Who Can Maintain this Code?
Assessing the Effectiveness of Repository-Mining Techniques for Identifying Software Maintainers

Guilherme Avelino[1,2], Leonardo Passos[3], Fabio Petrillo[4], Marco Tulio Valente[1]

[1]Federal University of Minas Gerais, Brazil

[2]Federal University of Piaui, Brazil

[3]Quantstamp Technologies, Canada

[4]Polytechnique Montreal, Canada

{gaa, mtov}@dcc.ufmg.br, leo@quantstamp.com, fabio@petrillo.com

Abstract

In large and complex systems, identifying developers capable of maintaining a piece of code is an important but challenging task. In this context, repository-mining techniques can help by providing some level of automation. Still, whether such techniques effectively identify skilled software maintainers is yet unclear. To shed light on this issue, we evaluate three techniques supporting software maintainers recommendation, namely (1) the number of changes a developer makes; (2) the number of lines a developer owns in the last version of a file; and (3) a linear regression approach for defining experts. We apply these techniques against the evolution history of ten systems, contrasting recommendations with an oracle built from surveying developers. We concluded that practitioners should use the approach based on linear regressions, since it has the best performance after controlling for size/recency, closely followed by number of commits.

Keywords— Software maintenance, mining software repositories, source-code expertise

1 Introduction

When software needs to be fixed or improved, identifying who is able to maintain, assist, or review a particular piece of code can be a wicked task. Particularly, many software projects follow collective code ownership practices, as encouraged for example by agile software development methodologies. In these projects, it is common to have files with multiple contributors, sometimes reaching dozens of them. Among these contributors, some are responsible for the major changes in the files, while others can be considered as peripheral contributors, who perform only minor and less important maintenance tasks. However, usually, there is no clear and easily defined frontier separating peripheral from major contributors. This separation is important for example when a critical bug is reported and project managers have to rapidly identify an expert developer capable to fix it.

In this context, different repository-mining techniques can be applied to identify skilled software developers from the historical data kept by version control systems (VCS) [8, 9, 5]. By mining the development history, the goal is to infer developers able to maintain a file or a more specific piece of code. However, the extent that existing techniques successfully recommend software maintainers, as well as possible limitations, remains unclear. We shed light on this matter by evaluating three popular techniques: (1) the Number of Commits [3, 4]; (2) the Number of Lines of Code in the Last Version [6, 10] and; (3) the Degree of Authorship (DOA) [5]—a linear regression approach for defining experts. We apply all three against the evolution history of 10 systems (2 commercial and 8 open-source), comparing results with an oracle we build from surveying developers. We point out for the use of recency and file size information as a strategy to improve the effectiveness of the compared techniques. We also recommend the use of the DOA technique to identify maintainers, since it has the best performance, after controlling for size/recency, closely followed by *Commits*.

2 Evaluated Techniques

From the existing literature, we find three main techniques to recommend expertise from version control systems. The selected techniques depend on measures that are available on git-based version control systems or that are straightforward to compute. Moreover, although the selected techniques do not represent an exhaustive list of approaches to recommend software maintainers, they cover the key concepts adopted by most of them.

Number of Changes (*Commit*)

This technique counts the number of changes to define the experts on a file. The expertise of a developer d over a file f is defined by counting the number of commits performed by d in f. Bird et al. [3] and Casalnuovo et al. [4] use it to identify experts at the level of modules and methods, respectively.

Number of Lines of Code in the Last Version (*Blame*)

This technique relies on blame-like tools to obtain the developer who last modified a line in a file. It considers expertise as the percentage of the number of lines associated to a given developer. Girba et al. [6] consider a file expert the developer with more associated blame lines in a given system snapshot. Rahman and Devanbu [10] rely on this technique to assess expertise of developers responsible for defective code.

Degree of Authorship (DOA)

As proposed by Fritz et al. [5], this technique considers three events to compute the degree of authorship (DOA) of a developer d in a file f: first authorship (FA), number of deliveries (DL), and number of acceptances (AC). If d is the creator of f, *FA* is 1; otherwise it is 0; *DL* is the number of changes in f made by d; and AC is the number of changes in f made by other developers. The DOA measure is defined as follow:

$$DOA(d, f) = 3.293 + 1.098 * FA + 0.164 * DL - 0.321 * \ln(1 + AC)$$

In this equation, *FA* and *DL* contribute to increment the DOA value, the former with higher importance than the latter. In an opposite way, changes made by other developers (AC) decrease the DOA value. The weights used in this equation were empirically derived from a study with Java developers [5]. Although their use in other systems is as threat, the authors of the DOA metric claim the model is robust enough to be applied in different systems. Additionally, we previously used DOA to compute the truck factor of popular systems on GitHub, obtaining positive feedback from developers [1].

