Identifying Critical Projects via PageRank and Truck Factor

Rolf-Helge Pfeiffer
IT University of Copenhagen
Copenhagen, Denmark
ropf@itu.dk

***Abstract*—Recently, Google's Open Source team presented the *criticality score* [1] a metric to assess *"influence and importance"*[1] of a project in an ecosystem from project specific *signals*, e.g., number of dependents, commit frequency, etc. The community showed mixed reactions towards the score doubting if it can accurately identify critical projects. We share the community's doubts and we hypothesize, that a combination of PageRank (PR) and Truck Factor (TF) can more accurately identify critical projects than Google's current Criticality Score (CS). To verify our hypothesis, we conduct an experiment in which we compute the PR of thousands of projects from various ecosystems, such as, Maven (Java), NPM (JavaScript), PyPI (Python), etc., we compute the TFs of the projects with the highest PR in the respective ecosystems, and we compare these to the scores provided by the Google project. Unlike Google's CS, our approach identifies projects, such as, six and idna from PyPI, com.typesafe:config from Maven, or tap from NPM, as critical projects with high degree of transitive dependents (highest PR) and low amount of core developers (each of them possessing a TF of one).**

I. Introduction

Our societies depend heavily on a multitude of open source projects. For example, in 2014 two-thirds of all webservers were running OpenSSL (today likely more). Back then, the OpenSSL project was run by few volunteers with only one full-time developer at a low income [2]. General public became first aware of the wide-spread dependency on OpenSSL, when the Heartbleed bug[2] was discovered. The combination of many direct or indirect (transitive) dependents in combination with being maintained by only few persons rendered OpenSSL a *critical* project. Here, OpenSSL is just one example of a *critical* software project.

To identify *critical* projects and to provide *"ways to connect critical [...] projects we all rely on, with organizations that can provide them with adequate support"*, recently, the Open Source team at Google[3] in collaboration with the Open Source Security Foundation[4] announced the Criticality Score project[5].

The official project announcement defines *criticality* as the *"influence and importance of a project"* and illustrates the role of critical projects with their many transitive dependents via Figure 1. For the CS project, Rob Pike defines: *"A package with higher criticality is one that is more important within its packaging system (NPM, RubyGems etc.) and therefore may deserve higher scrutiny and attention."* [1]. He also provides a generic formula to compute a CS of a package as a number between zero and one (higher values indicate higher criticality) based on *"signals"*, e.g., number of package downloads or number of dependents. However, in its current implementation[19], CSs are computed only for projects hosted on Github –instead of packages as defined by Pike– relying on ten signals, e.g., estimated number of dependents, commit frequency, etc.

Community feedback to the proposed CS was mixed[6]. It was criticized that the score ranks projects of low criticality high (and vice versa) and that it favors popularity over criticality.

Our goal is to contribute to improving the current CS by actually identifying critical projects, which we consider – just as the community and as suggested by the CS project announcement– as projects with high centrality (high amount of transitive dependents) and low amount of developers as illustrated in Figure 1. Inspired by community discussions[7], the call for community feedback[1], the openness of the scoring formula to other signals, and by the fact that the project comes from Google –whose founders invented the PR algorithm [3]–, we hypothesize that *critical* projects of an ecosystem can be more accurately identified via PR and TF [4], [5] than with the current CS. We believe so, since PR weighs "importance" of a node in a network based on transitive link structures [3], [6], [7] and TF describes the amount of developers that can be removed from a project before it is endangered [4].

In this paper we describe an experiment (section III) in which we: *a)* rank all packages of the dependency graphs of the package managers Cargo, NPM, Maven, Packagist, and PyPI with the PR algorithm, *b)* report PRs and TFs for the highest ranked projects, *c)* identify critical projects from multiple ecosystems, and *d)* compare these to their scores of Google's current CS, see section IV. We conclude, that the projects that our approach identifies are really critical in the

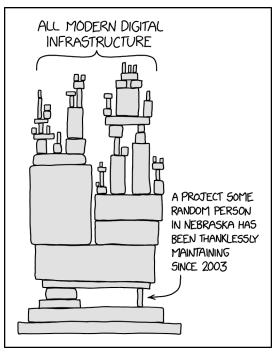

Figure 1. Critical projects popularly illustrated (from https://xkcd.com/2347/).

sense of being central to the respective ecosystem and lacking resources in terms of developers (projects that are worth to be supported). We discuss shortcomings of our approach in section VI.

