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

Kim Herzig
Saarland University
Saarbrücken, Germany
herzig@cs.uni-saarland.de

Sascha Just
Saarland University
Saarbrücken, Germany
just@st.cs.uni-saarland.de

Andreas Zeller
Saarland University
Saarbrücken, Germany
zeller@cs.uni-saarland.de

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.

Index Terms—Mining software repositories, bug reports, data quality, noise, bias

I. Introduction

In empirical software engineering, it has become commonplace 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. The accuracy of such measurements and predictions depends on the *quality of the data.* Therefore, mining software archives must take appropriate steps to assure data quality.

A general challenge in mining is to separate *bugs from non-bugs*. In a bug database, the majority of issue reports are classified as *bugs*—that is, requests for corrective code maintenance. However, an issue report may refer to "perfective and adaptive maintenance, refactoring, discussions, requests for help, and so on" [1]—that is, activities that are unrelated to errors in the code, and would therefore be classified in a non-bug category. 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. Such filtering needs nothing more than a simple database query.

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 (which mostly stem from issue reporters and developers interpreting "bug" differently) occurred frequently and systematically they would introduce *bias* in data mining models threatening the external validity of any study that builds on such data: Predicting the most error-prone files, for instance, may actually yield files most prone to new features. But how often does such misclassification occur? And does it actually bias analysis and prediction?

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

These are the questions we address in this paper. From five open source projects (Section II), 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 (Section III). Our findings indicate substantial data quality issues:

Issue report classifications are unreliable. In the five bug databases investigated, more than 40% of issue reports are inaccurately classified (Section IV)

Every third bug is not a bug. 33.8% of all bug reports do not refer to corrective code maintenance (Section V).

After discussing the possible sources of these misclassifications (Section VI), we turn to the consequences. We find that the validity of studies regarding the distribution and prediction of bugs in code is threatened:

Files are wrongly marked as fixed. Due to misclassifications, 39% of files marked as defective actually have never had a bug (Section VII).

Files are wrongly marked to be error-prone. Between 16% and 40% of the top 10% most defect-prone files do not belong in this category after reclassification (Section VIII).

Section IX details studies affected and unaffected by these issues. After discussing related work (Section X) and threats to validity (Section XI), we close with conclusion and consequences (Section XII).

II. Study Subjects

We conducted our study on five open-source JAVA projects described in Table I. We aimed to select projects that were under active development and were developed by teams that follow strict commit and bug fixing procedures similar to industry. We also aimed to have a more or less homogeneous
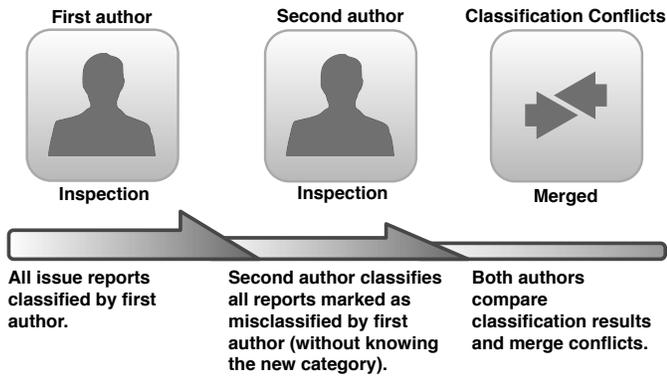
Fig. 1. The manual report inspection process.

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 backport (**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.

data set that eased the manual inspection phase. Projects from APACHE and MOZILLA seemed to fit our requirements best. Additionally, we selected the five projects such that we cover at least two different and popular bug tracking systems: Bugzilla[1] and Jira[2]. 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.

For each of the five projects, we selected all issue reports that were marked as being *RESOLVED*, *CLOSED*, or *VERIFIED* and whose resolution was set to *FIXED* and performed a manual inspection on these issues. We disregarded issues with resolution in progress or not being accepted, as their features may change in the future.

The number of inspected reports per project can be found in Table I. In total, we obtained 7,401 closed and fixed issue reports. 1,810 of these reports originate from the Rhino and Tomcat5 projects and represent Bugzilla issue reports. The remaining of the 5,591 reports were filed in a Jira bug tracker.

III. MANUALLY CLASSIFYING BUG REPORTS

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. For our manual inspections, we used (a) the issue report itself, (b) all the attached comments and discussions, as well as (c) the code change that was applied to the source code. We analyzed code changes if and only if neither the issue report nor its comments clarified the underlying problem of the reported issue. Each issue report was then categorized into one of eleven different issue report categories shown in Table II.

To assign issue reports to one of the categories, we used a *fixed set of rules* that describe how to classify issue reports based on specific issue report properties. If none of these rules applied, and if inspecting possible attached patches did not clarify the original problem, we left the original category unchanged. Hence, we favored possible original misclassification noise over new misclassification noise introduced by manual

[1]http://www.bugzilla.org/
[2]http://www.atlassian.com/JIRA

misclassification. The rule set used for classification is shown in Table III. For each category, we also present a typical real world example.