3 Study Design

3.1 Target Systems

We compare the described techniques in ten systems, including two commercial systems and eight open-source systems. The commercial systems (Commercial #1 and Commercial #2) are, respectively, a web platform for digital media and the client of a VoIP communication system. We omitted the names due to confidential reasons. They are developed by different teams and represent the main product of two companies, which are located in Brazil (Commercial #1) and Canada (Commercial #2). We also used the following open-source systems: Salt, Django, Moment, Ember.js, Faker, Monolog, Fog, and Puppet. These systems, which are implemented in four different programming languages, come from our previous truck factor study [1].

3.2 Oracle

Our investigation requires an oracle to compare the results produced by the techniques. We create this oracle by asking the system developers about their knowledge on a random set of files. The oracle construction comprises four steps:

Step 1: Extract development history. We extract the development history from the repositories using the `git log --no-merges --find-renames` command, which returns all no-merge commits of a repository and identifies possible file renames. We process each commit, extracting three pieces of information: (i) the file path; (ii) the developer who performed the change; and (iii) the type of the change—addition, modification, or rename. Rename information is used to join the development history of a file (old and new file names). Additionally, we discard files that do not contain source code (e.g., images, documentation) and third-party libraries. We also handle developers alias. To perform these tasks we follow the same automatic procedures described in previous works [1, 2].

Step 2: Generate survey sample. Given the development history of a system, we first discard developers with invalid e-mails and source code files that are touched by only one developer. The survey sample for a given system is generated executing the following procedure: (i) we randomly select a file and retrieve the list of developers who changed it; (ii) we discard the file if at least one of these developers reached the maximal limit of files ($files\_limit$); (iii) otherwise, we add the file to the list of each developer; (iv) we repeat the procedure until there are no more files to be verified. After consulting the commercial systems' managers, we decide to ask each developer on her knowledge about a list of at most 50 files ($files\_limit$). For the open-source systems, we set $files\_limit = 10$ to do not discourage the developers to answer the survey. This second step produces a list of pairs *(developer, file)* for each system in our dataset. In total, we generate a sample with 3,068 pairs, covering 1,109 files and 740 developers.

Step 3: Apply the survey. After producing the survey sample, we send an e-mail to each developer d asking him/her to assess his/her knowledge on each file f in the sample. The developers are invited to rank their knowledge using a scale from 1 (one) to 5 (five), where (1) means no knowledge about the file; and (5) means complete knowledge about the file. We also request them to explain or comment their answers in an optional text field. In total, we send 668 e-mails and received answers from 159 developers, resulting in a response rate of 24 %. Additionally, these answers correspond to 1,209 pairs (*developer*, *file*), covering 654 files.

Step 4: Process answers. Finally, we classify the answers in two disjointed sets: *declared maintainers* (O_M) and *declared non-maintainers* ($O_{\overline{M}}$). A *declared maintainer* is a developer who declared to have a knowledge greater than three in a file; otherwise, he is a *declared non-maintainer*. Most developers are *declared non-maintainers* (52 % and 54 %, for commercial and open-source systems, respectively). Therefore, although all the respondents had changed the files included in the study at least once, most of them answered they have limited knowledge on these files.

The data generated for this study is publicly available at: *https://github.com/gavelino/authorship-data*.

3.3 Inferring Maintainers and Non-maintainers

In this section, we describe how we use the techniques presented in Section 2 to classify the developers as *maintainers* or *non-maintainers* of a file. First, for a given file f we normalize the measures produced by each technique. We define that $\textit{expertise}(d, f)$ is 1 to the developer with the highest measure for f, otherwise it receives a proportional value. For example, if f_1 was modified by three developers d_1, d_2, and d_3 and they performed, respectively, five, four, and two commits, their normalized value using the *Commit* technique are $\textit{expertise}(d_1, f_1) = 1.0$, $\textit{expertise}(d_2, f_1) = \frac{4}{5} = 0.8$, and $\textit{expertise}(d_3, f_1) = \frac{2}{5} = 0.4$. A similar normalization happens with the *Blame* and DOA measures (i.e., the highest measure is normalized to 1; the other measures receive a proportional value in the range 0 to 1).

