Ownership and Defects in Open-Source Software

Eric Camellini
Delft University of Technology
Delft, The Netherlands
E.Camellini@student.tudelft.nl

Kaj Dreef
Delft University of Technology
Delft, The Netherlands
K.Dreef@student.tudelft.nl

ABSTRACT

<still missing>

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 corrections has a great impact on the economy: it costs tens of billions of dollars every year and 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, 12, 6, 13, 9, 15]. 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 [9, 13]. These past results showed that the developers behaviour has more impact on software quality then 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 [6].

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 dependant 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. [5] the same metrics do not result in the same kind of relationship when computed on Open-Source Software (OSS) artifacts. Foucault et al. [5] also introduced the idea of computing ownership metrics on artifacts of different granularities (Java files and packages). Greiler et al. [6] applied the same principle on different Microsoft Products, and their results showed that 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 how do they correlate with 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 er al. [12] (also called *fix-inducing* code [14]). 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 technique [4] (4) measure how the ownership metrics improve the model that uses only the classic ones.

Our results show that (1) (2) ...

We think that our approach can capture the characteris-

The 13th Working Conference on Mining Software Repositories '16 Austin, Texas USA
© 2016 ACM. ISBN .
DOI:

tics of the defective code in a better way, leading to more generalizable outcomes.

<MISSING: outline of the paper>

2. THE PROBLEM

The general problem that is targetted 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 [15]. 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 [6]. 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.

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, 5, 6]. What makes the problem tricky is that code ownership highly depend 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, the first study 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;
- *Minors*: number of contributors with a proportion of ownership that is below a certain threshold;
- *Majors*: number of contributors with a proportion of ownership that is above the threshold used for the minors;
- *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 modified 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 [5, 6], 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 limitations:

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.

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 the third, are addressed computing more ownership 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.

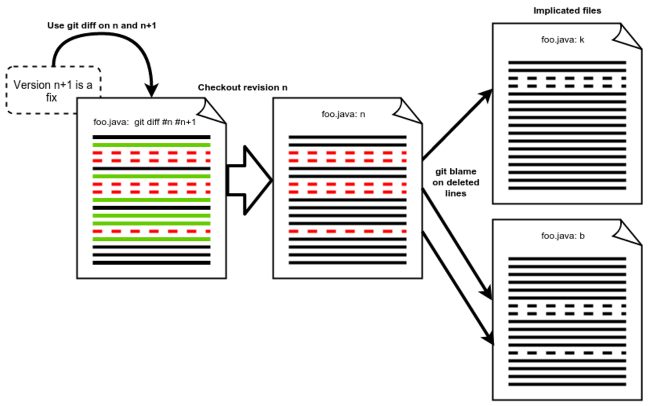
We focus on computing the ownership metrics considering the source files as target artifacts; we are first interested in determining which versions of the files follow the introduction of defects. To spot the introduction of defects we use the concept of *implicated code.*

3.1 Implicated code

We refer to the concept of *implicated code* as described by Rahman et al. [12]: "Implicated code is code that is modified to fix a defect". Rahman et al. also describe a technique to identify the implicated code (see Figure 1). The steps to do it 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 version of the file it was introduced: the same code in that version is the implicated code;
4. If that code was introduced after the fixed bug was reported, then mark it as *innocent* (not implicated).

Figure 1: Implicated code identification technique



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

In order 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 and in particular the bug convention that Apache projects use on it. Every Apache software projects 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 <CITE THE STUDY THAT EXTRACTED THE JIRA ISSUES IN JSON FORMAT>

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) for three different kind of variables: commit count, number of lines added and number of lines deleted. We also use five different thresholds to distinguish minor and major contributions: 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 another kind of ownership that we define as *authorship*. Authorship can be seen as memoryless 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. 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.

