It 's Not a Bug, It 's a Feature: How Misclassification Impacts Bug Prediction

Xindi Li
North Carolina State University
xli76@ncsu.edu

ABSTRACT

In a manual examination of more than 7,000 issue reports from the bug databases of five open-source projects, we found 33.8% of all bug reports to be misclassified - that is, rather than referring to a code fix, they resulted in a new feature, an update to documentation, or an internal refactoring. This misclassification introduces bias in bug prediction models, confusing bugs and features: On average, 39% of files marked as defective actually never had a bug. We discuss the impact of this misclassification on earlier studies and recommend manual data validation for future studies.

KEYWORDS

Mining software repositories, bug reports, data quality, noise, bias

1 INTRODUCTION

In empirical software engineering, it has become common- place to mine data from change and bug databases to detect where bugs have occurred in the past, or to predict where they will occur in the future.

If one wants to mine code history to locate or predict error prone code regions, one would therefore only consider issue reports classified as bugs. However, all this assumes that the category of the issue report is accurate. In 2008, Antoniol et al. [1] raised the problem of misclassified issue reports - that is, reports classified as bugs, but actually referring to non-bug issues. If such mix-ups occurred frequently and systematically they would introduce bias in data mining models threatening the external validity of any study that builds on such data. But how often does such misclassification occur? And does it actually bias analysis and prediction?

From five open source projects, we manually classified more than 7,000 issue reports into a fixed set of issue report categories clearly distinguishing the kind of maintenance work required to resolve the task

2 MANUALLY CLASSIFYING BUG REPORTS

We selected the five projects such that we cover at least two different and popular bug tracking systems: Bugzilla and and Jira. Three out of five projects (Lucene-Java, Jackrabbit, and HTTPClient) use a Jira bug tracker. The remaining two projects (Rhino, Tomcat5) use a Bugzilla tracker.

To validate the issue categories contained in the project 's bug databases, we manually inspected all 7,401 issue reports and checked if the type of each report reflects the maintenance task the developer had to perform in order to fix the corresponding issue.Each issue report was then categorized into one of eleven different issue report categories shown in Table II.

TABLE I
PROJECT DETAILS.

	Maintainer	Tracker type	# reports
HTTPClient	APACHE	Jira	746
Jackrabbit	APACHE	Jira	2,402
Lucene-Java	APACHE	Jira	2,443
Rhino	MOZILLA	Bugzilla	1,226
Tomcat5	APACHE	Bugzilla	584

TABLE II
THE ISSUE REPORT CATEGORIES USED FOR MANUAL CLASSIFICATION.

Category	Description
BUG	Issue reports documenting corrective maintenance tasks that require semantic changes to source code.
RFE	Issue reports documenting an adaptive maintenance task whose resolving patch(es) implemented new functionality (request for enhancement; feature request).
IMPR	Issue reports documenting a perfective maintenance task whose resolution improved the overall handling or performance of existing functionality.
DOC	Issue reports solved by updating external (e.g. website) or code documentation (e.g. JavaDoc).
REFAC	Issues reports resolved by refactoring source code. Typically, these reports were filed by developers.
OTHER	Any issue report that did not fit into any of the other categories. This includes: reports requesting a back-port (**BACKPORT**), code cleanups (**CLEANUP**), changes to specification (rather than documentation or code; **SPEC**), general development tasks (**TASK**), and issues regarding test cases (**TEST**). These subcategories are found in the public dataset accompanying this paper.

3 AMOUNT OF DATA NOISE

As overall noise rate we measured the false positive rate. The false positive rate represents the ratio between misclassified issue reports and all issue reports in the data set.

Over all five projects researched, we found 42.6% of all issue reports to be wrongly typed. Every third bug report is no bug report. The noise rate slice for bug reports is of great importance.

Foundations of Software Science, Fall 2018, Computer Science
© 2018 North Carolina State University
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$9999.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

TABLE IV
NOISE RATES FOR ALL PROJECTS AND FOR A COMBINED DATA SET.

Project	Noise rate
HTTPClient	47.8%
Jackrabbit	37.6%
Lucene-Java	46.4%
Rhino	43.2%
Tomcat5	41.4%
All projects combined	42.6%

TABLE V
RECLASSIFICATION OF REPORTS

(a) Reports originally filed as **BUG**

Classified category	HTTPClient	Jackrabbit	Lucene-Java	Rhino	Tomcat5	combined
BUG	63.5%	75.1%	65.4%	59.2%	61.3%	66.2%
RFE	6.6%	1.9%	4.8%	6.0%	3.1%	3.9%
DOC	8.7%	1.5%	4.8%	0.0%	10.3%	5.1%
IMPR	13.0%	5.9%	7.9%	8.8%	12.0%	9.0%
REFAC	1.7%	0.9%	4.3%	10.2%	0.5%	2.8%
OTHER	6.4%	14.7%	12.7%	15.8%	12.9%	13.0%
Misclassifications	36.5%	24.9%	34.6%	40.8%	38.7%	33.8%

Bug reports are one of the most frequently used instruments to measure code quality when being mapped to code changes.