We consider the developer d of a file f as a *maintainer candidate* if she has an *expertise* greater than or equal to a threshold k; otherwise she is a *non-maintainer candidate*. For instance, by taking the previous example, if we adopt $k = 0.5$, d_1 and d_2 are classified as *maintainer candidate* of f_1, while d_3 is classified as *non-maintainer candidate*. For the sake of clarity and brevity, *maintainer* and *non-maintainer* candidates are just called *maintainer* and *non-maintainer* in the remainder of this article.

3.4 Evaluation Metrics

Although in the survey we obtained a response ratio greater than the one common in software engineering studies [7], our oracle is not complete, because not all developers who changed a file answered the survey. Therefore, it is not possible to compare the techniques using precision and recall because these measures require a complete ground truth, which can be used to answer whether any recommendation is correct or not. Instead, using the oracle data, we calculate the ratio of correct classifications produced by each technique (*Commit*, *Blame*, and DOA). Specifically, we compute the *maintainers hit ratio* (HR_M) of a given technique using the following equation:

$$HR_M(k) = \frac{|\{(d,f) \in O_M \mid \textit{expertise}(d,f) \geq k\}|}{|O_M|}$$

Therefore, $HR_M(k)$ is the ratio of *declared maintainers* (O_M) that a given technique correctly identifies. As described in Section 3.3, we consider that d is a *maintainer* of a file f if $\textit{expertise}(d,f) \geq k$. The *hit ratio of non-maintainers* ($HR_{\overline{M}}$) is computed using a similar approach, but using the set $O_{\overline{M}}$ and considering as *non-maintainers* the ones whose $\textit{expertise}(d,f) < k$, as follows:

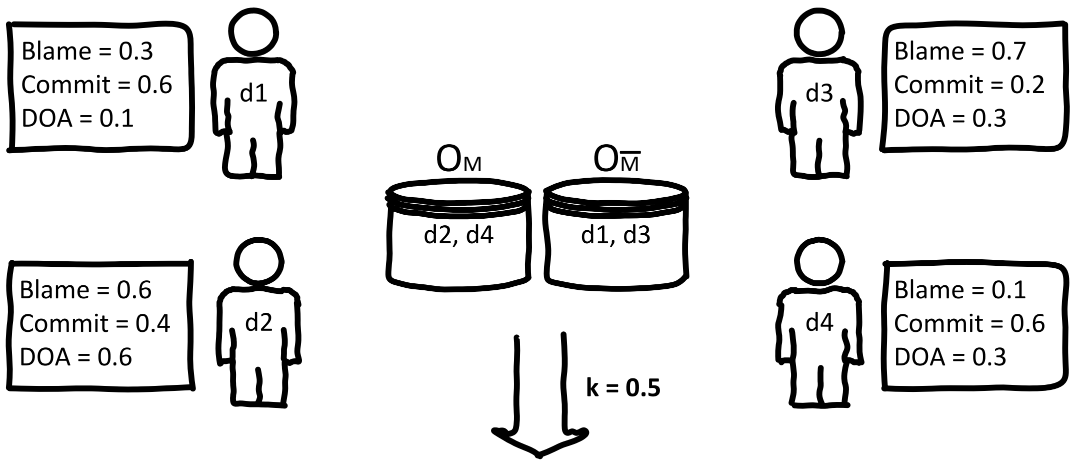
$$HR_{\overline{M}}(k) = \frac{|\{(d,f) \in O_{\overline{M}} \mid \textit{expertise}(d,f) < k\}|}{|O_{\overline{M}}|}$$

Both HR_M and $HR_{\overline{M}}$ are important to evaluate the results of the studied techniques. For example, a high HR_M but a low $HR_{\overline{M}}$ may indicate the technique is inflating the number of *maintainers*, erroneously classifying many developers with low knowledge as *maintainers*. Therefore, we also compute the *harmonic mean* (HM) of the hit ratios of *maintainers* and *non-maintainers*, as given by the following equation.

$$HM(k) = \frac{2 * HR_M(k) * HR_{\overline{M}}(k)}{HR_M(k) + HR_{\overline{M}}(k)}$$

We adopt an *harmonic mean* instead of the *arithmetic mean* because the former is less sensitive to outliers. For example, suppose that $HR_M = 90\,\%$ and $HR_{\overline{M}} = 20\,\%$. This is not an interesting result, because $HR_{\overline{M}}$ is very low. In this case, HM is 33 %, whereas the arithmetic mean is 55 %. Therefore, the *harmonic mean* better reflects and (and penalizes) the unbalanced nature of these HR_M and $HR_{\overline{M}}$ values.

