




Algorithms for estimating truck factors: a comparative study

Mívian Ferreira¹  · Thaís Mombach¹ · Marco Tulio Valente¹ · Kecia Ferreira²

Published online: 29 August 2019

© Springer Science+Business Media, LLC, part of Springer Nature 2019

Abstract

Software development is a knowledge-intensive industry. For this reason, concentration of knowledge in software projects tends to be very risky, which increases the relevance of strategies that reveal how source code knowledge is distributed among team members. The truck factor (also known as the bus factor) is an increasingly popular concept—proposed by practitioners—that indicates the minimal number of developers that have to be hit by a truck (or leave the team) 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 concept, algorithms have been proposed to automatically compute truck factors, using maintenance activity data extracted from version control systems. However, we still lack large studies that assess the results of truck factor algorithms. To fulfill this gap in the literature, this paper describes the results of three empirical studies. In the first study, we validate the results produced by three algorithms to estimate truck factors. To this purpose, we build an oracle of truck factors, gathered via a survey with 35 open-source project teams. In the second study, we provide a comparison between truck factors and core developers, a related concept commonly used to denote the key developers of open-source projects. Our results indicate that truck factor developers are in most cases a subset of core developers. Finally, as the algorithms proposed so far are based in commit data, in the third study, we investigate other factors that may impact the computation of truck factors.

Keywords Truck factor · Code ownership · Core developers · Developer turnover · Mining software repositories

1 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.

✉ Mívian Ferreira
mivian.ferreira@dcc.ufmg.br

¹ Department of Computer Science, UFMG, Belo Horizonte, MG, Brazil

² Department of Computing, CEFET-MG, Belo Horizonte, MG, Brazil

In this context, developer turnover is a serious risk to software projects (Williams and Kessler 2003; Mockus 2010; Coelho and Valente 2017). 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 (Williams and Kessler 2003; Zazworka et al. 2010; Ricca and Marchetto 2010; Ricca et al. 2011; Mens 2016). The measure is also known as bus factor/number or lottery factor. 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. By contrast, 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 (Williams and Kessler 2003; Beck and Andres 2004).

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, posted to the FindBugs mailing list: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.

In fact, TF events are not just theoretical and anecdotal situations, particularly in open-source projects. For example, Avelino et al. (2019a) estimated the TFs of a large and curated sample of 1932 GitHub projects. They report that 16% of these projects were abandoned by all of their TF developers, at least in a given moment of their development history. However, 41% of these projects were also able to survive, by attracting new TF developers. Usually, these new developers were motivated to increase their contributions due to their own usage of the projects.

Due to the relevance of truck factor, algorithms have been proposed to estimate this measure. Nevertheless, we lack studies to assess the differences among these algorithms and to identify whether the results presented by them are compatible with real scenarios of software project development. Furthermore, other concepts defined in the literature are closely related to truck factor, e.g., core developers (Yamashita et al. 2015). Both concepts aim to identify the most important developers in a software project. However, there are no works—to our knowledge—that provide an empirical investigation on the relationship between both concepts.

In this paper, we first compare three algorithms for estimating truck factors. Additionally, the concepts of core developers and truck factor are compared. Finally, we conduct a survey to provide feedback on why existing algorithms for estimating truck factor fail in the case of some projects.

Next, we briefly describe the studies reported in this paper.

¹Findbugs is a popular open-source bug-finding tool for Java systems): <https://mailman.cs.umd.edu/pipermail/findbugs-discuss/2016-November/004321.html>

Study #1: Comparison of algorithms for computing truck factor In this first study, we validate the results produced by three recent algorithms proposed in the literature to estimate truck factors: AVL Algorithm (Avelino et al. 2016), RIG Algorithm (Rigby et al. 2016), and CST Algorithm (Cosentino et al. 2015). To achieve our goals, we first build an oracle of truck factors by means of a survey with the developers of 35 well-known open-source systems hosted on GitHub. In this first study, we ask the following research questions:

RQ1. 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 value reported in the constructed oracle.

RQ2. 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 {Bob, Alice} are the developers responsible for this result. An algorithm can correctly estimate the system's TF, but by considering {Carlos, Carol} as the key developers. Therefore, this second research question investigates how often this situation happens with the studied algorithms.

RQ3. 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.

Study #2: Truck factor vs. core developers In the second study, we compare the concepts of truck factor and core developers. We seek to reveal the relationship between these concepts, since both aim to identify developers playing a central role in software projects. Particularly, we compare the results of two heuristics for computing core developers with the truck factors provided by our oracle. To guide this comparison, we ask two research questions:

RQ4. How accurate are the results provided by each heuristic? With this research question, we aim to identify how close is the number of core developers, estimated by the considered heuristics, to the truck factors reported in the oracle.

RQ5. How accurate is the identification of truck factor developers by each heuristic? This question aims to identify whether the core developers indicated by the studied heuristics are the same authors reported in the oracle of truck factors (i.e., our goal is similar to the one proposed in RQ2, but using core developers).

RQ6. What is the relation between truck factor and core developers sets? In this research question, we investigate the relation between truck factor and the core developer sets, by analyzing their intersection. Our hypothesis is that the truck factor developers are a subset of core developers.

Study #3: Other factors for estimating truck factor In this third study, we carry out a survey to identify other factors, not related with commits, that can affect the identification of truck factor developers. We ask the following research question:

RQ7. Why algorithms for estimating truck factors fail? With this research question, we aim to identify why the algorithms for estimating truck factors fail in some circumstances and systems.

Contributions We highlight the following contributions of the studies presented in this work: (i) a truck factor oracle obtained through a survey with the leading developers of 35 open-source projects hosted on GitHub; (ii) a comprehensive study that compares the main algorithms proposed in the literature to estimate truck factors; (iii) a study that compares truck factor with core developers; and (iv) the identification of other factors—besides commits—that can lead a developer to be part of truck factor sets.

Previous work The present work extends a previously published paper (Ferreira et al. 2017), in which three algorithms for estimating truck factors were compared (study #1). In the present paper, we additionally compare the concepts of truck factor with a related concept: core developers (study #2). Particularly, we evaluate two heuristics proposed in the literature to compute core developers: commit-based and LOC-based. The commit-based heuristic identifies core developers according to the number of commits of a developer in a software project. The LOC-based heuristic indicates core developers based on the number of lines of code added or removed by a developer in the project. Finally, we present the results of a novel survey designed to reveal other factors—not related to commits—that may impact the computation of truck factor sets (study #3).

Organization The remaining of this paper is organized as follows. Section 2 introduces the concepts of truck factor and core developers, and the main algorithms proposed in the literature to estimate them. Section 3 describes the method used to conduct the studies presented in this work. Section 4 presents a comparative study of three algorithms to compute truck factors. Section 5 describes a comparative study between the concepts of truck factor and core developers. Section 6 presents the results of a survey conducted with developers to identify factors that may impact the computation of truck factor sets. Section 7 provides further results about the conducted studies. In this section, we also discuss our results and the limitations of the algorithms compared in this work. Section 8 discusses threats to validity. Section 9 presents related work. Section 10 presents the conclusion of this paper, outlining its main contributions and future work.

2 Background

This section presents the main concepts and algorithms used in this work. First, we present the truck factor concept and the main algorithms proposed in the literature to compute this metric (Section 2.1). Second, we describe the core developers concept and two heuristics to estimate the core developers of a system (Section 2.2).

2.1 Truck factor

Truck factor is a software metric that aims to identify the distribution of knowledge among development team members. Also known as bus factor, bus number, truck number, or lottery factor, the metric is defined as “the minimum number of people on your team who must be hit by a truck so that your project gets into serious trouble” (Bowler 2005). Naturally, being “hit by a truck” is a metaphor used to represent the unexpected absence of a team member.

As reported by Ricca et al. (2011), in addition to identifying the distribution of knowledge among team members, Truck factor may also be used to identify other possible risks to the project by noting how much the project is dependent on certain developers.

High values of truck factor indicate that there is a homogeneous distribution of knowledge among team members and that if a team member leaves the project, it may suffer an impact, but it will not be dramatic. On the other hand, low values for this metric indicate a concentration of knowledge on individual team members. In this case, if one of those members leaves the project, it may be difficult or impossible to evolve the project.

2.1.1 Algorithms for estimating truck factor

In this section, we present the main algorithms proposed in the literature to estimate truck factor: ZWK Algorithm, proposed by Zazworka et al. (2010), CST Algorithm, proposed by Cosentino et al. (2015), AVL Algorithm, proposed by Avelino et al. (2016), and RIG Algorithm, proposed by Rigby et al. (2016). These algorithms are evaluated and compared in this paper.

2.1.2 ZWK algorithm

Zazworka et al. (2010) presented the first description of an algorithm for estimating truck factor. They used the metric to point out differences between the distribution of knowledge in projects that make use of extreme programming (XP) and projects that use other software development methodologies. Zazworka et al. (2010) describe the logic applied in the experiments execution presented in their work. However, they do not provide a concrete algorithm structure. Ricca et al. (2011) provide the first formalization of this algorithm, as shown in Algorithm 1.

