On the usefulness of ownership metrics in open-source software projects

Matthieu Foucault, Cédric Teyton, David Lo, Xavier Blanc, Jean-Rémy Falleri

On the Usefulness of Ownership Metrics in Open-Source Software Projects

Matthieu Foucault[a], Cédric Teyton[a], David Lo[b], Xavier Blanc[a], Jean-Rémy Falleri[a]

[a] *University of Bordeaux*
LaBRI, UMR 5800
F-33400, Talence, France
{mfoucaul,cteyton,xblanc,falleri}@labri.fr
[b] *School of Information Systems*
Singapore Management University
davidlo@smu.edu.sg

Abstract

Context: Code ownership metrics were recently defined in order to distinguish major and minor contributors of a software module, and to assess whether the ownership of such a module is strong or shared between developers. **Objective:** The relationship between these metrics and software quality was initially validated on proprietary software projects. Our objective in this paper is to evaluate such relationship in open-source software projects, and to compare these metrics to other code and process metrics. **Method:** On a newly crafted dataset of seven open-source software projects, we perform, using inferential statistics, an analysis of code ownership metrics and their relationship with software quality. **Results:** We confirm the existence of a relationship between code ownership and software quality, but the relative importance of ownership metrics in multiple linear regression models is low compared to metrics such as the number of lines of code, the number of modifications performed over the last release, or the number of developers of a module. **Conclusion:** Although we do find a relationship between code ownership and software quality, the added value of ownership metrics compared to other metrics is still to be proven.

Keywords: Software Engineering, Empirical Study, Process Metrics

1. Introduction

Process metrics, which measure developer's activity, were shown to have a strong relationship with software quality and, to be more useful than code metrics when it comes to defect prediction [1]. Among process metrics, the ones introduced by Bird et al. that measure code ownership (CO) are of a particular interest [2]. These metrics, called *CO metrics* in this paper, quantify the level to which developers own modules of a software project, by measuring the ratio of contributions they make to such modules. CO metrics split developers of a module into two distinct groups: major and minor developers, who perform more and less than 5% of the contributions, respectively.

The usefulness of these metrics was validated on Microsoft software projects, showing that they have a strong relationship with the number of bugs of a module, and that adding code ownership metrics to a regression model (with the number of bugs as the dependent variable) improves its quality [2]. Bird et al. also observed that the more minor developers contribute to a software module, the more bugs it contains. A possible explanation comes from the fact that minor developers have less knowledge of the modules they contribute to, and therefore may introduce more bugs. Moreover, Bird et al. also observed that for a given software module two other metrics are related to its number of bugs: the number of major developers, and the ratio of contributions preformed by the main developer of a module to the total amount of contributions on such module. Contrary to minor developers, major developers have more insight on the modules they contribute to, and therefore may introduce less bugs.

Such a finding has two main consequences. First, development team should be reorganized with the objective to increase code ownership by limiting the number of minor developers, or if it is not possible, to have major developers reviewing the contribu-

tions of the minor ones. Second, CO metrics should be used when predicting the number of bugs of software modules, as adding them to a model significantly improves its quality.

As these results were observed solely on two Microsoft projects, we therefore replicated the Bird et al. study but with open-source software systems [3]. However, our replication, made on seven open-source Java software projects, did not yield the same observations. In particular, we did not observe any significant correlations between the CO metrics and the number of post-release bugs. So far, our replication was not complete as we only observed Java open-source software projects.

We therefore propose in this paper a deeper study that goes further and that aims to generally question the usefulness of the CO metrics for open-source software systems.

First of all, to overcome the limitation of our previous study, we propose in this study a new dataset of open-source software projects developed in several programming languages. Another essential point strengthening the validity of our study is the technique used to collect bug-related information: based on previous research, we concluded that automatic techniques developed to measure the number of bugs per module are not accurate nor precise enough [4, 5, 6], and therefore relied only on manually crafted data.

Further, we push our investigation toward the relative importance of the CO metrics for estimating the number of bugs. Our previous study only tries to observe a correlation between CO metrics and number of bugs, and does not investigate on the importance of these metrics in a model accounting for several variables. In this study we check their relative importance as compared to metrics that are frequently used to measure the quality of a software module, using an automatic technique called PMVD [7], which evaluates the importance of each metric in a multiple linear regression model, with the number of bugs as the dependent variable.

In comparison to our previous study, we therefore propose the new following contributions:

- a completely new dataset that contains open-source projects developed in different programming languages, and manually crafted bug-related data.
- new results of correlation between CO metrics and post release bugs.
- an investigation on the relative importance of CO metrics

This paper is structured as follows: Section 2 presents the foundations of code ownership and the metrics related to it. Section 3 presents the detailed methodology of our study, including the construction of the dataset. Section 4 presents the main results of our study which shows that the usefulness of CO metrics is debatable in case of open-source software systems. Section 5 presents the threats to the validity of our study. Finally, Section 6 provides an overview of the related work and Section 7 concludes this paper.

2. Background and Theory

This section starts by presenting the code ownership (CO) metrics that have been defined to measure to which extent developers own software modules.

2.1. Ownership Metrics

Before explaining how CO metrics are measured, we need to define the model we use to represent a software development project and define the pertinent concepts, such as software module and developer contribution.

We assume that a software project is composed of a finite set of software modules that are developed by a finite set of developers who submit their code modifications by sending commits to a shared code repository.

Each module is defined by a finite set of source code files. When a developer modifies one of the files of a software module by committing her work, she is contributing to that module. The weight of the contribution made by a developer to a given module can be measured with different metrics. Bird et al. [2] chose to measure it by counting the number of files touched by the developer. For example, if Alice contributes to a module by modifying three files in a first commit and five files afterwards, she is contributing with a weight of eight. Another possibility is to measure the weight of a developer contribution by counting the number of line changes performed by the developer, also called code churn [8].

In our formal definitions, we use D as the set of developers that contribute to the project. For a given module, we define w_d as the weight of a developer d.

CO metrics mainly measure the ratio of contributions made by one developer compared to the rest of the team. More formally, for a given module, the ownership of a developer d is:

$$own_d = \frac{w_d}{\sum_{d' \in D} (w_{d'})}$$

Bird et al. [2] proposed three ownership-based metrics that are computed for each software module:

Most Valued Owner[1] This score is the highest value of the ratio of contributions performed by all developers. More formally, for a given software module, its *Most valued owner* value (MVO) is

$$\max(\{own_d \mid d \in D\})$$

Minor This score counts how many developers have a ratio of contributions that is lower than 5%. Such developers are considered to be minor contributors of the software module. More formally, for a given software module, its *Minor* value is

$$|\{0 < own_d \leq 5\% \mid d \in D\}|$$

Major This score counts how many developers have a ratio of contributions that is bigger than 5%. Such developers are considered to be major contributors of the software module. More formally, for a given software module, its *Major* value is

$$|\{own_d > 5\% \mid d \in D\}|$$

Bird et al. showed that varying the 5% threshold used by the metrics *Minor* and *Major* to other values from 2% to 10% did not impact the results they obtained regarding the relationship between code ownership and software quality.

