Chapter 6
Mining Bug Data

A Practitioner's Guide

Kim Herzig and Andreas Zeller

Abstract Although software systems control many aspects of our daily life world, no system is perfect. Many of our day-to-day experiences with computer programs are related to software bugs. Although software bugs are very unpopular, empirical software engineers and software repository analysts rely on bugs or at least on those bugs that get reported to issue management systems. So what makes data software repository analysts appreciate bug reports? Bug reports are development artifacts that relate to code quality and thus allow us to reason about code quality, and quality is key to reliability, end-users, success, and finally profit. This chapter serves as a hand-on tutorial on how to mine bug reports, relate them to source code, and use the knowledge of bug fix locations to model, estimate, or even predict source code quality. This chapter also discusses risks that should be addressed before one can achieve reliable recommendation systems.

6.1 Introduction

A central human quality is that we can *learn from our mistakes*: While we may not be able to avoid new errors, we can at least learn from the past to make sure the same mistakes are not made again. This makes software bugs and their corresponding bug reports an important and frequently mined source for recommendation systems that make suggestions on how to improve the quality and reliability of a software project or process. To predict, rate, or classify the quality of code artifacts (e.g., source files or binaries) or code changes, it is necessary to learn which factors influence code quality. Bug databases—repositories filled with issue reports filed by end users and developers—are one of the most important sources for this data. These reports of open and fixed code quality issues make rare and valuable assets.

K. Herzig (✉) • A. Zeller
Saarland University, Saarbrücken, Germany
e-mail: herzig@cs.uni-saarland.de; zeller@cs.uni-saarland.de

M.P. Robillard et al. (eds.), *Recommendation Systems in Software Engineering*,
DOI 10.1007/978-3-642-45135-5_6, © Springer-Verlag Berlin Heidelberg 2014

In this chapter we discuss the techniques, chances, and perils of mining bug reports that can be used to build a recommender system that suggests quality. Such systems can predict the quality of code elements. This information may help to prioritize resources such as testing and code reviews. In order to build such a recommendation system, one has to first understand the available content of issue repositories (Sects. 6.2) and its correctness (Sect. 6.3). The next important step is to link bug reports to changes, in order to get a quality indicator, for example, a count of bugs per code artifact. There are many aspects that can lead to incorrect counts, such as bias, noise, and errors in the data (Sect. 6.4). Once the data has been collected, a prediction model can be built using code metrics (Sect. 6.5). The chapter closes with a hands-on tutorial on how to mine bug data and predict bugs using open-source data mining software (Sect. 6.6).

6.2 Structure and Quality of Bug Reports

Let us start with a brief overview discussing the anatomy and quality of bug reports. We will then present common practices on mining bug data along with a critical discussion on bug mining steps, their consequences, and possible impacts on approaches based on these bug mining approaches.

6.2.1 Anatomy of a Bug Report

In general, a bug report contains information about an *observed misbehavior* or *issue* regarding a software project. In order to fix the problem, the developer requires information to reproduce, locate, and finally fix the underlying issue. This information should be part of the bug report.

To provide some guidance and to enforce that certain information be given by a bug reporter, a bug report is usually structured as a *form* containing multiple required and optional fields. A bug report can thus be seen as a collection of *fields* dedicated to inform developers and readers about particular bug properties. The value of each field usually classifies the observed issue with respect to a given property or contributes to the description or discussion of the underlying issue. Figure 6.1 shows the structure of a typical bug report. Fields include the following:

- Information on the *product*, version, and *environment* tell developers on which project and in which environment the issue occurs.
- The *description* typically contains instructions to reproduce the issue (and to compare one's observations against the reported ones).
- Fields such as *issue type* (from feature request to bug report), *assignee*, and *priority* help management to direct which bug gets fixed by whom and when.

Create Issue
Configure Fields

Project* mozkito
Issue Type* Bug
Summary*
Priority Major
Due Date
Component/s
Start typing to get a list of possible matches or press down to select.
Affects Version/s
Start typing to get a list of possible matches or press down to select.
Fix Version/s
Start typing to get a list of possible matches or press down to select.
Assignee Automatic
Assign To Me
Reporter* Kim Herzig
Start typing to get a list of possible matches.
Environment
For example operating system, software platform and/or hardware specifications (include as appropriate for the issue).
Description
Original Estimate (eg. 3w 4d 12h)
The original estimate of how much work is involved in resolving this issue.
Remaining Estimate (eg. 3w 4d 12h)
An estimate of how much work remains until this issue will be resolved.
Attachment Choose Files no files selected
The maximum file upload size is 10.00 MB.
Labels
Begin typing to find and create labels or press down to select a suggested label.
Create another Create Cancel