Algorithm 1 ZWK algorithm.

Input: coverage, devs (List of Developers), files (List of Files)

Output: TF (Truck Factor)

begin

```

    for  $j = 1$  to  $devs.size()$  do
        minimalCoverage  $\leftarrow$  100;
        for each  $comb \in Combination(j, devs)$  do
            covered  $\leftarrow$  0;
            for each  $file \in files$  do
                if  $getFileDevs(file)-comb \neq null$  then
                    covered  $\leftarrow$  covered + 1;
                end
            end
            currentCoverage = covered/files.size() * 100;
            if  $currentCoverage \leq minimalCoverage$  then
                minimalCoverage  $\leftarrow$  currentCoverage;
            end
        end
        minimalCoverageArray[j]  $\leftarrow$  minimalCoverage;
        if  $minimalCoverageArray[j] < coverage$  then
            break;
        end
    end
    return j;
end
```

ZWK Algorithm receives as input the coverage, i.e., percentage of files that should be abandoned to configure a disaster scenario; the list of the system developers (devs), i.e., developers who have executed at least one commit in the system; and the list of system files (files). The algorithm simulates all possible scenarios of developers leaving the team. Initially (line 2), the number of developers to be removed from the project is defined (j), $1 \leq j \leq n$, where n is the total number of developers. Then, for that value of j , it is considered that the system is totally covered, i.e., all files have at least an author (line 3). Given the set of developers whose output will be simulated, the algorithm checks if each system file is covered (line 6–10). In this case, a file is considered covered if after removing the developer combination from the set of developers of the file (given by *getFileDevs*()), it does not become an empty set (line 7–9). Then, it verifies if the current coverage (current coverage) is the minimal coverage for the combination size (j) (lines 11–14). The truck factor is given by the number of developers present in the first combination whose coverage (minimal coverage) is less than the value of variable coverage (line 17–19).

2.1.3 AVL algorithm

Avelino et al. (2016) proposed the AVL algorithm that relies on the Degree-of-Authorship (DOA) measure to define the authors, i.e., the key developers, of each file in a system (Fritz et al. 2010, 2014). 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 were defined after empirical experiments performed with developers of two proprietary software (Fritz et al. 2014). 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 was defined after an empirical experiment when the authors of AVL Algorithm manually validated DOA results produced for a sample of 120 source code files from six open-source systems (Avelino et al. 2016).

Algorithm 2 AVL algorithm.

Input: TA (list of top-authors of a system S)
Output: tf (Truck Factor), TFSet (TF developers)

```

begin
    tf ← 0;
    TFSet ← ∅;
    TA ← list of top-authors;
    while TA ≠ ∅ do
        dev ← head(TA);
        remove-author(dev);
        tf++;
        TFSet ← TFSet + dev;
        if rate-abandoned-files() ≥ 0.5 then
            break;
        end
        TA ← TA − dev;
    end
    return tf, TFSet;
end

```

AVL algorithm is presented in Algorithm 2. 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 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.

2.1.4 RIG algorithm

Rigby et al. (2016) proposed the RIG algorithm, as part of a piece of work in which the authors assessed and quantified 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 Algorithm 3. 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 1000 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 initially proposed to manage financial risks. We omit details on this specific likelihood computation (line 5) because our interest is the TF determination, despite the probability of its occurrence. We refer the interested reader to the original paper (Rigby et al. 2016). 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, like the one originally defined by Zazworka et al. (2010).

Algorithm 3 RIG algorithm.

Input: git-blame results for each file in the system
Output: g (Truck Factor), TFSet (TF developers)
begin
 for $g \leftarrow 1$ **to** 200 **do**
 for $i \leftarrow 1$ **to** 1,000 **do**
 TFSet \leftarrow random sample of g developers;
 likelihood \leftarrow compute-likelihood (TFSet);
 remove-authors(TFSet);
 if $\text{rate-abandoned-files}() \geq 0.5$ **then**
 return g , TFSet;
 end
 end
 end
 return null, null;
end

We emphasize 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 situation happens when all groups of developers (line 4) do not meet the conditions of a truck factor scenario (line 7).

2.1.5 CST algorithm

Cosentino et al. (2015) proposed, in a tool paper, the CST algorithm that computes the truck factor using two sets of developers: primary and secondary 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 of a file, Cosentino et al. (2015) propose a set of metrics, including last change takes it all and multiple changes equally considered. In last change takes it all, all knowledge on a file is assigned to the last developer who modified it. In 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. In their tool paper, Cosentino et al. (2015) do not either provide more details nor the pseudo-code of CST algorithm.

2.2 Core developers

Core developers are the ones who play a leading role in a project (Yamashita et al. 2015), making important contributions such as defining the system's architecture and implementing critical features (Joblin et al. 2017). The concept is related to truck factor because both concepts are tightly coupled to the distribution of knowledge in software projects. Nevertheless, we lack studies to investigate the extent and the characteristics of this relation. Therefore, in this paper, we carried out an empirical study with this purpose (Section 5).

In this study, we use two heuristics to identify core developers: commit-based heuristic and LOC-based heuristic (Yamashita et al. 2015).

Commit-based heuristic This heuristic identifies core developers according to their contributions to a software project. In this case, a contribution is defined as a commit. Yamashita et al. (2015) define that the core developers are the ones responsible for 80% of the commits of a project. Algorithm 4 presents the commit-based heuristic. The algorithm receives as input a list of developers (LD) and the total number of commits made in the system (size). First, the coreDev set is initialized with an empty set and the variable cumulativeRatio is initialized with zero (lines 2–3). Then, the list of developers is ordered according to the number of commits that each developer performed in the project (line 4). After that, the process of identifying core developers starts (5–10). In this process, the first developer on the LD list is selected (dev) (line 6). Then, his/her commit rate is calculated by dividing dev's number of commits by the total number of commits of the system (line 7). The core

developers set (line 11) is composed of developers who together are responsible for at least 80% of the commits in the project.

Algorithm 4 Commit-based heuristic.

Input: LD (list of developers of a system S), size (total of commits of a system S)

Output: coreDev (set of Core Developers)

begin

coreDev $\leftarrow \emptyset$;

cumulativeRatio $\leftarrow 0$;

sort(LD);

while cumulativeRatio ≤ 0.8 **do**

dev \leftarrow head(LD);

cumulativeRatio \leftarrow cumulativeRatio + devRatio(dev, size);

coreDev \leftarrow coreDev + dev;

remove-developers(dev);

end

return coreDev;

end

LOC-based heuristic The LOC-based heuristic identifies core developers according to the number of lines of code added and removed by the developer in the project. More precisely, Yamashita et al. (2015) identify a core developer by the sum of the added and removed lines, which they called churn. The algorithm is essentially the same of commit-based heuristic. The only difference is that the input parameter size receives the total churn of the analyzed project. The core developer set is composed of the developers who together are responsible for at least 80% of the churn of a software project.

3 Method

In this section, we describe the design of the studies conducted in this work. It is organized as follows: the dataset used in the paper and its cleaning steps are presented in Sections 3.1 and 3.2, respectively. Section 3.3 describes the implementation, configuration, and execution of the algorithms compared in this work. Finally, Section 3.4 presents the steps to conduct a survey about truck factors.

3.1 Dataset

The dataset used in this work is constituted by a previous dataset proposed by Avelino et al. (2016) and by an extension carried out in the present work. The whole dataset is described in this section.

3.1.1 AVL dataset

In this work, we reuse the dataset proposed by Avelino et al. (2016) to validate the results obtained by their algorithm with GitHub developers. The selection of the software systems included in this dataset was carried out in three steps, as follows.

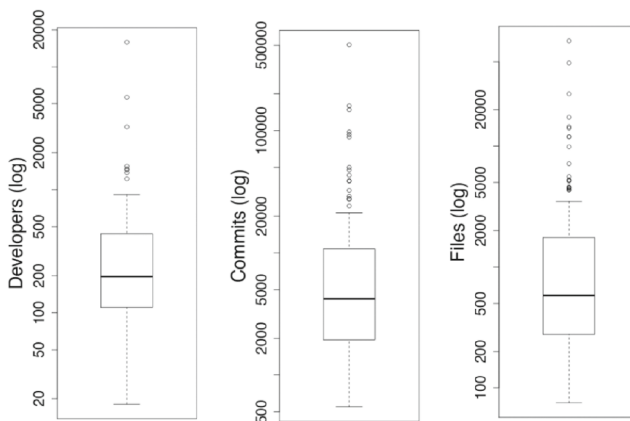
Table 1 AVL dataset (Avelino et al. 2016)

Language	#Systems	#Devs	#Commits	#Files
JavaScript	22	5740	108,080	24,688
Python	22	8627	276,174	35,315
Ruby	33	19,960	307,603	33,556
C/C++	18	21,039	847,867	107,464
Java	21	4499	418,003	140,871
PHP	17	3329	125,626	31,221
Total	133	63,194	2,083,353	373,115

1. Selecting the Programming Language: Avelino et al. (2016) chose systems from the six programming languages with the largest number of repositories in GitHub: JavaScript, Python, Ruby, C/C++, Java, and PHP.
2. Selecting Software Systems: Avelino et al. (2016) selected the top-100 most popular systems of each of the languages identified in the previous step. The adopted popularity criterion was the number of system's stars.
3. Filtering the Software Systems: From the systems obtained in the previous step, the most important ones in each programming language were selected, according to their number of developers, commits, and files. Basically, the systems that were in the first quartile of the distribution of the number of developers, commits, and files were discarded by Avelino et al. (2016). Finally, systems with evidence of incorrect migration (i.e., with more than 50% of system files added to the repository in less than 20 commits) were discarded.

