Ownership and Defects in Open-Source Software

Eric Camellini
Politecnico di Milano
Milan, Italy
Eric.Camellini@gmail.com

Kaj Dreef
Delft University of Technology
Delft, The Netherlands
Kaj.Dreef@gmail.com

ABSTRACT

Code ownership measures the proportion of contribution of the developers to a source code artifact, in terms of code changes (e.g. number of commits). It can be used to describe the responsibility for a certain piece of software or the expertise of the developers with respect to it. Responsibility and expertise are important factors in the development process, and can affect software quality: for this reason a lot of studies focused on determining how to measure the ownership in order to include it in defect prediction models in an effective way. Bird et al [3] and Greiler et al. [7] showed that ownership metrics are a good indicator for software quality in Microsoft software projects, while Foucault et al. [6] found contrasting results for what concerns open-source software projects. However, in our opinion these past studies compute the ownership without considering the software revisions that actually introduce defects. Furthermore, no study did an explicit analysis of the effect of the granularity chosen to consider the code changes (e.g. considering lines instead of commits).

In this paper we contribute to the past research in the following ways: (1) we build dataset, publicly available, that contains information about code changes and defects over the whole development history for five open-source software projects; (2) we describe and apply a novel technique to compute software metrics, using the above mentioned dataset, that can capture the state of the software right after the introduction of defective code; (3) we use this technique to empirically study the effect that ownership has on software quality for five open-source software projects, considering an exhaustive set of metrics in terms of granularity of code changes.

Using this approach and considering also a set of classic and effective code metrics we are able to classify defective files, using Random Forest, with an average out-of-bag error rate of 23% and an average relative improvement of 22% over a model that uses only the classic metrics.

The 13th Working Conference on Mining Software Repositories '16 Austin, Texas USA

DOI:

CCS Concepts

•**Software and its engineering** → **Software defect analysis;** *Open source model;* •**Information systems** → Data mining;

Keywords

Ownership; Software quality; Process metrics

1. INTRODUCTION

Software defects correction have a great impact on the economy: it costs tens of billions of dollars every year and the 50% of developers programming time [2, 1]. For this reason, in recent years, a wide number of studies focused on defining and comparing software metrics that can be useful to build models for *defect prediction* [3, 14, 7, 15, 11, 18]. A *software metric* is a measure of a property of the software that can be related to the code (*code metric*) or to its development process (*process or change metric*). Previous studies also reported that process metrics are better-suited for prediction models [11, 15]. These past results showed that the developers behaviour has more impact on software quality than the characteristics of the software itself. *Code ownership* metrics are process metrics, and can in general be defined as measurements of the proportion of contribution of the developers to a source code artifact over a certain period of time, in terms of code changes (e.g. number of commits) [7].

Previous studies reported contrasting results on the relation between software quality and ownership; this is probably due to the fact that ownership metrics are highly dependent on the process used to develop the software and on the organizational structure of the team. Bird et al. [3] showed that there is a significant correlation between ownership and number of defects in Microsoft Windows projects, while accordingly to Foucault et al. [6] the same metrics do not result in the same kind of relationship when computed on open-source software (OSS) artifacts. Foucault et al. [6] also introduced the idea of computing ownership metrics on artifacts of different granularities (Java files and packages). Greiler et al. [7] applied the same principle on different Microsoft Products, and their results showed that it can be more significant to use ownership metrics to classify defective and non-defective artifacts rather than to try to infer the number of defects.

All the previous studies computed the metrics on a certain release of the software and then tried to find a correlation between these metrics and the number of defects introduced

before that release. We think that the metrics should instead be computed on the software artifacts just after the commits that introduce a defect, to better capture the exact state of the code when it becomes defective. In addition to that, most of the previously cited works used the commit count to measure the contribution of the developers to the source code artifacts; none of them tried to compute the ownership metrics in a more fine-grained way (e.g. considering the amount lines of code added and deleted). Furthermore, to our knowledge, a classification approach like the one described in in the previous paragraph has not yet been applied to Open-Source Software.

In this work, we study the effect of code ownership on software quality by trying to classify defective source code files on different Open-Source software projects. We expand the cited previous work by: (1) computing ownership metrics on source files just after some defective code is introduced; (2) experimenting the effects of changing the granularity of the metrics; (3) applying a classification approach to distinguish defective and non-defective source files on Open-Source software projects.

To spot the introduction of defective code we use the concept of *implicated code* as described by Rahman et al. [14] (also called *fix-inducing* code [17]). We selected 5 OSS projects and, for each one of them, we: (1) create a dataset that captures the history of its development in terms of developers contribution and implicated code introduction (2) use this dataset to create a second dataset that contains the ownership metrics computed for every version of every file, with different granularities (3) use these metrics, together with some classic defect prediction code metrics, to classify the defective file versions with the Random Forests [4] and Logistic Regression techniques (4) measure how the ownership metrics improve the model that uses only the classic ones.

Our results show that ownership metrics are indicative for software defects, giving a significant improvement over a classification model built using only classic code metrics; this is particularly evident when using a more fine-grained, line-based, approach to compute the ownership, as we hypothesized.

We think that our approach can capture the characteristics of the defective code in a better way, leading to more generalizable outcomes.

Code and datasets created for this work are publicly available.[1]

2. THE PROBLEM

The general problem that is targeted in this paper is that building software defect prediction models is a challenging activity, and one of its main difficulties is that it is highly project-dependant [18]. In particular we address this problem for models built using *code ownership* metrics. Code ownership is a measurement of the proportion of contribution of the developers to a source code artifact over a certain period of time, in terms of code changes (e.g. number of commits) [7]. It describes whether the *responsibility* for a certain software artifact is spread around many developers, or if there is a single person that can be considered its "owner"; it can also be interpreted as a measure of the *expertise* of a developer with respect to the code artifact [3].