Fig. 6.1 Sample bug report and common bug report fields to be filled out when creating a new bug report

Typically, bug reports allow discussion about a particular issue. This discussion can but may not include the reporter. Comments on bug reports usually start with questions about an issue and the request of developers to provide additional information [15]. Later, many comments are dedicated to discussions between developers on possible fixes and solutions. This shows that bug-tracking systems should be seen primarily as a *communication platform*—first between bug *reporters* and *developers*, later between developers themselves. The reporter is usually the person that observed and reported the problem. She can be a developer (especially

when considering bugs reported before the software has been released) but might also be an end-user with varying degree of experience. Usually, the assignee is a developer that should be considered an expert who can verify the validity of an issue and knows how to resolve the underlying issue or at least which developer the report should be assigned to.

When mining issue repositories, it is important to realize that the different bug report fields and their content are filled by different groups with different expertise or following different usage patterns. Bettenburg and Begel [8] showed that the usage of issue management systems may differ greatly between individual teams and sub-teams leading to problems in understanding bug reports and their background.

Individual fields have a different impact on the bug report and its handling. The importance and impact of individual bug report fields is frequently the subject of research studies. There exists a large degree of regularity on bug report summaries [35] and on questions asked in report discussions between reporters and assignees [15]. Bettenburg et al. [10] and Marks et al. [39] showed that well formulated and easy to read bug reports get fixed sooner. Researchers have shown a similar effect dedicated to other bug report fields. Bug reports with higher priority get fixed quicker [39, 46]; the more people are involved in a bug report, the longer it takes to fix the bug [1]—an important motivation for recommendation systems to automatically determine assignees for bug reports [3, 28, 40, 53]. As bug report descriptions and attached discussions contain natural text, the use of natural language processing becomes more and more important. Natural language can contain important information about related bug severity [37], bug reports [58, 64], affected code [34, 55], etc.

Bug reports evolve over time: Fields get updated, comments get added, and eventually they should be marked as resolved. Thus, mining bug reports at a particular point in time implies the analysis of bug report snapshots. Considering the history of a bug report and frequently updating the analysis results is important. Knowing when and who changed which bug report field can be derived by parsing the history of a bug report and adds additional information allowing to examine a bug report of previous points in time and to capture its evolution. Consider a bug report that got marked as fixed and resolved weeks ago but was reopened recently. Not updating mined artifacts might leave data sources in a misleading state: bug reports once marked as resolved and fixed might be reopened and should be considered unresolved until being marked as resolved again.

It is also common to use values of bug report fields as criteria to filter bug reports of particular interest. To determine code artifacts that were changed in order to fix a bug (see Sect. 6.4), bug data analysts usually consider only bug reports marked as fixed and resolved, or closed [4, 22, 69]. Reports with other statuses and resolutions indicate that the reported issue is either not addressed, has been reopened, or is invalid; thus, the set of changed artifacts is incomplete or might contain false positives. The priority field is often used to filter out trivial bug reports and to restrict the analysis to severe bug reports [18] while fields dedicated to project revisions are frequently used to distinguish between pre- and post-release bugs [12, 52, 69]—bugs filed before or after a product was released.

When mining bug data,

- *Identify the semantics of the individual fields*
- *Identify the individuals who fill out the fields*
- *Use only reports that match your researches (e.g., closed and fixed bugs)*

6.2.2 Influence of Bug-Tracking Systems

In general, all bug reports, independent from their origin, share the purpose of documenting program issues. But individual bug-tracking systems and development processes reflect and create *individual process patterns and philosophies*. Thus, it is important to understand that bug reports across different teams and projects should be considered different, although the differences can be small. But it is essential to identify these small differences as they are important to determine how bug reports get created, handled, used, and finally resolved. Thus, bug-tracking systems impact bug report content.

Depending on the goal of an issue repository analyst, bug-tracking and bug report differences might be relevant or irrelevant. In this section, we briefly discuss important aspects when targeting code quality-related recommendation systems:

Default Values. Creating a new bug report usually requires the bug reporter to fill out a form similar to the one shown in Fig. 6.1. These forms usually populate specific fields (e.g., bug type, bug severity) with default values. These values commonly reflect the typical setting or expected default setting but also help non-expert end-users to fill out all required fields. The drawback is that reporters tend to fill out only those fields that are required and not already set, thus default values can influence the values chosen by reporters [59]. Consequently, the configuration of the issue-tracking system defining which default values to be set may already impact the content of bug reports.