To illustrate the use of these evaluation metrics, Figure 1 presents a hypothetical example. The figure shows developers d_1, d_2, d_3, and d_4 who changed a given file f and their *expertise* measures as computed by *Blame*, *Commit*, and DOA. We can also observe the sets of *declared maintainers* ($O_M = \{d_2, d_4\}$) and *declared non-maintainers* ($O_{\overline{M}} = \{d_1, d_3\}$) for f. In the bottom of the figure, we present the classification results produced by the proposed evaluation metrics, assuming $k = 0.5$. *Blame* correctly classifies one *maintainer* (d_2) and one *non-maintainer* (d_1). Since $|O_M| = |O_{\overline{M}}| = 2$, we have $HR_M = HR_{\overline{M}} = \frac{1}{2} = 0.50$,



TECHNIQUE	M	$\overline{M}$	HR_M	$HR_{\overline{M}}$	HM
Blame	**d2**, d3	**d1**, d4	0.50	0.50	0.50
Commit	d1, **d4**	d2, **d3**	0.50	0.50	0.50
DOA	**d2**	**d1**, **d3**, d4	0.50	1.00	0.67

Figure 1: Computing HR_M, $HR_{\overline{M}}$, and HM for a hypothetical file f. In the bottom part, underlined values denote developers correctly classified by the techniques as *maintainers* (M) and *non-maintainers* ($\overline{M}$) according to the oracles O_M and $O_{\overline{M}}$, and assuming a threshold $k = 0.5$.

and $HM = 0.50$ for *Blame*. The same results are obtained by *Commit*. DOA correctly classifies one *maintainer* (d_2) and two *non-maintainers* (d_1, d_3), obtaining the following results: $HR_M = \frac{1}{2} = 0.50$, $HR_{\overline{M}} = \frac{2}{2} = 1.00$ and $HM = 0.67$.

4 Results

To compare the techniques to infer *maintainers* and *non-maintainers*, we use the *harmonic mean* (HM), as described in the previous section. As this measure depends on a threshold k, we range k from 0 to 1, using steps of 0.1. By considering the results in Figure 2, we can identify the best technique for each group of systems (commercial and open-source). For commercial systems, *Blame* obtains the highest HM (67 %, $k = 0.1$), closely followed by *Commit* (66 %, $k = 0.2$), while DOA has the lowest one (63 %, $k = 0.6$). On the other hand, for open-source systems the best result is obtained by *Commit* (63 %, $k = 0.6$), closely followed by DOA (62 %, $k = 0.8$) and *Blame* (60 %, $k = 0.1$). In summary, although *Blame* and *Commit* provide slightly better results, respectively for commercial and open-source systems, the three techniques present similar performance on classifying developers as *maintainers* or *non-maintainers* (HM ranging from 60 % to 67 %).

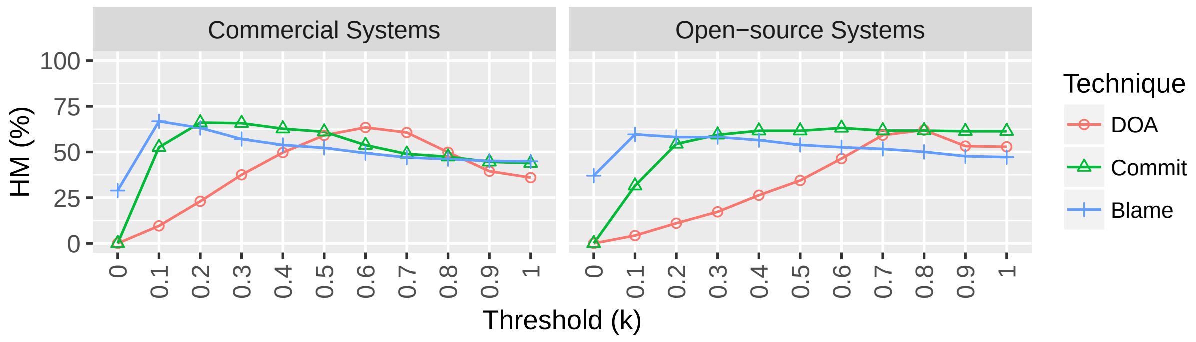


Figure 2: Harmonic mean (HM)