Table 1 characterizes the AVL dataset used in this study. It includes 133 systems, implemented in six different programming languages; Ruby is the language with more systems (33), and PHP is the language with fewer systems (17). Considering all systems, the dataset includes more than 63,000 developers, 373,000 files, and 2 million commits.

Figure 1 shows the distribution of the number of developers, commits, and files of the dataset. For number of developers, the first, second, and third quartiles are 111,

**Fig. 1** AVL dataset: distribution of number of developers, commits, and files (Avelino et al. 2016)

197, and 437, respectively. The systems with the highest number of developers are `torvalds/linux` (15.8K), `Homebrew/homebrew` (5.6K), and `rails/rails` (3.2K). Regarding the number of commits, the first, second, and third quartiles are 1950, 4210, and 10,800, respectively. The three systems with the highest number of commits are `android/platform_frameworks_base` (159.3K) and `JetBrains/intellij-community` (148.5k). Finally, for system source code files, the first, second, and third quartiles are 269, 574, and 1750, respectively. The three systems with the largest number of files are `JetBrains/intellij-community` (74K), `torvalds/linux` (49K), and `android/platform_frameworks_base` (27K).

3.1.2 Oracle of truck factors

To answer the research questions investigated in this paper, 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 in the AVL dataset (Avelino et al. 2016). In their work, Avelino et al. (2016) 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 the AVL algorithm. The developers answered the question based on their experience and knowledge about their projects, i.e., they did not use an algorithm to calculate TFs. Developers of 67 systems answered to these issues. To create our oracle, we carefully read these answers, aiming to retrieve TF results and developers. As the amount of answers is low, we did not apply any automation to retrieve information from these answers. The information were extracted from the answers of the developers manually. We were able to retrieve this information in two situations:

1. 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 value ("Yes. Let's just say we all hope here that neither of them get hit by a truck.").²
2. 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.").³

Out of 67 answers, 20 answers (29.9%) fit in the first described situation and seven 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 issues. All TF results included in the oracle were validated by the systems' principal developers. Therefore, although the basis for the survey are the results provided by AVL, the aforementioned procedure does not imply in a bias towards AVL. Indeed, as mentioned, seven results are different from the ones generated by AVL.

We also extended this initial oracle with new systems. For 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 the popularity of GitHub software (Borges et al. 2016; Borges and Valente 2018)) 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

²<https://github.com/netty/netty/issues/4069>

³<https://github.com/ipython/ipython/issues/8710>

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 main characteristics of the systems of our oracle, including the TF results according to the systems' main developers, are summarized in Table 2. The oracle includes well-known systems, such as junit-team/junit4, d3/d3, and ReactiveX/RxJava. The number of stars ranges from 2863 (nicolasgramlich/AndEngine) to 59,184 (d3/d3). The number of contributors range

Table 2 Oracle of TF results.

Systems in the first table's section are reused from Avelino et al. (2016). Age is measured in years

System	Language	Stars	Contributors	Age	TF
Alexreisner/geocoder	Ruby	4309	233	8	1
/androidannotations	Java	8781	58	6	2
Atom/atom-shell	C++	40,401	526	4	1
Bjorn/tiled	C++	4359	137	9	1
Celluloid/celluloid	Ruby	3457	90	6	1
Chef/chef	Ruby	4621	502	9	4
D3/d3	JavaScript	59,184	117	6	1
Dropwizard/metrics	Java	4562	131	7	1
Facebook/osquery	C++	7719	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	6756	119	5	1
Netty/netty	Java	8837	238	8	2
Nicolas.../AndEngine	Java	2863	21	7	1
Pallets/flask	Python	24,670	370	7	1
Powerline/powerline	Python	6892	84	4	1
Puphpet/puphpet	PHP	3729	147	4	1
ReactiveX/RxJava	Java	20,457	132	5	1
Requirejs/requirejs	JavaScript	10,121	98	7	1
Respect/Validation	PHP	3671	92	6	3
Saltstack/salt	Python	7265	1701	6	11
Sandstormio/capnproto	C++	4295	70	4	1
Sass/sass	Ruby	9176	179	10	1
SFTtech/openage	C++	5472	94	3	2
Thoughtbot/paperclip	Ruby	8300	348	9	1
Capistrano/capistrano	Ruby	9141	207	4	2
Deis/deis	Python	5936	163	4	3
Ruby-grape/grape	Ruby	7754	230	7	4
Cantino/huginn	Ruby	15,473	135	3	3
JUnit-team/junit4	Java	5602	133	16	4
Kennethreitz/requests	Python	22,874	440	6	3
Symfony/symfony	PHP	13,658	1339	7	15
Tornadoweb/tornado	Python	12,788	246	7	1

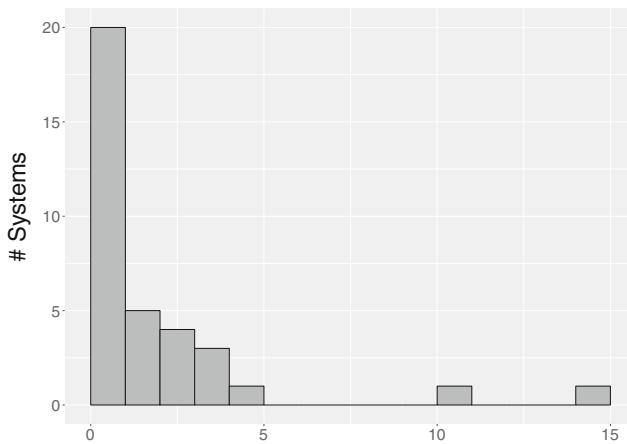


Fig. 2 Distribution of the system's Truck Factor

from 21 (nicolasgramlich/AndEngine) to 7265 (saltstack/salt) and the system's age ranges from 2 (facebook/osquery) to 16 years (junit-team/junit4).

Figure 2 shows a histogram of the distribution of the system's truck factor. As we can see, 20 systems (57.1%) have $TF=1$; and five 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 (Mockus et al. 2002).

3.2 Handling third-party libraries and aliases

Before running the truck factor algorithms over the commit history of the 35 systems evaluated in this work, we performed two cleaning steps. First, we used the open-source Linguist tool⁴ 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 GitHub repositories. The tool is implemented and evolved by GitHub with the main purpose of providing statistics on the amount of source code in a repository implemented in a 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.

3.3 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.⁵ 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.⁶ When answering RQ1 (How accurate are the results provided by each algorithm?) and RQ2 (How accurate is the identification of TF

⁴<https://github.com/github/linguist>

⁵<https://github.com/aserg-ufmg/Truck-Factor>

⁶<https://github.com/SOM-Research/busfactor>

developers by each algorithm?), we used the default configuration suggested by the algorithms' authors (including the thresholds reported in Section 2.1.1). In the case of CST, we used multiple changes equally considered as the metric to infer developer's knowledge. To compute the Core Developers estimated by commit-based heuristic and LOC-based heuristic, we implemented the algorithms based on the description of the heuristics presented by Yamashita et al. (2015).

When executing the algorithms over AVL's dataset, we used the commits available on GitHub in 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.

3.4 Survey with developers

In our third study, we aim to identify other factors, besides commits, that may influence the estimation of truck factors. To this purpose, we selected projects where the best algorithm for estimating truck factors has failed. As detailed in Section 4.1, AVL provides the best results when compared to the other algorithms evaluated in this work. We consider that the algorithm fails to estimate truck factors when its F -measure < 1 .

For each project with F -measure < 1 , we selected the false negatives produced by the algorithm, i.e., developers indicated as part of the truck factor in the oracle, but who are not identified by the best algorithm for truck factor estimation. We sent an e-mail to these developers, asking the following single question:

Could you please explain what other roles (not reflected in the number of commits) you played in the project that justified your inclusion among the Truck Factor developers?"

Additionally, this e-mail contained the project name, the truck factor of the system, and the position of the developer in the ranking of committers.

4 Comparison of algorithms for computing truck factor

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 (Cosentino et al. 2015; Avelino et al. 2016; Rigby et al. 2016). However, to the best of our knowledge, we still lack studies that (i) validate the results produced by such algorithms, and (ii) compare these results and discuss the main benefits and limitations of each algorithm. Therefore, in this section, we assess three recent algorithms proposed to estimate Truck Factors: AVL Algorithm, RIG Algorithm and CST Algorithm.