The manual classification was conducted in three phases as shown in Figure 1:

1) In the first phase, the first author inspected all 7,401 issue reports and assigned a report category using the set of report classification rules.
2) The second author re-classified the set of issue reports that were considered to be misclassified after phase one. Again, the second author was using the fixed set of classification rules and the issue reports only; he had no access to the classification results of the first phase. Overall, 3,093 misclassification candidates got reinspected.
3) We then compared the classification results from phase one and phase two to detect classification conflicts—issue reports that were classified differently by the first and the second author. This affected 340 of the 3,093 reinspected issue reports; the other 94% were independently classified identically by the first and second author and thus validated the accuracy and complexness of the rule set. Each classification conflict finally got resolved by a joint pair-inspection of both authors, partially inducing clarification and refinements of the rule set. (Table III lists the final rule set.)

The first and second phase of the inspection process were processed by one individual each. This ensures that all issue reports across all projects are treated and categorized equally. Every issue report reported as misclassified in this paper was independently verified. We did not double check whether the first author did oversee misclassified reports. This implies that the presented misclassification ratios and impact measurements can be considered as a lower bound. The effort

TABLE III
CLASSIFICATION RULES

A report is categorized as **BUG** (Fix Request) if...

1) it reports a *NullpointerException* (*NPE*).
2) the discussion concludes that code had to be changed semantically to perform a corrective maintenance task.
3) it fixes runtime or memory issues cause by defects (e.g endless loops).

Example: *Tomcat5 report 28147[1] is categorized as* **RFE** *but reports a bug that causes a "JasperException for jsp files that are symbolic links". The underlying issue was that tomcat used canonical instead of absolute paths. The applied fix touches one line replacing one method invocation. According to Rule 2, we classified the applied code change as a corrective maintenance task and thus the issue report as* **BUG**.

A report is categorized as **RFE** (Feature Request) if...

1) it requests to implement a new access/getter method.
2) it requests to add new functionality.
3) it requests to support new object types, specifications, or standards.

Example: *Lucene-Java report LUCENE-2074[2] is categorized as* **BUG**. *But the applied patch and the discussion unveil that a new versioning mechanism had to be implemented. The first comment by Uwe Schindler makes it explicit: "Here the patch. It uses an interface containing the needed methods to easyliy [sic] switch between both impl. The old one was deprecated [...]". This is reclassified as* **RFE** *by Rule 2.*

A report is categorized as **IMPR** (Improvement Request) if...

1) it discusses resource issues (time, memory) caused by non optimal algorithms or garbage collection strategies.
2) it discusses semantics-preserving changes (typos, formatting) to code, log messages, exception messages, or property fields.
3) it requests more or fewer log messages.
4) it requests changing the content of log messages.
5) it requests changing the type and/or the message of Exceptions to be thrown.
6) it requests changes supporting new input or output formats (e.g. for backward compatibility or user satisfaction).
7) it introduces concurrent versions of already existent functionalities.
8) it suggests upgrading or patching third party libraries to overcome issues caused by third party libraries.
9) it requests changes that correct/synchronize an already implemented feature according to specification/documentation.

Example: *Jackrabbit report JCR-2892[3] is filed as* **BUG** *under the title "Large fetch sizes have potentially deleterious effects on VM memory requirements when using Oracle". The algorithm fetches data from a database with a large amount of columns and rows, which caused the Oracle driver to allocate a large buffer. The resolution was to develop a new algorithm consuming less memory. This is an* **IMPR** *according to Rule 1 since no new functionality was implemented and since the program did not contain any defect.*

A report is categorized as **DOC** (Documentation Request) if...

1) its discussion unveils that the report was filed due to missing, ambiguous, or outdated documentation.

Example: *Tomcat5 bug report 30048[4] fixes the problem "Setting compressableMimeTypes is ignored." by "Docs updated in CVS to reflect correct spelling." This is a* **DOC**.

A report is categorized as **REFAC** (Refactoring Request) if...

1) it requests to move code into other packages, classes, or methods.
2) it requests to rename variables, methods, classes, packages, or configuration options.

Example: *Tomcat5 report 28286[5] is filed as* **BUG** *and contains a patch adding a new interface SSOValve. But in comment 4, Remy Maucherat refuses to apply the patch and the idea to introduce a new interface. Instead, he commits a patch that refactors class AuthenticatorBase to allow subclassing. This is a* **REFAC** *as per Rule 2.*

A report is categorized as **OTHER** if...

1) it reports violations of JAVA contracts without causing failures (e.g. "*equals()* but no *hashCode()*").
2) complains about compatibility fixes (e.g. "should compile with GCJ").
3) the task does not require changing source or documentation (like packaging, configuration, download, etc.)

Example: *Lucene-Java report LUCENE-1893[6] complains that "classes implement equals() but not hashCode()". This violated JAVA contracts but does not cause failures. Lucene-Java report LUCENE-289[6] requests "better support gcj compilation". According to our rules this is considered to be an compatibility improvement classified as* **OTHER**.

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%

for the 10,884 inspections was 4 minutes per issue report on average, totaling 725 hours, or 90 working days.

