# A Comparison of Three Algorithms for Computing Truck Factors

Mívian Ferreira, Marco Tulio Valente
Department of Computer Science
Federal University of Minas Gerais
Belo Horizonte, Brazil
{mivian.ferreira, mtov}@dcc.ufmg.br

Kecia Ferreira
Department of Computing
Federal Center of Technological Education
Belo Horizonte, Brazil
kecia@decom.cefetmg.br

***Abstract*—Truck Factor (also known as Bus Factor or Lottery Number) is the minimal number of developers that have to be hit by a truck (or leave) before a project is incapacitated. Therefore, it is a measure that reveals the concentration of knowledge and the key developers in a project. Due to the importance of this information to project managers, algorithms were proposed to automatically compute Truck Factors, using maintenance activity data extracted from version control systems. However, to the best of our knowledge, we still lack studies that compare the accuracy of the results produced by such algorithms. Therefore, in this paper, we evaluate and compare the results of three Truck Factor algorithms. To this end, we empirically determine the truck factors of 35 open-source systems by consulting their developers. Our results show that two algorithms are very accurate, especially when the systems have a small Truck Factor. We also evaluate the impact of different thresholds and configurations in algorithm results.**

***Index Terms*—Truck Factor; Code ownership; Developer turnover; Mining software repositories.**

## I. Introduction

Software development is a knowledge-intensive industry, which makes developers the most important asset of software organizations. Moreover, after 25 years of modern Internet technologies, software continues to "eat the world" and we are everyday observing the rise of companies totally centered on software and, by consequence, on their developers.

In this context, developer turnover is a serious risk to software projects [1], [2]. To mitigate this risk, project managers should measure and monitor the concentration of knowledge in specific team members. An interesting indicator of this concentration is a measure known as Truck Factor (TF), which is defined as the minimal number of developers that have to be hit by a truck (or leave the team) in order to put the project in trouble [2]–[6]. The measure is also known as Bus Factor or Lottery Number. Essentially, a low Truck Factor means that the project's knowledge is concentrated in few team members and therefore the project faces a serious risk of discontinuation in case these developers leave. On the other hand, a high Truck Factor means that every developer is contributing to the project in similar terms. High Truck Factors may be the result of collective code-ownership practices, commonly advocated by agile software-development methodologies [2], [7].

In the case of open-source software, Truck Factor has additional relevance, since these projects are commonly maintained by volunteer developers. These developers do not have legal contracts with the project and usually do not have financial benefits, which makes turnover risks higher than in commercial software development. An example of a "Truck Factor episode" faced by an open-source project is illustrated by the following mail, recently posted to the FindBugs mailing list (Findbugs is a popular open-source bug finding tool for Java systems):[1]

> *I'm really sorry to say, but FindBugs project in its current form is dead. ... It looks like the project leader is not interested in the project anymore, and we can't reach him. ... We requested his help for the project many times (via direct mails, postings to the list and to the GitHub issues) but haven't received any sign of life from him since a year.*

Due to the relevance of Truck Factor data to project managers and users of open-source software, algorithms have been proposed to automatically estimate Truck Factors by using code-ownership data retrieved from version control repositories [8]–[10]. However, to the best of our knowledge, we still lack studies that (a) validate the results produced by such algorithms; (b) compare these results and discuss the main benefits and limitations of each algorithm. Therefore, in this paper we assess three recent algorithms proposed to estimate Truck Factors: (a) an algorithm proposed by Avelino et al. [8], which is based on a degree-of-authorship metric and on a greedy heuristic to infer the key developers in a project; (b) an algorithm proposed by Rigby et al. [9], as part of a study to assess the impact of turnover in software projects and that uses a git-blame-based heuristic to infer code ownership; and (c) an algorithm proposed by Cosentino et al. [10], which computes Truck Factors for files and then scales the results to the level of branches and projects. To validate and compare the algorithms, we build an oracle with the Truck Factor of 35 open-source systems, as indicated by the principal developers of these systems. We use open-source software in the study due to the importance of Truck Factor information in this context of software development. Moreover, we survey the main developers of these projects

[1] https://mailman.cs.umd.edu/pipermail/findbugs-discuss/2016-November/004321.html

to construct the oracle because they are probably the only authority capable to provide this information.

The study aims to answer the following research questions:

1) *How accurate are the results provided by each algorithm?* To answer this question, we define accuracy as a measurement of how close an estimated Truck Factor (by an algorithm) is to the true value (reported in the constructed oracle).
2) *How accurate is the identification of TF developers by each algorithm?* This research question targets the cases where an algorithm correctly estimates the Truck Factor, but not the list of key authors. For example, suppose a system with TF= 2 and that $\{Mary, John\}$ are the developers responsible for this result. An algorithm can correctly estimate the system's TF, but by considering $\{Lisa, Mark\}$ as the key developers. Therefore, this second research question investigates how often this situation happens with the studied algorithms.
3) *What is the impact of different thresholds and configurations in the results of each algorithm?* The studied algorithms depend on specific thresholds and configurations to produce their results. In the answers of the previous questions, we consider the default thresholds and configurations, as suggested by the algorithms' authors. Therefore, in this third research question, we explore the impact of different thresholds on the algorithms' results.

The remainder of this paper is structured as follows. Section II presents the main algorithms for estimating truck factor covered in the literature. Section III describes the study design. We present the results in Section IV and discuss them in Section V. Threats to validity are discussed in Section VI. Section VII presents related work. Finally, we conclude the paper in Section VIII.

## II. Algorithms for Estimating Truck Factors

This section describes the three algorithms for estimating truck factors compared in this paper (Section II-A to II-C). We also describe a fourth algorithm reported in the literature and justify why it is not included in the study (Section II-D). To help on the identification of the algorithms, we name them using three letters of the last name of their first author.

### A. AVL Algorithm

Proposed by Avelino et al. [8], the AVL algorithm relies on the Degree-of-Authorship (DOA) measure to define the *authors*—i.e., the key developers—of each file in a system [11], [12]. DOA values are computed from commit histories as follows: the creation of a file $f$ by a developer $d$ initializes the value of $DOA(d, f)$; further commits on $f$ by $d$ increase $DOA(d, f)$; finally, commits by other developers decrease $DOA(d, f)$. The weights used to increase/decrease the DOA values are defined after empirical experiments performed somewhere [12]. As the last step, DOA values are normalized per file; the developer with the highest DOA in a file $f$ has its normalized DOA equal to 1. A developer is considered an author of a file if its normalized DOA is greater than 0.75. This threshold is defined after an empirical experiment when AVL's authors manually validated DOA results produced for a sample of 120 source code files from six open-source systems [8].