This work addresses the more specific problem that currently it is difficult to generalize the effect of code ownership on software quality; previous studies show contrasting result when trying to correlate these two aspects [3, 6, 7]. What makes the problem complicated is that code ownership highly depends on the organizational structure of the development team and on the developers behaviour.

2.1 Existing solutions and limitations

The main source of inspiration for us comes from Bird et al. [3], who did, to our knowledge, one of the first empirical studies of the effects that code ownership has on software quality. They used the concept of ownership in a defect prediction model built for Microsoft Windows; to do that they extracted the following metrics from the software artifacts:

- *Ownership*: proportion of ownership for the highest contributor;
- *Minor*: number of contributors with a proportion of ownership that is below a certain threshold (minor contributors);
- *Major*: number of contributors with a proportion of ownership that is above the threshold used for the minors (major contributors);
- *Total*: total number of contributors.

As artifacts they considered the software binaries of a Microsoft Windows release and as variable to measure the proportions of ownership on every artifact they used the number of commits that changed it before the release, with a 5% threshold to identify minor and major contributors. These metrics have been then reused and revisited in further studies [6, 7], targeting different projects (Microsoft and OSS), different kind of artifacts (source files, source code folders and Java packages) and changing the threshold (5%, 20% and 50%), but using the same variable to measure the ownership and again computing it on the artifacts of a specific software release.

In the cited works we see the following main shortcomings:

1. The metrics are computed on the code artifacts of a specific software release and then correlated with the presence of defects in it. The problem is that in this way the metrics are not extracted when the defects are introduced, but later, so they don't capture the state of the code in the moment that it becomes defective;
2. The variable used to measure the ownership is the number of commits to the code artifact, a coarse-grained measure of the code changes, and none of the cited studies experimented different granularities;
3. None of the previous works did an explicit analysis on the impact that changing the threshold used to distinguish minor and major contributors has on the study results;

In this work we try to solve these problems; we think that taking into account these three factors results in ownership metrics that better adapt to the specific characteristics of the software project. This leads to more generalizable outcomes, so we address the more general problem described at the beginning of this Section.

[1] github.com/kajdreef/IN4334-MSR

3. PROPOSED SOLUTION

In essence, our approach focuses on solving the first problem described in Section 2.1, so on when ownership metrics are computed. Once solved this problem, the second and third are addressed computing more metrics for different granularities and thresholds and studying their effects on the results.

To solve the first problem we cannot simply compute the metrics on all the artifacts of a specific version of the software, as the previous studies did, so we use a different approach: we extract them from every version of every artifact over the history of the project development. To correlate them with the defects we then mark all the artifacts versions that follow the introduction of defective code and we try to build a classification model that, using the metrics, can distinguish them from the other ones. In this way the metrics are used to build a model that can ideally determine when a commit introduces a bug, because we extract them on the software artifacts as soon as they become defective and compare them with the normality.

We focus on computing the ownership metrics considering the Java source files as target artifacts; we are first interested in determining which versions of the files follow the introduction of defects. To spot the introduction of defects we use the concept of *implicated code*.

3.1 Implicated code

We refer to the concept of *implicated code* as described by Rahman et al. [14]: "Implicated code is code that is modified to fix a defect". Rahman et al. also describe a technique to identify the implicated code using git (see Figure 1). The steps to do it using git [2] are the following:

1. Identify a commit that fixes a bug: this commit changes one or more files from version n to version $n+1$;
2. Identify the lines that are deleted or changed by this commit using the *git diff* command between the versions n and $n+1$;
3. Checkout revision n, since it still contains the defective code (changed or removed lines), and use the *git blame* command to identify in which versions of the file it was introduced: the same code in that versions is the implicated code;
4. If that code was introduced after the fixed bug was reported, then mark it as *innocent* (not implicated).

In this work we don't consider the last step: this because we assume that if the lines are added after the issue reporting date, they are still based on a defective file, and so they contribute to the defect. We want to highlight the fact that a line of code is marked as implicated only in the version of the file in which it is introduced.

Since we need to spot defective file versions we define the new concept of **implicated file**: a file in a specific version is implicated if that version contains implicated code.

3.2 Bug and commit linking technique

To apply the technique described to extract the implicated files we must be able to identify which commits are bug fixes. To do so we decided to use the JIRA issue tracking system [3] and in particular the bug convention that Apache projects use on it [4]. Every Apache software project that uses JIRA has an issue key that is, by convention, mentioned in commits that address an issue, together with the issue id. In particular, the convention is to include these information with the following notation: KEY-ID (e.g. LUCENE-1234 if the commit addresses the issue 1234 of the Lucene project).