2.2. Code Ownership and Software Quality

When the amount of developers of a software system rises, work must be divided between contributors. Whether a shared or strong ownership is preferable is a matter of debate where two theories come face to face. On the one hand the XP movement [9] and Raymond [10] advocate shared ownership, and the latter introduced "Linus' Law", which states that "given enough eyeballs, all bugs are shallow", i.e., increasing the number of contributors accelerates the detection and correction of bugs. On the other hand Bird et al [2]. advocate for a strong ownership, and aim to confirm the "too many cooks spoil the broth" theory stating that when the number of developers increases, coordination in the development efforts becomes too complex to ensure. Further, both theories are backed by empirical findings: Rahman and Devanbu [11], considered ownership at the level of individual lines of code, and found that code implicated in bugs was strongly associated to a single developer's contribution.

In this paper we focus on the second theory, and deeply investigate on the Bird et al. CO metrics. As these metrics were only validated on Microsoft software system, we here check their usefulness for open-source software systems. Bird et al. validate their claim by observing relationships between CO metrics and Software Quality, which is measured by counting the number of bugs. In particular, they base their claim on the following hypotheses:

Most Valued Owner This metric measures the highest percentage of contributions that a developer made to a software module. If the *MVO* of a software module is close to 100%, this means that one developer performed almost all the changes made to that module. If the *MVO* is low that means that the greater contributor makes few contributions and therefore that the module is shared between several developers that all perform few contributions. A high value therefore reveals that the software module has strong ownership while a low value reveals that it has shared ownership. Therefore, the impact of ownership on Software Quality is formulated by the following hypothesis:
H_{MVO}: The *MVO* metric is negatively correlated to the number of bugs.

Minor If there are lots of minor contributors, this implicitly means that many contributions are made by minor contributors and therefore the software module is shared between many developers. Work is thus fragmented between many developers with little knowledge of the module they are working on, and therefore overseeing all these contributions becomes an obsta-

[1] This metric originally called *ownership* has been renamed here for sake of clarity.

cle. Thus, the impact of ownership on Software Quality is formulated by the following hypothesis:
H_{minor}: The *Minor* metric is positively correlated to the number of bugs.

Major If there are lots of major contributors, this means that they all perform a significant amount of contributions and therefore that the software module has a shared ownership, and therefore that coordinating the work of developers is more difficult. Hence, the hypothesis regarding the *Major* metric is:
H_{Major}: The *Major* metric is positively correlated to the number of bugs.

Our objective is therefore to check these hypotheses for open source system but also to detect the relative importance of CO metrics, as it is clearly expressed in the next section.

3. Design of the methodology

We design our methodology around the two following research questions:

RQ1 Does code ownership, measured via the CO metrics *MVO*, *Minor*, and *Major*, have a relationship with software modules quality, measured with their number of post-release bugs?

RQ2 If so, do these metrics provide an added value (compared to other state-of-the-art metrics) for predicting the number of post-release bugs of a software module?

We propose to answering these two research questions using statistical inference on a dataset drawn from open-source software projects. This section then presents the methodology we designed to obtain and to measure such a dataset. It presents the corpus of software projects we used to perform our study, the approach we defined for identifying software modules within software projects, and how we compute the different metrics. The computation of the tests as well as their interpretation is presented in the Section 4.

To ease the replication of our own study, the artifacts we used and the data we measured is available online.[2]

3.1. Corpus of software projects

Performing our study requires a corpus of software projects with clearly identified software modules, contributions made by developers to the modules, and, for each module, the number of bugs it contains.

3.1.1. Reliability of Existing Datasets

Such a corpus is available in public datasets such as the PROMISE repository [12], on which we relied on in our previous study [3]. However, these datasets have two main issues regarding the design of our study.

First, they lack in clearly identifying authors of commits. Most projects included in these datasets use Subversion as a centralized version control system (VCS), which is not adequate for computing ownership metrics as Subversion does not make any distinction between the author and the committer of a change.[3] This is an issue in open-source projects hosted with centralized VCSs such as Subversion, as only the developers having "write" access to the repository appear as authors. Other developers thus contribute by sending patches, who are applied by the core developers of the project [13]. Although techniques to retrieve the submission and acceptance of such patches exist [14], this information is difficult to extract and much reliable than the one provided by decentralized VCSs such as Git.

Second, they lack in providing accurate bug-related information. The state-of-the-art technique used in these datasets consists in parsing the commit messages, looking for a bug identifier in the bugtracker (e.g. "Bug #42") [15, 16]. This technique assumes that developers reference bugs in commit messages, which is not always the case [4, 5]. Therefore, there are probably many bugs missing from the dataset. Moreover, Herzig et al. showed that a large proportion of issues available in bugtrackers are misclassified: many issues are classified as bugs although they are in fact features or improvements [6]. As a consequence, there is probably a substantial amount of false bugs are in such datasets.

3.1.2. Software Projects Selection Criteria

We therefore decide to build a new dataset dedicated to our study, and that covers these two is-

[2] `http://se.labri.fr/data/articles/IST-2014`
[3] `http://subversion.apache.org/`

Table 1: The FLOSS projects included in our dataset.

Project	Language	Release	Previous release	#Commits	#Modules	#Bugfixes	#LoC
Angular.js	JavaScript	1.0.0	0.10.0	783	26	147	11,041
Ansible	Python	1.5.0	1.4.0	1241	29	62	50,553
Jenkins	Java	1.509	1.480	1341	60	74	79,774
JQuery	Javascript	1.8.0	1.7.0	567	23	46	5,306
PHPUnit	PHP	3.6.0	3.5.0	500	16	46	11,885
Rails	Ruby	2.3.2	2.2.0	1072	46	390	33,919
Mono	C#	2.10.0	2.8.0	2800	184	351	1,777,719

sues. Regarding the first issue, our dataset must contain projects that use a VCS which is able to make the distinction between the author and the committer of a change. Regarding the second issue, the bug-related information should be as accurate as possible. In other words, only true bugs must be included within it (no false positive).

Identification of authors. To cover the first constraint we simply choose to rely on Git that natively distinguish authors from committers.[4]

Identification of Bugs. The second constraint is much more complex to address. The objective is to identify bug-fixing commits stored in a VCS with the intent to be as precise as possible. We assume that the number of bug-fixing commits is a fair representation of the actual number of bugs within a software module.

As we were not able to find an automatic approach which would not introduce a bias in our study, we decided to manually analyze commits to constitute our dataset. For instance, among the best automatic approaches, the one developed by Tian et al. has a precision of only 0.53 in the tested project (the Linux kernel), which in our case would increase the number of bug-fixing commits identified, and would be a bias to our study [17].