AVL algorithm is defined in Listing 1. It receives as input the list of top-authors of a system $S$, i.e., the list of authors in $S$ ordered by the number of files they have authorship on. This list should be previously computed using the DOA measure, as described in the previous paragraph. AVL algorithm continually simulates the removal of the top-author from the system. After each top-author removal (line 7), the current TF estimate is incremented by one (line 8) and the removed author is added to the set of TF developers (line 9). The algorithm terminates when more than 50% of the files become abandoned (line 10-12). A file is considered *abandoned* when all its authors have been removed from the system.

**Algorithm 1:** AVL Algorithm

```
   Input: TA (list of top-authors of a system S)
   Output: tf (truck factor), TFSet (TF developers)
1  begin
2      tf ← 0;
3      TFSet ← ∅;
4      TA ← list of top-authors;
5      while TA ≠ ∅ do
6          dev ← head(TA);
7          remove-author(dev);
8          tf++;
9          TFSet ← TFSet + dev;
10         if rate-abandoned-files() ≥ 0.5 then
11             break;
12         end
13         TA ← TA − dev;
14     end
15     return tf, TFSet;
16 end
```

### B. RIG Algorithm

RIG algorithm was proposed by Rigby et al. [9], as part of a work where the authors assess and quantify the susceptibility of software projects to developer turnover. The algorithm uses a blame-based approach to compute code authorship (instead of a commit-based approach, as in the AVL algorithm). As implemented in git-based version control systems, the *blame* feature indicates the developer who last changed each line in a file. RIG algorithm defines that a line of code is *abandoned* when the `git-blame` command attributes the line to a developer who has left the project. The algorithm also defines that a file is *abandoned* when at least 90% of its lines are abandoned. RIG authors claim they use this high threshold to exclude developers with trivial contributions to a file.

RIG algorithm is presented in Listing 2. The algorithm works by simulating several Truck Factor scenarios. First, it varies the group size, $g$, of developers who leave from 1 to 200 developers (line 2). Then, the algorithm randomly

selects groups of developers to leave, each one with size $g$ (line 4). This selection procedure is performed 1,000 times, for each group size (line 3). The algorithm also computes the likelihood of each disaster scenario, i.e., the departure of the selected group of developers (line 5), using a statistical technique originally proposed to manage financial risks.[2] To make fair the comparison of RIG with AVL, we consider the TF value returned by RIG as being the lowest $g$ that implies in more than 50% of the files being abandoned in the system (lines 7-9), using the blame-based definition of abandoned files, explained in the first paragraph of this section. Indeed, RIG authors discuss the use of this solution to compare their results with other TF algorithms, as the one originally defined by Zazworka and colleagues [3].

**Algorithm 2:** RIG Algorithm

```
   Input: git-blame results for each file in the system
   Output: g (truck factor), TFSet (TF developers)
1  begin
2      for g ← 1 to 200 do
3          for i ← 1 to 1,000 do
4              TFSet ← random sample of g developers;
5              likelihood ← compute-likelihood (TFSet);
6              remove-authors(TFSet);
7              if rate-abandoned-files() ≥ 0.5 then
8                  return g, TFSet;
9              end
10         end
11     end
12     return null, null;
13 end
```

It is worth noting two important characteristics of RIG algorithm. First, it is a non-deterministic algorithm, i.e., due to the use of a random sample of developers (line 4), the results produced by the algorithm vary from one execution to another. Second, RIG can finish without computing a valid TF result (line 12). This happens when all groups of developers (line 4) do not meet the conditions of a Truck Factor scenario (line 7).

### C. CST Algorithm

Proposed by Cosentino et al. [10] in a tool paper, the CST algorithm computes the truck factor using two sets of developers. *Primary developers (P)* are those that have a minimum knowledge $K_p$ on a given software artifact (e.g., a file). *Secondary developers (S)* are the ones that have at least a knowledge $K_s$, where $K_s < K_p$. The authors suggest that $K_p$ should be set to $1/D$, where $D$ is the number of developers that have ever changed the artifact and that $K_s$ should be set to $K_p/2$. The truck factor of an artifact is defined as $|P \cup S|$, i.e., the number of developers classified as primary or secondary developers. To calculate the developer's knowledge on a file, the authors propose a set of metrics, including *last change takes it all* (all knowledge on a file is assigned to the last developer who modified it) and *multiple changes equally considered* (the knowledge of a developer $d$ on a file is defined as $C_d/C$, where $C_d$ is the number of commits on the artifact performed by $d$ and $C$ is the total number of commits on the artifact). They also propose a strategy to aggregate knowledge data to the level of directory, branch, or project. This aggregation is computed by summing up the knowledge on individual files and scaling the results considering the total number of files in a directory, branch or project. We do not provide more details and the pseudo-code of CST algorithm because they are not available in the paper describing a supporting tool [10].

### D. Other Algorithms

Zazworka et al. [3] were the first to propose an algorithm to automate the computation of Truck Factors from version histories. They define Truck Factor as the greatest number of developers a project may lose such that the remaining developers still have knowledge on at least a fraction of the project's files. However, the authors make a very strong assumption: all developers who changed a file—despite their number of commits—have knowledge on it. For this reason, the algorithm requires testing the impact of each possible combination of $j$ developers leaving the project, for $j$ ranging from 1 to the total number of developers. As a result, the algorithm only applies to small projects. Indeed, Ricca et al. [5] show that the algorithm in practice scales only to projects with less than 30 developers. Hannebauer and Gruhn [13] further explore the scalability problems of Zazworka's definition, showing that its implementation is NP-hard. We implemented Zazworka and used it to compute Truck Factors for the systems in the oracle of Truck Factors used in this paper (see Section III-A). We were only able to run the algorithm in one out of 35 systems in this oracle. This single system (nicolasgramlich/AndEngine) has only 21 contributors. For the remaining systems, the algorithm did not finish even after three days of execution. For this reason, we decided not to evaluate Zazworka in this paper.

## III. Study Design

### A. Oracle of Truck Factors

To answer the research questions proposed in the study, we created an oracle with the truck factor of open-source systems hosted on GitHub. First, we populated this oracle with a subset of the systems used in the work proposing the AVL algorithm [8]. In this work, the authors asked the developers of 133 open-source systems—by means of issues that are publicly available on GitHub—if they agree with the TF results estimated by their algorithm. Developers of 67 systems answered to these issues. We carefully read all these answers, aiming to retrieve TF results and developers. We were able to retrieve this information in two situations. First, when the GitHub developers agreed with the result produced by AVL. For example, for system netty/netty, AVL estimates TF = 2 and the project's developers promptly agreed with this

[2]We omit details on this specific likelihood computation (line 5) because our interest is the TF determination, despite the likelihood of its occurrence. We refer the interested reader to the original paper [9].

value (*"Yes. Let's just say we all hope here that neither of them get hit by a truck."*).[3] Second, when the developers did not agree with the result produced by AVL, but mentioned the correct value in their answer, together with the name of the developers responsible for this value. For example, developers of ipython/ipython did not agree with the result suggested by AVL (TF = 4) and argued that the correct value is 5 (*"... though I would add* [*developer name*] *who is the expert on many pieces of the code."*).[4]

Out of 67 answers, 20 (29.9%) fit in the first described situation and 7 answers (10.4%) fit in the second one. Therefore, 40 answers (59.7%) were discarded because they do not fit any of the situations, i.e., in these cases it is not possible to figure out the TF number and the TF developers by inspecting the GitHub answers. It is also important to clarify that the aforementioned procedure does not imply in bias towards AVL. All TF results included in the oracle were validated by the systems' principal developers. Indeed, as mentioned, seven results are different from the ones generated by AVL.