We consider a commit as a bug fix if its message mentions the key and id of an issue that is marked as a fixed bug on JIRA. To navigate through the JIRA issues we use the data extracted in JSON format in the work by

<MISSING: Cite the technical report for the JIRA JSON issues extraction.>

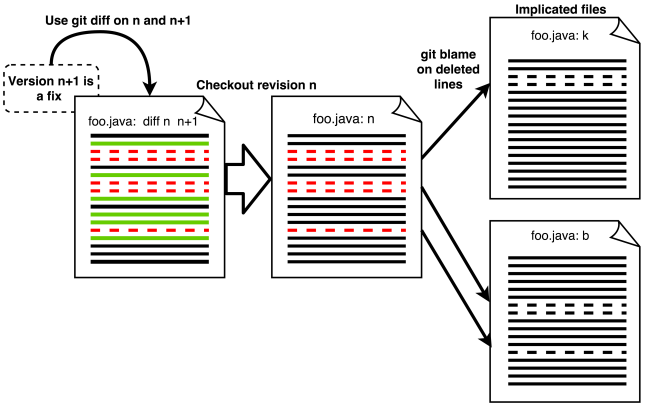

Figure 1: Implicated code identification technique

3.3 Ownership metrics

As ownership metrics we decided to compute the ones defined by Bird et al. [3] and already described in Section 2.1. In particular we compute the first three ones (Ownership, Minor and Major) using three different variables to quantify code changes: commit count, number of lines added and number of lines deleted. We also use five different thresholds to distinguish minor and major contributors: 5%, 10%, 20%, 30% and 40%. In this way we address also the second and the third of the problems described Section 2.1.

To expand our study we also decided to compute the *authorship*. Authorship can be seen as memory-less ownership: the metrics of this class are not based on the history of the development but only on the content of the file on which they are computed. The concept of authorship was already introduced by Rahman et al. [14], but was applied only to implicated code chunks and not at a file level. In particular we compute the two following metrics:

- Line authorship: proportion of lines in the file authored by the developer with the highest proportion of lines authored;
- Total authors: total number of authors of the lines in the file.

We consider these two measures as part of our ownership metrics, following the memory-less vision described above. To clarify the difference between the two classes of metrics lets suppose to compute them on the version V of the file F, using the number of lines added as variable to quantify the code changes:

[2] git-scm.com

[3] atlassian.com/software/jira

[4] issues.apache.org/jira/secure/BrowseProjects.jspa#all

- the proportion of ownership of the contributor C is computed as the number of lines added to F by C over all the considered history (all the versions of F that precede V), divided by the total number of lines added to F by all the contributors over all the considered history;

- the proportion of authorship of the contributor C is computed as the number of lines that are actually in F in its version V and that are authored by C, divided by the size of F in the same version (in terms of lines);

3.4 Classification

As already stated previously in this section, we use the concept of implicated code to mark code that can be considered defective, therefore in this research the defective files are the implicated ones. This means that the classification model that we want to build should distinguish which versions of the project files are implicated using the metrics that we described.

To be able to evaluate our results in a way that is more sound in the defect prediction field, we first build a model that uses some classic code metrics that are known to be effective, then we add our metrics and we quantify the improvement in the model accuracy (a similar approach was also adopted by Bird et al. [3]).

The classic metrics that we decided to use are (1) file size; (2) comment-to-code ratio; (3) number of previous defects (implications, in our case). We selected these three because are known to be indicative of software defects [15, 5].

4. METHODOLOGY

4.1 Research questions

We structure our research through the following research questions:

RQ1 Are ownership metrics indicative for the presence of implicated code in source files for open-source software projects?

RQ2 Can ownership metrics be used to build a classification model to identify implicated source files?

- **2a** For which level of granularity (line-based or commit-based) do ownership metrics give more accurate results when used to build the classification model?
- **2b** How does the value of the threshold, used to distinguish minor and major contributors, impacts on the accuracy of the classification model?

The first research question is a more general question, which is needed to determine if in general ownership metrics can be a good indication of the presence of implicated code in source files for open-source software. The second research question focuses on if it is possible to classify the implicated files based on ownership metrics. The sub-questions focus on the type of metrics included in the classifier: in particular on the parameters used to compute them (i.e. code changes granularity and minor-major threshold) and on the effects resulting from changing these parameters.

4.2 Study subjects

A software project, in order to be used for this research, should (1) be and open-source software project; (2) use git as version control system, or have a git mirror of the repository (requirement for the technique described in Section 3.1; (3) use JIRA as issue tracking system and adhere to the Apache JIRA convention, so that we can apply the technique described in Section 3.2.

We selected five different software projects as study subjects, all written using the Java programming language. In Table 1 we show some information about each one of them. When choosing the projects we checked if they satisfied the previous mentioned requirements, but also tried to choose them with different sizes and application fields, in order to have a diverse group.

For each project we decided to consider its whole development history until the 01/01/2015: this because we had issues extracted in JSON until the half of the year, and we only wanted to consider code for which the most of the issues were already discovered and fixed. We also decided to discard some of the first days of commits from the data extracted from the repositories, because these are the commits needed to setup the project repository or the git mirror. We did this last step by manually checking the messages of the first available commits.

4.3 Research steps

What we want to do is to build a model that is able to identify the implicated files in the history of the selected software projects, and to evaluate its effectiveness. In this Section we describe in a detailed way all the steps followed in this research.

4.3.1 Project history dataset

In the first step of our research we create, for every software project, a dataset that contains all its Java files history (over the considered time period, see Section 4.2) in terms of commits, lines, bug fixes and implications. To do that we go through all the commits in its *git log*, and for every commit we:

1. extract the list of the Java files affected;
2. for each affected file update the information about the contribution that the author of the commit performed to it in terms of lines added, lines deleted and number of commits;
3. extract information about the file itself: size (LOC), authorship information, comment lines, if it is a bug fix (using the technique described in Section 3.2);
4. add to the data set, for each one of these files, as many lines as the number of authors that contributed to it until the considered commit (included): each one of these lines contains the information about the contribution of the corresponding author to the corresponding file, at the moment of the corresponding commit, plus the information related to the commit itself.

We then add to this dataset, once extracted, another column that says when a file is implicated, using the approach described in Section 3.1.