Our manual approach therefore aims to identify commits that are true bugfixes. We choose to focus on post-release bugfixes, as it is the case for the study of Bird et al. [2]. Identifying post-release bug fixes can be eased by the development process of a software project. In particular, in some projects, a maintenance branch is created for each release of a software. These maintenance branches differ from development branches in the fact that they usually do not contain new features. Further, the operations performed in such branches are usually bug-fixing, documentation, optimizations, or compatibility updates related to third party dependencies (e.g., the 2.3.x maintenance branch of Rails contains updates related to new versions of the Ruby programming language). Therefore, to ease our manual analysis, we choose to integrate in our dataset only software projects where the chosen release has a maintenance branch associated to it. Moreover, we restrict our search to maintenance branches where no commit was performed for the past six months, in order to have branches where most of the bugs were fixed.

Our definition of a bug-fixing commit includes any semantic changes to the source code which fixes an unwanted behavior. The type of bugs considered includes any arithmetic or logic bug (e.g., division by zero, infinite loops, etc.), resource bugs (e.g., null pointer exceptions, buffer overflows, etc.), multi-threading issues such as deadlocks or race conditions, interfacing bugs (e.g., wrong usage of a particular API, incorrect protocol implementation or assumptions of a particular platform, etc), security vulnerabilities, as well as misunderstood requirements and design flaws.

Practically, the identification of bug-fixing commits is performed manually, discarding commits where new features are implemented. We choose to ignore commits where performance optimizations are performed, as we consider performance issues as a different aspect of code quality. Moreover, we also ignore commits that resolve compatibility issues due to the evolution of a third-party dependency (although, we consider OS or hardware compatibility issues as bug), as these bugfixes are not due to the lack of quality of the changed code, but to the modification of an external requirement. Finally, it occurs that bug-fixing commits are lated

[4] `http://git-scm.com/`

discarded by the developers due to a regression introduced by the bugfix. In such cases, the developers perform a "revert" operation of such commits, and we ignore both the "revert" and the "reverted" commits.

We consider that bug-fixing commits are atomic, in the way that we do not consider the possibility that a bug-fixing commit may in fact include two bug-fixes. Moreover, if a bug-fixing commits affects two modules, the number of bug-fixing commits will be incremented in both modules.

3.1.3. Selected Corpus of Projects

According to these requirements, we manually select seven open-source projects. These projects, summarized in Table 1, are written in six different programming languages: JavaScript, Java, PHP, Python, Ruby and C#. The developers contributions we considered are the ones performed between the "Release" and the "Previous Release" shown in the table[5]. The "#Commits" column in the table corresponds to the number of commits performed between the two releases.

The main criterion for the choice of these releases, besides the availability of a maintenance branch, is the fact that Git was indeed the VCS used when they were developed, as in many of the projects selected here, the older part of the development history was done with Subversion, and then imported to Git. The selected releases are minor releases (i.e., no breaking changes have been performed in the selected development period) in Ansible, JQuery, PHPUnit and Rails. They are major release in Angular.JS, Jenkins and Mono.

The selected releases are, with the exception of the one in Ansible, considered to be long term supported (LTS) releases. For these LTS releases, bugfixing commits are backported from the main development branch even after these new releases are available. In Ansible, although the maintenance of the 1.5.x releases stopped a couple of week before the availability of the 1.6.0 release, it was performed simultaneously with the development of the 1.6.0 release.

3.2. Modules Definition

The metrics used in our study all target software modules. In order to perform our analysis we therefore have to decompose each project in a finite set of software modules, which is well known to be a hard task that requires some subjective choices [18].

We therefore chose to use a manual process that aims to decompose a project into a finite set of software modules. We asked three members of our research group to provide, for each of the six projects in the corpus, a list of software modules. After comparing the resulting decompositions, we ended up with two distinct lists of modules (two members returned quite the same list), one slightly finer grained than the other. The results provided in this paper are the ones obtained with the finer decomposition of modules, which is the one generated by two members. As the results obtained with the coarser decomposition are similar, they are available in the appendix of this paper.

The manual process used by the three participants is a quite simple approach that looks at the directory tree of a given project. We consider that a software module is either a file or a directory, with the possibility to include or not its subdirectories.

3.3. Metrics Computation

To reach our objective, which is to state whether the code ownership has an influence on the software quality, we need to measure CO metrics, as well as other process and code metrics, such as the number of lines of code (LoC) of a module, its number of touches, its code churn, and the total number of developers who edited such a module.

As defined in Section 2, CO metrics require to weigh the contributions of the developers. For the sake of completeness, we chose to compute two alternatives of such weight. The first alternative consists in counting the number files touched by a developer (*Touch*). The second alternative consists in counting the number of lines changed by a developer (*Churn*). Section 4 only present the numbers using the *Touch* to compute the weight, and the results computed using *Churn*, which are similar, are available in the appendix.

4. Analysis

This section presents the results of our analysis, and then answer the two research questions that aim to question the usefulness of code ownership metrics.

[5] In the case of Mono, the 2.10.0 release in on a different branch than the 2.8.0, so we considered the commits since the common ancestor of these two releases instead.

Table 2: Spearman correlation coefficients between metrics and the number of post-release bugfixes.

	Spearman Correlation						
Project	**LoC**	**NumDevs**	**Touches**	**Churn**	**MVO**	**Major**	**Minor**
Angular	**0.56 ****	**0.77 *****	**0.78 *****	**0.71 *****	-0.46 *	**0.68 *****	**0.63 *****
Ansible	**0.77 *****	**0.79 *****	**0.84 *****	**0.81 *****	-0.25	0.16	**0.8 *****
Jenkins	**0.72 *****	**0.73 *****	**0.67 *****	**0.58 *****	**-0.54 *****	**0.63 *****	**0.64 *****
JQuery	**0.79 *****	**0.85 *****	**0.83 *****	**0.87 *****	**-0.52 ****	**0.64 *****	**0.71 *****
Rails	**0.71 *****	**0.76 *****	**0.69 *****	**0.68 *****	**-0.62 *****	**0.62 *****	**0.63 *****
PHPUnit	**0.8 *****	0.42	**0.61 ***	**0.52 ***	-0.02	0.04	**0.79 *****
Mono	**0.5 *****	0.47 ***	0.42 ***	0.37 ***	-0.38 ***	0.34 **	**0.56 *****

p-value: 0 *** 0.001 ** 0.01 * 0.05 . 0.1 ' ' 1

4.1. RQ1: Is there any relationship between Code Ownership and Software Quality?

To answer this question, we perform a correlation test (Spearman) between code ownership metrics and the number of post-release bug-fixing commits for each of the project of our corpus. The results of the correlation tests are summarized in Table 2.

The correlations we obtain clearly exhibit the expected effects of code ownership as we describe in Section 2. There is a negative correlation between MVO and the number of bugs, confirming the theory that modules with stronger ownership have less bugs. There are positive correlations between both Minor and Major and the number of bugs, meaning that the more minor and major owners, the more bugs.