We also extended the initial oracle with new systems. To this purpose, we started by considering systems from the top-6 most popular programming languages on GitHub, i.e., JavaScript, Ruby, Python, PHP, Java, and C/C++. For each language, we selected 100 systems with the highest number of stars (a common measure of popularity of GitHub software [14]) and attending the following criteria: at least 100 contributors and three years of development history. The goal was to select mature systems with a large community of developers. We only considered systems that use GitHub to handle issues and that are not already in the oracle. This initial selection produced 27 systems. We then opened issues for each system on GitHub, asking two questions: (a) What is the TF of [project-name]? (b) Who are the developers responsible for this TF result? We received answers from 8 systems, which corresponds to a response ratio of 29.6%.

The final oracle provides the Truck Factor of 35 systems, including 27 systems reused from AVL's paper and 8 new systems. The main characteristics of these systems, including the TF results according to the systems' principal developers, are summarized in Table I. The oracle includes well-known systems, such as junit-team/junit4, d3/d3, and ReactiveX/RxJava. The number of stars ranges from 2,863 (nicolasgramlich/AndEngine) to 59,184 (d3/d3). The number of contributors range from 21 (nicolasgramlich/AndEngine) to 7,265 (saltstack/salt) and the system's age ranges from 2 (facebook/osquery) to 16 years (junit-team/junit4). Figure 1 shows a histogram of the distribution of the system's Truck Factor. As we can see, 20 systems (57.1%) have TF=1; and 5 systems (14.3%) have TF=2. Only two systems have TF $>$ 10, saltstack/salt (TF = 11) and symfony/symfony (TF = 15). The primacy of systems with a small Truck Factor (e.g., TF $\leq$ 2) is common in open-source systems, which are usually maintained and evolved by a small team of core developers [15].

[3]https://github.com/netty/netty/issues/4069

[4]https://github.com/ipython/ipython/issues/8710

| System | Language | Stars | Contrib | Age | TF |
|---|---|---|---|---|---|
| alexreisner/geocoder | Ruby | 4,309 | 233 | 8 | 1 |
| ./androidannotations | Java | 8,781 | 58 | 6 | 2 |
| atom/atom-shell | C++ | 40,401 | 526 | 4 | 1 |
| bjorn/tiled | C++ | 4,359 | 137 | 9 | 1 |
| celluloid/celluloid | Ruby | 3,457 | 90 | 6 | 1 |
| chef/chef | Ruby | 4,621 | 502 | 9 | 4 |
| d3/d3 | JavaScript | 59,184 | 117 | 6 | 1 |
| dropwizard/metrics | Java | 4,562 | 131 | 7 | 1 |
| facebook/osquery | C++ | 7,719 | 126 | 2 | 2 |
| gruntjs/grunt | JavaScript | 11,271 | 64 | 5 | 1 |
| ipython/ipython | Python | 11,005 | 473 | 9 | 5 |
| Leaflet/Leaflet | JavaScript | 17,295 | 446 | 5 | 1 |
| less/less.js | JavaScript | 14,371 | 208 | 7 | 1 |
| mailpile/Mailpile | Python | 6,756 | 119 | 5 | 1 |
| netty/netty | Java | 8,837 | 238 | 8 | 2 |
| nicolas.../AndEngine | Java | 2,863 | 21 | 7 | 1 |
| pallets/flask | Python | 24,670 | 370 | 7 | 1 |
| powerline/powerline | Python | 6,892 | 84 | 4 | 1 |
| puphpet/puphpet | PHP | 3,729 | 147 | 4 | 1 |
| ReactiveX/RxJava | Java | 20,457 | 132 | 5 | 1 |
| requirejs/requirejs | JavaScript | 10,121 | 98 | 7 | 1 |
| Respect/Validation | PHP | 3,671 | 92 | 6 | 3 |
| saltstack/salt | Python | 7,265 | 1,701 | 6 | 11 |
| sandstormio/capnproto | C++ | 4,295 | 70 | 4 | 1 |
| sass/sass | Ruby | 9,176 | 179 | 10 | 1 |
| SFTtech/openage | C++ | 5,472 | 94 | 3 | 2 |
| thoughtbot/paperclip | Ruby | 8,300 | 348 | 9 | 1 |
| capistrano/capistrano | Ruby | 9,141 | 207 | 4 | 2 |
| deis/deis | Python | 5,936 | 163 | 4 | 3 |
| ruby-grape/grape | Ruby | 7,754 | 230 | 7 | 4 |
| cantino/huginn | Ruby | 15,473 | 135 | 3 | 3 |
| junit-team/junit4 | Java | 5,602 | 133 | 16 | 4 |
| kennethreitz/requests | Python | 22,874 | 440 | 6 | 3 |
| symfony/symfony | PHP | 13,658 | 1,339 | 7 | 15 |
| tornadoweb/tornado | Python | 12,788 | 246 | 7 | 1 |

TABLE I: Oracle of TF results. Systems in the first table's section are reused from [8]. Age is measured in years.

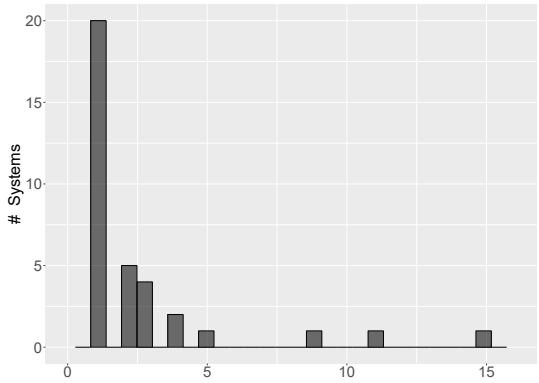



Fig. 1: Distribution of the system's Truck Factor

### B. Handling Third-Party Libraries and Aliases

Before running the Truck Factor algorithms over the commit history of the systems in Table I, we performed two cleaning steps. First, we used the open-source Linguist tool[5] to remove third-party libraries and other external source code from the cloned repositories. Linguist relies on an extensive set of pattern matching rules to detect external code copied to

[5]https://github.com/github/linguist

GitHub repositories. The tool is implemented and evolved by GitHub with the main purpose to provide statistics on the amount of source code in a repository implemented in different programming language (which also requires discarding external code, as in our case). Second, we also handled developer's names alias in the evaluated commits. In this step, we listed all developers of each project and automatically grouped those who had the same name linked with more than one e-mail and those who had the same e-mail linked with different names.