Report Types. Most bug-tracking systems allow not only bug reports but also other types of issues, such as feature requests, improvements, or tasks. Bug-tracking systems have different mechanisms to allow reporters to distinguish between these report types. A good example is the difference between Bugzilla and Jira, two commonly used bug tracking systems. In their standard configurations, Bugzilla supports only bug reports but allows the user to mark reports as enhancement requests using the severity field. In contrast, the Jira tracker not only supports bug and enhancement reports as full types but also offers report types like "task" and "improvement":

- To file an enhancement request instead of a bug report in Jira, the reporter has to set the field `issue type` accordingly.
- To perform the same task in Bugzilla, the reporter has to set the severity field choosing the value `enhancement`.

This unusual mechanism in Bugzilla has two consequences:

- To distinguish between bug reports and enhancement requests, we have to analyze the severity field and not the issue report type.
- Bugzilla does not allow the distinction between high and low severe enhancement requests.

This distinction between bug reports and enhancement requests might also leave many enhancement requests filed as bug reports. Unexperienced reporters might not know to use the severity field to file an enhancement request and relying on the default severity value will automatically mark a report as bug.

Ambiguous Terms. Many fields offer ambiguous terms and vague definitions. In the default configuration, Bugzilla marks bug reports as enhancement requests using the severity field (see above). But the term "enhancement" is ambiguous. Fixing a bug can be seen as an enhancement or improvement but software repository analysts would like to see bug fixes being classified as "bug." It is up to the bug data analyst whether to mark Bugzilla enhancements as feature request, improvement, or any other issue report type. But no matter how he decides, he will most likely end up with noise due to misclassification.

Missing Fields. Bug-tracking systems like Google tracker or SourceForge lack common bug report fields. These fields (e.g., report type, report priority, and affected product) are managed by *labels* instead of explicitly given fields. The advantage is that no default values exist. The disadvantage is that bringing Bugzilla and Google reports to a uniform object model requires detailed knowledge about possible fields and development processes. SourceForge also abandons the report type and forces projects to use different issue management system instances for different report types. While a bug is reported in the issue management system, feature requests are reported in a different tracker. Although an issue repository analyst can consider issue reports in a tracker to belong to the corresponding report type category, it complicates the process of turning a bug report into a feature request, or vice versa. A developer would have to delete the original report, file a new report and transfer properties, fields, and discussion—a process that can be considered to rarely happen. And even if developers would transfer reports between trackers, timestamp values would become unreliable.

Default field settings and predefined report structures impact mined bug data.

6.2.3 Peril of Duplicate Bug Reports

Once a software contains a bug, it is not unlikely that the issue is detected by multiple users. Although the underlying bug is the same, the user experience may vary—a bug can cause multiple crashes and failures. Consequently, it is not

uncommon that a bug gets reported more than once. But are duplicate bug reports harmful? The answer to this question depends on the perspective.

Bettenburg et al. [11] showed that most developers do not consider duplicate bug reports as a serious problem pointing out that duplicate bug reports add additional bug description and details that help developers in their debugging process. Dhaliwal et al. [20] performed an empirical study of crash-reports and showed that grouping bug reports can effectively reduce the fix time of bugs by more than 5 %. In cases in which bug reports provide only high-level details about the issue, developers might benefit from additional reports providing additional details and thus help with progress on that particular issue. On the other hand, duplicate bug reports can cause unnecessary cost because detecting bug reports that should be considered duplicates can be expensive.

From an issue repository analyst's perspective, duplicate bug reports complicate the mining and in particular the analysis processes. Within an issue repository, duplicate bug reports are independent development artifacts. But for the purpose of analyzing bug reports, these independent artifacts should be marked as dependent. There exist a wide variety of approaches to automatically detect duplicate bug reports [56, 58, 62, 64]. Software repository analysts and developers can use these techniques to determine and validate duplicate bug reports. Once these bug reports are marked as duplicates, analysts face the problem of how to aggregate the information: should comments contained in different duplicate reports be seen as one larger discussion group? What priority or severity is to be assumed for a set of duplicate bug reports with different priority and severity values? Which report is the *master report* and should be treated as such [11, 58, 64]? Depending on the purpose of a study, the problem of duplicate bug reports is irrelevant and ignored. For example, when identifying the number of fixed bugs per source artifact, not dealing separately with duplicate bug reports may make sense because code changes and their commit messages refer to only one of the related reports. Thus, related and duplicate bug reports will not be associated with the corresponding code changes (see Sect. 6.4.1) causing no duplicate bug count.