In Figure 2, we also observe that DOA is more susceptible to threshold variations, while *Commit* and *Blame* provide more stable results, specially when we consider $k \geq 0.2$. Additionally, *Commit* and mainly *Blame* achieve high HM measures ($> 50\,\%$) with a low k, while DOA requires $k > 0.5$ to produce values higher than $50\,\%$.

5 Discussion

In this section, we analyze examples in which the studied techniques fail and we also shed light on their limitations. Then, we evaluate the impact of using different thresholds to judge the technique results in scenarios where they are more likely to fail.

5.1 When and Why the Techniques Fail

We start by contrasting the cases where the three techniques succeed (*AllHit*) on identifying the *maintainers* of the studied systems against the cases where they all fail (*AllMiss*). To compute these cases we configured the techniques with their best thresholds, as pointed in the previous section. We identified 270 pairs (*developer*, *file*) in *AllHit* and 96 pairs in *AllMiss*. For each pair (*developer*, *file*) in *AllHit* or *AllMiss*, Figure 3 shows the distribution of the percentage of commits performed in the selected *file* by the respective *developer*. We can see that when all the techniques fail (first plot) the percentage of commits by the considered developers is usually lower than when they all succeed (second plot). Indeed, in most of the *AllMiss* cases the percentage of commits by the selected developers is low—75 % of the missed *maintainers* have less than 8 % of the files' commits. In other words, in the *AllMiss* cases the studied techniques failed because the developers classified themselves as *maintainers* despite having a small percentage of the files' commits. To clarify the failure reasons, we manually inspected the 96 pairs (*developer*, *file*) in *AllMiss*. We found three major reasons:

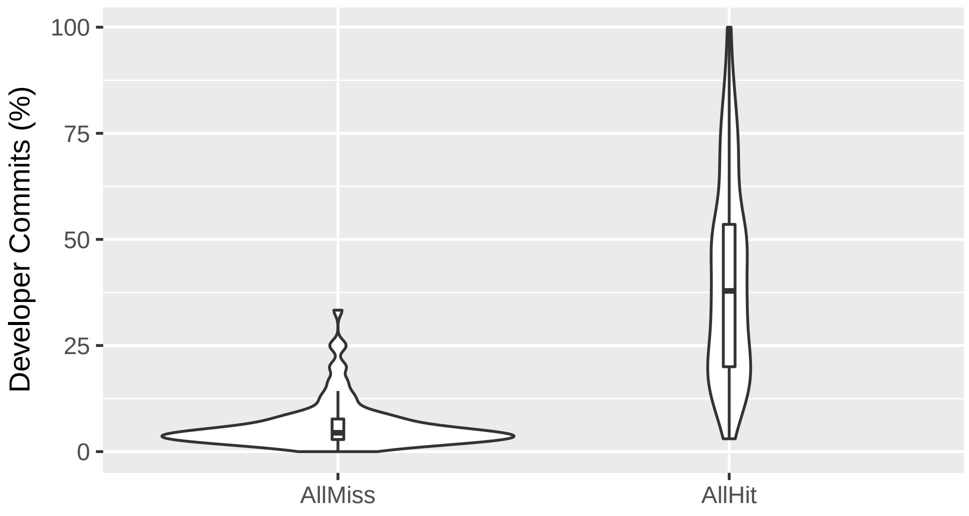


Figure 3: Distribution of the percentage of commits by a developer d who is a *declared maintainer* of a file f when all techniques failed to identify this condition (*AllMiss*) and when all techniques correctly identified this condition (*AllHit*).

- **Recency:** in 24 cases (25 %), there is less than one month that the developers modified the file. Although most of them had done few and minor contributions, they considered to have good knowledge on these files. Furthermore, *recency* influences other 11 cases (11 %) where the developer is the last one to change the file. However, the *recency* of the contributions is not caught by any of the studied techniques. For example, a concern about this question is mentioned in the following answer of a Django developer: *"I don't know if you are taking time into account, but I'd expect this to be a significant factor"*. We also found evidences that the familiarity with a file decays over time. For example, we received the following comment of a Puppet developer: *"I believe I submitted a patch about 3 years. At the time, I probably understood what I was doing but it's too long ago now"*.