A complete list of the columns contained in the dataset can be found in Table 4. Some of them contain information

Table 1: Characteristics of the studied projects (until 01/01/2015)

Project	LOC	Contributors	Commits	
Camel	103883	125	18371	Rule-based r
Lucene-solr	220632	55	13841	Information retri
Mahout	177317	31	3163	Implementation of s
Maven	129132	51	9909	
Zookeeper	108132	13	1283	Distributed s

that are not related to the single author, but to the file or the commit: in this case their value will be the same for more than one line.

An example of this procedure is shown in Figure 2 and Table 2, it shows only information about commits and lines added but it can be used to better understand the described methodology.

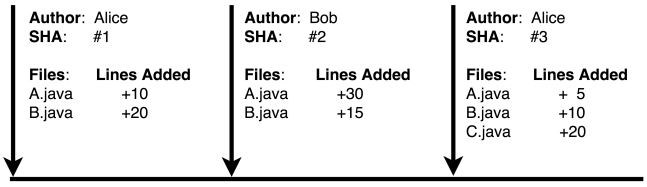

Figure 2: Example of a small portion of development history (i.e. commits), with lines added information. The corresponding history dataset is shown in Table 2

		columns (see Table 4)					
file	sha	4	5	6	7	11	12
A.java	#1	Alice	10	10	10	1	1
B.java	#1	Alice	20	20	20	1	1
A.java	#2	Alice	10	0	40	1	2
A.java	#2	Bob	30	30	40	1	2
B.java	#2	Alice	20	0	35	1	2
B.java	#2	Bob	15	15	35	1	2
A.java	#3	Alice	15	5	45	2	3
A.java	#3	Bob	30	0	45	1	3
B.java	#3	Alice	30	10	45	2	3
B.java	#3	Bob	15	0	45	1	3
C.java	#3	Alice	20	20	20	1	1

Table 2: Example of a small portion of history dataset with a reduced set of columns. The corresponding development history is shown in Figure 2 ·

		columns (see Table 5)						
sha	file	3	4	6	11	12	13	14
#1	A.java	1 / 1	10 / 10	1	1	0	1	0
#1	B.java	1 / 1	20 / 20	1	1	0	1	0
#2	A.java	1 / 2	30 / 40	2	2	0	1	1
#2	B.java	1 / 2	20 / 35	2	2	0	2	0
#3	A.java	2 / 3	30 / 45	2	2	0	2	0
#3	B.java	2 / 3	30 / 45	2	2	0	2	0
#3	C.java	1 / 1	20 / 20	1	1	0	1	0

Table 3: Example of a small portion of metrics dataset with a reduced set of columns, extracted from the history dataset example shown in Table 2 using a 30% threshold to distinguish minor and major contributors.

4.3.2 Metrics computation

In this second step, for every project and using the respective history dataset, we compute the dataset that contain the metrics that we need for the classification. We need to compute the metrics for every version of every file, so the computation is done grouping by commit SHA and file (columns 2 and 3 of the history dataset, see Table 4).

As metrics we compute the ownership ones described in Section 3.3 together with the classic code metrics listed in Section 3.4. This will produce another data set, with fewer lines, that contains the metrics and the column "implicated", defined as in Table 4. A complete set of the columns of the metrics dataset can be seen in Table 5, where the terms ownership, minor, major and total must be interpreted as defined in Section 3.3.

A different version of this dataset is computed for every threshold listed in Section 3.3 (the threshold used to distinguish minor and major contributors). Table 3 shows an example of the metrics dataset resulting from the history dataset example shown in Table 2, using a threshold of 30%; it contains a reduced set of columns: only the ones based on the lines added and on the commit count.

4.3.3 Classification

Using the metrics dataset we build, for every project, a classification model using the Random Forests approach [4], and we then evaluate how effective it is when used to distinguish which file versions are implicated. We also use Logistic Regression to determine which features are significant. The choice of these techniques is based on the fact that they were already used respectively in [7] and [3]. An accurate description of this process is provided in Section 5.

5. EVALUATION

5.1 Experiment 1: Ownership and granularity

5.1.1 Experiment design

In this experiment we address research questions 1 and 2 building a model that can classify implicated files using the classic code metrics, and evaluating if the ownership metrics can improve its effectiveness. We also address question 2a comparing the improvement given by different ownership granularities (commit-based and line-based).

Table 6: Class sample size per project

Project	Implicated files	Sample size
Camel	7076	4000
Lucene-Solr	14416	4000
Mahout	2114	2000
Maven	2383	2000
Zookeeper	819	800

This experiment is performed for every considered project using its metrics dataset with the 5% threshold. We define different subsets of features (see Table 7), and for each one of them we cross-validate (10-folds) the Random Forest [4] model on a sample of the dataset. The sample contains the same number of lines for both the classes (implicated and not implicated, see Table 6).