This study aims to answer the following research questions, as detailed in Section 1.

- RQ1. How accurate are the results provided by each algorithm?
- RQ2. How accurate is the identification of TF developers by each algorithm?
- RQ3. What is the impact of different thresholds and configurations in the results of each algorithm?

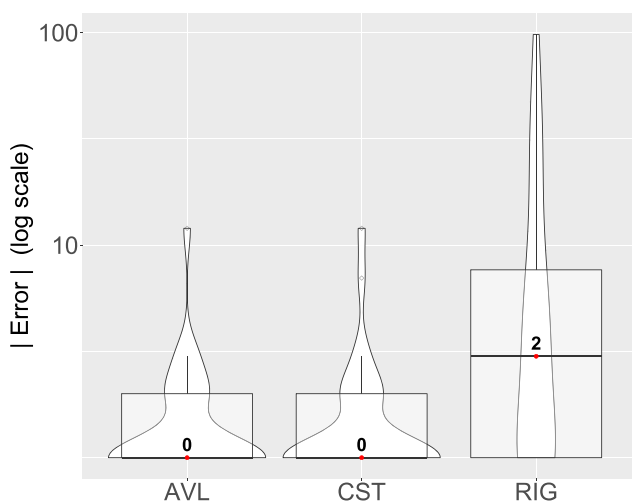


Fig. 3 Absolute error results

4.1 Results

RQ1. How accurate are the results provided by each algorithm?

To answer this first question, we compute the error of the truck factors estimated by each algorithm, compared with the oracle values, as follows: $TF_{\text{algorithm}} - TF_{\text{oracle}}$. Figure 3 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 3. 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 3.3). 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.

Table 4 shows how the algorithms perform for groups of systems with different TF results. The table divides the 35 systems in three groups: systems with $TF = 1$ (20 systems), systems with $2 \leq TF \leq 5$ (13 systems), and systems with $TF \geq 6$ (two systems). For each group, the table

Table 3 Error measures for outlier systems

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 4 Percentage of results with error = 0 per group of systems. Size is the number of systems in each group

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

presents the percentage of systems each algorithm was able to estimate their TF precisely, 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 the systems in TF1. In Table 4, 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.

RQ2. How accurate is the identification of TF developers by each algorithm?

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

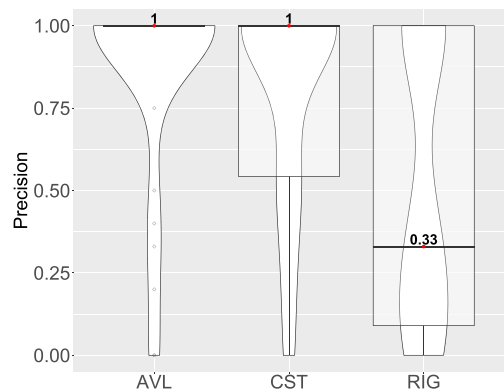
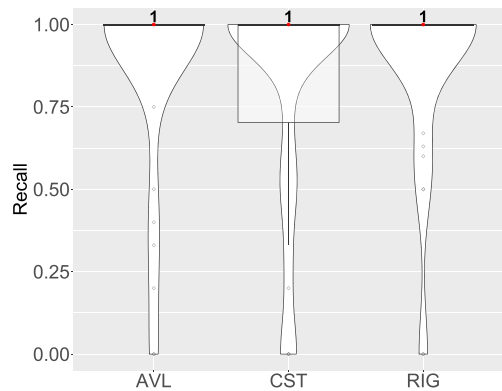
Fig. 4 Precision

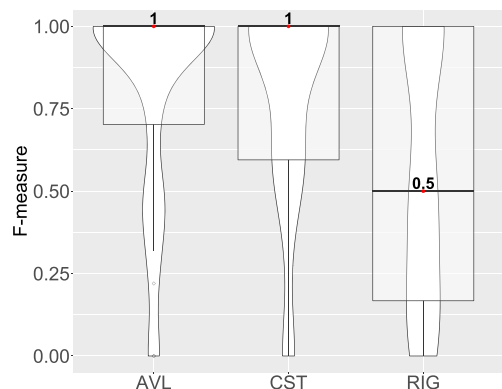
Fig. 5 Recall

each algorithm with the TF sets in the oracle. For 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 occurs when a developer indicated by the oracle is not in the algorithm's TF set.

Figure 4 shows violin plots with the results of precision for the three algorithms. 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.

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. Figure 5 shows violin plots with the results of recall. 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 first 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), defined as $2 * P * R / P + R$. As shown in Fig. 6, for AVL, the first, median, and third quartiles are 0.70, 1, and 1, respectively.

Fig. 6 F -measure

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 first quartile measures, AVL has the highest precision (1.00); CST has a lower precision (0.54), and RIG has the worst one (0.09). RIG and AVL are the algorithms with the highest recall (1). CST has the second best recall (0.70). Regarding *F*-measure, AVL is the best algorithm (0.70), followed by CST (0.60), and RIG is the less accurate algorithm (0.17).

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

AVL algorithm

The algorithm depends on a threshold that defines the rate of abandoned files that configures a truck factor disaster (see the algorithm in Section 2.1.1). AVL's authors suggest to set this threshold to 0.5, but they do not provide detailed evidence that this is indeed the best option. 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" Avelino et al. (2016).

Figure 7 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 $\text{error} = 0$ and the number of systems where $|\text{error}| \leq 1$, for each threshold. Therefore, the second curve tolerates a small error in the truck factor results. 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.

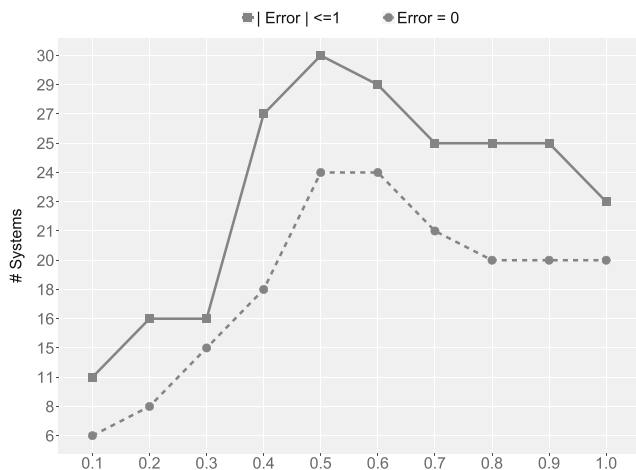


Fig. 7 Number of systems with $\text{error} = 0$ and $|\text{error}| \leq 1$, after varying AVL's threshold on abandoned files

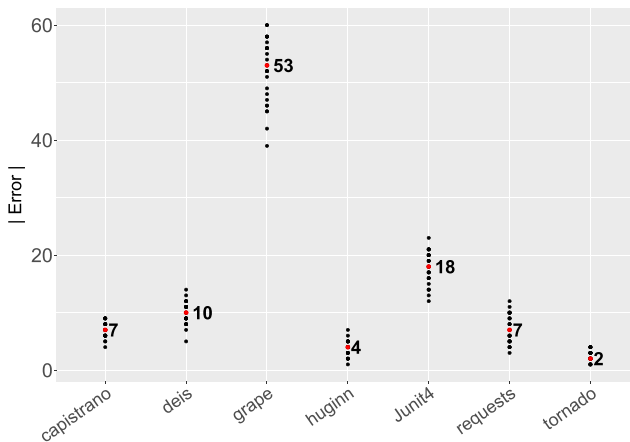


Fig. 8 Dispersion of RIG errors, considering 30 runs. The highlighted value (in red) is the median error

RIG algorithm

We reran RIG varying the number of random samples of developers tested by the algorithm. In its original configuration, RIG generates 1000 samples of developers (see Section 2.1.1, Algorithm 3, line 3). First, we increase this threshold to 2000 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 8 reports the absolute error of each run, for the oracle systems that are not reused from AVL's work. We restrict the figure to these systems for the sake of legibility.

RIG does not produce results for symfony/symfony (as in RQ1 and RQ2). The results in Fig. 8 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).

Figure 9 shows the results of varying the number of samples tested by RIG from 1000 (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 $|\text{error}| \leq 1$, 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 (1000 samples) to 17 systems (5000 samples). After this threshold, the curve remains constant.⁷

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

⁷We also ran experiments for 100,000 samples. After increasing the number of analyzed samples, the results presented by the RIG algorithm did not show significant improvements: 13 systems presented error = 0, and 21 systems have $|\text{error}| \leq 1$.