### C. Implementing, Configuring, and Executing the Algorithms

To compute the TF results estimated by AVL, we used the tool provided by the algorithm's authors, which is publicly available on GitHub.[6] To compute the results estimated by RIG, we implemented the algorithm, since the authors do not provide a public tool. For CST, we use the tool provided by the authors, also available on GitHub.[7] When answering RQ1 and RQ2, we used the default configuration suggested by the algorithms' authors (including the thresholds reported in Section II). In the case of CST, we used *multiple changes equally considered* as the metric to infer developer's knowledge.

When executing the algorithms over AVL's dataset, we used the commits available on GitHub on August, 2015 (when the survey with the developers was concluded). For the remaining systems, we used the commits available on September, 2016 (date of the second survey).

Due to its non-deterministic behavior, RIG results reported when discussing RQ1 and RQ2 refer to the median measure of 30 executions. Specifically, in RQ2 we report the median precision, recall, and F-measure results for these executions. Even after 30 runs, RIG failed to produce valid results for two systems: symfony/symfony and saltstack/salt. This happened because the 200,000 groups of developers tested by the algorithm for each of these two systems do not meet the conditions of a Truck Factor scenario (i.e., at least 50% of files abandoned, according to the git-blame criterion defined by RIG authors). Therefore, when discussing the first two research questions, RIG results do not include these two systems.

## IV. Results

### A. How accurate are the results of each algorithm? (RQ1)

To answer this first question, we compute the *error* of the Truck Factors estimated by each algorithm, compared to the oracle values, as follows: $error = \text{TF}_{algorithm} - \text{TF}_{oracle}$. Figure 2 shows violin plots with the absolute error measures. AVL and CST have very similar results. In both algorithms, the first quartile and the median of the error measures are 0 and the third quartile is 1. By contrast, RIG presents worst results, with a median error of 2.

Six systems with an outlier behavior regarding their error measures are listed in Table II. The table shows the TF of each system (as indicated by their developers) and the error of the results computed by AVL, RIG, and CST. As we can see, RIG presents the highest error measures. For the first five systems in this table, the algorithm's error ranges from 21 (junit-team/junit4) to 58 (ruby-grape/grape). Furthermore, RIG was not able to compute a TF for symfony/symfony (as mentioned in Section III-C). Indeed, symfony/symfony is the system with the highest error produced by AVL and CST. It has TF=15, but both AVL and CST estimated a TF of 11 for the system.

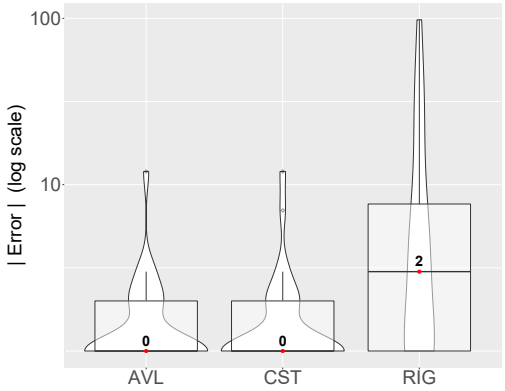



Fig. 2: Absolute error results

| System | TF | AVL | RIG | CST |
|---|---|---|---|---|
| ruby-grape/grape | 4 | 1 | 58 | 2 |
| ipython/ipython | 5 | 1 | 34 | 1 |
| alexreisner/geocoder | 1 | 0 | 31 | 0 |
| Leaflet/Leaflet | 1 | 0 | 23 | 0 |
| junit-team/junit4 | 4 | 2 | 21 | 1 |
| symfony/symfony | 15 | 11 | - | 11 |

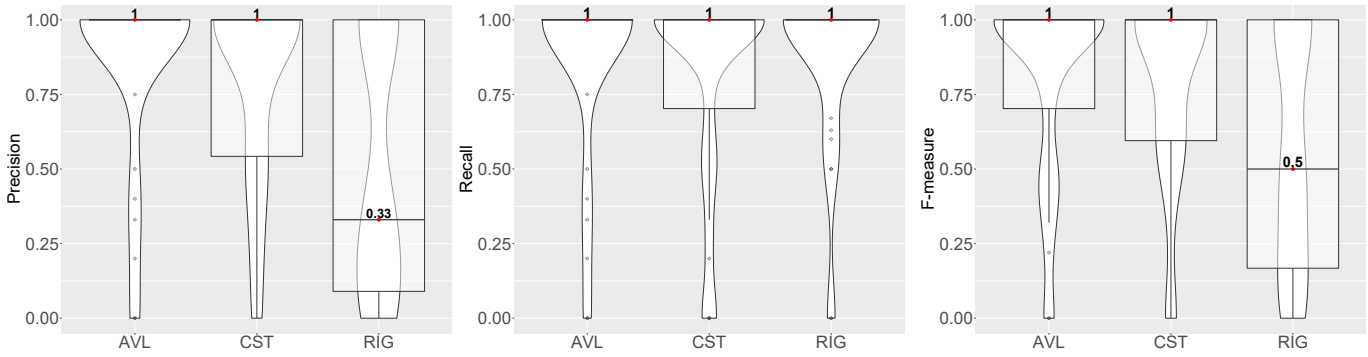
TABLE II: Error measures for outlier systems

| Group | Range | Size | Alg | % |
|---|---|---|---|---|
| TF1 | $\text{TF} = 1$ | 20 | AVL | 100 |
| | | | RIG | 60 |
| | | | CST | 85 |
| TF2-5 | $2 \leq \text{TF} \leq 5$ | 13 | AVL | 30 |
| | | | RIG | 40 |
| | | | CST | 46 |
| TF6+ | $\text{TF} \geq 6$ | 2 | AVL | 50 |
| | | | RIG | 0 |
| | | | CST | 50 |

TABLE III: Percentage of results with $error = 0$ per group of systems. Size is the number of systems in each group.