Table 4: Columns of the history data set

#	Column	Description
1	project	project name
2	file	file name (full path)
3	sha	commit SHA (can be interpreted as file version or revision)
4	author	name of the contributor **(NOTE: it is not the author of the commit, but one of the file contributors)**
5	author_file_tot_added	total lines added by the author to the file
6	author_file_added_this_commit	lines added by the author to the file with the commit (different from 0 only for the author of the commit)
7	file_tot_added	total lines added to the file
8	author_file_tot_deleted	total lines deleted by the author from the file
9	author_file_deleted_this_commit	lines deleted by the author from the file with the commit (different from 0 only for the author of the commit)
10	file_tot_deleted	total lines deleted from the file
11	author_file_commits	total commits to the file by the author
12	file_tot_commits	total commits to the file
13	current_lines_authored	number of lines actually present in the file and authored by the author (obtained from the *git blame* output)
14	current_file_size	file size measured in LOC (lines of code)
15	current_comment_lines	comments size in terms of lines
16	max_current_author	number of lines actually present in the file and authored by the author that authored the highest number of lines actually in the file (obtained from the *git blame* output)
17	total_current_authors	number of authors of the lines actually present in the file (obtained from the *git blame* output)
18	commit_date	date of the commit
19	bug_fix	1 if the commit fixes one or more bugs, zero otherwise
20	fixed_bugs	JIRA KEY-ID of the bugs fixed, empty if the commit is not a bug fix
21	affected_versions	list of the project releases affected by the bugs fixed by the commit, empty if the commit is not a bug fix
22	implicated	1 if the file version is implicated

Table 7: Considered set of metrics

Group	Metrics
Classic	file_size, comment_to_code_ratio, previous_implications
Commit based	Classic + commit_ownership, minor_contributors, major_contributors
Deleted	Classic + line_ownership_deleted, lines_deleted_minor_contributors, lines_deleted_major_contributors
Added	Classic + line_ownership_added, lines_added_minor_contributors, lines_added_major_contributors,
Line authorship	Classic + line_authorship, total_authors
Line based	Classic + Added + Deleted
All metrics	All the above mentioned metrics minus the highly correlated ones (cut-off=0.75)

We evaluate the constructed classifiers with the out-of-bag error rate (OOB) averaged over the 10-folds, and use it to determine if the improvement over the classic metrics is statistically significant. This is done by comparing with the t-test the outcome of every set of metrics with the outcome obtained considering only the classic ones. For the test we assume a significance level of *0.05*.

5.1.2 Results

In Table 8 the results for the project Lucene-Solr can be found. It can be noted here that the different granularities between metrics have a significant influence on the performance of the classifier. Line-based metrics (deleted, added, authorship) give a larger performance increase than the commit-based ones. This effect is highlited also in Figure 3: it shows the importance of the single metrics for Lucene-Solr. Eventually, Figure 4 shows the corresponding ROC curve. Similar results come from the other projects.

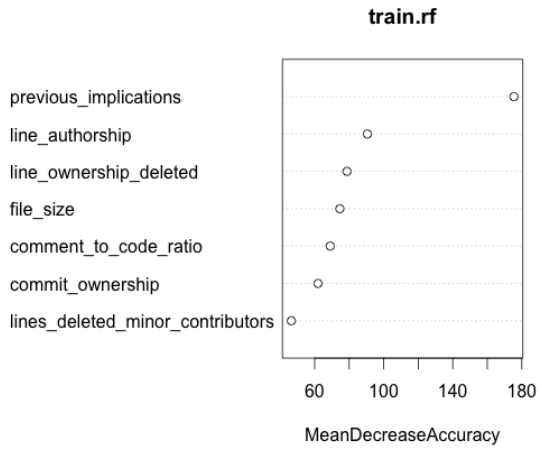

Figure 3: Importance of variables for Lucene-Solr

Table 9 shows a summary of the results of the experiment. Using all the metrics the average OOB is 22.99% which gives an average increase of 21.71% in performance over the classic metrics. For what concerns the granularity, line-based metrics perform on average 15% better than commit-based metrics. All the p-values resulting from the significance test performed on the improvements are far below the 0.05 significance level, meaning that all the improvements over the classic metrics are statistically significant.

Table 5: Columns of the metrics dataset; See Section 3.3 and Table 4 for the terms used in the definitions.

#	Column	Description
1	sha	commit SHA (can be interpreted as file version or revision)
2	file	file name (full path)
3	commit_ownership	max(author_file_commits / file_tot_commits)
4	line_ownership_added	max(author_file_tot_added / file_tot_added)
5	line_ownership_deleted	max(author_file_tot_deleted / file_tot_deleted)
6	total_contributors	total number of contributors (count the of the lines grouped together)
7	line_authorship	max_current_author / file_size
8	total_authors	total_current_authors
9	file_size	file_size
10	comment_to_code_ratio	current_comment_lines / (current_file_size - current_comment_lines)
11	major_contributors	count where author_file_commits / file_tot_commits >= threshold
12	minor_contributors	count where author_file_commits / file_tot_commits < threshold
13	lines_added_major_contributors	count where author_file_tot_added / file_tot_added >= threshold
14	lines_added_minor_contributors	count where author_file_tot_added / file_tot_added < threshold
15	lines_deleted_major_contributors	count where author_file_tot_deleted / file_tot_deleted >= threshold
16	lines_deleted_minor_contributors	count where author_file_tot_deleted / file_tot_deleted < threshold
17	previous_implications	count how many times this file was implicated before that version
18	implicated	1 if the file version is implicated

Table 8: Lucene-Solr experiment 1 results

Features	Error Rate with 5% threshold	Improvement over Classic	Precision	Recall
Classic	32.59%	0.00%	67.85%	67.26%
Classic + Commit based	30.38%	6.78%	67.21%	70.62%
Classic + Deleted	29.69%	8.88%	70.09%	70.40%
Classic + Added	29.49%	9.51%	69.40%	70.98%
Classic + Line authorship	29.46%	9.59%	74.29%	69.11%
Classic + Line based	26.12%	19.86%	80.41%	71.13%
All metrics	26.34%	19.16%	80.50%	70.81%

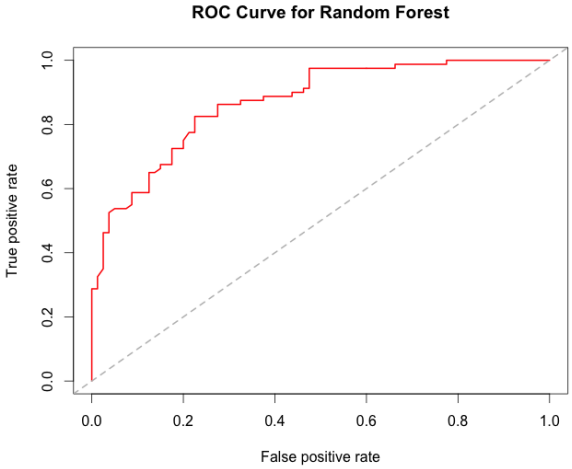

Figure 4: ROC curve resulting from experiment 1 on Lucene-Solr, considering all the metrics.