Combining PR and TF, we identify, e.g., the projects Six and IDNA (PyPI packages six and idna), Config (Maven package com.typesafe:config), or Node-Tap (NPM package tap) as critical projects. Each of these has a low amount of core developers ($TF = 1$) and ranks amongst the top PRs of the respective ecosystem. Since none of these projects is identified as critical by Google's current CS implementation, we suggest to include PR and TF as signals with high weight in the CS formula. Note, a replication package including the experiment setup and more results than can be presented in this short paper is available online[8].

II. Background

Pike [1] describes a generic formula to calculate the CS of a package (C_{pkg}) as a normalized weighed sum (α_i) of the ratio of the logarithm of signals (S_i), the logarithm of the maximum of signal value and corresponding thresholds (T_i):

$$C_{pkg} = \frac{1}{\sum_i \alpha_i} \sum_i \alpha_i \frac{\log(1 + S_i)}{\log(1 + \max(S_i, T_i))} \quad (1)$$

In his paper, Pike mentions the number of package downloads or the number of its dependents as possible signals. He does not provide other signals, weights or thresholds. However, the current implementation of the CS[19] relies on ten signals, e.g., time since last update, average number of comments per issue, commit and release frequency, number of dependents mentioned in commit messages etc., with corresponding weights and thresholds[9]. For example, *commit frequency* is currently implemented as the average number of commits per week over the last 52 weeks and the *dependents count* is implemented as a query to the Github API over all commits of any repository mentioning the name of the project for which the CS is computed[10]. The latter leads to false results, e.g., for projects with common names, such as tasks[11]. The project does not specify how given weights and thresholds are created.

Note, while Pike describes CS for packages in ecosystems available via package managers the current implementation computes it for projects hosted on Github, i.e., the terms *package* and *project* are used interchangeably by Google's CS project. In this paper we call a Git repository a *project*, and an entity that is registered on a package manager is called a *package*. Multiple packages may belong to a single project, e.g., org.scala-lang:scala-library, org.scala-lang:scala-reflect or org.scala-lang:scala-compiler all belong to the Scala project[12].

Based on Github's popularity API, the CS project lists Criticality Scores for the top 200 projects in the languages C, C++, C#, Go, Java, JavaScript, PHP, Python, Ruby, Rust, and Shell[13]. The computed Criticality Scores are biased towards popularity since they are based on activity signals, such as, average number of comments per issue and based on Github's popularity API.

III. Method

We download the *libraries.io* dataset [8][14], which contains package metadata including project URL and dependency links, from 15 ecosystems, such as, Cargo, NPM, Maven, PyPI, etc., see experiment configuration[15]. More package managers are present in the dataset but only for 15 of these it contains dependency links.

After preprocessing the *libraries.io* dataset for import into a Neo4j graph database [9] (we represent each package as a node and each versioned dependency link as a separate relation) we compute the PR for all packages of the 15 ecosystems with the PR algorithm from the Neo4j Graph Data Science plugin[16] (with default values for maximum iterations 20 and damping factor 0.85). Per package manager dependency graph, we sort the packages by decreasing PR and export the 1,000 (configurable value) highest ranked packages[17].

For comparability, we compute TFs for the projects corresponding to the 100 (configurable value) packages with the highest PR from the five package managers Cargo, Maven, NPM, Packagist, and PyPI (these are present in both, the results of Google's CS project[13] and the *libraries.io* dataset). TFs are computed using the CLI tool `truckfactor`[18], which was originally developed for teaching concepts of mining software repositories. It computes a TF for a given Git repository by calculating *knowledge ownership* [10] per file. *Knowledge ownership* of a file is assigned to the developer that added most lines over the history of a file. Similar to Avelino et al. [11], TF is set to the amount of developers possessing knowledge ownership over most of the files and covering at least 50% of all files of a project. Finally, we calculate a CS with all signals of the current score for each project for which we calculate a TF with the `criticality_score` tool[19].