Table III shows how the algorithms perform for groups of systems with different TF results. The table divides the 35 systems in three groups: systems with $\text{TF} = 1$ (20 systems), systems with $2 \leq \text{TF} \leq 5$ (13 systems), and systems with $\text{TF} \geq 6$ (2 systems). For each group, the table presents the percentage of systems each algorithm was able to precisely estimate their TF, i.e., $error = 0$. AVL is the most accurate algorithm, with 100%, 30%, and 50% of perfect accuracy for the groups TF1, TF2-5, and TF6+, respectively. However, CST has a very close performance, with results of 85%, 46%, and 50% for the same groups. RIG results are worse; for example, the algorithm correctly infers the results of 60% of

[6]https://github.com/aserg-ufmg/Truck-Factor
[7]https://github.com/SOM-Research/busfactor




(a) Precision (b) Recall (c) F-measure

Fig. 3: Precision, Recall, and F-measure

the systems in TF1. In Table III, we also observe that the accuracy declines for the systems with high TF results. For example, although AVL is able to precisely infer the TF of all systems in TF1, the algorithm infers correctly the TF of four system (out of 13 systems) in TF2-5. A similar behavior happens with CST, although this algorithm outperformed AVL for TF2-5.

*Summary:* AVL and CST are the most accurate algorithms. They correctly estimated ($error = 0$) the Truck Factor of 71.4% (AVL) and 68.6% (CST) of the systems in the oracle. However, their accuracy decreases for systems with high TF values. For both algorithms, the highest error is 11, produced for a system with TF=15. RIG has the worst accuracy; it estimates correctly the TF of only 34.3% of the systems.

### *B. How accurate is the identification of TF developers by each algorithm? (RQ2)*

Besides the Truck Factor, the algorithms report the TF sets, i.e., the developers responsible for the estimated Truck Factor. In this second research question, we compare the TF sets reported by each algorithm with the TF sets in the oracle. To this purpose, we use three metrics: precision, recall, and F-measure. These metrics are based on the analysis of true positive, false positive, and true negative cases. A true positive occurs when a developer in the TF set computed by the algorithm is also in the oracle; a false positive occurs when a developer indicated in the TF set by the algorithm is not in the oracle; a false negative is a developer indicated by the oracle but who is not in the algorithm's TF set.

Figure 3 shows violin plots with the results of precision, recall, and F-measure. Precision is defined as $|TP|/|TP \cup FP|$, where $TP$ and $FP$ are the set of true and false positives returned by the algorithm, respectively. For AVL, the first quartile, the median, and the second quartile are 1. For CST, precision is slightly lower than AVL; the first quartile is 0.54, the median and the third quartile are 1. For RIG, the first quartile is 0.09, the median is 0.33, and the third quartile is 1. Therefore, AVL has the highest precision, followed by CST. Furthermore, RIG presented the worst precision results.

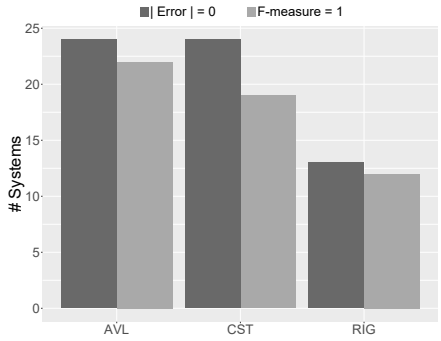



Fig. 4: Nb of systems with $(error = 0) \wedge (F\text{-}measure = 1)$

Recall measures the rate of true TF developers that the algorithms are able to find. Recall is defined as $|TP|/|TP \cup FN|$, where $FN$ is the set of false negatives returned by the algorithm. The three algorithms have median recall of 1; the third quartiles are also 1. For RIG and AVL, the first quartile is 1, and for CST, the fist quartile is 0.70. Therefore, RIG and AVL have the best recall, followed by CST.

F-measure is the harmonic mean of precision (P) and recall (R), as follows: $(2 * P * R)/(P + R)$. For AVL, the first, median, and third quartiles are 0.70, 1, and 1, respectively. For CST, the same measures are 0.60, 1, and 1. For RIG, they are 0.17, 0.5, and 1. Therefore, AVL has the highest results for F-measure, closely followed by CST and then by RIG.

*Summary:* Regarding the identification of the TF developers, AVL is the most accurate algorithm, closely followed by CST and then by RIG. The first quartile results for F-measure are 0.70, 0.60, and 0.17, respectively; and the median (second quartile) measures are 1, 1, and 0.5.

To complement the investigation of RQ2, Figure 4 shows for each algorithm the number of systems with $error = 0$ and the number of systems with F-measure $= 1$. The goal is to check how often the algorithms estimate correctly the Truck Factor results ($error = 0$), but not the Truck Factor developers (F-measure $< 1$). For AVL, this situation happens in two systems (out of 24 systems). For CST, it happens in 19

systems (out of 24 systems). Finally, for RIG it happens in a single system (out of 12 systems).

*Summary:* Usually, when the algorithms correctly estimate the Truck Factor, they also correctly identify the developers responsible for them. Notably, this happens with RIG (91% of the systems) and AVL (also 91%).

### C. What is the impact of different thresholds and configurations in the results of each algorithm? (RQ3)

*AVL:* The algorithm depends on a threshold that defines the rate of abandoned files that configures a Truck Factor disaster (see the algorithm in Section II-A). AVL's authors suggest to set this threshold to 0.5, but they do not provide detailed evidences that this is indeed the best option.[8] Figure 5 shows the results of varying this threshold in the systems of our oracle. In the x-axis, the threshold ranges from 0.1 to 1.0. The curves represent the number of systems where $error = 0$ and the number of systems where $|error| \leq 1$, for each threshold. Therefore, the second curve tolerates a small error in the Truck Factor results. As can be observed, both curves achieve their maximal value for the threshold of 0.5. Therefore, this result provides additional confidence to the default threshold recommended by AVL's authors.

*Summary:* AVL assumes a project will face serious maintenance problems if its current set of contributors cover less than 50% of the files in the system. In our study, this threshold indeed has the best results, producing an accurate Truck Factor result in 24 out of 35 systems.

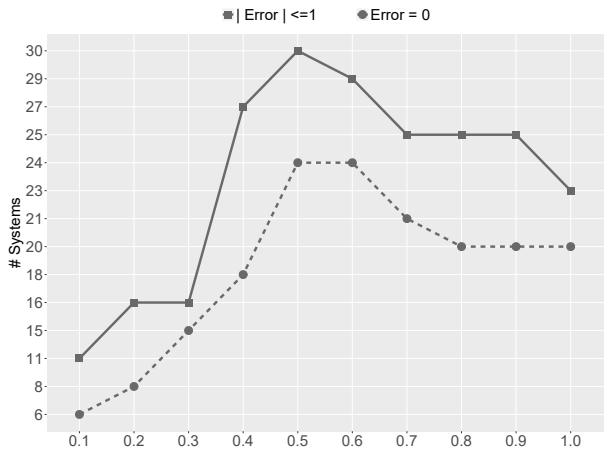



Fig. 5: Number of systems with $error = 0$ and $|error| \leq 1$, after varying AVL's threshold on abandoned files.