- **File size:** the number of lines of a file also seems to influence the results. In fact, in 13 failures (14 %) the changed files are small (≤ 26 lines of code, which is the first quartile of the *file size* distribution in the entire oracle). This factor is mentioned by a Django developer to justify his answer: *"This is a tiny file, so having added one line of code, I've contributed a significant portion of it I guess"*. By contrast, a large file requires more effort to gain knowledge on its content, as exposed by another Django maintainer: *"On a relatively big piece (such as this 1K+ lines file) with a bunch of authors, I might be very knowledgeable about the piece of code I touched but know basically nothing about the rest"*.

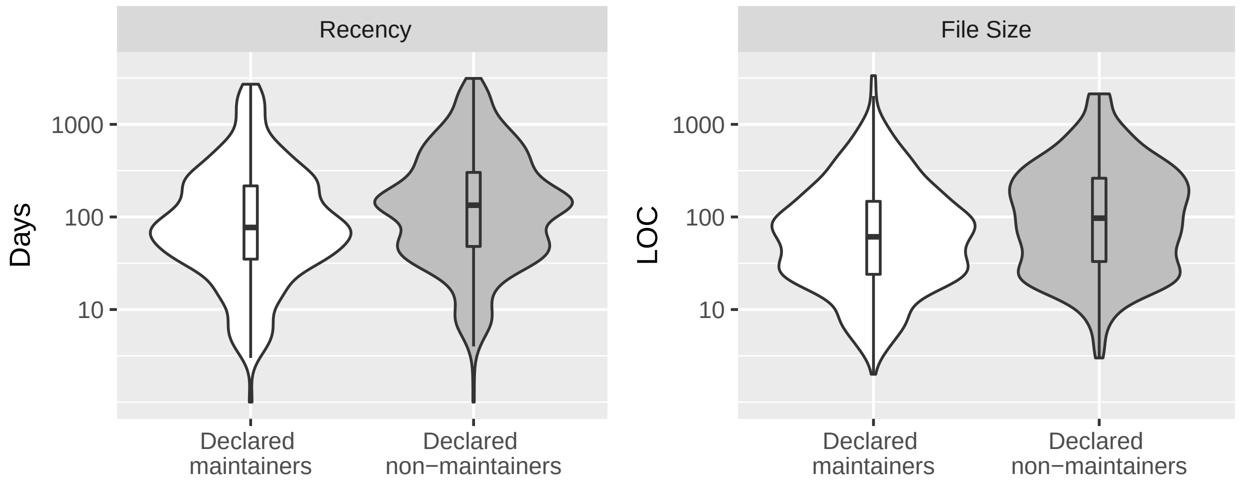


Figure 4: Distribution of *declared maintainers* and *declared non-maintainers* according to the following factors: *recency* (in days from the last developer commit) and *file size* (in LOC).

- **Extra-repository activities:** in the remaining cases (56 %), we could not find evidences from the collected data to support the failure results. For example, in Commercial #2, a developer rated his knowledge with a score four but performed only one commit, months ago, in a highly modified and large file (344 commits, by 25 different developers, 700 lines of code). Analyzing this case, we found that he recently worked in a new client-side module, and the file in question is a facade widely used by this new module. This suggests that activities not reflected in the commit history can also be a source of code knowledge. Additionally, two Commercial #1 developers mentioned their participation in the design of the system as the source of knowledge in specific files (e.g., *"this file represents a concept defined in the beginning of the project. The knowledge came from participating in the file definition"*).

To better evaluate the failure reasons identified in the manual investigation, Figure 4 presents the distribution of the pairs (*developer*, *file*) regarding *recency* (in days from the last commit) and *file size* (in LOCs). The entire oracle is represented, but we show separately *declared maintainers* and *declared non-maintainers* pairs. According to a Wilcoxon-Mann-Whitney test, the presented distributions are statistically different (*p-value* $< 10^{-8}$, in both cases). Regarding *recency*, the median number of days from the last commit is 71 days for *declared maintainers* and 134 days for *declared non-maintainers*. Regarding *file size*, the median size is 60 LOC for *declared maintainers* and 96.5 LOCs for *declared non-maintainers*.