In the remainder we call the resulting dataset the *results*. The results combine PRs, TFs, and CSs, they are available online[20], and they are the basis for the next section.

IV. Results

The ecosystems are of different size. Ordered by number of packages, sizes are: NPM (1,275,082), Packagist (313,278), PyPI (231,690), Maven (184,871), Cargo (35,635). Not all packages are actually connected to the respective dependency graph. The amount of disconnected packages, i.e., packages with no requirements and no dependents (in- and outdegree = 0) is different across the ecosystems. In increasing order, ca. 18% of NPM (224,718), 22% of Cargo (7,833), 40% of Maven (74,647), 43% of Packagist (134,352) and ca. 79% of PyPI (182,498) packages are disconnected.

An excerpt of the results[20] is listed in Table I. It shows the 25 most central packages (ranked by PR) from the respective ecosystems. PR values indicate centrality via transitive link structures representing importance. The varying magnitudes of PR values across ecosystems indirectly indicate the size of an ecosystem. For example, NPM with most packages and most links scores higher than Cargo with least packages and links.

Table I
PACKAGES WITH HIGHEST PR BY ECOSYSTEM.

Platform	Name	PR	TF	CS
Cargo	rand	627.58	2	0.64
	serde	603.13	1	0.66
	serde_derive	536.54	1	0.66
	winapi	509.88	1	0.52
	libc	452.96	2	0.72
Maven	junit:junit	2,972.38	3	0.59
	org.hamcrest:hamcrest-core	2,487.32	2	0.47
	org.hamcrest:hamcrest	2,114.50	2	0.47
	org.scala-lang:scala-library	1,654.49	6	0.84
	com.typesafe:config	1,428.11	1	0.55
NPM	tap	24,944.12	1	0.58
	mocha	24,488.46	3	0.76
	eslint	10,987.09	5	0.86
	tape	9,946.41	2	0.65
	mkdirp	7,084.05	1	0.29
Packagist	phpunit/phpunit	16,099.03	1	0.80
	symfony/phpunit-bridge	1,659.07	3	0.62
	phpunit/php-text-template	1,358.10	1	0.37
	sebastian/exporter	1,299.93	1	0.43
	symfony/yaml	1,232.69	1	0.61
PyPI	requests	664.90	1	0.72
	six	582.92	1	0.57
	numpy	288.01	6	0.86
	setuptools	237.72	4	0.72
	soupsieve	174.72	1	0.44

Table I also illustrates, that the kind of most central packages is different per ecosystem. For Maven, NPM, and Packagist multiple testing packages (junit:junit, org.hamcrest:hamcrest-core, phpunit/phpunit, tap, mocha, and tape) rank highest whereas in Cargo libraries for random number generation and data serialization (rand and serde which in other languages would likely be part of a standard library) rank highest, and in PyPI tools for HTTP (requests), language version compatibility (six), and scientific computing (numpy) rank highest. In all ecosystems, utility packages for data serialization or configuration (e.g., serde, sebastian/exporter, or com.typesafe:config) are most central.

Table II, lists the amount of direct ($\circ \to \circ$) and (in)direct dependents ($\circ - [1..2] \to \circ$, $\circ - [1..3] \to \circ$) of the respective package. The column headers for the two last columns shall illustrate that the number of dependents up to depth two/three are given. Values are provided in absolute and in relative numbers, where relative numbers are with respect to the total amount of packages in the ecosystem. Tiny relative numbers in parenthesis are with respect to only those packages that are connected in the dependency graph (in- or outdegree > 0). The property of PR to rank packages based on transitive link structures instead of direct number of dependents is well illustrated by that only 0.5% (1,038) of all Maven packages depend directly on org.hamcrest:hamcrest-core. However, in total 18% (32,596) of all Maven packages depend either directly or indirectly via another package on it. Via up to two other packages over a third of the Maven ecosystem depends on it. Also, the serde package is so central to the Cargo ecosystem that almost two thirds of all packages depend via up to two other packages on it. Some values for NPM are not available in Table II, since computation of transitive dependencies with Neo4j is memory intensive (experiment DBMS is configured with 4GB heap memory) and querying NPM's 1,275,082 packages with 154,387,701 dependency links exceeds the provided memory.