Based on these results, we can conclude that there is a significant relationship between CO metrics and the number of bugs. However, as in our previous study [3], other metrics are also correlated with the number of bugs, without having a single metric that outperforms the others. The fact that Minor is not the metric with the strongest correlation is a difference with the results obtained by Bird et al. [2].

4.2. RQ2: What is the importance of Code Ownership metrics for predicting bugs?

Although CO metrics have a relationship with the number of bugs, other metrics exhibit such a relationship. Our intent here is to determine if CO metrics provide an added valued compared to other metrics, and to find whether splitting developers into major and minor contributors is useful or not.

To answer this question, we propose to use multiple linear regression with the objective to measure the importance of each metric regarding the estimation of the number of bugs.

Using multiple linear regression, one can assess the relative importance of metrics, by evaluating how the R^2 of the regression model improves when adding each variable to the model. The R^2 measures the proportion of variation in the dependent variable (i.e., in our case the number of bugs) explained by the regressors (i.e., the software metrics) in the model. However, this technique is effective only when the regressors are uncorrelated, which is not the case with software metrics. When metrics are correlated, the order in which the regressors are added to the model may have a strong impact on the R^2 added to the model, which would be misleading regarding the importance of the metrics.

To assess the relative importance of regressors in a multiple linear regression model, we use the PMVD technique, developed by Feldman [7], and implemented in the R package `relimpo` [19]. To overcome the issue of the ordering of regressors, PMVD automatically computes all possible permutations of the regressors, and performs an average of the R^2 of each regressor over all possible orderings.

To answer our second research question we then run PMVD for each of the project by considering not only code ownership metrics as regressors, but also the *LoC*, *Touch*, *Churn* and *NumDevs* metrics.

Before applying the PMVD technique, we observed that *NumDevs* and *Minor* are collinear. These two metrics have a Pearson correlation coefficient ranging from 0.896 to 0.996, depending on the project. Two metrics with such a high degree of collinearity can be considered identical, which means that, with the current dataset, *Minor* is redundant with the simple metric that is the number of developers.

We still need to determine the relative importance of *MVO* and *Major* compared to the simple metrics that are *LoC*, *Touches*, *Churn* and *NumDevs*. Therefore, we build a multiple linear regression model including the aforementioned metrics as the regressors, and the number of bug-fixing commits as the dependent variable. For each project, we use PMVD to decompose the R^2 of the model into non-negative contributions that sum to the total R^2. Figure 1 presents the result of PMVD for each project, where each bar shows the contribution of each metric to the total R^2 of the model.

In all seven projects, the best metric (in terms of relative importance) is either *LoC*, *Touches*, or *NumDevs*. *Churn* has a relatively small contribution to the total R^2 of the models in six out of seven projects, whereas *MVO* and *Major* have even smaller contributions, with negligible contributions in five and four out of seven projects, respectively.

Based on these observations, the benefits of computing code ownership metrics is highly debatable in open-source software projects. Although our previous findings [3] were relatively moderate — we observed that code ownership metrics were not better than simple metrics — the conclusions of this experiment seem much more radical: *Minor* is redundant with the number of developers, and cases where *MVO* and *Major* contribute significantly to the precision of multiple linear regression models seem incidental.

5. Threats to validity

In this section we cover the different factors that may affect the validity of our study. We emphasize on three main categories of threats to validity: internal, construct and external validity.

5.1. Internal Validity

The internal validity of our study can be threatened by confounding factors, i.e. additional variables that may explain our results, or the differences between them and the results obtained by Bird et al. [2]. In this section we uncover possible confounding factors.

5.1.1. Minor and Major Contributors

In the Windows projects, most developers were major contributors of at least one module, and few developers were exclusively minor contributors [2]. Open-source projects follow a different model, with a part of the contributors being the core developers of the project, and another part being incidental contributors, who perform a small amount of contributions. In the projects of our corpus, the proportion of developers being only minor contributors vary between 50% and 79%. Ownership metrics, as defined in this paper, do not make the difference between minor contributors who belong to the core team of developers, i.e., who are major contributors of another module, and developers who are only minor contributors. We defined two simple metrics to take into account this difference, which are the number of minor contributors who are also major contributors of another module, and the number of developers who are only minor contributors. However, correlations between these metrics and the number of bug-fixing commits are either not statistically significant or have a smaller effect size than existing ownership metrics.

5.1.2. Volunteers and Paid Contributors

Another difference between developers is the fact that many open-source projects are *industry-led* or *industry-involved* [20], meaning that some developers are being paid to contribute to the projects, while others are volunteers. This could be a confounding factor as the motivations of both categories of developers are different, and may impact the quality of the code they produce. Although we do not compare in this paper the quality of the code performed by paid and volunteer developers, we do have some insight of the proportion of developers in each category, based on public information we could retrieve from social media and developers and projects websites (i.e. GitHub, Twitter, LinkedIn). First, all the projects in our corpus are at least *industry-involved*, and projects such as Mono (Novell/Xamarin) and Ansible (Ansible, inc.) are clearly *industry-led*. Second, for the developers with the most commits in each project (10 commits or more in the studied period) we sought evidences of employment by one of the companies involved in the project: in all the projects with the exception of PHP Unit where the involvement of companies seems to be less strong, most developers who performed more than 10 commits were paid by a company involved in the project.

5.1.3. Code Ownership Guidelines

Each project defines its development guidelines, which indicate to developers the rules to follow when contributing. These rules include the patch