IV. AMOUNT OF DATA NOISE

In this section, we show the amount of data noise and bias (with respect to issue report types) that is evident in the bug databases of the five analyzed projects (see Section II). We start analyzing the issue report data sets by measuring the false positive rates and slicing individual categories to show how many issue reports were misclassified and which categories these misclassified reports belong to. Later, we will discuss the impact and bias rates for data sets that map issue reports to code changes and source files. At the end, we will show how a simple model identifying the most defect-prone files is impacted by misclassified bug reports.

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. The noise rate is independent from individual issue report categories. We will discuss individual categories in Section V. The higher the noise rate, the higher the threat that the noise might cause bias in approaches based on these data sets.

RQ1 *Do bug databases contain data noise due to issue report misclassification, and how much?*

Table IV shows the noise rate values for all five projects and for a combined data set containing the issue reports of all five projects. The noise rates for all projects lie between 37% and 47% and are surprisingly similar. The overall noise rate lies at 42.6%—that is, two out of five issue reports are wrongly typed. This unexpected high ratio raises threats to any approach based on raw issue report data sets.

> *Over all five projects researched, we found 42.6% of all issue reports to be wrongly typed.*

The noise rates of the individual report categories and their variances are shown in Figure 2 as box plot. We excluded the categories **DOC** and **REFAC** from this plot since none of the analyzed bug tracking systems supported these report types.

[1] https://issues.apache.org/bugzilla/show_bug.cgi?id=28147
[2] https://issues.apache.org/jira/browse/LUCENE-2074
[3] https://issues.apache.org/jira/browse/JCR-2892
[4] https://issues.apache.org/bugzilla/show_bug.cgi?id=30048
[5] https://issues.apache.org/bugzilla/show_bug.cgi?id=28286
[6] https://issues.apache.org/jira/browse/LUCENE-1893
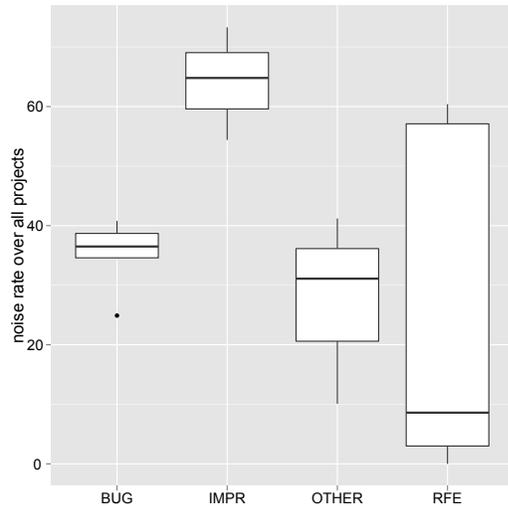[7] https://issues.apache.org/jira/browse/LUCENE-289

Fig. 2. The noise rates split by category over all projects. The categories **DOC** and **REFAC** are not present since none of the bug trackers supports these categories. The Bugzilla projects are only included in the analysis of **BUG** and **RFE** reports since Bugzilla does not support any of the other issue report categories.

The boxes representing the categories **IMPR** and **OTHER** are based on the Jira projects only since Bugzilla does not support these types of reports. The noise rate for **BUG** reports is stable across all projects. Although the noise rates for **IMPR** reports show more variance all projects show comparable noise rates. The variance for **RFE** reports is huge and is partially caused by the fact that the overall number of **RFE** reports is low. Most feature requests have their origin from within the project, especially in open source projects, and it is questionable if or how many such feature requests are documented using a bug tracker.

V. BUGS VS. FEATURES

We have seen that two out of five issue reports are misclassified. And we have seen that there exist misclassified **BUG** reports. This is a threat for all empirical studies based on raw, unchecked bug data sets. To increase the level of detail, we sliced issue categories to show the percentage of issue reports that were associated with a category but marked as misclassified. We also include the individual categories the reclassified bug reports belong to when using our classification rule sets shown in Table III.

RQ2 *Which percentage of issue reports associated with a category was marked as misclassified? Which category do these misclassified reports actually belong to?*

Each slice contains the set of all issue reports originally associated to a given category and shows to which category the individual issue report actually belong to (Tables V(a)— V(c)). Thus, each slice table cell contains the percentage of issue reports originally associated to a given category that were manually classified into the issue category indicated by the row name. The lengths of the bars behind the percentage numbers

represent the individual percentage visually. The last row of each slice states the percentage of reports originally associated to the corresponding category that were assigned a different category during manual classification. The values of this last row correspond to the category boxes in the box plot shown in Figure 2.

A. Bugs

V(a) contains the noise rate slice for the **BUG** issue category. We already discussed in Section IV that the noise rate for **BUG** reports is surprisingly stable across all projects. Tracking bug reports and their target categories shows that 13% of **BUG** reports are manually classified into the **OTHER** category containing multiple sub-categories (see Section III). Between 6% and 13% of filed **BUG** reports are improvement requests and up to 10% contain documentation issues. The fraction of bug reports containing feature requests lies between 2% and 7%. The striking number, however, is that on average 33.8% of all issue reports are misclassified.