Table II
ABSOLUTE AND RELATIVE NUMBER OF DIRECT AND INDIRECT DEPENDENTS OF PACKAGES (DEPTH TWO AND THREE) PER ECOSYSTEM.

Name	$\circ \to \circ$		$\circ - [1..2] \to \circ$		$\circ - [1..3] \to \circ$	
rand	3,398	10% (12%)	13,683	38% (49%)	22,646	64% (81%)
serde	6,427	18% (23%)	16,634	47% (59%)	23,221	65% (84%)
junit:junit	27,192	15% (25%)	61,126	33% (55%)	73,295	40% (66%)
org.hamcrest: hamcrest-core	1,038	0.5% (0.9%)	32,596	18% (30%)	63,041	34% (57%)
tap	8,725	0.7% (0.8%)	402,193	32% (38%)	n/a	
mocha	207,094	16% (20%)	n/a		n/a	
phpunit/phpunit	77,341	25% (43%)	142,230	45% (79%)	157,398	50% (88%)
symfony/phpunit-bridge	2,262	0.7% (1%)	43,516	14% (24%)	132,043	42% (74%)
requests	10,644	5% (22%)	14,615	6% (30%)	15,700	7% (32%)
six	5,014	2% (10%)	16,846	7% (34%)	22,628	10% (46%)

More than half of the 25 most central packages (Table I) possess a $TF = 1$, i.e., a single person stands for the majority of development in projects like serde, tap, or phpunit/phpunit. Highly central packages with $TF = 1$ are *critical* since large parts of the ecosystem depend on the work of a single core developer. For example $> \frac{2}{3}$ of Cargo packages depend on serde, $> \frac{2}{5}$ of NPM packages depend on tap, $> \frac{1}{2}$ of Packagist packages depend on phpunit/phpunit, see Table II. Only org.scala-lang:scala-library and numpy from Table I spread knowledge widely over six core developers ($TF = 6$). TFs are also different across ecosystems. Projects in Cargo, Packagist, or PyPI have more often a $TF = 1$ whereas knowledge in Maven or NPM packages is usually spread over multiple developers ($TF_{avg}(Cargo) = 1.74$, $TF_{avg}(Maven) = 2.6$, $TF_{avg}(NPM) = 2.75$, $TF_{avg}(Packagist) = 1.3$, $TF_{avg}(Packagist) = 1.8$)[20].

In our results[20], CS values are centered around moderate criticality ($avg \approx 0.58$, $std \approx 0.14$, $min \approx 0.27$, $q_1 \approx 0.48$, $q_2 \approx 0.57$, $q_3 \approx 0.68$, $max \approx 0.93$). However, the collection of TypeScript types (NPM project @types/node) has the highest CS (≈ 0.93 on rank 20) and sports simultaneously the highest $TF = 22$. The Cargo serialization framework rustc-serialize preceding serde has the lowest CS (≈ 0.27 on rank 16) and a $TF = 1$. These two projects illustrate the community critique of CS favoring popularity. By CS, @types/node is highly critical, on Github it is certainly popular (31.7k stars and 23.6k forks) but its TF suggests that it is of low criticality since knowledge is spread over 22 developers. Contrary, rustc-serialize is of low criticality according to CS, it is of low popularity on Github (174 stars on Github and 102 forks) but it is developed mainly by a single person. Other interesting examples of likely critical projects with moderate CS can be found in the results. For example, idna (rank 14 in PyPI, $TF = 1$, $CS \approx 0.48$), is so important that its inclusion into the standard library is discussed[21] or pytz (rank 10 in PyPI, $TF = 1$, $CS \approx 0.5$), is so important that it triggered enhanced standard library functionality[22].