Identify if and how duplicate issue reports should be handled.

6.3 How Reliable Are Bug Reports?

Bug reports play an important role in software maintenance but also in recommendation systems related to code quality. Many mining approaches and recommendation systems are based on issue repositories in some way (e.g., [3, 10, 22, 69]), either as standalone artifact or as measurement for code quality. But how reliable are bug reports? The quality of bug reports is a frequent topic of research studies [9, 10, 22, 27]. Many of these studies show that bug reports often contain too little or incomplete information to reproduce and fix the reported issues. This raises

further questions regarding the correctness of bug reports. If a bug report contains incomplete information, can we expect the data that is available to be correct? Antoniol et al. [2] and Herzig et al. [26] report that there exists a significant amount of incorrectly classified bug reports—reports marked as "bug" but not referring to any corrective maintenance task. Other fields in bug reports have been reported to contain many incomplete or even incorrect data.

The quality of bug reports is an issue not only in open-source projects. Aranda and Venolia [4] showed that even at industry "repositories hold incomplete or incorrect data more often than not." Bachmann and Bernstein [6] confirmed that data quality issues can be an impacting factor in industry datasets and presented a systematic investigation of software process data quality and measures project-specific characteristics that may be used to develop project-specific mining algorithms taking the individual characteristics of software project into account. In a similar study, Bernstein and Bachmann [7] also showed that "product quality—measured by number of bugs reported—is affected by process data quality measures."

6.3.1 Matter of Perspective

Many studies of bug report quality mention a gap between *reporters* and *developers* (at least in cases in which reporters are not developers themselves). We already discussed that the reputation of a reporter heavily impacts the probability that a report gets fixed [10,22,27]. Consequently, submitting bug reports seems to be more complicated than expected. Does a non-developer reporting a bug understand the differences between and the meanings of the required bug report fields? Reporters that are not developers are likely to be neither software nor development experts and thus might not know the difference between a bug and a documentation issue. For a user, a failure or unexpected behavior is a bug. But using the perspective of a user to determine the quality of the source code might cause mismatches. A user who observed bug stemming from outdated documentation does not refer to code issues, although the developer might have to change the documentation in the source file. Thus, mapping the source file change to the "bug" report and thus counting it as a bug fix introduces false bug identifications, because the source code in the source file is kept unchanged. However, since the user determines the issue report type when submitting the issue, the report is submitted as bug report and thus suggests a code issue.

The different perspective of reporters and developers might cause the reporter to select *wrong or misleading values* when filling out the bug report form. Herzig et al. [26] manually inspected over 7,000 bug reports of five open-source projects and found a significant amount of incorrectly classified issue reports. Table 6.1 shows their reclassification results for "bug" reports. Each column of Table 6.1 refers to the "bug" reports of one investigated open-source project. The rows of Table 6.1 represent the categories an issue report could be assigned to during manual

Table 6.1 Reclassification of reports originally filed as "bug". Taken from [26]

Classified category	HttpClient	Jackrabbit	Lucene	Rhino	Tomcat	Combined
bug	63.5%	75.1%	65.4%	59.2%	61.3%	66.2%
feature	6.6%	1.9%	4.8%	6.0%	3.1%	3.9%
documentation	8.7%	1.5%	4.8%	0.0%	10.3%	5.1%
improvement	13.0%	5.9%	7.9%	8.8%	12.0%	9.0%
refactoring	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%

Table 6.2 Fractions of resolved issue reports whose type field got changed

	HttpClient	Jackrabbit	Lucene	Rhino	Tomcat	Combined
reports type changed	1/750	9/2413	1/2683	11/622	57/1242	79/7710
changed to bug	0	2/9	0	0	4/57	6/79
changed to non-bug	1/1	7/9	1/1	11/11	53/57	73/79
misclassified after type change [according to 25]	0	1/9	0	0/11	23/57	24/79

inspection. Thus, each cell of the table contains the proportion of original bug reports and the category these reports were assigned to during manual inspection. 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 reports filed against the investigated open-source projects are misclassified.

Herzig et al. [26] reported similar results for feature requests and improvement requests. Again, the reporter of a bug report might not know the difference between a bug, a feature (adding new functionality), or an improvement request (improving existing functionality)—even among developers, there exist different opinions on when a bug is a bug or when an improvement is a new feature.