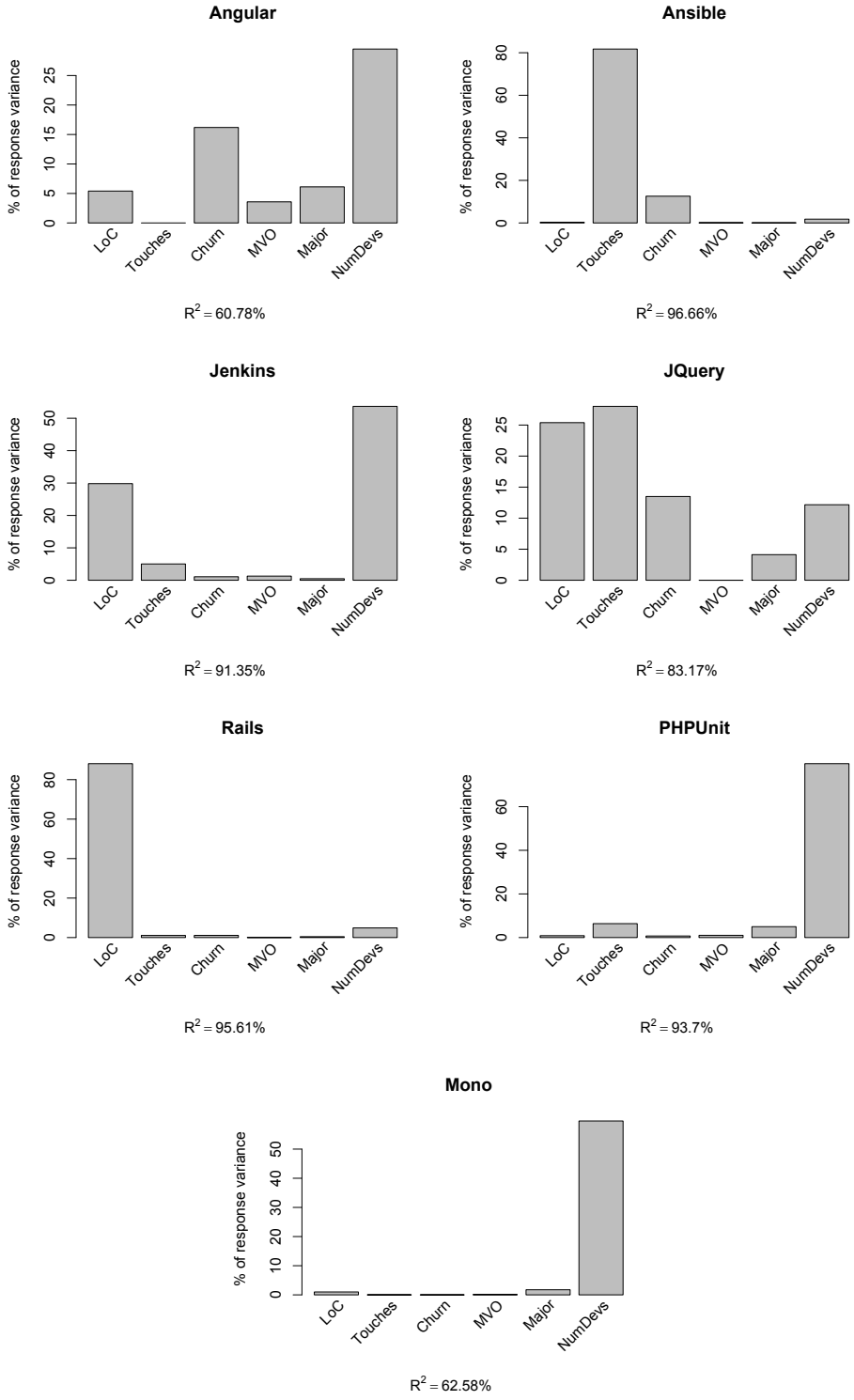

Figure 1: Relative importance of metrics in regression models with the number of bugfixes as the dependent variable.

submission process, code style, etc. As pointed out by Mockus et al. [21], development guidelines may include indications relative to code ownership, which can be enforced. In that case, module owners are defined by the project core team, and have more responsibilities than other developers, such as reviewing patch submissions, fielding bug reports, etc.

We sought for such guidelines in our dataset. In most projects, there was no mention of code ownership in the guidelines. In Jenkins, development guidelines advocate shared ownership, and there is no assigned tasks to core developers: the Jenkins governance document states that "Core committers generally use their own judgment to decide what to work on". In Mono, the guidelines are more in favor of strong ownership, and state that the author of a piece of source code automatically becomes its owner, and that further modifications to this source code must be discussed with such owner. However, we did not find strong ownership enforcement such as the one described by Mockus et al. [21].

5.2. Construct Validity

The construct validity of a study refers to whether the measurements performed are consistent with the theory. We now reveal the threats encountered in our empirical study.

With Git, developers can submit pull requests, so that the project leaders, who have written permission on the repository, can add their contributions to the project. As the identity of the initial author is maintained through the *pull* operation, she is identifiable even though she does not have access to the main repository. However, it may happen that a developer exchanged with a pull request author to agree on its acceptance. Even though this developer spent time to fix or improve the pull request content, all the credits will go to the pull request author. This may also introduce a bias in the results.

The modules listed from the selected project release may have evolved through the period of time in which the CO metrics have been computed. In case they underwent refactoring operations such as renames or moves, information about developers contributions could be lost. In order to avoid such cases, we used Git's rename detection to follow files that were renamed during the studied period.

We deliberatively did not rely on the information provided by bugtrackers, as several studies showed that their use can introduce an important bias [4, 6]. The drawback of our technique is that the number of bug-fixing commits may not reveal the actual number of bugs that appeared in the software modules. There may exist bugs that are tedious to fix and remain to be resolved. In addition, the manual analysis has some limits due to the subjective evaluation to decide whether or not a commit is a bug-fixing commit. Finally, we only went through a maintenance branch to collect such commits for each project. However, it there may exists bug-fixing commits from the main development branch that have not been backported to the maintenance branch.

5.3. External Validity

The external validity of a study concerns the extent to which the findings are generalizable to other subjects and settings.

As we targeted different languages, we are confident of the generalization of the findings across languages. The only concern we have regarding the external validity of our study is that the projects included in our corpus were not selected using random sampling. Although the results regarding ownership metrics corroborate the ones we found in our previous study [3], they might not be generalizable to all kinds of projects as we only analyze a few projects in this study (14 projects in total, including our previous study).

6. Related Work

In the late 2000s, several studies have shown evidence of a relationship between the number of developers of a software artifact and its fault-proneness. Illes-Seifert and Paech [22, 23] found a correlation between the number of faults identified on a file and its number of authors. Later, they explored the relationship between several process metrics and fault-proneness, and did not find a metric where the relationship with fault-proneness existed in all projects. However, they found that the number of distinct authors of a file was correlated to the number of faults in *almost* every project. Weyuker et al. [24] found that adding the number of developers who edited a file to their prediction model provides a slight improvement to the model's precision.

Many studies used fault prediction models to validate the relevance of process metrics for measuring software quality. Moser et al. [25] compared

the predictive power of two sets of software metrics — code and process metrics — on several Eclipse projects. They found that process metrics are better indicators of software quality than code metrics. Similar results have been found by other researchers who also used fault prediction as a quality indicators for their metrics, such as in [26, 1].

D'Ambros et al. [16] evaluated different sets of metrics in a thorough study on fault prediction. They compared the process metrics introduced by Moser et al. [25] to other metrics, such as the classical source code metrics by Chidamber and Kemerer [27], the measure of entropy of changes introduced by Hassan [28], the churn of source code metrics and the entropy of source code metrics. They found that the process metrics, the churn metrics, and the entropy of source code metrics are the best performers for fault prediction. However, the authors expressed concerns with the external validity of their study (i.e., whether the results are generalizable), which calls for more empirical studies on that matter.

As the number of developers is not always the process metric that shows the highest correlation, ownership metrics rely on other information such as the proportion of contributions made by the developers. Using this information it is possible to classify developers as major and minor contributors. The relationship between measures of code ownership and faults was studied by Bird et al. [2] on Windows Vista and Windows 7 binaries. Their study showed that the number of minor contributors of a binary is strongly correlated to the number of pre- and post-release faults of Windows binaries.