> *Every third bug report is no bug report.*

The noise rate slice for bug reports is of great importance. Bug reports are one of the most frequently used instruments to measure code quality when being mapped to code changes. But feature requests, improvement requests, and even documentation issues can also be mapped to code changes implementing a new feature, implementing an improvement, or fixing code comments. Thus, we cannot rely on natural filtering mechanisms that rule out misclassified **BUG** reports belonging to any report category that will not cause code changes being applied to source files. Studies that use bug data sets might be impacted by data noise as shown in V(a). The noise rates in this section include issue reports that might not be mapped to code changes or files. We will discuss the bias caused by bug data noise later in this paper.

B. Feature Requests

The noise rate slice for issues originally categorized as **RFE** is interesting because it shows a fundamental difference between Bugzilla and Jira trackers. As you can see in V(b) the false positive rates for all three Jira projects lie between zero and nine percent. The corresponding false positive rates jump to 57% and 60% for Bugzilla trackers. Interpreting these values, it seems that issue reports in Bugzilla trackers are less reliable than Jira reports. This matches the fact that the false positive rates for **BUG** reports in V(a) where larger for Bugzilla trackers, too. By default, Bugzilla trackers support less issue report types than Jira. This has the consequence that reporters and developers that file issue tickets not being bug reports use the only alternative label **RFE**.

C. Improvement Requests

The last noise rate slice shows how many improvement requests were differently categorized during manual inspection (see V(c)). The columns for Bugzilla tracker projects remain zero since by default Bugzilla trackers do not support these issue categories. For the remaining three projects, between 19%

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%

(b) Reports originally filed as **RFE**

Classified category	HTTPClient	Jackrabbit	Lucene-Java	Rhino	Tomcat5	combined
BUG	0.0%	0.7%	0.0%	3.6%	8.1%	2.8%
RFE	100.0%	91.3%	97.0%	42.9%	39.6%	72.6%
DOC	0.0%	2.0%	0.0%	0.0%	18.1%	5.3%
IMPR	0.0%	0.7%	0.6%	19.0%	20.8%	8.6%
REFAC	0.0%	0.0%	0.0%	15.5%	3.4%	3.2%
OTHER	0.0%	5.3%	2.4%	19.0%	10.1%	7.5%
Misclassifications	0.0%	8.6%	3.0%	57.1%	60.4%	24.7%

(c) Reports originally filed as **IMPR**.

Classified category	HTTPClient	Jackrabbit	Lucene-Java	Rhino	Tomcat5	combined
BUG	2.6%	2.8%	1.8%	0.0%	0.0%	2.3%
RFE	45.3%	18.8%	28.6%	0.0%	0.0%	26.1%
DOC	11.6%	3.7%	7.2%	0.0%	0.0%	6.2%
IMPR	26.7%	45.6%	35.2%	0.0%	0.0%	38.8%
REFAC	4.3%	9.2%	14.2%	0.0%	0.0%	10.9%
OTHER	9.5%	19.8%	13.0%	0.0%	0.0%	29.4%
Misclassifications	73.3%	54.4%	64.8%	0.0%	0.0%	61.2%

and 45% of improvement requests were manually categorized as **RFE** issue reports. Only a very marginal low fraction of 2% were manually classified as bug reports. On average, more than 60% of improvement requests were reclassified during manual inspection.

VI. SOURCES OF MISCLASSIFICATION

The misclassification ratios presented in the last section shed a low light on the data quality of bug databases. But why do bug tracking systems contain so many misclassified reports? 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. If the software does not comply with their expectations or with the provided documentation they file a bug report. It is the *reporter* who assigns an issue category.

On the other side, the developer is the expert of any technical detail of the program; she designed and implemented it. This difference between reporter and resolver already is a source of uncertainty. In contrast to a reporter, a developer certainly has the ability to distinguish between different problems and the required maintenance task to solve the issue. The developer would be the right person to categorize issue reports. But this is not how bug trackers work. Of course, the developer could change the issue category after resolution— but this happens rarely. In many cases there exists no real motivation to change the issue category once the cause for a problem is found and fixed.

We suspect this conceptional problem to explain many of the misclassification patterns we observed during manual inspection. It might also explain the high misclassification noise rates for issue reports originally marked as **BUG**. 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.

The question of whether an issue is a bug or not is a hard one. And the definition of a bug not only differs between users and developers but also between developers themselves. In principal, if an issue bothers the user, the developer should fix it, whether or not he considers it to be a bug or not. From this perspective, the issue report category does not matter. But a data miner building a defect prediction model must distinguish

between bugs and non-bugs. Otherwise, the prediction model would predict changes, not defects.

Thus, for a data miner the question is whether these misclassification sources impact issue reports that can actually be mapped to source code changes. A documentation issue might require to change the documentation in the source file. So a data miner would map the report into source code and count it as a bug fix although the plain source code did not change.

VII. IMPACT ON MAPPING

The issue report misclassification noise presented in Sections IV and V can impact studies and tools that use these or similar data sets without validating them. As a first category, we discuss how misclassified issue reports impact approaches that map issue reports to source code changes—for instance, to identify files that had the most bugs in the past.

