understanding Artifact Composition (GUAC) [12]. GUAC is a database of projects and their dependencies that we will query GUAC to record the usage of a particular package. This is a great indicator of criticality because it measures how much a package is used by other projects, regardless of the activity trying to maintain the package. We will also perform a detailed analysis comparing the original criticality score algorithm and the alternative algorithm. Additionally, we will do more graphical analysis on each feature to determine statistically accurate max threshold (T_i) and potentially minimum threshold values. Furthermore, we will use the more reliable data from this new criticality score to train a machine learning model to predict a project's criticality score non-deterministically.

Acknowledgements

Thank you to Dr. Berkay for mentoring me throughout this research project. Special thanks to my sister, Brea, for reviewing my paper and code as well as encouraging and supporting me throughout the process.

References

[1] Ossf. Ossf/criticality_score: Gives criticality score for an open source project.

[2] Sean Michael Kerner Saheed Oladimeji. Solarwinds hack explained: Everything you need to know, Jun 2023.

[3] David A Wheeler and Jason N Dossett. Core infrastructure initiative (cii)nbsp; open source softwarenbsp; census ii strategy. *Institute for Defense Analysis*, Oct 2017.

[4] Caleb Brown. Doc: Criticality score and security risk, improving criticality score. · issue #102 · ossf/criticality_score, Feb 2022.

[5] David A Wheeler and Samir Khakimov. Open source software projectsnbsp; needing security investments. *Institute for Defense Analyses*, Jun 2015.

[6] Rolf-Helge Pfeiffer. Identifying critical projects via pagerank and truck factor. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 41–45, 2021.

[7] Luuk Derkson. Using t-sne in python to visualize high-dimensional data sets, Sep 2022.

[8] Simplilearn. Sklearn regression models: Methods and categories: Sklearn tutorial, Feb 2023.

[9] Andy McDonald. Seaborn heatmap for visualising data correlations, Aug 2022.

[10] Jason Brownlee. Feature selection for machine learning in python, Aug 2020.

[11] Jessica Wilkerson and James Dana. Vulnerabilitiesnbsp; in the core: Preliminary report and census iinbsp; of open source software. *The Linux Foundation*, 2021.

[12] GUAC Team. Guac.