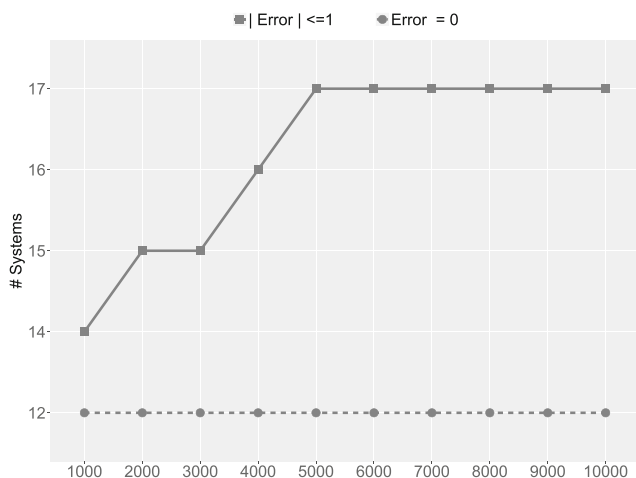


Fig. 9 Number of systems with error = 0 and $|error| \leq 1$, after varying RIG's threshold on the number of tested samples

CST algorithm

As described in Section 3, 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 the 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 5 shows the number of systems with error = 0 and error ≤ 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.

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.

5 Truck factor vs. core developers

Developers who contribute to open-source projects are usually categorized as core or peripheral developers. Core developers are those “who take a leading role in the development and mainte-

Table 5 Number of systems with error = 0 and error ≤ 1 , when CST is configured to use LCTA and MCEC metrics

Metric	error = 0	error ≤ 1
LCTA	18	25
MCEC	24	30

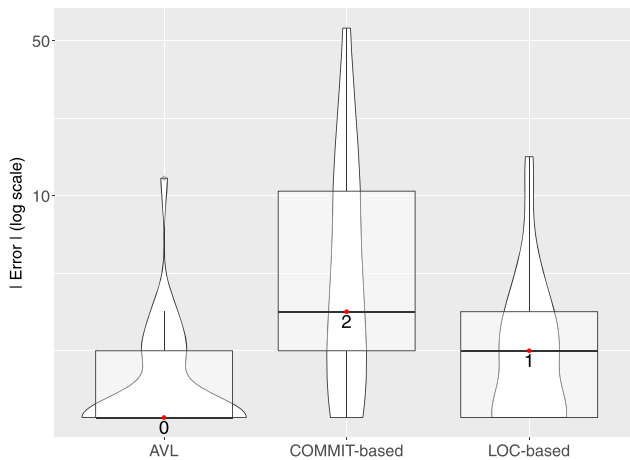


Fig. 10 Error results

nance of a software project” (Yamashita et al. 2015). In this section, we answer the following research questions:

- RQ4. How accurate are the results provided by heuristics for computing core developers when used to estimate TF developers?
- RQ5. How accurate is the identification of truck factor developers by heuristics originally proposed to identify core developers?
- RQ6. What is the relation between truck factor and core developers sets?

Only the results of the AVL algorithm are included in RQ4 and RQ5 because it was the algorithm with the best results in our first study.

RQ4. How accurate are the results provided by each heuristic?

To answer RQ4 and RQ5, we used the oracle of truck factors of 35 open-source systems (Section 3.1.2). First, we compute, for each of these systems, the core developers set according to commit-based heuristic and LOC-based heuristic. After obtaining these sets, we calculate the error, precision, recall, and *F*-measure of the heuristics in relation to the oracle of truck factors, as described in Section 4.1. The error metric ($|CD_{\text{heuristic}} - TF_{\text{oracle}}|$) is the absolute value of the difference between the number of developers indicated by the core developers (CD) heuristic and the number of truck factor developers present in the oracle.

Figure 10 shows violin plots with the absolute error measures. For the commit-based heuristic, the first quartile is 1, the median is 2, and the third quartile is 9.5. The largest error presented by this heuristic is 56. For the LOC-based heuristic, the first quartile, the median, and the third quartile are 0, 1, and 2, respectively. The maximum error presented by this heuristic is 14. For the AVL algorithm, the first quartile is 0, and the median and the third quartile are 1. The maximum error presented by AVL is 11. Therefore, LOC-based is the heuristic whose error distribution most closely resembles the AVL algorithm, presenting better results than a commit-based heuristic. However, AVL continues to be the most recommended algorithm for estimating truck factors, since it has the best accuracy.

Table 6 shows how the heuristics and the AVL algorithm perform for groups of systems with different TF results. As shown, AVL is the most accurate algorithm, identifying the TF correctly for all systems in TF1 group, 30% of systems in TF2-5, and 50% of systems in TF6+ group. The

Table 6 Percentage of results with error = 0 per group of systems. Size is the number of systems in each group

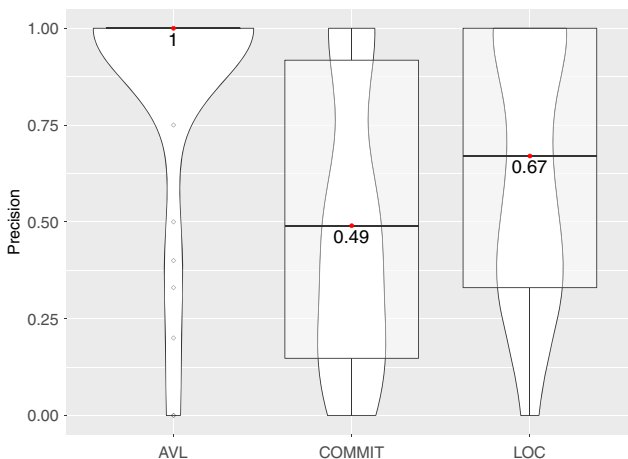
Group	Range	Size	Alg	%	
TF1	TF = 1	20	AVL	100	<div></div>
			COMMIT	17	<div></div>
			LOC	40	<div></div>
TF2-5	$2 \leq TF \leq 5$	13	AVL	30	<div></div>
			COMMIT	6	<div></div>
			LOC	3	<div></div>
TF6+	TF ≥ 6	2	AVL	50	<div></div>
			COMMIT	0	<div></div>
			LOC	0	<div></div>

LOC-based heuristic has the second best results, being able to identify the TF for 40% of the systems in TF1 group and 3% (i.e., only one system) in TF2-5. Commit-based heuristic has the worst results. For TF1, this heuristic identified the results correctly for only 17% of the systems, and for 6% in TF2-5. Both heuristics were not able to correctly identify the TF results for the systems with truck factor equal or greater than 6 (TF6+).

Summary: AVL algorithm has superior accuracy than both Core Developers heuristics, in all systems evaluated in this work.

RQ5. How accurate is the identification of TF developers by each heuristic?

Figure 11 shows violin plots with the precision results. The commit-based heuristic presents the worst results: the first quartile, the median, and the third quartile are 0.15, 0.49, and 0.9, respectively. For LOC-based heuristic, the first quartile is 0.33, the median is 0.67, and the third quartile is 1. Therefore, this heuristic presents slightly better results than commit-based. For AVL algorithm the first quartile, the median, and the third quartile are 1. Recall results are shown in

**Fig. 11** *F*-measure

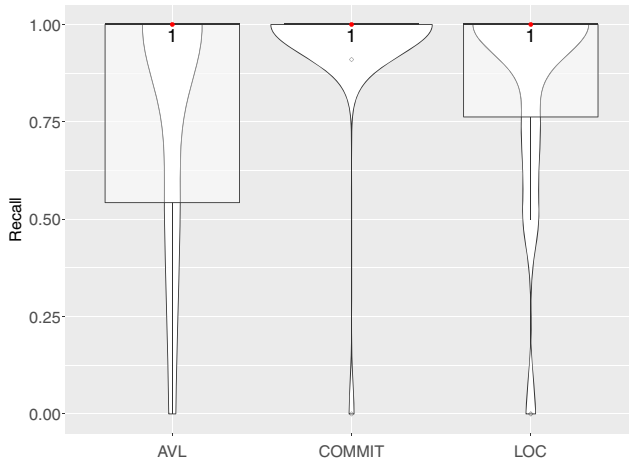


Fig. 12 Recall

Fig. 12. The first quartile of AVL algorithm, commit-based heuristic, and LOC-based heuristic are, respectively, 0.54, 1, and 0.76. The median and the third quartile are 1 for AVL, commit-based, and LOC-based heuristic. Finally, Fig. 13 shows the F -measure results. For the AVL algorithm, the first quartile is 0.70, the median and the third quartile are 1. For the commit-based heuristic, the first quartile is 0.54, the median is 0.68, and the third quartile is 0.9. The LOC-based heuristic presents better results than the previous one; its first quartile is 0.5, the median is 0.76, and the third quartile is 1.

Summary: Among the considered heuristics, LOC-Based has the best results. Furthermore, AVL and the studied heuristics present different results.