*RIG:* We rerun RIG varying the number of random samples of developers tested by the algorithm. In its original configuration, RIG generates 1,000 samples of developers (see Section II-B, Algorithm 2, line 3). First, we increase this threshold to 2,000 samples, aiming to double the likelihood of selecting the correct group of developers. We kept the maximal TF tested by the algorithm ($g = 200$), since in the oracle the highest TF is 15. Moreover, we also kept the threshold of 50% of abandoned files, which is used to define a Truck Factor scenario. After setting this configuration, we executed the algorithm 30 times, as performed when answering RQ1 and RQ2. Figure 6 reports the absolute error of each run, for the oracle systems that are not reused from AVL's work.[9] RIG does not produce results for symfony/symfony (as in RQ1 and RQ2). In Section V, we elaborate on the reasons why RIG does not finish with a valid TF in some systems. For now, we mention that the results in Figure 6 confirm that RIG is a non-deterministic algorithm, producing different outputs, from one execution to another. Considering 30 runs, the dispersion of the error results (maximal error minus minimal error) ranges from 3 developers (tornadoweb/tornado) to 21 developers (ruby-grape/grape).

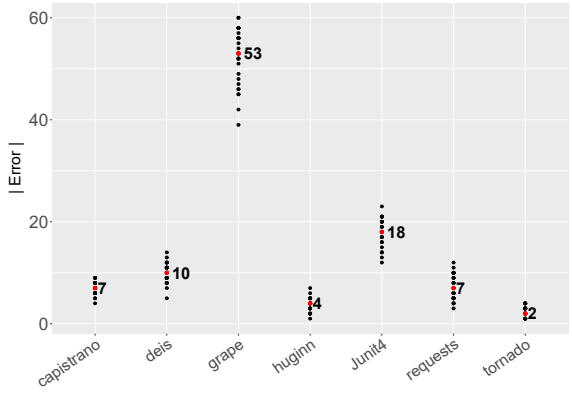



Fig. 6: Dispersion of RIG errors, considering 30 runs. The highlighted value (in red) is the median error.

*Summary:* RIG has a non-deterministic behavior, which can cause, for example, a difference of 21 developers in the Truck Factor estimated by the algorithm, from one execution to another.

Figure 7 shows the results of varying the number of samples tested by RIG from 1,000 (default value proposed by the algorithm) to 10,000 samples. The curves represent the number of systems where $error = 0$ and the number of systems where $|error| \leq 1$, for each threshold and considering the median results of 30 runs. For the first curve, varying the number of samples has no impact on the algorithm results. In all tested thresholds, RIG matches the Truck Factor of 12 systems (out of 35 systems). For the second curve, there is an increment in the number of matched systems, from 14 systems (1,000 samples) to 17 systems (5,000 samples). After this threshold, the curve remains constant.

[8]They only mention that "our estimation relies on a coverage assumption: a system will face serious delays or will be likely discontinued if its current set of authors covers less than 50% of the current set of files in the system." [8]

[9]We restrict the figure to these systems for the sake of legibility.

*Summary:* Increasing the number of tested samples—from 1,000 to 10,000 samples—does not have a major positive impact on RIG results.

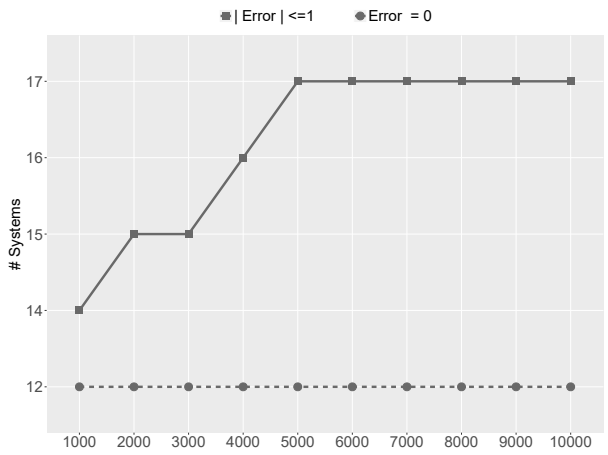



Fig. 7: Number of systems with $error = 0$ and $|error| \leq 1$, after varying RIG's threshold on the number of tested samples.

*CST:* As described in Section II-C, the algorithm depends on two thresholds: primary developers knowledge ($K_p$) and secondary developers knowledge ($K_s$). It is also possible to configure the metrics used to compute knowledge on source code. In this case, the options are *last change takes it all (LCTA)* and *multiple changes equally considered (MCEC)*. However, CST is implemented as a GUI-based tool and the algorithm is tightly coupled to the interface layer. For this reason, it is not simple to experiment with different thresholds, by means of scripts that call the core algorithm's implementation. Given this context, we investigate only the impact of changing the knowledge metrics on CST results. Table IV shows the number of systems with $error = 0$ and $error \leq 1$. In both scenarios, MCEC leads to more accurate result than LCTA. For example, 24 systems have their Truck Factor correctly estimated when CST is configured to use MCEC. When LCTA is used, this number decreases to 18 systems.

| **Metric** | $error = 0$ | $error \leq 1$ |
|---|---|---|
| LCTA | 18 | 25 |
| MCEC | 24 | 30 |

TABLE IV: Number of systems with $error = 0$ and $error \leq 1$, when CST is configured to use LCTA and MCEC metrics.

*Summary:* When using CST, MCEC is the knowledge metric that leads to the best results. When MCEC is used (as in RQ1 and RQ2), there is an increment of six systems with $error = 0$, when compared to the alternative metric LCTA.

## V. Discussion and Complementary Questions

### A. What is the runtime performance of each algorithm?

We did not create a specific research question to investigate runtime performance because the three algorithms execute very fast, for all systems in the oracle. For each combination of algorithm and system, the execution requires less than one minute (in a HP Xeon six-Core server, with 64 GB RAM, and Ubuntu 12.04). However, this time does consider the check out of the systems from GitHub and the preprocessing steps to remove non-source code files, third-party APIs and to handle alias. Specifically, the tool that implements the CST algorithm uses a database to store data about the source code and the developers of the target systems. Creating and populating this dataset is the most time consuming task when using CST. It required for example around eight hours in the case of symfony/symfony and saltstack/salt, which are the two system with more contributors in the oracle.

### B. Why and when the algorithms fail?

*RIG:* The algorithm is inspired on Zazworka's precursor algorithm to estimate Truck Factors. As discussed in Section II-D, Zazworka has a serious scalability problem, since it tests all combinations $\binom{N}{g}$, where $N$ is the number of contributors and $g$ ranges from 1 to $N$. To tackles this problem, RIG selects—for each $g$—exactly 1,000 random samples of developers. Therefore, the algorithm can easily miss the group of TF developers. For example, rub-grape/grape has TF=4 and 230 contributors (see Table III-A). An exhaustive algorithm must test $\binom{230}{4} = 113,525,855$ combinations to guarantee the selection of the correct group of four developers. However, RIG only tests 1,000 combinations. As a result, RIG suggests a TF = 56 for this system, i.e., it only evaluates the TF developers when testing for combinations of 56 developers.