5.2 Experiment 2: Logistic regression

5.2.1 *Experiment design*

In the previous experiment we used Random Forest to determine the performance of different granularities. In this experiment we will take it a step further by taking a look at the individual metrics to see how statistically significant they are. The goal is to determine which metrics are significant, checking also if the significance changes depending on the threshold, and to determine if there is a common group of metrics that is significant for all the selected study subjects.

We determine the statistically significant metrics by running Logistic Regression on the dataset without feature selection, since this technique automatically detects which ones are not linearly dependent with each other. Logistic regression is a good choice in this case: this because it automatically computes the significance of every variable in terms of p-value.

Table 10: Logistic Regression outcomes explanation.

Pr(>|z|)	Symbol
0	* * *
0.001	**
0.01	*
0.05	.
0.1	
NA	NA

5.2.2 *Results*

Table 10 shows how to interpret the results of the Logistic Regression: NA means that the metric is considered linearly dependent with another one and because of that it is not taken into consideration. Table 11 summarizes the results of

Table 9: Summary of the results of experiment 1

Project	Classic	Commit based	Line based	All metrics	Improvement (all metrics with FS)	Improvement (Line based)
lucene-solr	32.59%	30.38%	26.12%	26.34%	19.16%	19.86%
mahout	31.68%	29.44%	25.42%	29.33%	7.43%	19.77%
Camel	29.36%	25.16%	19.38%	19.25%	34.43%	34.00%
Maven	25.51%	23.19%	19.27%	19.09%	25.15%	24.45%
Zookeeper	26.94%	22.35%	20.67%	20.92%	22.37%	23.27%
Average	29.22%	26.10%	22.17%	22.99%	21.71%	24.27%

the significance of the different metrics for the 5% threshold.

The classic metrics are, as expected, highly significant. It can be noted that the total contributors and line authorship are also highly significant for all the projects, while the rest of the ownership metrics significance values vary a lot, meaning they are probably more project dependent. Despite that, for every project it is possible to see that at least five of these ownership metrics are statistically significant (considering a 0.05 significance level).

Table 12 shows the influence that changing the threshold has on the significance. It can be seen there is no clear improvement in significance for the metrics, meaning that when the threshold changes the change in the significance of the metrics is not relevant: while some of them become more significant, others result to be less important. For example, when considering a 10% threshold, *lines_added_major_contributors* becomes more significant for Lucene-Solr but also less significant for Mahout, in comparison to the 5% threshold scenario. For other threshold values the results are similar.

5.3 Experiment 3: Minor-major thresholds

5.3.1 Experiment design

Research question 2b has to do with the threshold used to distinguish minor and major contributors and with how much does it influence the performance of the classifier. The goal of the experiment is to determine the effect of the different thresholds that we selected to use: `0.05`, `0.10`, `0.20`, `0.30`, and `0.40`.

As said in Section 4.3.2, for every project we computed a metrics dataset for each one of the thresholds. In this experiment we use again the Random Forest technique to build a classifier for every threshold and every project, considering only the metrics that depend on the threshold, so all the ones that measure minor and major contributors (see Table 5). This results in 25 classifiers.

The performance of every classifier is measured using 10-folds cross-validation and the OOB error: since we used only features that depend from the threshold, the difference between these resulting models is only related to its variation.

To determine if the 5 groups of OOB errors, corresponding to different thresholds, differ in a statistically significant way we apply an *ANOVA* test, which will tell if the outcomes are statistically significant. Based on the output p-values we can determine if some thresholds are statistically better than other ones for some of the projects.

5.3.2 Results

In Table 13 shows the out-of-bag error for the different projects and thresholds. It can be noticed that changing the threshold doesn't give a clear increase in performance in any case. The ANOVA test shows that there is no significant outcome: all p-values are above 0.05, meaning that major-minor threshold doesn't have any significant impact on the performance.

5.4 Summary of results

RQ1,2: Based on the results of the experiments we can say that ownership metrics are an indication for the presence of implicated code. Over the 5 projects we got an average classifier OOB of 22% with a significant improvement over the one that uses only classic metrics (22%).

RQ2a: The level of granularity used to compute the ownership metrics is important: our results show that line-based metrics give a significant improvement and are on average 15% more effective than the commit-based ones. The authorship can also be considered a line-based ownership metric; it gives even a better improvement and it is statistically significant for all the projects.

RQ2b: For what concerns the threshold used to distinguish minor and major contributors, the results show that its value doesn't influence the classifier accuracy in a statistically significant way; this is due to the fact that the metrics that depend on the threshold are in general not so effective in this setup.