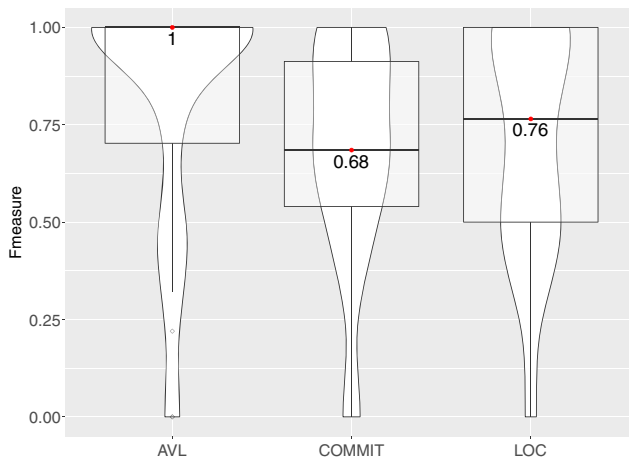


Fig. 13 F-measure

RQ6. What is the relation between truck factor and core developers sets?

The results of the investigation of RQ5 confirmed that truck factor and core developers are different concepts. This research question aims to better identify the relation between these concepts. To answer the question, we identify the true positives (TP), false positives (FP), and false negatives (FN) regarding the sets of truck factor developers from the oracle and core developers. A true positive occurs when a developer is both in the truck factor set and in the core developer set. A false positive occurs when the core developer heuristic indicates a developer that is not part of the truck factor set in the oracle. In contrast, a false negative indicates that the heuristic did not find a developer that is in the truck factor oracle. Thus, the relationship between the two sets can take the following forms:

- **True Positive > 0 (TP1+)**

In this case, the heuristic has developers in common with the truck factor set. This case occurs in the following situations:

- **FP = 0 and FN = 0**

The absence of false positives (FP0) and false negatives (FN0) indicates that the heuristic did not provide incorrect results, i.e., the core developer set and the truck factor set are equal.

- **FP = 0 and FN ≥ 1**

The presence of false negatives ($FN \geq 1$) indicates that the heuristic has failed to detect developers who are in the truck factor set. The absence of false positives ($FN = 0$) indicates that there is no overestimation of developers by the heuristic. These two situations indicate that the set of developers reported by the heuristic is contained in the truck factor set. That is, the set of core developers is a subset of the oracle of truck factor.

- **FP ≥ 1 and FN = 0**

The absence of false negatives ($FN = 0$) shows that the heuristic indicated all developers that are in the oracle. The presence of false positives ($FP \geq 1$) indicates that heuristics reported more developers than those in the truck factor set. In other words, the core developer set contains all developers who are in the truck factor set, but it is larger than the truck factor set. In this case, the truck factor is a subset of core developers.

- **FP ≥ 1 and FN ≥ 1**

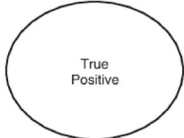
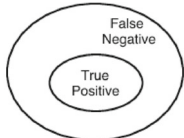
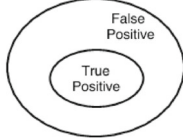
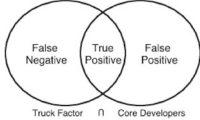
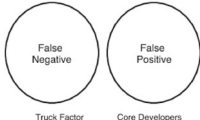
The presence of false positives ($FP \geq 1$) and false negatives ($FN \geq 1$) indicates that the heuristic has failed to identify truck factor developers and that it also introduced developers who are not part of this set. In this case, there is an intersection between the two sets.

- **True Positive = 0 (TP0)**

In this case, the heuristic does not report developers in common with the truck factor set. The absence of true positives (TP0) indicates that the sets are disjoint.

Table 7 presents the results for each of aforementioned cases. Among the 35 evaluated systems, there are no cases, for both heuristics, in which it was possible to observe the existence of core developers as a subset of truck factor (row $TP \geq 1, FP = 0, FN \geq 1$). The LOC-based heuristic presents the following results for the other cases: 13 systems are $FP = 0$ and $FN = 0$, 11 systems are $FP \geq 1$ and $FN = 0$, and eight systems are $FP \geq 1$ and $FN \geq 1$. These results indicate that in 37% of the systems the heuristic found exactly the same sets as the truck factor oracle; 31% of the truck factor sets are subsets of core developers; and in 23% of the evaluated systems there is an intersection between the two sets.

Table 7 Summary of intersection results

			Scenario	Heuristic results	
				Commit-based	LOC-based
	FP = 0	FN = 0	 Core Developers = Truck Factor	9	13
TP ≥ 1	FP = 0	FN ≥ 1	 Core Developers ⊂ Truck Factor	0	0
	FP ≥ 1	FN = 0	 Truck Factor ⊂ Core Developers	23	11
	FP ≥ 1	FN ≥ 1	 Truck Factor ∩ Core Developers	1	8
TP = 0	—	—	 Truck Factor Core Developers	2	3

The results obtained for the commit-based heuristic are significantly different from the ones of LOC-based heuristic. In 9 systems (26%), the commit-based heuristic showed that core developers and truck factor are the same set (FP = 0 and FN = 0). In only one system (3%), it was found a small intersection between the sets (FP ≥ 1 and FN ≥ 1). In 23 systems (66%), truck factor is a subset of core developers. The LOC-based and commit-based heuristics present completely different results from truck factor set (TP = 0) in only 2 (5%) and 3 (9%) cases, respectively.

Summary: In general, Truck Factor is a subset of Core Developers. In 94% of the systems evaluated by the Commit-Based Heuristic, the Truck Factor set is the same or it is contained in the Core Developers set. In other words, truck factor developers tend to be the “core of the core developers”.

6 Other factors impacting truck factor estimation

In this last study, we investigate the reasons of failures in TF algorithms, i.e., we provide answer to the following research question:

RQ7. Why algorithms for estimating truck factors fail?

By analyzing the projects in our oracle, we found 12 projects with TF developers having very few commits. Then, we conducted a second survey with these developers asking them about other roles they played in the projects. Particularly, we sent an e-mail to 17 contributors. We received seven responses (41%). These answers were analyzed by two authors of this paper in order to identify other factors—besides number of commits—that should be considered when computing truck factors. We identified five factors, as follows:

- *Social interaction (4 answers)*. These responses refer to contributors who said they were active in their projects through discussions in mailing lists, issues, Stack Overflow questions, blogs, or websites, as well as in the promotion of the projects in conferences and meetings.
- *Code Review (2 answers)*. These answers refer to contributors who claimed to be responsible for accepting and reviewing pull requests.
- *Documentation (2 answers)*. These responses are from developers who claimed to be a major contributor of the system documentation.
- *Tests (2 answers)*. Two developers mentioned they were responsible for testing the systems.
- *Supporting tools (1 answer)*. One contributor answered he worked on plugins, installers, and demo versions of the project.⁸

Summary: Besides number of commits, social interaction is another factor that might be considered when estimating Truck Factors. Code Review, documentation, tests, and supporting tools might also be considered by TF algorithms.

7 Complementary results and discussion

In this section, we provide further results about the three studies we carried out. We also discuss our results and the limitations of some algorithms.

7.1 Algorithms for estimating truck factor

7.1.1 What is the runtime performance of each algorithm?

We did not define 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 an HP Xeon six-Core server, with 64 GB RAM, and Ubuntu 12.04). However, this time does consider the checkout 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 data set is the most time-consuming task when using CST. It required, for example,

⁸Some answers mentioned more than one role, so the sum exceeds 100%.

around 8 h in the case of *symfony/symfony* and *saltstack/salt*, which are the two system with more contributors in the oracle.

7.1.2 Why and when RIG algorithm fails?

RIG algorithm is inspired by Zazworka's precursor algorithm to estimate truck factors. Zazworka's algorithm 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 tackle 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 2). 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 1000 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 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 2000, RIG started to estimate the truck factor of *saltstack/salt*.

7.1.3 Are truck factor scenarios realistic?

Rigby et al. (2016) 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.

7.2 Truck factor vs. core developers

Core developers designate the developers who play a critical role in software development. Therefore, this concept is directly related to the truck factor metric. In this study, we compared an algorithm for estimating truck factor with two heuristics for the detection of core developers. In particular, we investigated (i) whether heuristics for the detection of core developers are also capable of correctly identify truck factor developers; and (ii) whether truck factor developers are a subset of the core developers.

Among the evaluated heuristics, LOC-based heuristic is the one that presents the best accuracy in identifying the truck factor developers. When answering RQ4, it was possible to observe that the error of LOC-based heuristic is lower than the error of the commit-based one. In RQ5, we observe that LOC-based heuristic has better precision, lower recall, but better F -measure than commit-based heuristic. However, the heuristic for core developer detection that obtained the

best accuracy in the detection of truck factor was not able to present results as good as those presented by the AVL algorithm. This fact indicates that truck factor and core developers are different concepts. Understanding how these concepts are related was the goal of RQ6. Our findings indicate that, in most systems considered in the study, there is an intersection between truck factor developers and core developers. In fact, we could observe that in most cases the truck factor set is a subset of core developers.