The random selection of developers also leads RIG to do not present results for some systems. In such cases, all tested samples do not meet the conditions of a Truck Factor disaster. Indeed, this happens for two systems, symfony/symfony and saltstack/salt, when answering RQ1 and RQ2. In RQ3, after increasing the number of samples to 2,000, RIG started to estimate the Truck Factor of saltstack/salt.

*AVL and CST:* We acknowledge that explaining the reasons for AVL and CST failures would require a second round of surveys with the developers of the systems in our oracle, asking then why certain developers with relevant contributions to their systems—counting the number of commits—are not ranked as TF developers, while other developers with less relevant contributions are. For example, in symfony/symfony (TF=15) one of the TF developers is the 22nd developer with more commits in the system. However, he seems to play an important social role in the system's community. For example, the announcement of his engagement in the group of symfony/symfony's core developers mentions that "*he is well known for being the one able to get hundreds of comments on his pull requests; he knows how to engage the community!*"[10]. In other words, developers can play specific roles in a project, which are important enough to raise them to the level of TF

[10]http://symfony.com/blog/new-symfony-core-team-member-grgoire-pineau

developers, but that are not reflected in their commit history. In such cases, AVL, CST and RIG will fail to identify them.

### C. Are Truck Factor scenarios realistic?

Rigby et al. [9] claim that Truck Factor scenarios (i.e., the loss of all TF developers) computed using loss percentages are unrealistic. They mention that for large projects these algorithms produce results with hundreds of developers, for instance. Therefore, only outsourcing an entire project or Google removing funding for Chrome could trigger such massive loss of developers. For this reason, they suggest that Truck Factor should be renamed to "Airplane Factor", to refer to an airplane disaster with all developers on board.

In fact, the likelihood of a disaster decreases as the Truck Factor increases. For example, it is more likely a Truck Factor scenario in a single-man project than in projects with 200 key developers, as seems to be the case of Google Chrome. However, the Truck Factor computation also reveals the key developers in a project (and the project's area where they have expertise on). This is useful both in small and in large projects. For example, it is indeed unlikely that 200 developers will leave a project at once. However, even in this case, managers can benefit from information on project's key developers (among possibly thousands of other developers) and about the code locations where they are making their contributions.

## VI. Threats to Validity

### A. Construct Validity

The data gathering of AVL results was performed by a tool developed by the algorithm's authors. We could access the source code of the tool, so that we were able to inspect it and verify that it implements AVL as described in this work. The data of CST was also gathered by a tool developed by the authors of the algorithm; however, we did not inspect the source code of this tool. Nevertheless, as the tool is implemented by the authors who proposed CST, we consider that it follows the algorithm presented by Cosentino et al. [10]. We did not find any public tool that implements RIG. Then, we implemented the algorithm by ourselves, trying to carefully follow the algorithm description presented by its authors. Since the algorithm's core is not complex, we claim the risks of misunderstandings are small. In fact, the most complex part of RIG is the one that computes the likelihood of each disaster scenario, which is not used in our comparison. Our central interest is the determination of TF values, despite the likelihood of their occurrence.

### B. Internal Validity

We use an oracle of Truck Factors that comprises 35 open-source software systems hosted on GitHub. The oracle has two types of data: the Truck Factor number and the name of the developers that are part of the Truck Factor. The data were gathered in two ways: (i) an initial oracle was constructed by presenting AVL results to the contributors of the software systems and asking them to indicate whether the results are correct or not; (2) by asking the contributors of eight open-source systems to indicate the data, i.e., the Truck Factor value as well as the developers they consider to be part of the Truck Factor. One may consider that using two approaches to construct the oracle might be misleading. However, these approaches are complementary and, in both cases, the data are based in the opinion of key team members. To mitigate a threat related to inaccurate answers we only considered consensual responses coming from the top-10 contributors of the projects. Finally, the contributors are identified by their names on GitHub. However, there are cases of contributors with more than one name. To solve this issue, we pre-process the data in order to merge names referring to the same contributor.

### C. External Validity

The data analyzed in this work are from 35 open-source software systems hosted on GitHub. Aiming representativeness, we considered systems implemented in six popular programming languages. The systems have different popularity, varying from 2,863 to 59,184 stars on GitHub. The same occurs with the team size and the maturity of the projects: the teams vary from 21 to 1,701 contributors, and the systems have from 2 to 16 years. Even though, it is not possible to claim that the study results generalize to any open-source software. In the same vein, they may not generalize to proprietary software.

## VII. Related Work

The interest on studying source code knowledge is due to many reasons. As pointed by Fritz et al. [12], it is unusual that developers have all knowledge about a software system, and therefore they usually need to know who to ask about doubts on the code. Previous research on code knowledge was carried out with three main purposes: (a) to define models and metrics to assess the level of developer's knowledge on the code; (b) to define algorithms to compute such metrics; (c) and to evaluate the usefulness of the proposed models and metrics. Truck Factor is one of the main metrics proposed in the literature in this context. However, we still lack studies that evaluate and compare the existing algorithms to calculate Truck Factors. The study presented in this paper aims to contribute to overcome such problem.

Besides Truck Factor, two other concepts are commonly associated to source code knowledge: *ownership* and *authorship*. Authorship is the result of the contribution of a developer (author) to a piece of software, e.g., a line of code, a method, or a class. Ownership refers to the level that a contributor is responsible (or owns) a piece of software [16], [17]. Rahman and Devanbu [18] propose metrics for authorship and ownership. For them, authorship is the rate of contribution of a developer to a code fragment, calculated as a function of the number of lines contributed by him. They define ownership as the author who has the highest authorship on a code element. Truck Factor is related to both concepts, since it aims to identify the key contributors of a software project. Torchiano et al. [19] refer to Truck Factor as the "collective code ownership" of

a project. Therefore, Truck Factor is a system-level metric, whereas authorship and ownership are fine-grained metrics.

Rahman and Devanbu [18] report the results of a study on code ownership and defects. They investigate whether the number of developers that work on a file impacts in the number of defects. Among their findings, they conclude that implicated code (i.e., lines modified to fix bugs) tends to have contributions by a single developer, contrasting with the idea that more developers working in a piece of code leads to more defects. Bird et al. [16] investigate the role of ownership in software faults, by analyzing two large commercial projects. They report that the number of minor contributors of a module is positively correlated with the number of faults. Foucault et al. [17] replicate this study in seven open-source software systems. However, in this case they did not find a correlation between ownership and faults. Avelino et al. [20] use the DOA metric to study code authorship in the Linux kernel. They show that the kernel has 13,436 developers, but only 3,459 (26%) are authors of at least one file.