To check whether the signals PR or TF are already encoded in CS, we compute Spearman's ρ to test for correlations between PR and CS and between TF and CS of the results[20]

respectively. PR and CS are not correlated ($\rho \approx 0.0346, n = 92, p \approx 0.7436$) but we find a moderate positive monotonic correlation between TF and CS $\rho \approx 0.5416, n = 92, p < 0.05$. The latter correlation is likely due to TF and CS being both based on the number of contributors.

V. Related Work & Discussion

Besides for Google's search engine, the PR algorithm was previously applied to assess importance of files in code repositories hosted on Maven [12], to identify influential projects [13] and developers [13], [14] on Github, to identify important academic papers [15], to study architectural smells within software [16], etc. Therefore, we consider PR a suitable centrality index for this kind of work, even though other centrality indices, such as, subtree centrality [17], or a combination of such [18] may be applied instead.

The authors of CS state[23] that the score does not use PR as a signal yet, since package dependency information is not readily available in the C/C++ realm. However, many C++ packages are in the NuGet dependency graph that is distributed with this experiment's setup. Alternatively, JFrog's Conan[24] or Microsoft's Vcpkg[25] are recent C/C++ package managers, from which dependencies can be extracted to extend[26] the *libraries.io* dataset in future.

There is no generally agreed upon algorithm or tool to compute TFs [19]. Previous work [20], [21] computes TFs for SVN repositories. We cannot reuse these directly since the majority of projects in the *libraries.io* dataset are stored in Git repositories. We apply our `truckfactor`[18] tool instead of other available tools[27,28] [11], [22], since it is readily installable from PyPI and thereby directly integrable into our setup.

Decan et al. [23] study the topology of CRAN, NPM, and PyPI and confirm that PyPI comprises the highest number of disconnected packages compared to other package managers, see Table II. They call ecosystems with many transitive dependencies on few packages *"fragile"* [24]. Identification of *criticality* of projects can be considered the inverse of identification of *fragility* of an ecosystem.

We believe that the results in section IV, e.g., idna and pytz with high PRs, low TFs but mediocre criticality according to CS illustrate, that a combination of PR and TF can more accurately identify *critical* projects than the current CS signals. Consequently, we envision that PR can be included directly into the CS formula and TF via its inverse, i.e., lower TF with higher PR indicates higher criticality. Determination of respective weights and thresholds for PR and TF remains future work since according to Ricca et al. [21] more work is needed to identify valid thresholds for TF and since the CS project currently does not describe how weights and thresholds are established.

Threats to validity: TF weighs initial contribution higher than maintenance contributions, which are typically smaller in size. For example, requests was initially developed mainly by K. Reitz but is now maintained by a group of developers. Still, due to the size of his initial contributions, TF algorithms (ours[18] as well as Avelino et al.[27] [11]) assign knowledge ownership of the majority of files to Reitz ($TF = 1$) overshadowing the many smaller contributions by the many active maintainers. It remains future work to better adapt TF to projects in maintenance.

The quality of our results depend on the accuracy of dependency link information in the *libraries.io* dataset, which we trust to be accurate.

To identify critical projects, we compute PR globally per ecosystem, which might underrate projects that are only critical in certain domains. Our experiment setup can be modified[29] to compute *personalized pagerank* [3], a PR biased towards certain packages.

Since the PR algorithm iteratively consumes in- and out-degrees of nodes, our choice of representing dependencies between different package versions as separate relations impacts the calculated PR values. Potentially biased PR values (towards release activity) could be mitigated by adjusting the weight of the respective current CS signal.

We do not map Git user names to logical users before TF computation as, e.g., Avelino et al. [11]. We observe Git user name changes to be not directly automatically identifiable as in [11] since their edit distance is usually longer than one. We plan to extend `truckfactor` to be parameterizable with rename mappings.