RQ3 *What is the impact of misclassified issue reports when mapping issue reports 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. Ignoring report severity we then count the number of distinct issue reports originally classified as **BUG** (*num_original_bugs*) for each source file of a given software project. Additionally, we count the number of distinct issue reports that were classified as **BUG** during manual inspection (*num_classified_bugs*). We measure the issue mapping bias using five different bias measurements.

MappingBiasRate: This bias rate expresses the percentage of false positive original **BUG** reports that could be mapped to code files. The *mappingBiasRate* corresponds to the false positive rates shown in Figure 2 but is limited to **BUG** reports that can be mapped to code changes.

DiffBugNumRate: The *diffBugNumRate* represents the number of files for which

$$num\_original\_bugs - num\_classified\_bugs \neq 0.$$

The measure ignores files for which the set of issue reports differ but not its size. Counting the number of fixes does not require the individual report to be known.

MissDefectRate: The *missDefectRate* is defined as

$$missDefectRate = \frac{numMissDefect}{numZeroOriginalDefect}$$

where *numMissDefect* represents the number of source files for which no original bug report could be mapped but that have at least one manually classified bug report assigned and where *numZeroOriginalDefect* is the number of source files having no original bug report assigned. This measure is important for defect classification models (distinction between has bug or has no bug).

FalseDefectRate: Analog to *missDefectRate*, we compute the *falseDefectRate* as

$$falseDefectRate = \frac{numFalseDefect}{numOriginalDefect}$$

where *numFalseDefect* is the number of source files with at least one original bug report assigned but no manually classified bug report and where *numOriginalDefect* is the total number of source files that got at least one original bug report assigned.

The values of these bias measures for our five target projects are shown in Table VI along with an additional column containing the average bias measures. For all projects, the number of misclassified **BUG** reports that can be mapped to source files (*mappingBiasRate*) lies above 20%. On average, every third bug report that can be associated to code changes is misclassified. This is a threatening high fraction and confirms that the misclassification noise rates presented in Section IV also affect issue reports that can be mapped to source code changes. On average, the *mappingBiasRate* is only five percent points below the average false positive rate for bug reports shown in V(a). The *mappingBiasRate* is also stable across different bug tracking systems indicating that bug tracking systems and their different usage behavior seem to have no impact on the mapping bias.

The second row of Table VI shows the fraction of files having a different number of mapped bug reports. The *diffBugNumRate* shows how many files will change their defect-prone ranking. This value might also have severe consequences for defect prediction models based on concrete bug count numbers (see Section VIII). On average 37% of all source files have biased bug count numbers. For the projects HTTPClient and Rhino the *diffBugNumRate* well exceeds the 50% margin.

Row three and row four of Table VI are interesting for approaches using classification models grouping source files into two groups of defect-prone and non defect-prone entities. The fractions of files that were falsely marked as defect free (*missDefectRate*) is very low and can be disregarded, except for Tomcat5. But the fraction of false classified defect-prone using a threshold of one to distinguish between defect-prone and non defect-prone entities (*falseDefectRate*) is significant. 20% to 70% of the original defect-prone marked source files contained no defect. An average *falseDefectRate* of 39% shows that mapping bias is a real threat to any defect prone classification model.

> *On average, 39% of all files marked as defective actually never had a bug.*

To give some more details on the differences between original and classified bug counts (*diffBugNumRate*), Figure 3 shows stacked bars displaying the distribution of bug count differences among source files. Each stacked bar contains intervals reflecting the difference between the number of original bugs and the number of manual classified bugs (*num_original_bugs* − *num_classified_bugs*). A positive difference indicates that the number of defects fixed in the corresponding

TABLE VI

| Measure	HTTPClient	Jackrabbit	Lucene-Java	Rhino	Tomcat5	Average
MappingBiasRate	24%	36%	21%	38%	28%	29%
(False positive rate for mappable **BUG** reports)						
DiffBugNumRate	62%	17%	14%	52%	39%	37%
(How many files will change their defect-prone ranking?)						
MissDefectRate	1%	0.3%	0.7%	0%	38%	8%
(How many files with no original **BUG** have at least one classified **BUG**?)						
FalseDefectRate	70%	43%	29%	32%	21%	39%
(How many files with at least one original **BUG** have no classified **BUG**?)						
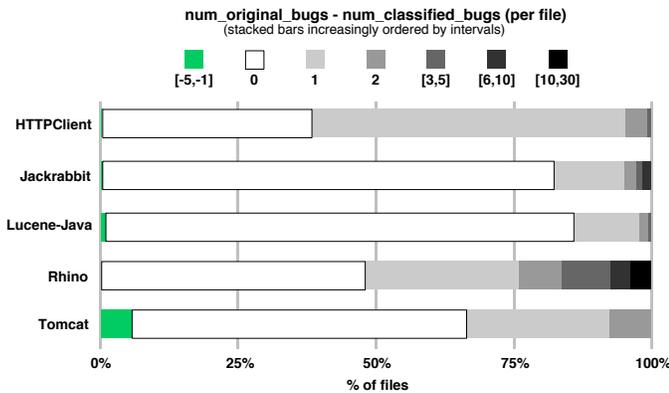
Fig. 3. The stacked bars show the *diffBugNumRates* and their frequencies. A *diffBugNumRate* of zero indicates no changes in the number of associated bugs; a positive diff rate indcates taht too many bug reports were assigned.