Mockus et al. [21] observed two code ownership patterns in open-source projects: In the Apache project, they found that almost every source code file with more than 30 changes had several contributors who authored more than 10% of the changes. In the Mozilla project they found that code ownership was enforced by the development guidelines, which stated that all contributions should be reviewed and approved by the module owner. Although the focus of their work was FLOSS projects and ownership was also investigated, the authors did not attempt to examine the connection between the ownership patterns and fault-proneness.

In a previous study, we examined the relationship between ownership metrics and fault proneness in open-source projects [3]. Although the results of both studies confirm each other, the dataset of our current study was more carefully constructed than in the previous study, which strengthens the importance of our new findings.

7. Conclusion and Future Work

The study presented in this paper, in which we aimed to improve the methodology presented in our previous paper [3], reaches similar conclusions with a different dataset of projects. First, we confirm that there is a relationship between ownership metrics and software quality. However, the usefulness of code ownership is very debatable: The *Minor* metric is highly collinear with the number of developers, making its computation redundant with a simpler metric. This result is mainly due to the intrinsic characteristics of the open-source projects: they have many contributors but most of them are minor developers. Overall, simple metrics perform better or as well as code ownership metrics, which not only confirm our previous findings, but questions the point of computing ownership metrics.

Other code ownership metrics, such as *MVO* and *Major* may however help to improve regression models for some projects, although not in a drastic way. To confirm that these metrics do improve the quality of regression models, and that they should be used for bug prediction, we plan to perform a study including a larger set of software metrics. As the method we used to measure quality, i.e. manually counting the number of bugfixes in a maintenance branch, has not been validated yet in terms of accuracy, further studies are required to confirm these results.

We also plan to increase the amount of projects in subsequent studies, in order to improve the generalization of our findings. Another trail would be to run a large scale experiment with the goal to select projects where *Minor* and the number of developers are not collinear, which would allow to compare both metrics.

Finally, we would like to stress that, in this study, we only discard the use of the *Minor* metric. We did not explore the application of major and minor contributors to social network metrics, as Bird et al. did in their study of code ownership [2].

8. Acknowledgments

The authors would like to thank Alan Charpentier for his help in the realization of several manual tasks performed in this study, as well as for his helpful comments regarding the methodology we used.

The authors also thank the anonymous reviewers for their comments which helped to improve the quality of this contribution.

9. References

[1] F. Rahman, P. Devanbu, How, and why, process metrics are better, in: Proceedings of the 2013 International Conference on Software Engineering, 2013, p. 432–441.

[2] C. Bird, N. Nagappan, B. Murphy, H. Gall, P. Devanbu, Don't touch my code!: examining the effects of ownership on software quality, in: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ESEC/FSE '11, ACM, 2011, p. 4–14. doi:10.1145/2025113.2025119.

[3] M. Foucault, J.-R. Falleri, X. Blanc, Code ownership in open-source software, in: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, EASE '14, ACM, 2014, p. 39:1–39:9. doi:10.1145/2601248.2601283.

[4] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, P. Devanbu, Fair and balanced?: bias in bug-fix datasets, in: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, 2009, p. 121–130.

[5] T. F. Bissyandé, F. Thung, S. Wang, D. Lo, L. Jiang, L. Réveillère, Empirical evaluation of bug linking, in: Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR 2013), 2013, pp. 1–10.

[6] K. Herzig, S. Just, A. Zeller, It's not a bug, it's a feature: how misclassification impacts bug prediction, in: Proceedings of the 2013 International Conference on Software Engineering, 2013, p. 392–401.

[7] B. Feldman, Relative importance and value, Available at SSRN 2255827.

[8] J. C. Munson, S. G. Elbaum, Code churn: A measure for estimating the impact of code change, in: Software Maintenance, 1998. Proceedings. International Conference on, 1998, p. 24–31.

[9] K. Beck, Embracing change with extreme programming, Computer 32 (10) (1999) 70–77. doi:10.1109/2.796139.

[10] E. Raymond, The cathedral and the bazaar, Knowledge, Technology & Policy 12 (3) (1999) 23–49.

[11] F. Rahman, P. Devanbu, Ownership, experience and defects: a fine-grained study of authorship, in: Proceedings of the 33rd International Conference on Software Engineering, 2011, p. 491–500.

[12] T. Menzies, B. Caglayan, Z. He, E. Kocaguneli, J. Krall, F. Peters, B. Turhan, The PROMISE repository of empirical software engineering data (Jun. 2012).

[13] C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, G. Hsu, Open borders? immigration in open source projects, in: Mining Software Repositories, 2007. ICSE Workshops MSR'07. Fourth International Workshop on, IEEE, 2007, p. 6–6.

[14] C. Bird, A. Gourley, P. Devanbu, Detecting patch submission and acceptance in OSS projects, in: Proceedings of the Fourth International Workshop on Mining Software Repositories, MSR '07, IEEE Computer Society, 2007, p. 26–. doi:10.1109/MSR.2007.6.

[15] T. Zimmermann, R. Premraj, A. Zeller, Predicting defects for eclipse, in: International Workshop on Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007, 2007, p. 9. doi:10.1109/PROMISE.2007.10.

[16] M. D'Ambros, M. Lanza, R. Robbes, Evaluating defect prediction approaches: a benchmark and an extensive comparison, Empirical Software Engineering 17 (4-5) (2012) 531–577.

[17] Y. Tian, J. Lawall, D. Lo, Identifying linux bug fixing patches, in: Software Engineering (ICSE), 2012 34th International Conference on, 2012, p. 386–396.

[18] D. L. Parnas, On the criteria to be used in decomposing systems into modules, Communications of the ACM 15 (12) (1972) 1053–1058.

[19] U. Groemping, Relative importance for linear regression in r: The package relaimpo, Journal of Statistical Software 17 (1) (2006) 1–27.

[20] A. Capiluppi, K.-J. Stol, C. Boldyreff, Exploring the role of commercial stakeholders in open source software evolution, in: Open Source Systems: Long-Term Sustainability, no. 378 in IFIP Advances in Information and Communication Technology, Springer Berlin Heidelberg, 2012, pp. 178–200.

[21] A. Mockus, R. T. Fielding, J. D. Herbsleb, Two case studies of open source software development: Apache and mozilla, ACM Transactions on Software Engineering and Methodology (TOSEM) 11 (3) (2002) 309–346.

[22] T. Illes-Seifert, B. Paech, Exploring the relationship of history characteristics and defect count: an empirical study, in: Proceedings of the 2008 workshop on Defects in large software systems, 2008, p. 11–15.

[23] T. Illes-Seifert, B. Paech, Exploring the relationship of a file's history and its fault-proneness: An empirical method and its application to open source programs, Information and Software Technology 52 (5) (2010) 539–558. doi:10.1016/j.infsof.2009.11.010.

[24] E. J. Weyuker, T. J. Ostrand, R. M. Bell, Using developer information as a factor for fault prediction, in: Proceedings of the Third International Workshop on Predictor Models in Software Engineering, 2007, p. 8.

[25] R. Moser, W. Pedrycz, G. Succi, A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction, in: ACM/IEEE 30th International Conference on Software Engineering, 2008, p. 181–190.