5.5 Discussion

The obtained results show that our work contributes to the research in this field, confirming some past outcomes and contradicting some others. Bird et. al [3] and Greiler et al. [7] showed that ownership metrics are indicative for software quality in Microsoft projects while Foucault et al. [6] showed the opposite regarding OSS projects. Our results confirm the fact that the effect of code ownership on software quality is highly project dependant, but we can consider ownership metrics indicative for software quality also for OSS projects, since for each one of our case studies three or more of them are significant for the classification (see Table 11) and considering them improves a classic code metrics model in a significant way (see Table 9). Our result are in contrast with the ones from Focault et al. [6], and the reasons could be that: (1) we try to classify defective files instead of correlate the metrics with the number of defects; (2) we consider also ownership at a line-based granularity; (3) we consider also memory-less ownership (authorship).

Looking at the metrics significance in Table 11 we can see that *total contributors* and *line authorship* are the only non-classic metrics that are always significant: this confirms the results from Greiler et al. [7] and Rahman et al. respectively. This outcome makes sense, since the two mentioned studies tried to distinguish defective and non-defective artifacts (i.e. Microsoft source files and C language implicated code chunks) with approaches similar to our (i.e. classification). *Commit ownership* was also shown to be important by Bird et al. [3] and Greiler et al. [7], and for us the same holds in three out of five projects. Minor and major con-

Table 11: Results of logistic regression with the metrics with a 5% threshold, see Table 10 for the interpretation.

Metrics	Lucene-Solr	Camel	Mahout	Maven	Zookeeper
file size	***	***	***	***	
previous implications	***	***	***	***	***
comment to code ratio	.	***	***	***	***
commit ownership		***	*	***	.
minor contributors	NA	NA	**	NA	NA
major contributors	***	***	NA	***	***
total contributors	***	***	***	***	***
total authors	***				***
line authorship	***	***	***	***	***
line ownership added		***	***	***	.
lines added minor contributors		NA		NA	NA
lines added major contributors		**	.		**
line ownership deleted	.	***	***		.
lines deleted minor contributors	NA	NA		NA	NA
lines deleted major contributors	*		*	**	
Accuracy	**0.6325**	**0.65**	**0.6925**	**0.6425**	**0.59375**

Table 12: Major contributors metrics with different granularities and the influence of the threshold for different projects

Threshold	Metrics	Lucene-solr	Camel	Mahout	Maven	Zookeeper
5%	major contributors	***	***	NA	***	***
	lines added major contributors		**	.		**
	lines deleted major contributors	*		*	**	
10%	major contributors	***				***
	lines added major contributors	**	***		**	***
	lines deleted major contributors				***	
20%	major contributors	***	**	**	***	.
	lines added major contributors	***		.		
	lines deleted major contributors			***	***	
30%	major contributors	***		*		
	lines added major contributors	***	***	**		
	lines deleted major contributors			***	***	
40%	major contributors	***		**	*	
	lines added major contributors	***	***	**		.
	lines deleted major contributors			*	***	

tributors are highly dependant from each other (reasonable, since they sum to the *total contributors*): for every project one of them is discarded, while the other one results significant, confirming what stated by Bird et al. [3].

6. THREATS TO VALIDITY

Like every empirical study, this one too has threats to validity that must be considered.

External validity: All the five subjects of this study are Apache OSS projects that use JIRA with the same convention and are mainly programmed in Java (we only consider Java files). Because of that it could be the case that for different scenarios the described technique leads to different or contrasting results (e.g. on proprietary software or considering a different programming language).

Construct validity. Not all the commits that mention a JIRA key and id couple are necessarily fixes to the related bug. Furthermore, in a bug fix we consider all the deleted or changed lines to spot the corresponding implicated code, but not all the code touched is necessarily related to the bug fix. The fact that single commits often include unrelated changes was already identified as the problem of Tangled Changes [9], and it is currently an active research topic. This work also includes all the JIRA related threats, like the fact that some developers could not adhere to the bug convention.

When determining the implicated files we do not consider the fact that a file version can be implicated more than one time. We do that because ultimately our goal is to determine when a file is defective and not to predict how many defects will it introduce, but taking into account that factor could be important: in Lucene-Solr, for example, in the 27.3% of the cases a file is implicated more than once.

To build our dataset we discard the first commits, the ones that set up the repository, but we consider all the other outliers because we assume that unusual commits are often the ones that lead to defects; this could have biased some of our results. We also considered the whole available development history for every project, and this could lead some metrics to converge, especially the line based ones: a developer could be marked as a major contributors even if there are no more lines authored by him in the file. Further investigation with different time windows should be done. We also do not perform any distinction between test and non-test Java files, but it could be interesting to consider this factor.

<MISSING: Check the division between validity classes (external, construct, content etc.)>

7. RELATED WORK

A number of prior studies focused on code ownership and its relationship with software quality. Bird et al. [3] first

Table 13: Average performance of minor-major threshold dependent metrics

Projects	OOB (0.05)	OOB (0.10)	OOB (0.20)	OOB (0.30)	OOB (0.40)
Lucene-Solr	44.60%	43.35%	43.60%	44.99%	46.17%
Camel	39.15%	37.75%	39.14%	40.26%	41.23%
Mahout	41.37%	39.89%	39.67%	43.63%	42.83%
Maven	38.25%	38.89%	36.66%	40.81%	42.00%
Zookeeper	33.44%	33.72%	34.82%	36.93%	34.73%

examined this topic by defining ownership as a proportion of code contribution, and of minor and major contributors. Their results show that if a Microsoft Windows code artifact does not have a well-defined owner, then it is more likely to be defect prone, and the same holds if a lot of minor contributors have worked on it.