Fig. 4. The cutoff_difference for the top x%.

source files is actually lower. For files showing a negative difference more defect fixes could have been found. While most files show no or only little changes to their bug count there also exist files for which we registered up to 26 wrongly counted bugs. The number of files for which more bugs could have been found is marginal.

VIII. IMPACT ON BUG COUNTS

The results presented in the last section indicate that models based on noisy bug data sets containing misclassified bug reports might be severely biased. To verify this threat, we conducted an experiment that uses a simple quality model that identifies the most defect-prone source files by counting the number of distinct bug reports mapped to the corresponding file. If we can show that such a simple bug count model is affected, more complex models based on similar count or classification schemata might be affected too.

RQ4 *How does bug mapping bias introduced by misclassified issue reports impact the TOP 5%, 10%, 15%, 20% of most defect prone source files?*

The experiment to answer RQ4) is visually described in Figure 4. We duplicate the set of source files and sort each copy by two different criteria. One set gets sorted in descending order using the number of original bug reports (*num_original_bugs*). The other set clone gets sorted in descending order using the number of manually classified bug
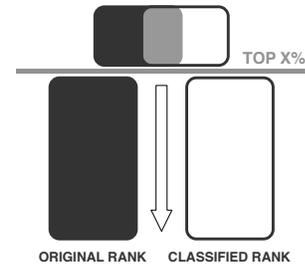
reports (*num_classified_bugs*). In each set, the most defect-prone file is the top element. Comparing the top X% of both file sets (containing the same elements but in potentially different order) allows us to reason about the impact of mapping bias on models using bug counts to identify the most defect-prone entities. Since both cutoffs are equally large (the number of source files does not change, only their ranks), we can define the *cutoff_difference* as:

$$\frac{\text{size of cutoff} - \text{size of intersection}}{\text{size of cutoff}}.$$

The result is a number between zero and one where zero indicates that both cutoffs are identical and where a value of one implies two cutoffs with an empty intersection. A low cutoff_difference is desirable.

Table VII contains the *cutoff_differences* for all five projects using the top 5%, 10%, 15%, and 20%. Considering the top 5% cutoff the *cutoff_differences* lie between 11% and 29% and rise to a range between 16% and 35% for a cutoff size of 20%. The variance between the different cutoff sizes per project lies around 15% for HTTPClient and Tomcat5 and 5% for Lucene-Java and Rhino. The bias measured for all projects and cutoffs lies well above 10%. The results across all projects and cutoff sizes show that quality-measuring approaches using biased bug mappings would report a false positive rate between 16% and 40% for the top 10% most defect-prone files.

> *When predicting the top 10% most defect-prone files, 16% to 40% of the files do not belong in this category because of misclassification.*

Table VIII shows the Spearman rank correlations for all source files in the corresponding intersections. These rank

TABLE VII
THE CUTOFF_DIFFERENCES FOR ALL FIVE PROJECTS.

	TOP 5%	TOP 10%	TOP 15%	TOP 20%
HTTPClient	20%	20%	11%	25%
Jackrabbit	29%	40%	29%	35%
Lucene-Java	24%	20%	21%	18%
Rhino	11%	16%	14%	16%
Tomcat5	14%	21%	29%	21%

TABLE VIII
SPEARMAN RANK CORRELATIONS FOR ALL SOURCE FILES REMAINING IN
THE INTERSECTION OF ORIGINAL AND CLASSIFIED MOST DEFECT-PRONE
ENTITIES.

	TOP 5%	TOP 10%	TOP 15%	TOP 20%
HTTPClient	1	0.8	0.8	0.7
Jackrabbit	0.2	0.3	0.5	0.6
Lucene-Java	0.4	0.5	0.4	0.2
Rhino	−0.1	0.7	0.4	0.4
Tomcat5	0.8	0.5	0.5	0.6

correlations indicate the relative order for files that remain in the top X% most defect-prone source files. A correlation value of one means that there exist files that should not belong to the top cutoff but at least the relative order of the correctly classified files remains stable. Only the rank correlation values for HTTPClient remain above 0.7 for all cutoff sizes. For all other projects, the rank correlations lie below 0.7 (except Rhino top 10%) and reach correlation values close to zero.

> *Misclassification also impacts the relative order
> of the most defect-prone files.*

IX. IMPLICATIONS ON EARLIER STUDIES

The results presented in the previous sections show that misclassified issue reports can affect the assessment and prediction of code quality based on bug data sets. Hence, empirical studies that use or used bug data sets without validating them might suffer from bias.

A. Studies Threatened to Be Biased

Mapping bugs to code changes was first introduced by Fischer et al. [3] and Čubranić et al. [4] who described procedures to map bug reports to code changes. These two approaches do not interpret the mapped artifacts and are per se not threatened; however, any study using one of these approaches to derived code quality measures is likely to be threatened if it did not perform additional data validation.