This result reinforces that developers are more likely to declare themselves as *maintainers* when they recently modified a file or when this file is small.

5.2 Controlling for File Size and Recency

The results presented so far depend on a threshold to classify a developer d as a *maintainer* of a file *f*. Different thresholds are used for each technique; but for a given technique, the same threshold is used to classify developers as *maintainers* or *non-maintainers* of all files. However, as concluded in Section 5.1, the techniques tend to fail when the modified files are small (or large); and when the last modification by a developer in a file was performed recently (or a long time ago). Therefore, we decided to experiment different thresholds for the mentioned scenarios. Instead of having a single threshold, we adjust this value according to the following thresholds: small files (k_1), large files (k_2), files recently modified by the developer (k_3), files modified a long time ago (k_4); plus the previously used threshold for the remaining cases (k). We used the first quartile of the *file size* distribution in lines of code to classify the small files in a system; the last quartile is used to classify the large files. The first quartile of the number of days of the last commit by the developers of a system is used to classify the recently modified files; the last quartile classifies the files modified a long time ago. For each system, we experiment different values of k_i, ranging from 0.0 to 1.0 with steps of 0.05. The following test is then used to decide whether a developer d is a maintainer of a file f:

```
function isMaintainer (d, f)
    begin
        adj ← 1;
        if "f is a small file" then
            adj ← adj - $k_1$;
        else if "f is a large file" then
            adj ← adj + $k_2$;
        if "d recently modified f" then
            adj ← adj - $k_3$;
        else if "d modified f a long time" then
            adj ← adj + $k_4$;
        adjThreshold ← $k$ * adj;
        return expertise(d,f) ≥ adjThreshold;
    end
```

The improvements on the *HM* values (harmonic mean of the hit ratio of *maintainers* and *non-maintainers*) using these thresholds are as follows:

- For *Blame*, the *HM* improvements range from 0 % (Ember.js) to 14 % (Puppet).

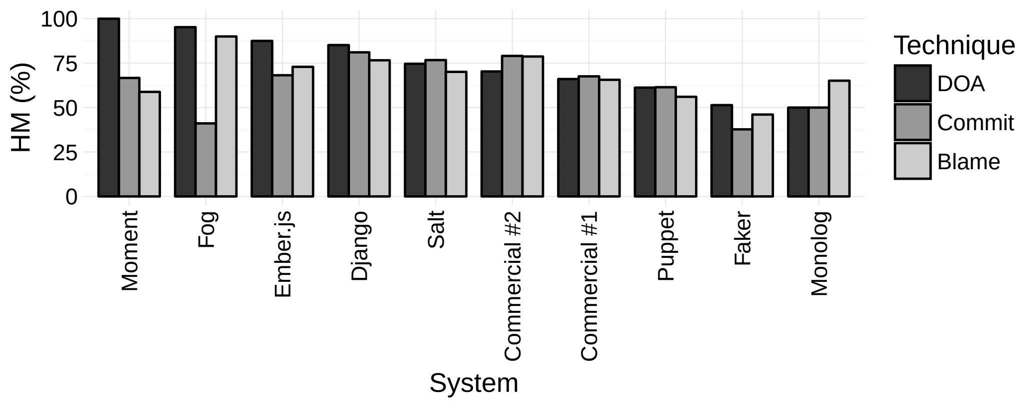


Figure 5: Harmonic mean (*HM*) of the hit ratio of *maintainers* and *non-maintainers* achieved when controlling for file size and recency

- For *Commit*, the improvements range from 0 % (Monolog, Salt, Moment, and Django) to 41 % (Fog).
- For DOA, the improvements range from 0 % (Monolog and Salt) to 95 % (Fog). Fog is the system with the highest improvement because most of the answers for the system refer to recently modified files.

Figure 5 shows the *HM* results of each system considering the proposed improvements. As we can see, DOA presents the highest gain after controlling for size and recency. It achieves the best *HM* values for five systems, followed by *Commit* (four systems) and *Blame* (two systems, one shared with *Commit*).

6 Threats to Validity

To construct the oracle we split the answers in two sets: *maintainers* ($score > 3$) and *non-maintainers* ($score \leq 3$). Although a score three may represent an acceptable knowledge, we followed a more conservative criterion, only classifying as *maintainers* the developers that informed a higher knowledge on the files. A second threat relates to the fact that some developers might provide unreliable answers. For example, developers might overestimate their scores aiming to obtain personal credits. To minimize this threat, we informed in the beginning of the survey that the study has not a commercial purpose and we also avoid to send a large number of questions to the developers. Finally, the study results are based on the data extracted from 10 real-world systems. Therefore, our findings may not generalize to other systems.