Focault et al. [6] replicated the above study on seven open-source projects. They used the same granularity for the metrics and the same threshold to distinguish minor and major contributors while changing only the code artifacts on which the study was focused (Java files and packages). The outcome is contrasting with the previous results: it shows no strong correlation between ownership and defects, but it states that it is more significant when the metrics are computed on more coarse-grained artifacts. In contrast, Rahman et al. [14] confirmed Bird et al.'s findings by examining the concept at a finer granularity (line-level) and on open source software.

Another replication of the study from Bird et al. was recently performed by Greiler et al. [7], including also the intuition from Focault et al. of changing the granularity of the code artifacts: they used folders and files. This study was again targeted on Microsoft projects and confirms the result of [3]. They show that it is also possible to classify with a high precision defective files using the ownership metrics.

In this paper, we took a different approach; we investigated how risky a commit can be depending on its author's ownership. Thus, here we measure ownership-quality relationship at commit granularity, as opposed to release granularity like our predecessors. To build our model we adapted the concept of *implicated code* by Rahman et al. [14] (see Section **??**). The concept of implicated code was also used in other bug prediction literature including Sliwersky et al. [17], Purushothaman et al. [13], Ray et al. [16], etc. ***Todo 7-1:*** *Bray: Review related work might have some relevance here.* ***Todo 7-2:*** *Bray: Blend this later.* A number of prior studies also tried to use a different granularity to compute the ownership metrics or to perform bug prediction using line-based approaches: Munson et al. [12] introduced the concept of code churn as a measure of code line changes, Meng et al. [10] considered fine-grained code changes over-time to measure the authorship in an accurate way and Hata et al. [8] computed ownership at a method level for bug prediction.

For what concerns the bug linking technique, D'Ambros et al. [5] described a method to identify bug-fixes using information from the JIRA and Bugzilla issue tracking systems: our technique is simpler but has more requirements (i.e. JIRA and the Apache convention, see Section 3.2).

8. CONCLUSION & FUTURE WORK

In this paper we extended the past studies of the effect of code ownership on software quality. We used the concept of implicated code to identify defective file versions over the development history of five open-source software project, and we built a model capable of distinguishing them from the non-defective ones. Our model is built using the Random Forest technique and it includes an exhaustive set of ownership metrics, considering different granularities (line-based and commit-based) and different thresholds to distinguish minor and major contributors, together with some authorship (memory-less ownership) and classic code metrics.

All the metrics are computed with a novel approach that ensures that defective files are really characterized in the revision where defective code is introduced.

This classifier with all metrics reached an average OOB of the 23% over the considered projects, with a relative improvement of the 22% over a model that consider only classic code metrics. Our results also show that ownership metrics computed with line-based granularity are more effective than the commit-based ones and that changing the threshold used to distinguish minor and major contributors doesn't affect the results in a significant way.

Future research should be done to consider more projects with programming languages different from Java, and to study how changing the granularity of the considered artifacts (e.g. folders or packages instead of files) and the history time period affects the results.

9. ACKNOWLEDGMENTS

We thank Alberto Bacchelli for his advices and feedback during the research and Tommaso dal Sasso for his help with the JIRA issues JSON data.

10. REFERENCES

[1] Financial content: Cambridge university study states software bugs cost economy $312 billion per year. http://insight.jbs.cam.ac.uk/2013/financial-content-cambridge-university-study-states-software-bugs-cost-economy-312-billion-per-year/. Accessed: January 26, 2016.

[2] Software errors cost u.s. economy $59.5 billion annually. http://www.abeacha.com/NIST_press_release_bugs_cost.htm. Accessed: January 26, 2016.

[3] C. Bird, N. Nagappan, brendan murphy, H. Gall, and P. Devanbu. Don't touch my code! examining the effects of ownership on software quality. In *Proceedings of the the eighth joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ACM, September 2011.

[4] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

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

[6] M. Foucault, J.-R. Falleri, and X. Blanc. Code ownership in open-source software. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, pages 39:1–39:9, New York, NY, USA, 2014. ACM.

[7] M. Greiler, K. Herzig, and J. Czerwonka. Code ownership and software quality: A replication study. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, Piscataway, NJ, USA, May 2015. IEEE.

[8] H. Hata, O. Mizuno, and T. Kikuno. Bug prediction based on fine-grained module histories. In *Proceedings of the 34th International Conference on Software Engineering*, pages 200–210. IEEE Press, 2012.

[9] K. Herzig and A. Zeller. The impact of tangled code changes. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 121–130. IEEE, 2013.

[10] X. Meng, B. P. Miller, W. R. Williams, and A. R. Bernat. Mining software repositories for accurate authorship. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 250–259. IEEE, 2013.

[11] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 181–190. IEEE, 2008.

[12] J. C. Munson and S. G. Elbaum. Code churn: A measure for estimating the impact of code change. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 24–31. IEEE, 1998.

[13] R. Purushothaman and D. E. Perry. Towards understanding the rhetoric of small changes-extended abstract. In *International Workshop on Mining Software Repositories (MSR 2004), International Conference on Software Engineering*, pages 90–94. IET, 2004.

[14] F. Rahman and P. Devanbu. Ownership, experience and defects: A fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 491–500, New York, NY, USA, 2011. ACM.

[15] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 432–441, Piscataway, NJ, USA, 2013. IEEE Press.

[16] B. Ray, V. Hellendoorn, Z. Tu, S. Godhane, A. Bacchelli, and P. Devanbu. On the "naturalness" of buggy code. 2016.

[17] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.

[18] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. 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*, ESEC/FSE '09, pages 91–100, New York, NY, USA, 2009. ACM.