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 for the ownership metrics:

- 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 file are the implicated ones. This means that the classification model be built to 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 see 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). <CITE A WORK THAT SAYS THAT THESE ARE KNOWN TO BE EFFECTIVE>

4. METHODOLOGY

4.1 Research Questions

We structure our research through the following research questions:

1. **Are ownership metrics indicative for the presence of implicated code in source files for open-source software projects?**
2. **Can ownership metrics be used to build a classification model to identify implicated source files?**
   (a) **For which level of granularity (line-based or commit-based) do ownership metrics give more accurate results when used to build the classification model?**
   (b) **How does the value of the threshold, used to distinguish minor and major contributors, impact 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 files based on ownership metrics. The sub-questions focuses on parameters of the classifier like which granularity (line or based) gives the best results, and if the minor-major threshold plays a significant role in classifying.

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[1] as version control system, or have a git mirror of the repository; (3) use JIRA[2] as issue tracking system and adhere to the Apache JIRA convention[3], so that we can apply the technique described in Section 3.2;

<DO WE NEED TO MENTION THAT THEY NEED TO BE IN THE SET OF PROJECTS FOR WHICH THE ISSUES WERE ALREADY AVAILABLE IN JSON?>

We selected five different software projects as study subjects (see Table 1), 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 (see Table 1).

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 wanted only to consider code for which the most of the discovered 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 projects 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 build 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);

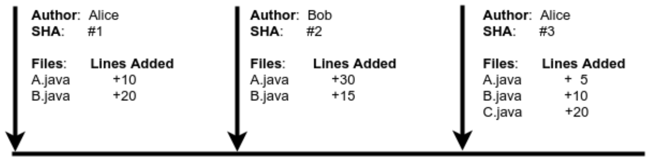
4. add to the data set, for each one of these files, as many lines as the number of authors that contributed to it until the considered commit (included): each one of these lines contains the information about the contribution of the corresponding author to the corresponding file, at the moment of the 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 3. Some of them contain information 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 in terms of commits, with lines added information. The corresponding history dataset is shown in Table 2



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 3).

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" as already described in Table 3. A complete set of the columns of the metrics dataset can be seen in Table 4, 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). In Table 5 an example is shown of the metrics dataset resulting from the history dataset example shown in Table 2 using a threshold of 30% (with 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],

[1]http://git-scm.com/
[2]https://www.atlassian.com/software/jira
[3]https://issues.apache.org/jira/secure/BrowseProjects.jspa#all

Table 1: Characteristics of the studied project

Project	LOC	Implicated files	Contributors	Commits	Application	Period
Camel	2552376	7076	219	13992		Integration framework
Lucene-solr	2458078	14416	73	16066		Search engine server/library
Mahout		2114				library of scalable machine-learning algorithms
Maven		2383				Build manager
Zookeeper		819				open-source server

Table 2: Example of a small portion of history dataset with a reduced set of columns (see Table 3). The development history is shown in Figure 2

file	sha	col. 4	col. 5	col. 6	col. 7	col. 11	col. 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

and we then evaluate how effective it is when used to distinguish which file versions are implicated. An accurate description of this process is provided in Section 5.

5. EVALUATION

5.1 Experiment 1: Ownership granularity

5.1.1 *Experiment design*

To answer research questions 1 and 2a we need to determine what the influence is of ownership granularity on the classification model. The goal of this experiment is to determine if there is a significant improvement by using line-based instead to commit-based granularity.

Table 6: Sample size per project

Project	Sample size
Lucene-solr	4000
Camel	4000
Mahout	2000
Maven	2000
Zookeeper	800

The experiment consists of two parts. First, a classifier is build with the random forest classifier. It is trained and validated with the dataset that was previously constructed for that project. From this dataset an even amount of samples from implicated and implicated code were sampled. Based on the size of the project and number of implicated files the sample size was determined, see Table 6. For this experiment multiple classifiers are trained based on the same data, but different features.