4 SOURCES OF MISCLASSIFICATION

We suspect the main reason to be that users and developers have different views and insights on bug classification, and that classification is not rectified once a bug has been resolved.

Bug tracking systems are a communication tools that allow users to file bug reports that will be fixed by developers. But users and developers might not share the same perspective regarding the project internals. In many cases, users have no project insight and might not have the ability to understand technical project details. Users tend to consider every problem as a bug.

Also, using their default configuration, many bug tracking systems set the report type to BUG by default. Combined with the problem of who assigns the bug report type, we are left with many BUG reports that should have been filed as improvement request or even feature request.

5 IMPACT ON MAPPING

For a data miner the question is whether these misclassification sources impact issue reports that can actually be mapped to source code changes.

For this purpose, we followed the issue report mapping strategy described by Zimmermann et al. [2], a mapping method frequently replicated by studies. Scanning through commit messages, we detect issue report identifiers using regular expressions and key words. Once we mapped issue reports to revision, we can identify the set of issue reports that caused a change within the source file. Additionally, we count the number of distinct issue reports that were classified as BUG during manual inspection. We measure the issue mapping bias using five different bias measurements.

An average falseDefectRate of 39% shows that mapping bias is a real threat to any defect prone classification model.

TABLE VI
IMPACT OF MISCLASSIFIED ISSUE REPORTS ON MAPPING STRATEGIES AND APPROACHES

Measure	HTTPClient	Jackrabbit	Lucene-Java	Rhino	Tomcat5	Average
MappingBiasRate (False positive rate for mappable **BUG** reports)	24%	36%	21%	38%	28%	29%
DiffBugNumRate (How many files will change their defect-prone ranking?)	62%	17%	14%	52%	39%	37%
MissDefectRate (How many files with no original **BUG** have at least one classified **BUG**?)	1%	0.3%	0.7%	0%	38%	8%
FalseDefectRate (How many files with at least one original **BUG** have no classified **BUG**?)	70%	43%	29%	32%	21%	39%

6 IMPLICATIONS ON EARLIER STUDIES

A. Studies Threatened to Be Biased

As of August 2012, the ACM digital library lists more than 150 published studies citing these two approaches. Zimmermann et al. [2] is a particular important case, as a large number of papers built on the accompanying (now found to be biased) bug data set.

The threat to validity for all these papers is that the bug data set they have been evaluated on contains a mix of bugs and non-bugs. Hence, in their evaluation, they map and predict non-bugs as well as bugs. Users would be generally interested in predicting bugs rather than non-bugs, however; and we now no longer know how these approaches perform and compare when using a data set consisting only of true bugs. This threatens their external validity.

B. Preventing Misclassification Threats

Test cases. Approaches like iBugs validate bug reports using test cases to replicate bugs. This straight-forward filtering mechanism ensures each bug is valid. Conse- quently, studies relying on the iBugs data sets are not affected by issues discussed in this paper.
Code history. Kim et al. uses version control history to verify that applied code changes are actual fixes. This approach solely relies on code evolution and thus is not sensitive to bug database issues. Again, this is a recommended procedure to prevent misclassification.
Automatic classification. Automatic classification models as described by Antoniol et al. [1] can be used to categorize issue reports based on the text of the issue report itself with precision rates between 77% and 82%. Although, constructing classification training sets requires initial human effort, such predictors should quickly reduce the required human interaction.
Rectified Data Sets. Our data sets rectified by manual bug classification are publicly available (Section XII); we encourage their use for further research.

7 CONCLUSION

Our findings suggest widespread issues with the separation of bugs and non-bugs in software archives, which can severely impact the accuracy of tools and studies that leverage such data.

Hence, dealing with bug databases will always require human effort - an investment that, however, pays off in the end.

REFERENCES

[1] M. Di Penta F. Khomh G. Antoniol, K. Ayari and Y.-G. Gueheneuc. 2008. Is it a bug or an enhancement? A text-based approach to classify change requests. *in Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds.* (2008), 23:304 – 23:318.
[2] R. Premraj T. Zimmermann and A. Zeller. 2007. Predicting defects for Eclipse. *in Proceedings of the Third International Workshop on Pre- dictor Models in Software Engineering* (2007), 9–.