7.3 Other critical factors for estimating truck factor

Considering commits that developers of a project perform is not enough to identify the most important contributors of a project. The results of our study to answer RQ7 have shown that other aspects, such as social interaction, pull requests, documentation, tests, and complementary tools should be considered to the truck factor estimation. Therefore, the algorithms proposed so far to calculate truck factor should be improved or new algorithms should be proposed to estimate truck factor properly, by considering these aspects. Nevertheless, it is worthwhile to notice that, even without considering these additional aspects, the algorithms proposed so far perform well.

7.4 Implications to practitioners

Open source projects have an increasing importance in modern software developers. Even commercial projects normally depend on a variety of open source software, including libraries, frameworks, development tools, web servers, and databases. On the other hand, the sustainability of such projects is a constant concern. Bus/truck factor is a key and simple metric to assess the risks of depending on an open source project. For example, if a library's TF is just one, project managers will certainly raise questions about its usage in a commercial and critical project. Therefore, TF algorithms have a key importance, because they automatically provide TF estimates for open source projects. To our knowledge, we are the first to compare and shed light on the precision and recall of such algorithms. Essentially, we concluded that AVL is the best algorithm.

8 Threats to validity

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. (2015). We did not find any public tool that implements RIG. Then, we implemented the algorithm, following the algorithm description presented by its authors carefully. 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.

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 constructing the oracle might be misleading. However, these approaches are complementary and, in both cases, the data are based on 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 to 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.

In order to identify other relevant factors to the truck factor's estimation, developers were asked which other roles, not reflected in the commits, they played in the rated system. The answers obtained were manually categorized and, then, they were subject of the personal interpretation of the person who classified them. In order to mitigate this threat, the classification of responses was performed separately by two of the authors of this article. The classifications were, then, compared to obtain the final classification. If there was a difference between the authors' classifications, they were again analyzed by both authors in order to obtain a consensus.

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 2863 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 1701 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.

9 Related work

The interest of Software Engineering community in studying knowledge of code is due to many reasons. As pointed by Fritz et al. (2014), it is hard to a developer to have knowledge about the software system, and the developers usually need to know who they should report their doubts about the code. Therefore, evaluating the distribution of the knowledge of a software system among its contributors is relevant both to developers and to managers.

Previous research on knowledge of code has been carried out with three main purposes: (i) to define models and metrics to assess the level of the developer's knowledge of the code, (ii) to define algorithms to calculate such metrics, and (iii) 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 lack studies that analyze and evaluate the behavior of the algorithms proposed to calculate truck factor. The present work aims to contribute to overcome this problem. As this work focuses on evaluating truck factor algorithms, we dedicated Section 2.1.1 to describe them. Therefore, in this section, we discuss other important related work.

9.1 Authorship and ownership

Besides truck factor, two other concepts are commonly associated with 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 class. Ownership refers to the level that a contributor is responsible (or owns) a piece of software (Bird et al. 2011; Foucault et al. 2014). Rahman and Devanbu (2011) 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. (2011) 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.

9.2 Developer categorization

Assessing how the knowledge of code is distributed among the developers is also important to identify the major and the minor contributors. The minor contributor has small contributions in the code, whereas the major contributor centralizes the knowledge of the code (Foucault et al. 2014). A correlated concept is core and peripheral developers in software projects. Core developers are those who 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. The 80th percentile threshold is, then, used to separate the developers. Developers with a number of commits above this threshold are core developers; the others are considered peripheral (Joblin et al. 2017).

In this work, we investigate the relation between core developers and truck factor by means of an empirical study. For this reason, the main heuristics for core developers have already been detailed in Section 2.2.

9.3 Experimental studies on source code knowledge

Rahman and Devanbu (2011) report the results of an experimental study about code ownership and defects. They investigated whether the quantity of developers that work on a file leads to more defects in software systems. Their study was carried out at a fine-grained level, considering lines of implicated code, i.e., lines that are modified to fix a bug. In the experimental study, they considered four open-source projects hosted on GitHub. Among their findings, they concluded that, in general, an implicated code is contributed by a single developer, contrasting with the idea that more developers working in a code lead to more defects. They also found that specialized knowledge in the code is more important than the general knowledge of the system to avoid defects.

Bird et al. (2011) also investigated the role of ownership in faults. They proposed a metric for ownership that is based on the concepts of minor and major contributors. A minor contributor of a module has ownership on it less than 5%; a major contributor has ownership of a module equals to or above 5%. They, then, define the ownership of a module as the “proportion of ownership for the contributor with the highest proportion of ownership”. They analyzed two large commercial projects, Windows Vista and Windows 7, which were developed by thousands of developers. In their analysis, they considered the Windows compiled binaries as modules. The main finding of their work is that the number of minor contributors of a module is positively correlated with the number of faults in the module. Foucault et al. (2014) replicated the study of Bird et al. (2011) in seven open-source software systems developed in Java. The classes of the systems were considered the modules in their study. In contrast to the results of the study of Bird et al. (2011) and Foucault et al. (2014) have not found a strong correlation between ownership and faults, and they concluded that the total number of developers is better than ownership to predict failures.

Using the DOA metric, Avelino et al. (2019b) studied code authorship in the Linux Kernel. They report that only a small portion of developers (26%) makes significant contributions to the kernel’s code; that 2% of the authors have authorship on hundreds of source code files; and that most Linux’s authors are specialists, i.e., they are authors of files in the same Linux subsystem. They replicate the same study to 118 popular GitHub projects and found similar code authorship results and patterns.

9.4 Experimental studies on truck factor

Some works have applied truck factor in experimental studies. Torchiano et al. (2011) proposed a theoretical model to infer the maximum possible truck factor value of a system. The model is based on variables such as the number of source files, the number of developers, and the number of missing developers, (i.e., the number of developers “hit with a truck”). They evaluated the model in 20 open-source software systems, with team size varying from 2 to 38 developers. Truck factor was calculated in their work by the algorithm proposed by Zazworka et al. (2010). They compared the results of their proposed model with another proposal referenced in their paper as Govindaray threshold. Although the problem of identifying a threshold for truck factor is important, the results of their study are not conclusive and, as the authors themselves recognize, further investigation is necessary. Moreover, their work is based on the algorithm of Zazworka et al. (2010) that has been proved to be applicable only to small projects.

After investigating 2496 projects hosted on GitHub, Yamashita et al. (2015) 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 (2003) mention that the development of GIMP (GNU Image Manipulation Program) was once halted for about 20 months because two developers have left the project.

10 Conclusion

In modern software projects, it is crucial to have reliable data on how source code knowledge is distributed among the team members. This information can be used to avoid “islands of knowledge” and to prevent the risks associated to the loss of key developers. The truck factor is a key measure to estimate such risks. In this paper, we assessed the distribution of knowledge in software projects through an in-depth analysis of the truck factor metric. To accomplish that, we carried out three empirical studies, whose key findings are presented next.

Study #1: Comparison of algorithms for computing the truck factors The first study was a comparative study of three algorithms to estimate truck factors. We built an oracle of truck factors, gathered via a survey with 35 open-source project teams. By comparing the accuracy, precision, recall, and *F*-measure results, we concluded that AVL is the algorithm that presents the best results for truck factor estimation. We also analyzed the algorithms when their thresholds are modified and we concluded that (i) the original threshold (50% of abandoned files) proposed by Avelino et al. (2016) has the best results in identifying truck factor developers, producing an accurate result in 68% of the evaluated systems; (ii) the CST Algorithm shows the best results when using the MCEC criteria (multiple changes equally considered), with error = 0 in 24 out of 35 systems; and (iii) RIG algorithm has a non-deterministic behavior, and increasing the number of tested samples does not have a positive impact in its accuracy.

Study #2: Truck factor vs. core developers The second study investigated the relationship between the truck factors and core developers. We considered two heuristics proposed by Yamashita et al. (2015) to infer core developers. We analyzed the accuracy of these heuristics regarding our oracle of truck factors. We found evidences that algorithms for estimating the truck factors and core developers present different results. By inspecting the detected true positives, false positives, and false negatives, we concluded that truck factor developers are in most cases a subset of core developers, i.e., they tend to be the “core of the core developers”.

Study #3 Other relevant factors for estimating truck factors In the final study, we investigated why the TF algorithms fail in some cases. We asked TF developers with few commits why they were ranked with this status. The most important reason relates to their social role in the projects, including participation in mailing listings, comments on issues and on Stack Overflow posts, for example.

Future work To conclude, we suggest the following themes for future work:

- *Replication of our work with proprietary software.* The studies presented in this paper were conducted using GitHub projects. It would be interesting to check out how the Truck Factor algorithms perform with proprietary software.
- *Consider a time window for the artifacts used by Truck Factor algorithms.* The algorithms evaluated in this work do not discard commits in stable files. In addition, they do not discard developers who are no longer contributing to the projects. Both aspects may have a negative impact on the algorithms results.
- *Consider social interaction on TF algorithms.* Commits are not the only way to gain knowledge on a project. Therefore, factors such as participation in mailing lists, issues discussions and similar social activities, might also contribute to improve the accuracy of TF algorithms.