Assessing code knowledge is also important to identify *major* and *minor* contributors [17]. A correlated concept is *core* and *peripheral developers*. Core developers are the ones that make the strategic and long-term decisions, including defining the system's architecture and implementing the most critical features. In contrast, peripheral developers are responsible for bug fixes and simple enhancements. A commonly used approach to distinguish core and peripheral developers relies on the number of commits made by each developer [21]. The 80th percentile threshold is often used to separate the developers. Developers with a number of commits above this threshold are in the core; the other ones are peripheral.

After investigating 2,496 projects hosted on GitHub, Yamashita et al. [22] report that several projects are susceptible to Truck Factor scenarios. They report that 26%-58% of the projects have core teams that are too small ($\leq$ 10% of active contributors) to be considered compliant with the Pareto principle. Ye and Kishida [23] mention that the development of GIMP (GNU Image Manipulation Program) was once halted for about 20 months because two developers left the project.

## VIII. Conclusion

In this work, we provide a comprehensive study comparing three Truck Factor algorithms: AVL (proposed by Avelino et al. [8]), RIG (proposed by Rigby et al. [9]), and CST (proposed by Cosentino et al. [10]). For this purpose, we relied on an oracle with the Truck Factor of 35 open-source projects hosted on GitHub. The oracle data are based on information provided by the projects' teams. We investigated three aspects in an experimental study: (i) accuracy of the computed Truck Factors, i.e., the ability of the algorithms to predict a Truck Factor close to the one in the oracle data; (ii) accuracy of the Truck Factor sets, i.e., the ability of the algorithms to predict the developers responsible for the Truck Factors; and (iii) the impact of different thresholds and configurations on the algorithms results.

The main findings of this work are as follows.

1) AVL and CST are the most accurate algorithms for predicting the Truck Factor of open-source projects.
2) AVL is the most accurate algorithm to predict the Truck Factor sets, i.e., the developers responsible for a given Truck Factor, closely followed by CST.
3) The best threshold for AVL is 50%, that is, in order to get the most accurate results, we should configure AVL to consider that a project will be in risk if its current developers cover less than 50% of the files in the system.
4) RIG has a non-deterministic behavior and changing the number of samples tested by the algorithm has a minor impact on its results.
5) The *multiple changes equally considered* metric used by CST to infer code knowledge leads to the best results.

Our results may direct further research in this area as well as the application of the Truck Factor metric in the practice. The main future work we envision includes the investigation of the impact of software architecture characteristics in the Truck Factor; the investigation of the relationship between Truck Factor and other software quality dimensions, such as fault-proneness; and the role of other properties, including social proprieties, in the computation of Truck Factors.

The oracle with the Truck Factors (and their respective developers) used in this paper is publicly available at: https://github.com/aserg-ufmg/datasets.

## Acknowledgments

This study is supported by grants from FAPEMIG, CAPES, and CNPq. We thank the GitHub developers who answered our issues about Truck Factors. We also thank G. Avelino for providing us the first oracle of Truck Factor results and V. Consentino for helping us to use the tool that implements the CST algorithm.

## References

[1] A. Mockus, "Organizational volatility and its effects on software defects," in *18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2010, pp. 117–126.

[2] L. Williams and R. Kessler, *Pair Programming Illuminated*. Addison Wesley, 2003.

[3] N. Zazworka, K. Stapel, E. Knauss, F. Shull, V. R. Basili, and K. Schneider, "Are developers complying with the process: an XP study," in *4th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2010, pp. 14:1–14:10.

[4] F. Ricca and A. Marchetto, "Are heroes common in FLOSS projects?" in *4th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2010, pp. 1–4.

[5] F. Ricca, A. Marchetto, and M. Torchiano, "On the difficulty of computing the truck factor," in *12th Product-Focused Software Process Improvement (PROFES)*. Springer, 2011, vol. 6759, pp. 337–351.

[6] T. Mens, "An ecosystemic and socio-technical view on software maintenance and evolution," in *32nd International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 1–8.

[7] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2004.

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

[9] P. C. Rigby, Y. C. Zhu, S. M. Donadelli, and A. Mockus, "Quantifying and mitigating turnover-induced knowledge loss: case studies of Chrome and a project at Avaya," in *38th International Conference on Software Engineering (ICSE)*, 2016, pp. 1006–1016.

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

[11] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill, "A degree-of-knowledge model to capture source code familiarity," in *32nd International Conference on Software Engineering (ICSE)*, 2010, pp. 385–394.

[12] T. Fritz, G. C. Murphy, E. Murphy-Hill, J. Ou, and E. Hill, "Degree-of-knowledge: modeling a developer's knowledge of code," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 2, pp. 14:1–14:42, 2014.

[13] C. Hannebauer and V. Gruhn, "Algorithmic complexity of the truck factor calculation," in *15th Product-Focused Software Process Improvement (PROFES)*. Springer, 2014, vol. 8892, pp. 119–133.

[14] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of GitHub repositories," in *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 334–344.

[15] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and Mozilla," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 3, pp. 309–346, 2002.

[16] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: Examining the effects of ownership on software quality," in *19th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2011, pp. 4–14.

[17] M. Foucault, J.-R. Falleri, and X. Blanc, "Code ownership in open-source software," in *18th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2014, pp. 39:1–39:9.

[18] F. Rahman and P. Devanbu, "Ownership, experience and defects: A fine-grained study of authorship," in *33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 491–500.

[19] M. Torchiano, F. Ricca, and A. Marchetto, "Is my project's truck factor low?: Theoretical and empirical considerations about the truck factor threshold," in *2nd International Workshop on Emerging Trends in Software Metrics (WETSoM)*, 2011, pp. 12–18.

[20] G. Avelino, L. Passos, A. Hora, and M. T. Valente, "Assessing code authorship: The case of the Linux kernel," in *13th International Conference on Open Source Systems (OSS)*, 2017, pp. 1–12.

[21] M. Joblin, S. Apel, C. Hunsen, and W. Mauerer, "Classifying developers into core and peripheral: An empirical study on count and network metrics," in *39th International Conference on Software Engineering (ICSE)*, 2017, pp. 1–12.

[22] K. Yamashita, S. McIntosh, Y. Kamei, A. E. Hassan, and N. Ubayashi, "Revisiting the applicability of the pareto principle to core development teams in open source software projects," in *14th International Workshop on Principles of Software Evolution (IWPSE)*, 2015, pp. 46–55.

[23] Y. Ye and K. Kishida, "Toward an understanding of the motivation open source software developers," in *25th International Conference on Software Engineering (ICSE)*, 2003, pp. 419–429.