7 Conclusion

We summarize our major findings as follows:

1. When used without control for particular cases, DOA, *Commit*, and *Blame* have similar performance, with the harmonic mean (*HM*) of *maintainers* and *non-maintainers* hit ratios ranging from 60 % to 67 %.

2. However, when controlling for *file size* and *recency*, the improvements on *HM* are relevant. These improvements reach 95 % (DOA), 51 % (*Commit*), and 14 % (*Blame*).

3. After controlling for *file size* and *recency*, DOA presents the highest *HM* results in five systems, followed by *Commit* (four systems) and *Blame* (two systems).

Practical Implications: In many contexts, project managers have to identify possible maintainers for source code files. In this paper, we investigated three techniques for this purpose, based on data extracted from version control systems. As a first practical implication, we showed that practitioners with interest in these techniques should consider file size and recency information. As a second implication, we showed that practitioners should use the DOA technique to identify maintainers, since it has the best performance (after controlling for size/recency, closely followed by *Commits*).

Acknowledgments

Our research is supported by FAPEMIG and CNPq.

References

[1] Guilherme Avelino, Leonardo Passos, André Hora, and Marco Tulio Valente. A novel approach for estimating truck factors. In *24th International Conference on Program Comprehension (ICPC)*, pages 1–10, 2016.

[2] Guilherme Avelino, Leonardo Passos, Andre Hora, and Marco Tulio Valente. Assessing Code Authorship: The Case of the Linux Kernel. In *13th International Conference on Open Source Systems (OSS)*, pages 151–163, 2017.

[3] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code! In *19th International Symposium on Foundations of Software Engineering (FSE)*, pages 4–14, 2011.

[4] Casey Casalnuovo, Prem Devanbu, Abilio Oliveira, Vladimir Filkov, and Baishakhi Ray. Assert use in GitHub projects. In *37th International Conference on Software Engineering (ICSE)*, pages 755–766, 2015.

[5] Thomas Fritz, Gail C. Murphy, Emerson Murphy-Hill, Jingwen Ou, and Emily Hill. Degree-of-knowledge: modeling a developer's knowledge of code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(2):14:1–14:42, 2014.

[6] T. Girba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How developers drive software evolution. In *8th International Workshop on Principles of Software Evolution (IWPSE)*, pages 113–122, 2005.

[7] Barbara A. Kitchenham and Shari L. Pfleeger. Personal opinion surveys. In Forrest Shull, Janice Singer, and Dag I. K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, chapter 3, pages 63–92. 2008.

[8] David W. McDonald and Mark S. Ackerman. Expertise recommender: A flexible recommendation system and architecture. In *2000 Conference on Computer Supported Cooperative Work (CSCW)*, pages 231–240, 2000.

[9] Audris Mockus and James D. Herbsleb. Expertise browser: A Quantitative Approach to Identifying Expertise. In *24th International Conference on Software Engineering - (ICSE)*, pages 503–512, 2002.

[10] Foyzur Rahman and Premkumar Devanbu. Ownership, experience and defects. In *33rd International Conference on Software Engineering (ICSE)*, pages 491–500, 2011.

About the Authors

Guilherme Avelino is an assistant professor at the Federal University of Piaui and a Ph.D. candidate at the Federal University of Minas Gerais, Brazil. His research interests include software maintenance and evolution, software quality analysis, and programming languages. Contact him at *gaa@ufpi.edu.br*.

Leonardo Passos is a Software Engineer at Quantstamp Technologies. He received a Ph.D. in Computer Engineering from the University of Waterloo, in 2016. His main research interests include security auditing, automated bug finding, compilers, and software evolution.

Contact him at *leo@quantstamp.com*.

Fabio Petrillo is a Research Assoaciate at Concordia University (Ptidej Team). He received a Ph.D. degree in Computer Science in 2016 from Federal University of Rio Grande do Sul (Brazil). His research interests are mainly on debugging, software comprehension and evolution, software visualization, and empirical software engineering. Contact him at *fabio@petrillo.com*.

Marco Tulio Valente received his Ph.D. 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. Contact him at *mtov@dcc.ufmg.br*.