The list of papers affected encompasses published literature in mining software archives of the past years; 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. Taking the blame, other typical examples with one of us as co-authors are: [5], [6], [7], [8], [9].

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.

The approaches and techniques are still original, present excellent contributions, and can still provide good results. It may even be that removing data noise will improve their results. Assessing such effects for all earlier studies is beyond the scope of this paper; however, we provide rectified data for future studies.

How about studies using industrial bug data? In general, one might hypothesize that industry has stronger process rules and incentives (speak: measurements and goal metrics) that encourage accurate issue classification. This could also explain why predictors such as change bursts [8] or network metrics [10] work extremely well on industrial data sets, but poorly on open source data sets. Exploring the quality of such data sets and differences between industrial and open source projects again is a topic for future research.

B. Preventing Misclassification Threats

How can we improve the quality of bug datasets?

Test cases. Approaches like *iBugs* [11] validate bug reports using test cases to replicate bugs. This straight-forward filtering mechanism ensures each bug is valid. Consequently, studies relying on the *iBugs* data sets are not affected by issues discussed in this paper.

Code history. Kim et al. [12] 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.

We strongly recommend to use additional data (e.g. tests) or human effort to reduce the high amount of misclassified issue reports. This means that the contributions of the mining software archive field can still all be applied; one just needs a bit of validation in the first place.

X. RELATED WORK

Bug reports are a key factor of software maintenance. Many mining approaches are based on bug databases in some way, either as standalone artifact or as a combination between

bug reports and code changes. The quality of bug reports is frequently topic of research studies [13], [14], [15], [16]. Many of these studies show that bug reports often contain too little or too incomplete information in order to reproduce and fix them. It is possible to automatically detect bug report duplicates [17], [18] that, when combined, might fill information gaps that prevent bug report fixes. But bug fixes have also been used to estimate development efforts [19], [20], [21], [22], bug triage [23], [24] or automatic bug assignment [25], [26].

Combining bug reports with source code changes opens the possibility to map bug fixes to individual code artifacts. Fischer et al. [3] and Čubranić and Murphy [27] were among the first that introduced heuristics that allow mapping bug reports to applied code changes. But as Mockus [28] and Liebchen and Shepperd [29] mentioned, data quality in empirical software engineering can be low and might impact the outcome of many empirical studies. Liebchen and Shepperd [29] found that only a tiny fraction of software engineering papers suggest data quality issues and their possible effect on their analysis results. Nguyen et al. [30] reported similar issues on commercial projects that usually follow more strict development guidelines compared to open source projects.

Recent studies investigates bug and commit bias and their potential implication for defect prediction models which might interfere with the results presented in this paper [31], [32], [33]. Their results showed the presence of systematic data noise and bias in several open source data sets affecting the performance of defect prediction models. As a consequence, strategies to improve mappings between bug reports and code changes have been proposed. Bird et al. [34] developed a tool to manually annotate bug reports and code changes to reduce the overhead of manual data point inspection. Later, Wu et al. [35] presented an automatic link recovery algorithm that learns bug report and code change criteria to recover missing links between bug reports and code changes. Kim et al. [36] provided guidelines for acceptable noise levels and proposed a noise detection and elimination algorithm.

In 2008, Antoniol et al. [1] showed that a significant number of bug reports refer to maintenance tasks that are not corrective. In other words, a significant number of bug reports are not documenting software bugs but to other maintenance tasks. This is in particular important since many of the bias reducing approaches are seeking for good and correct mapping strategies but rely on the fact that reported bugs are indeed documenting corrective maintenance tasks. In this paper, we want to extend the research initially conducted by Antoniol et al. [1] by measuring the amount of data noise introduced by misclassified bug reports. We also extend their work by showing the possible impact of misclassified bug report types on bug mapping strategies and machine learning models based on such noisy data sets—such as defect prediction models.

XI. Threats to Validity

Empirical studies like this one have their own threats to validity. We identified three noteworthy threats:

Manual inspection. First and most noteworthy, the manual inspection phase is crucial. To counter the threat of us making classification mistakes, we chose a setup that ensures that every misclassified bug report is cross-validated and that classification conflicts have to pass a third inspection. Still, we cannot rule out that the manual inspection contains errors. Additionally, we make our entire dataset available for independent assessment.

Classification rules. Second, the set of classification rules is only one possibility to classify issue reports. There exists no clear definition-separating feature and improvement requests. Using a different classification rule set will certainly impact the results presented in this paper. We counter this threat by listing the complete rules verbatim.

Study subjects. Third, the projects and bug tracking systems investigated might not be representative, threatening the external validity of our findings. Although Jira and Bugzilla are popular bug tracking systems, we cannot ensure that other projects using the same or even other bug tracking systems contain comparable amount and distribution of misclassified issue reports.