[26] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, M. Nakamura, An analysis of developer metrics for fault prediction, in: Proceedings of the 6th International Conference on Predictive Models in Software Engineering, 2010, p. 18.

[27] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, IEEE Trans. Softw. Eng. 20 (6) (1994) 476–493. doi:10.1109/32.295895.

[28] A. E. Hassan, Predicting faults using the complexity of code changes, in: Proceedings of the 31st International Conference on Software Engineering, 2009, p. 78–88.

Appendix A. Results with different settings

Section 4 only presents the results obtained with the finer granularity of modules, and with ownership weights computed using the *Touches* metric.

Table A.3: Spearman correlation coefficients between metrics and the number and density of post-release bug-fixes, using the **finer granularity** of modules, and ownership weights computed using the ***Churn*** metric

	Spearman Correlation		
Project	**MVO**	**Major**	**Minor**
Number of bug-fixes			
Angular	**-0.52 ****	**0.69 *****	**0.57 ****
Ansible	**-0.53 ***	0.48 *	**0.81 *****
Jenkins	**-0.52 *****	**0.55 *****	**0.73 *****
JQuery	**-0.57 ****	**0.58 ****	**0.84 *****
Rails	**-0.54 *****	**0.55 *****	**0.74 *****
PHPUnit	-0.21	0.37	**0.66 ****
Mono	-0.33 **	0.3 **	0.47 ***
Density of bug-fixes			
Angular	-0.44 *	**0.69 *****	0.42 *
Ansible	-0.34	0.42 .	0.28
Jenkins	-0.34 *	0.32 *	**0.55 *****
JQuery	-0.4 .	0.39 .	0.45 *
Rails	-0.14	0.18	0.29 *
PHPUnit	-0.05	0.22	**0.54 ***
Mono	-0.18	0.18	0.3 **

p-value: 0 *** 0.001 ** 0.01 * 0.05 . 0.1 ' ' 1

For the sake of completeness, we present here the results obtained with the larger granularity of modules, and the ones obtained with ownership weights computed using the *Churn* metric.

We also show correlations obtained with the density of bug-fixing commits, rather than with the absolute number of bug-fixing commits.

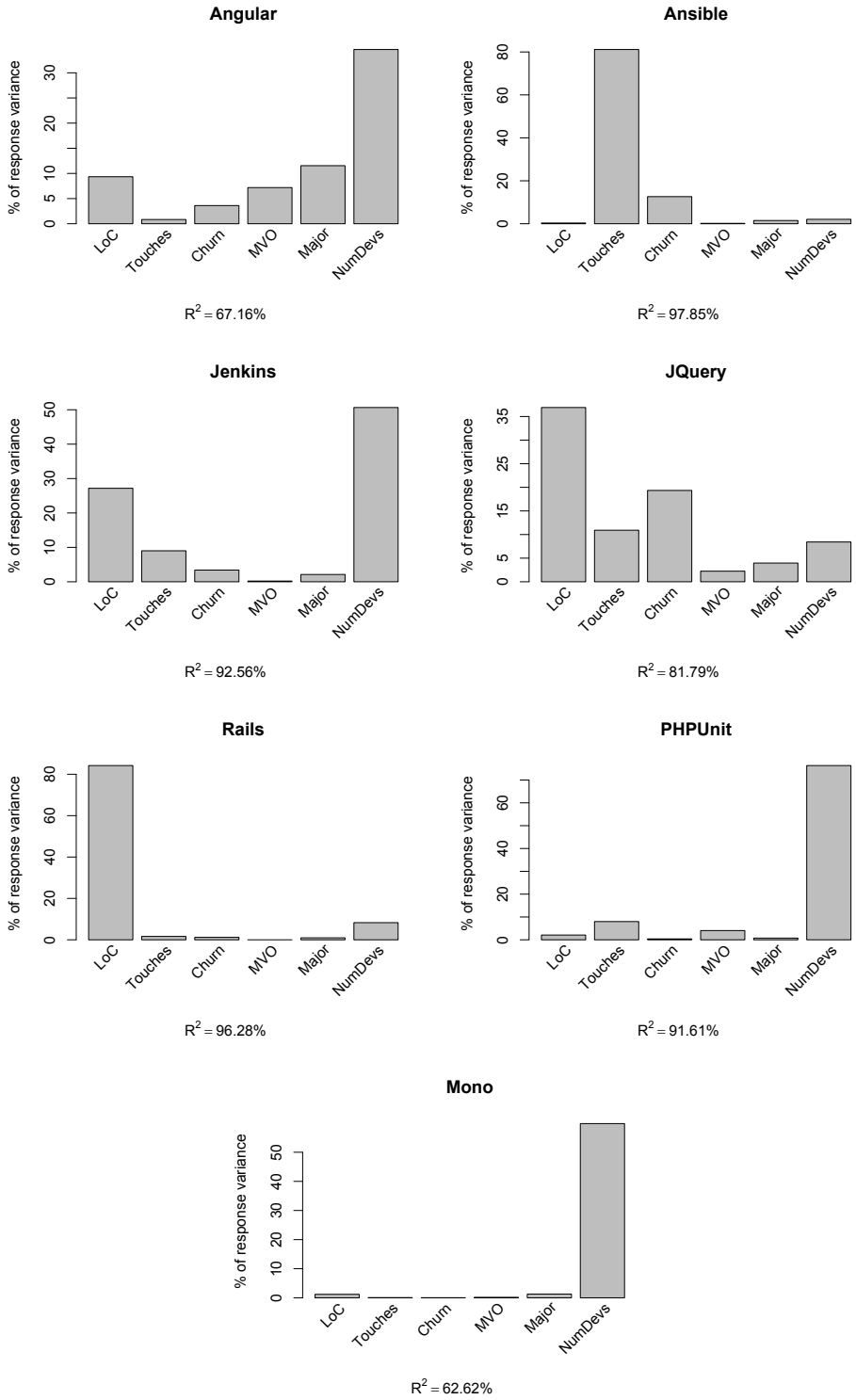

Figure A.2: Relative importance of metrics, using the **finer granularity** of modules, using ***Churn*** to weigh developers' ownership.

Table A.4: Spearman correlation coefficients between metrics and the **density** of post-release bug-fixes, using the **finer granularity** of modules, and ownership weights computed using the ***Touches*** metric.