Second, from the constructed classifiers the out-of-bag error rate (OOB) for a group of features is derived. In Table 7 the features are shown representing the group. With the different OOB it is possible determine if the improvement is statistically relevant.

Verifying if the performance gain is indeed statistically significant will be done by taking the group of interest and comparing them against the classic metrics with the t-test. The result will be a p-value and if this value is below the `0.05` it can be concluded they differ in a statistically significant way.

Table 7: Groups and which metrics belong to it

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 above mentioned metrics minus the highly correlated metrics

5.1.2 *Results*

Table 3: 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

In Table 8 and 9, some of the results for experiment 1 can be found. For the other projects similar results are found with an average of 23.3%, which is an increase of 36% over just using the classic metrics. For line granularity versus commit-granularity, line-granularity performs on average 17% better than commit-granularity.

Table 8: Mahout results with different metrics

Features	**OOB with 5% threshold**	**Improvement over CLASSIC**
CLASSIC	32.39%	0.00%
CLASSIC + COMMIT BASED	30.65%	5.66%
CLASSIC + DELETED	29.42%	10.10%
CLASSIC + ADDED	29.88%	8.41%
CLASSIC + LINE AUTHORSHIP	29.26%	10.68%
CLASSIC + LINE BASED	26.15%	23.84%
ALL FEATURES	26.11%	24.04%

To verify the statistical significance of the results the t-test was used. Here the classic metrics (our base case) is compared against one of the other groups. In Table **??**, the p-values can be found. here it can be noted that all the

Table 9: Mahout results with different metrics

Features	**OOB with 5% threshold**	**Improvement over CLASSIC**
CLASSIC	29.50%	0.00%
CLASSIC + COMMIT BASED	19.96%	47.81%
CLASSIC + DELETED	17.15%	71.98%
CLASSIC + ADDED	16.08%	83.42%
CLASSIC + LINE AUTHORSHIP	16.22%	81.85%
CLASSIC + LINE BASED	14.32%	106.01%
ALL FEATURES	14.65%	101.33%

p-values are below 0.05, meaning all the improvements are statistically significant. One of the resulting ROC curves can be found in Figure 3

5.2 Experiment 2: Minor-major thresholds

5.2.1 Experiment design

Research question 2b. has to do with the minor-major threshold and how much influence it has on the performance of those metrics. The goal of the experiment is to deter-

Table 4: Columns of the metrics data set, defined using the columns of the history dataset listed in Table 3. See Section 3.3 for a definition of the key terms.

#	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 5: Example of a small portion of metrics dataset with a reduced set of columns (see Table 4), extracted from the history dataset example shown in Table 2 using a 30% threshold to distinguish minor and major contributors.

sha	file	col. 3	col. 4	col. 6	col. 11	col. 12	col. 13	col. 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

mine the influence of the different thresholds. Five different thresholds will be tested, namely `0.05`, `0.10`, `0.20`, `0.30`, and `0.40`.

To be able to determine the influence of the minor-major threshold 5 datasets are created as described in Section 4.3.1 where the metrics (lines added/deleted by major/minor contributor, and major and minor contributor) are computed with the different minor-major threshold. For this experiment multiple classifier will be created based on the random forest algorithm and trained and validated with the different datasets, resulting that for every project five different classifiers are built with the different thresholds. From the constructed classifier we then derive multiple OOB's.

To determine if the 5 groups of OOBs with different minor-major thresholds will differ in a statistically significant way an *ANOVA* test will be applied to the groups.

5.2.2 Results

In Table 10, the out-of-bag error rate for different project can be found with five different minor-major thresholds. As can be seen is that the threshold doesn't give a clear increase in performance. As mentioned before, to determine if the results are statistically significant the outcomes are subjected to an ANOVA test, which will tell if the outcomes are statistically significant. The ANOVA test showed that there is no significant outcome, meaning that major-minor threshold doesn't have any significant impact on the performance.