XII. Conclusion and Consequences

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. The consequences are straightforward:

- First and foremost, automated quantitative analysis should always include human qualitative analysis of the input data—and of the findings. Approaches relying on bug datasets should be preceded by a careful manual validation of data quality; at least of a significant sample. Data quality should be discussed as a threat to validity.
- Bug prediction models trained and evaluated on biased data sets are threatened to predict changes instead of bugs. Filtering out non-bugs when estimating code quality might even improve results.
- The categorization of bug reports is dependent on the perspective of the observer. Approaches using bug data sets should validate whether the perspective of the prediction model matches the perspective of the bug creator.

Generally, one should always be aware that not all bugs should be treated equal. Many bugs are of little to no consequence, while a few ones—such as security or privacy issues—can easily damage the reputation of the entire product or even threaten the existence of the company. Assessing such consequences cannot be left to machines alone.

Hence, dealing with bug databases will always require human effort—an investment that, however, pays off in the end. Our motivation for this work was to have a well-classified set of bug reports and features, which we now can leverage (and share) for future research. In the long run, better data will lead to better recommendations that will make developers more conscious of maintaining data quality. Detailed refer-

ences, all data sets (original and rectified), all slices and more information can be found at:

```
http://www.softevo.org/bugclassify
```

References

[1] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "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*. ACM, 2008, pp. 23:304–23:318.

[2] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for Eclipse," in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*. IEEE Computer Society, 2007, pp. 9–.

[3] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 2003, pp. 23–32.

[4] D. Čubranić, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A project memory for software development," *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 446–465, Jun. 2005.

[5] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 international workshop on Mining software repositories*. ACM, 2005, pp. 1–5.

[6] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting faults from cached history," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 489–498.

[7] R. Premraj and K. Herzig, "Network versus code metrics to predict defects: A replication study," in *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, 2011, pp. 215–224.

[8] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change bursts as defect predictors," in *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*. IEEE Computer Society, 2010, pp. 309–318.

[9] A. Schröter, T. Zimmermann, and A. Zeller, "Predicting component failures at design time," in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*. ACM, 2006, pp. 18–27.

[10] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 531–540.

[11] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 433–436.

[12] S. Kim, E. J. Whitehead, Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 181–196, Mar. 2008.

[13] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 34–43.

[14] N. Bettenburg, S. Just, A. Schröter, C. Weiß, R. Premraj, and T. Zimmermann, "Quality of bug reports in eclipse," in *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*. ACM, 2007, pp. 21–25.

[15] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 308–318.

[16] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ACM, 2010, pp. 495–504.

[17] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 499–510.

[18] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 461–470.

[19] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in *Proceedings of the Fourth International Workshop on Mining Software Repositories*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 1–.

[20] J. J. Amor, G. Robles, and J. M. Gonzalez-Barahona, "Effort estimation by characterizing developer activity," in *Proceedings of the 2006 international workshop on Economics driven software engineering research*. ACM, 2006, pp. 3–6.

[21] H. Zeng and D. Rine, "Estimation of software defects fix effort using neural networks," in *Proceedings of the 28th Annual International Computer Software and Applications Conference - Workshops and Fast Abstracts - Volume 02*. IEEE Computer Society, 2004, pp. 20–21.

[22] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*. ACM, 2010, pp. 52–56.

[23] D. Čubranić, "Automatic bug triage using text categorization," in *In SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*. KSI Press, 2004, pp. 92–97.

[24] J. Anvik and G. C. Murphy, "Reducing the effort of bug report triage: Recommenders for development-oriented decisions," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 10:1–10:35, aug 2011.

[25] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 361–370.

[26] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, ""Not my bug!" and other reasons for software bug report reassignments," in *Proceedings of the ACM 2011 conference on Computer supported cooperative work*. ACM, 2011, pp. 395–404.

[27] D. Čubranić and G. C. Murphy, "Hipikat: recommending pertinent software development artifacts," in *Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society, 2003, pp. 408–418.

[28] A. Mockus, "Missing data in software engineering," *Guide to Advanced Empirical Software Engineering*, pp. 185–200, 2008.

[29] G. A. Liebchen and M. Shepperd, "Data sets and data quality in software engineering," in *Proceedings of the 4th international workshop on Predictor models in software engineering*. ACM, 2008, pp. 39–44.

[30] T. H. D. Nguyen, B. Adams, and A. E. Hassan, "A case study of bias in bug-fix datasets," in *Proceedings of the 2010 17th Working Conference on Reverse Engineering*. IEEE Computer Society, 2010, pp. 259–268.

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

[32] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links: bugs and bug-fix commits," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 97–106.

[33] T. H. Nguyen, B. Adams, and A. E. Hassan, "A Case Study of Bias in Bug-Fix Datasets," in *2010 17th Working Conference on Reverse Engineering*. IEEE Computer Society, 2010, pp. 259–268.

[34] C. Bird, A. Bachmann, F. Rahman, and A. Bernstein, "Linkster: enabling efficient manual inspection and annotation of mined data," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 369–370.

[35] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: recovering links between bugs and changes," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 15–25.

[36] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *Proceeding of the 33rd international conference on Software engineering - ICSE '11*. New York, New York, USA: ACM Press, 2011, p. 481.