Funding information This research is supported by grants from FAPEMIG, CAPES, and CNPq.

References

- Avelino, G., Passos, L., Hora, A., Valente, M.T. (2016). A novel approach for estimating truck factors. In: 24th international conference on program comprehension (ICPC), pp. 1–10.
- Avelino, G., Constantinou, E., Valente, M.T., Serebrenik, A. (2019a). On the abandonment and survival of open source projects: An empirical investigation. In: 13th international symposium on empirical software engineering and measurement (ESEM), pp. 1–11.
- Avelino, G., Passos, L., Hora, A., Valente, M.T. (2019b). Measuring and analyzing code authorship in 1+118 open source projects. *Science of Computer Programming*, 176(1), 14–32.
- Beck, K., & Andres, C. (2004). *Extreme programming explained: Embrace change*. Addison-Wesley Professional.
- Bird, C., Nagappan, N., Murphy, B., Gall, H., Devanbu, P. (2011). 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), pp. 4–14.
- Borges, H., & Valente, M.T. (2018). What's in a GitHub star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146, 112–129.
- Borges, H., Hora, A., Valente, M.T. (2016). Understanding the factors that impact the popularity of GitHub repositories. In: 32nd international conference on software maintenance and evolution (ICSME), pp. 334–344.
- Bowler, M. (2005). Truck factor. Online. <http://www.agileadvice.com/2005/05/15/agilemanagement/truck-factor/>, date Accessed: December 03 2016.
- Coelho, J., & Valente, M.T. (2017). Why modern open source projects fail. In: 25th international symposium on the foundations of software engineering (FSE), pp. 186–196.
- Cosentino, V., Izquierdo, J.L.C., Cabot, J. (2015). Assessing the bus factor of git repositories. In: 22nd international conference on software analysis, evolution, and reengineering (SANER), pp. 499–503.
- Ferreira, M., Valente, M.T., Ferreira, K. (2017). A comparison of three algorithms for computing truck factors. In: 25th international conference on program comprehension (ICPC), pp. 207–217.
- Foucault, M., Falleri, J., Blanc, X. (2014). Code ownership in open-source software. In: 18th international conference on evaluation and assessment in software engineering (EASE), pp. 1–9.

- Fritz, T., Ou, J., Murphy, G.C., Hill, E. (2010). A degree-of-knowledge model to capture source code familiarity. In: 32nd international conference on software engineering (ICSE), pp. 385–394.
- Fritz, T., Murphy, G.C., Murphy-Hill, E., Ou, J., Hill, E. (2014). Degree-of-knowledge: modeling a developer's knowledge of code. *ACM Transactions on Software Engineering and Methodology*, 23(2), 1–42.
- Joblin, M., Apel, S., Hunsen, C., Mauerer, W. (2017). Classifying developers into core and peripheral: An empirical study on count and network metrics. In: 39th international conference on software engineering (ICSE), pp. 1–12.
- Mens, T. (2016). An ecosystemic and socio-technical view on software maintenance and evolution. In: 32nd international conference on software maintenance and evolution (ICSME), pp. 1–8.
- Mockus, A. (2010). Organizational volatility and its effects on software defects. In: 18th ACM SIGSOFT international symposium on foundations of software engineering (FSE), pp. 117–126.
- Mockus, A., Fielding, R.T., Herbsleb, J.D. (2002). Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3), 309–346.
- Rahman, F., & Devanbu, P. (2011). Ownership, experience and defects: A fine-grained study of authorship. In: 33rd international conference on software engineering (ICSE), pp. 491–500.
- Ricca, F., & Marchetto, A. (2010). Are heroes common in FLOSS projects? In: 4th international symposium on empirical software engineering and measurement (ESEM), pp. 1–4.
- Ricca, F., Marchetto, A., Torchiano, M. (2011). On the difficulty of computing the truck factor. In: 12th product-focused software process improvement (PROFES), pp. 337–351.
- Rigby, P.C., Zhu, Y.C., Donadelli, S.M., Mockus, A. (2016). Quantifying and mitigating turnover-induced knowledge loss: case studies of Chrome and a project at Avaya. In: 38th International Conference on Software Engineering (ICSE), pp. 1006–1016.
- Torchiano, M., Ricca, F., Marchetto, A. (2011). 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), pp. 12–18.
- Williams, L., & Kessler, R. (2003). Pair programming illuminated. Addison Wesley.
- Yamashita, K., McIntosh, S., Kamei, Y., Hassan, A.E., Ubayashi, N. (2015). 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 (IWPE), pp. 46–55.
- Ye, Y., & Kishida, K. (2003). Toward an understanding of the motivation open source software developers. In: 25th international conference on software engineering (ICSE), pp. 419–429.
- Zazworka, N., Stapel, K., Knauss, E., Shull, F., Basili, V.R., Schneider, K. (2010). Are developers complying with the process: an XP study. In: 4th international symposium on empirical software engineering and measurement (ESEM), pp. 1–10.

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Mívia Ferreira received her Master degree in Computer Science from the Federal University of Minas Gerais, Brazil (2017) where she is a PhD candidate since 2017. Her research interests include software maintainability, evolvability, and quality.



Thaís Mombach is a M.Sc student at the Federal University of Minas Gerais, Brazil. Her research interests include software maintenance and evolution, software quality analysis, and optimization.



Marco Tulio Valente received his PhD degree in Computer Science from the Federal University of Minas Gerais, Brazil (2002), where he is an associate professor in the Computer Science Department, since 2010. His research interests include software architecture and modularity, software maintenance and evolution, and software quality analysis. Currently, he heads the Applied Software Engineering Research Group (ASERG), at DCC/UFMG.



Kecia Ferreira received her Ph.D. in Computer Science from Federal University of Minas Gerais (Brazil) in 2011. She is a professor of Computer Engineering at Federal Center for Technological Education of Minas Gerais. Her main research interest is on software measurement and its applications in software development and maintenance. Her most recent research has focused on software metrics, software evolution, and software maintenance.

Terms and Conditions

Springer Nature journal content, brought to you courtesy of Springer Nature Customer Service Center GmbH ("Springer Nature").

Springer Nature supports a reasonable amount of sharing of research papers by authors, subscribers and authorised users ("Users"), for small-scale personal, non-commercial use provided that all copyright, trade and service marks and other proprietary notices are maintained. By accessing, sharing, receiving or otherwise using the Springer Nature journal content you agree to these terms of use ("Terms"). For these purposes, Springer Nature considers academic use (by researchers and students) to be non-commercial.

These Terms are supplementary and will apply in addition to any applicable website terms and conditions, a relevant site licence or a personal subscription. These Terms will prevail over any conflict or ambiguity with regards to the relevant terms, a site licence or a personal subscription (to the extent of the conflict or ambiguity only). For Creative Commons-licensed articles, the terms of the Creative Commons license used will apply.

We collect and use personal data to provide access to the Springer Nature journal content. We may also use these personal data internally within ResearchGate and Springer Nature and as agreed share it, in an anonymised way, for purposes of tracking, analysis and reporting. We will not otherwise disclose your personal data outside the ResearchGate or the Springer Nature group of companies unless we have your permission as detailed in the Privacy Policy.

While Users may use the Springer Nature journal content for small scale, personal non-commercial use, it is important to note that Users may not:

1. use such content for the purpose of providing other users with access on a regular or large scale basis or as a means to circumvent access control;
2. use such content where to do so would be considered a criminal or statutory offence in any jurisdiction, or gives rise to civil liability, or is otherwise unlawful;
3. falsely or misleadingly imply or suggest endorsement, approval, sponsorship, or association unless explicitly agreed to by Springer Nature in writing;
4. use bots or other automated methods to access the content or redirect messages
5. override any security feature or exclusionary protocol; or
6. share the content in order to create substitute for Springer Nature products or services or a systematic database of Springer Nature journal content.

In line with the restriction against commercial use, Springer Nature does not permit the creation of a product or service that creates revenue, royalties, rent or income from our content or its inclusion as part of a paid for service or for other commercial gain. Springer Nature journal content cannot be used for inter-library loans and librarians may not upload Springer Nature journal content on a large scale into their, or any other, institutional repository.

These terms of use are reviewed regularly and may be amended at any time. Springer Nature is not obligated to publish any information or content on this website and may remove it or features or functionality at our sole discretion, at any time with or without notice. Springer Nature may revoke this licence to you at any time and remove access to any copies of the Springer Nature journal content which have been saved.

To the fullest extent permitted by law, Springer Nature makes no warranties, representations or guarantees to Users, either express or implied with respect to the Springer nature journal content and all parties disclaim and waive any implied warranties or warranties imposed by law, including merchantability or fitness for any particular purpose.

Please note that these rights do not automatically extend to content, data or other material published by Springer Nature that may be licensed from third parties.

If you would like to use or distribute our Springer Nature journal content to a wider audience or on a regular basis or in any other manner not expressly permitted by these Terms, please contact Springer Nature at

onlineservice@springernature.com