Limitations: In this work, we study only intra-ecosystem dependencies. For example, all of PyPI packages depend on a Python interpreter, which in turn depends on a C compiler and other C libraries, e.g., OpenSSL, BZip2, SQLite. Our current experiment setup cannot identify, e.g., OpenSSL as a critical project for various ecosystems since the *libraries.io* dataset does not include dependencies crossing technologies.

We present only a tiny subset of the top ranked projects from five ecosystems due to constrained space. Our replication package[8] contains more results and we plan to continuously expand the results there.

VI. Conclusions

In this paper, we demonstrate that critical projects, i.e., projects that are central in an ecosystem and that are developed by few persons, can be identified by PR and TF in combination. Unlike the current CS, our approach can identify projects, such as, serde (Cargo), com.typesafe:config (Maven), tap (NPM), phpunit (PHP), or six or idna from PyPI as critical. Each of these are amongst the most central in their respective ecosystems, with a considerable amount ($> \frac{1}{3}$) of packages in the dependency graphs depending directly or indirectly on them and each of these packages is developed mainly by a single person often in her spare time[30]. Consequently, we suggest to incorporate PR and TF as signals into the CS formula with higher weights than the current signals.

With the dataset resulting from our experiment, we provide the MSR community an alternative for selecting repositories for research projects. Instead of selecting popular repositories from Github, researchers can easily select central or critical projects for future research from our dataset[20].

References

[1] R. Pike, "Quantifying criticality," Open Source Security Foundation (OpenSSF), Tech. Rep., 2020. [Online]. Available: https://github.com/ossf/criticality_score/blob/a02c8311fbbbd5d569ebaad3106ec08532b3a10c/Quantifying_criticality_algorithm.pdf

[2] N. Eghbal, "Roads and bridges," *The Unseen labor behind our digital infrastructure*, 2016.

[3] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[4] L. Williams and R. R. Kessler, *Pair programming illuminated*. Addison-Wesley Professional, 2003.

[5] N. Zazworka, K. Stapel, E. Knauss, F. Shull, V. R. Basili, and K. Schneider, "Are developers complying with the process: an xp study," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2010, pp. 1–10.

[6] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," 1998.

[7] D. F. Gleich, "Pagerank beyond the web," *Siam Review*, vol. 57, no. 3, pp. 321–363, 2015.

[8] J. Katz, "Libraries.io Open Source Repository and Dependency Metadata," Jan. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3626071

[9] J. J. Miller, "Graph database applications and concepts with neo4j," in *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, vol. 2324, no. 36, 2013.

[10] A. Tornhill, *Your code as a crime scene: use forensic techniques to arrest defects, bottlenecks, and bad design in your programs*. Pragmatic Bookshelf, 2015.

[11] G. Avelino, L. Passos, A. Hora, and M. T. Valente, "A novel approach for estimating truck factors," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, 2016, pp. 1–10.

[12] S. Raemaekers, A. Van Deursen, and J. Visser, "The maven repository dataset of metrics, changes, and dependencies," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 221–224.

[13] F. Thung, T. F. Bissyande, D. Lo, and L. Jiang, "Network structure of social coding in github," in *2013 17th European conference on software maintenance and reengineering*. IEEE, 2013, pp. 323–326.

[14] Y. Yu, G. Yin, H. Wang, and T. Wang, "Exploring the patterns of social behavior in github," in *Proceedings of the 1st international workshop on crowd-based software development methods and technologies*, 2014, pp. 31–36.

[15] P. Chen, H. Xie, S. Maslov, and S. Redner, "Finding scientific gems with google's pagerank algorithm," *Journal of Informetrics*, vol. 1, no. 1, pp. 8–15, 2007.

[16] F. A. Fontana, I. Pigazzini, C. Raibulet, S. Basciano, and R. Roveda, "Pagerank and criticality of architectural smells," in *Proceedings of the 13th European Conference on Software Architecture-Volume 2*, 2019, pp. 197–204.