	Spearman Correlation						
Project	**LoC**	**NumDevs**	**Touches**	**Churn**	**MVO**	**Major**	**Minor**
Angular	0.11	**0.65 *****	0.45 *	0.39 *	**-0.57 ****	**0.73 *****	0.4 *
Ansible	0.13	0.37	0.24	0.25	-0.02	0.16	0.23
Jenkins	0.45 **	**0.53 *****	0.48 **	0.4 **	-0.36 *	0.47 **	0.41 **
JQuery	0.3	**0.51 ***	**0.5 ***	**0.65 *****	-0.27	**0.53 ****	0.25
Rails	0.09	0.31 *	0.31 *	0.33 *	-0.21	0.29 .	0.16
PHPUnit	**0.73 ****	0.26	**0.5 ***	0.42	0.08	-0.07	**0.64 ****
Mono	0.23 *	0.29 *	0.27 *	0.26 *	-0.23 *	0.19 .	0.38 ***

p-value: 0 *** 0.001 ** 0.01 * 0.05 . 0.1 ' ' 1

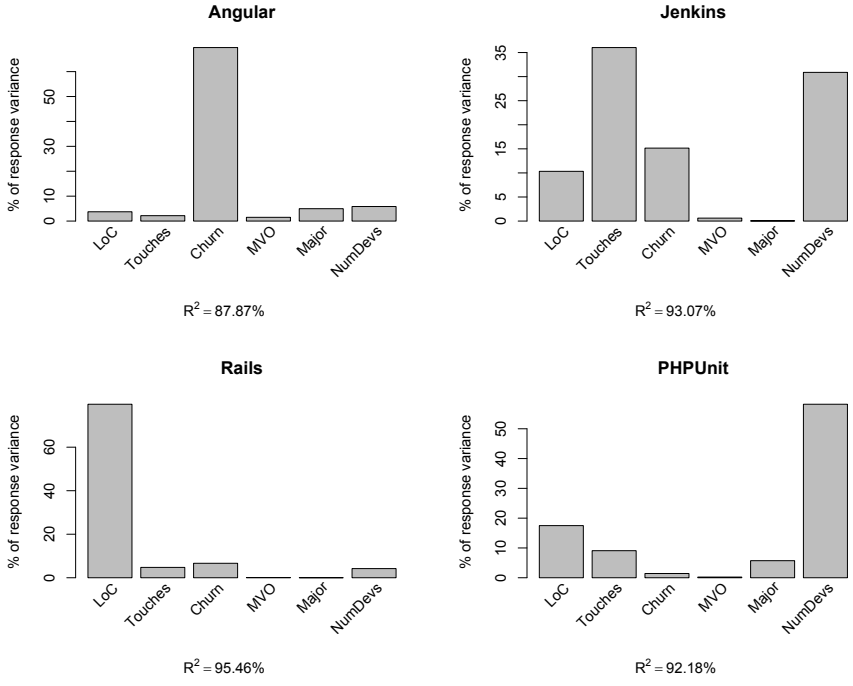

Figure A.3: Relative importance of metrics, using the **larger granularity** of modules, using ***Touches*** to weigh developers' ownership.

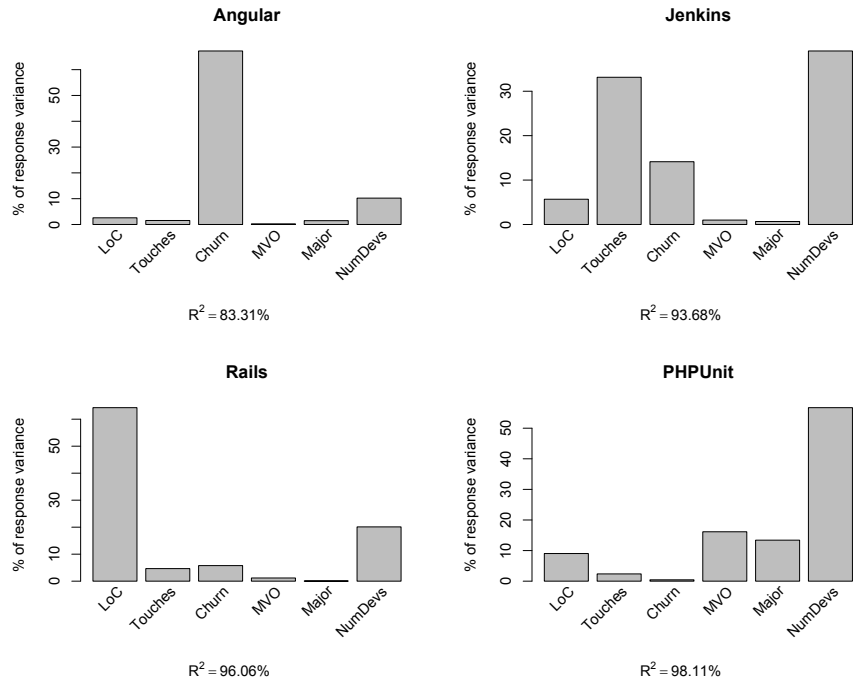

Figure A.4: Relative importance of metrics, using the **larger granularity** of modules, using ***Churn*** to weigh developers' ownership.

Table A.5: Spearman correlation coefficients between metrics and the number and density of post-release bug-fixes, using the **larger granularity** of modules, and ownership weights computed using the ***Churn*** metric

	Spearman Correlation		
Project	**MVO**	**Major**	**Minor**
Number of bug-fixes			
Angular	**-0.51 ***	**0.6 ****	**0.57 ***
Jenkins	**-0.52 ***	**0.52 ***	**0.77 *****
Rails	**-0.69 ****	**0.64 ****	**0.89 *****
PHPUnit	-0.36	0.49	**0.62 ***
Density of bug-fixes			
Angular	-0.36	**0.59 ****	**0.62 ****
Jenkins	-0.36 .	0.36 .	0.42 *
Rails	-0.02	0.08	0.22
PHPUnit	-0.06	0.24	0.36

p-value: 0 *** 0.001 ** 0.01 * 0.05 . 0.1 ' ' 1

Table A.6: Spearman correlation coefficients between metrics and the number and density of post-release bug-fixes, using the **larger granularity** of modules, and ownership weights computed using the ***Touches*** metric.

	Spearman Correlation						
Project	**LoC**	**NumDevs**	**Touches**	**Churn**	**MVO**	**Major**	**Minor**
Number of bug-fixes							
Angular	0.16	**0.74 *****	**0.55 ***	0.44 .	**-0.52 ***	**0.59 ***	**0.56 ***
Jenkins	**0.82 *****	**0.77 *****	**0.73 *****	**0.67 *****	**-0.53 ****	**0.55 ****	**0.77 *****
Rails	**0.88 *****	**0.89 *****	**0.87 *****	**0.88 *****	**-0.75 *****	**0.58 ****	**0.83 *****
PHPUnit	**0.78 ****	0.49	**0.55 .**	**0.53 .**	-0.12	-0.07	**0.81 ****
Density of bug-fixes							
Angular	-0.2	**0.75 *****	0.31	0.31	**-0.6 ****	**0.66 ****	**0.52 ***
Jenkins	**0.5 ***	0.47 *	0.42 *	0.33	-0.35 .	0.37 .	0.4 .
Rails	0.1	0.18	0.12	0.21	-0.14	0.14	0.15
PHPUnit	**0.58 .**	0.2	0.29	0.29	0.09	-0.27	**0.55 .**

p-value: 0 *** 0.001 ** 0.01 * 0.05 . 0.1 ' ' 1