5.3 Summary of results

Based on the results from experiment 1 and 2 we can say that ownership metrics are indeed a good indication for the presence of implicated code. Over the 5 projects we got an average accuracy of 23.3%.

The level of granularity is for a classifier important. The line based metrics do give a significant improvement with on average 17% over commit-based ownership. For the thresholds we determined that changing the thresholds doesn't have any statistically significant influence on the performance of minor-major metrics.

6. THREATS TO VALIDITY

1. First commit of the projects are extremely large (400.000+ LOC) without any information of authors etc.

2. A file in a certain commit can be multiple times implicated. We do not take this into consideration, because ultimately the goal is to classify the data as buggy or not (this happens in 27.3% of the implicated file cases in Lucene-solr).

3. Open source project vs commercial software can maybe

Table 10: 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%

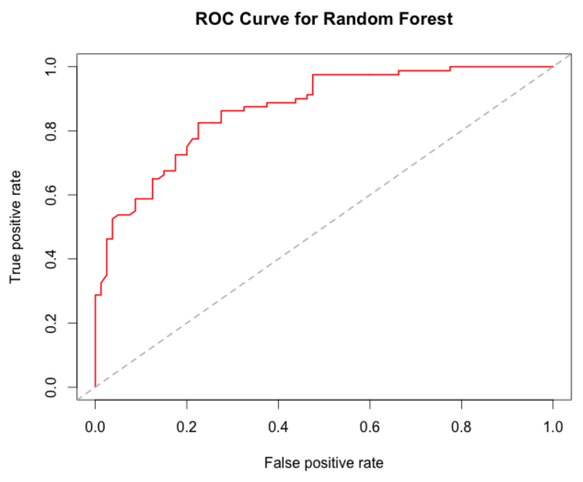


Figure 3: ROC curve of lucene-solr with all metrics and threshold of 0.05

not be compared to each other because of different organizational structure.

4. Results can be related to the specific work method of Apache projects that work with JIRA.

7. RELATED WORK

A number of prior studies focused on code ownership and its relationship with software quality. Bird et al. [3] first examined this topic defining the concepts of ownership as proportion of contribution, and of minor and major contributors. These are the concepts that we use in this work, but with a different granularity and on different type of software artifacts. Their results shows that if a Microsoft Windows code artifact does not have a well defined owner then it is more defect prone, and the same holds if a lot of minor contributors have worked on it.

Rahman et al. [12] examined the effects of ownership and authorship on software quality using a fine-grained approach and computing their metrics on chunks of implicated code. Our approach to determine software artifacts that contain implicated code is based on that work. They report findings similar to the ones reported by Bird et al. and described above. However, they consider ownership in a different way and use different metrics.

Focault et al. [5] replicated the study Bird et. al [3] on seven open-source projects, but using the same granularity for the metrics and the same threshold to distinguish minor and major contributors, and 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 metrcs are computed on more coarse-grained artifacts.

Another replication of the study from Bird et al. was recently performed by Greiler et al. [6], 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.

None of the studies that consider ownership as intended in this work [3, 5, 6] tried to compute it on artifacts that contain implicated code.

The concept of implicated code was used also by more previous works, for different purposes and with different names. Changes that introduce code that causes a fix are called fix-inducing by Sliwersky et al. [14] and dependencies by Purushothaman et al. [11].

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. [10] introduced the concept of code churn as a measure of code line changes, Meng et al. [8] considered fine-grained code changes over-time to measure the authorship in an accurate way and Hata et al. [7] computed ownership at a method level for bug prediction.

<MISSING: related work about bug-linking with commits or revisions, other related work if needed>

8. CONCLUSION & FUTURE WORK

9. ACKNOWLEDGMENTS

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 9, 2016.

[2] Software errors cost u.s. economy $59.5 billion annually. http://www.abeacha.com/NIST_press_release_bugs_cost.htm. Accessed: January 9, 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. 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.