[17] A. Spitz and E.-Á. Horvát, "Measuring long-term impact based on network centrality: Unraveling cinematic citations," *PloS one*, vol. 9, no. 10, p. e108857, 2014.

[18] I. Zakhlebin and E.-Á. Horvát, "Network signatures of success: Emulating expert and crowd assessment in science, art, and technology," in *International Conference on Complex Networks and their Applications*. Springer, 2017, pp. 437–449.

[19] M. Ferreira, T. Mombach, M. T. Valente, and K. Ferreira, "Algorithms for estimating truck factors: a comparative study," *Software Quality Journal*, vol. 27, no. 4, pp. 1583–1617, 2019.

[20] F. Ricca and A. Marchetto, "Are heroes common in floss projects?" in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2010, pp. 1–4.

[21] F. Ricca, A. Marchetto, and M. Torchiano, "On the difficulty of computing the truck factor," in *International Conference on Product Focused Software Process Improvement*. Springer, 2011, pp. 337–351.

[22] V. Cosentino, J. L. C. Izquierdo, and J. Cabot, "Assessing the bus factor of git repositories," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 499–503.

[23] A. Decan, T. Mens, and M. Claes, "On the topology of package dependency networks: A comparison of three programming language ecosystems," in *Proccedings of the 10th European Conference on Software Architecture Workshops*, 2016, pp. 1–4.

[24] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, no. 1, pp. 381–416, 2019.

Referenced URLs

[1] https://github.com/ossf/criticality_score/blob/4a3e3e171ac403344060dbcfc43f47944afe77d0/README.md

[2] https://nvd.nist.gov/vuln/detail/CVE-2014-0160

[3] https://opensource.googleblog.com/2020/12/finding-critical-open-source-projects.html

[4] https://openssf.org/

[5] https://github.com/ossf/criticality_score

[6] https://news.ycombinator.com/item?id=25381397

[7] on the project's issue tracker https://github.com/ossf/criticality_score/issues/8, https://github.com/ossf/criticality_score/issues/31 and on HackerNews https://news.ycombinator.com/item?id=25382712

[8] https://doi.org/10.5281/zenodo.4621650 or https://github.com/HelgeCPH/critical-projects

[9] Due to constrained space, we refer to a complete list all ten signals online: https://github.com/ossf/criticality_score/blob/4a3e3e171ac403344060dbcfc43f47944afe77d0/README.md

[10] https://github.com/ossf/criticality_score/blob/a02c8311fbbbd5d569ebaad3106ec08532b3a10c/criticality_score/run.py#L142

[11] https://github.com/ossf/criticality_score/issues/55

[12] https://github.com/scala/scala

[13] https://storage.cloud.google.com/ossf-criticality-score

[14] https://libraries.io/data

[15] https://github.com/HelgeCPH/critical-projects/blob/master/analysis_conf.yml

[16] https://neo4j.com/docs/graph-data-science/1.4/algorithms/page-rank/

[17] https://github.com/HelgeCPH/critical-projects/tree/master/data/output

[18] https://github.com/HelgeCPH/truckfactor

[19] https://github.com/ossf/criticality_score/releases/tag/v1.0.7

[20] For humans: https://github.com/HelgeCPH/critical-projects/tree/master/data/output/comparison.adoc or for machines: https://github.com/HelgeCPH/critical-projects/tree/master/data/output/comparison.csv

[21] https://github.com/kjd/idna/issues/67

[22] https://github.com/stub42/pytz/issues/48

[23] https://github.com/ossf/criticality_score/issues/53

[24] https://conan.io/

[25] https://github.com/Microsoft/vcpkg

[26] https://github.com/librariesio/libraries.io/blob/master/docs/add-a-package-manager.md

[27] https://github.com/aserg-ufmg/Truck-Factor

[28] https://github.com/SOM-Research/busfactor

[29] https://neo4j.com/docs/graph-data-science/current/algorithms/page-rank/#algorithms-page-rank-examples-personalised

[30] https://github.com/serde-rs/serde/issues/1025