Table 6.2 shows the number of issue reports for which the issue report type was changed at least once by a developer. Compared to the fraction of misclassified reports, this fraction is very low and lies between 0.04 % for Lucene and 4.6 % for Tomcat. Combining all five projects, the fraction of issue reports whose type was changed at least once lies at 1 %—in contrast to the combined false classification rate of 30 %. This evidence shows that developers rarely change the type of an issue report. Thus, bug data analysts should not rely on developers to detect and in particular to automatically correct issue report types. Interestingly, there exist many more issue reports being newly marked as non-bugs than reports moved from a non-bug category to "bug."

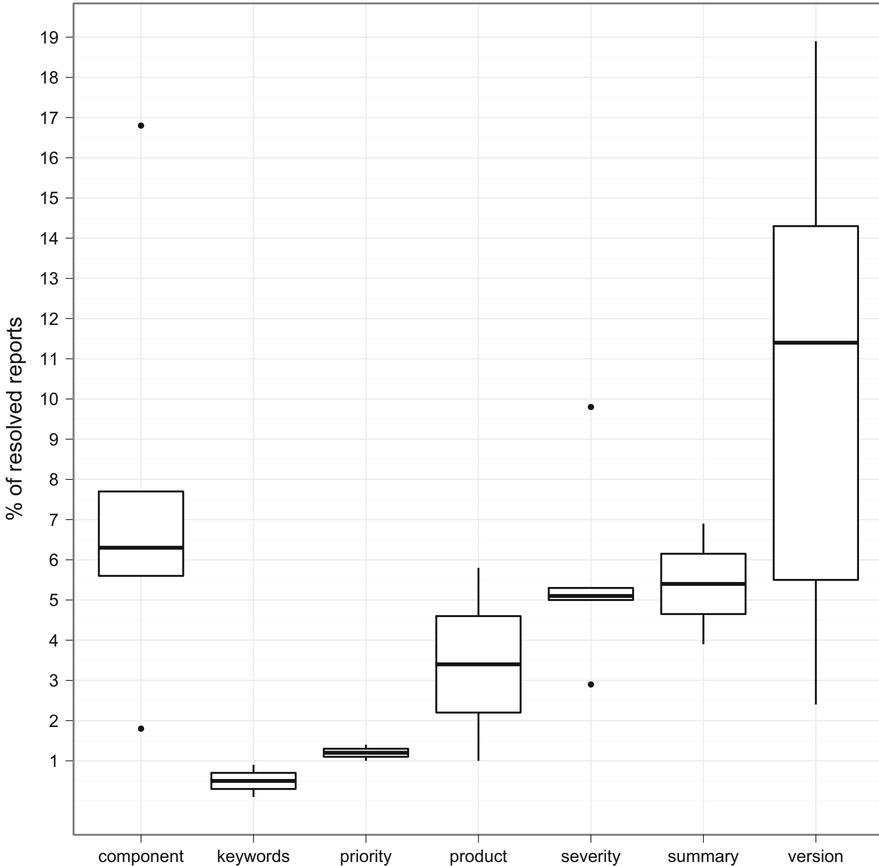

Fig. 6.2 Percent of resolved issue reports with respect to field changes. Priority, product, and summary only changed in Bugzilla tracker projects Rhino and Tomcat

How about other bug report fields? Figure 6.2 shows box plots that represent the fraction of reports for which the corresponding field was changed at least once. The box plot shows that at most 20 % of the resolved issue report fields get updated. Important fields like severity and priority (used to train recommendation systems on severe or important bugs only) are updated for 5 % of the issue reports.

We discuss the consequences of misclassified issue reports on quality-related models in Sect. 6.4.6.

If bugs are classified by reporters, check for possible misclassification.

6.3.2 Recommending Bug Report Fields

There exist multiple approaches to predict the correct values of issue report fields. Antoniol et al. [2] used linguistic data extracted from the issue report content to classify issue reports into bugs and non-bugs with a precision between 77 % and 82 %. Common keywords marking issue reports as bug reports are "exception," "fail," "npe" (null-pointer exception), or "error" [2].

Other studies have shown that it is also possible to predict the severity or priority of issue reports [37, 44, 67], who should be assigned [3, 28, 40, 53], and duplicate bug reports [56, 58, 62, 64]. These and other approaches can be used to verify the correctness of specific report field values but should be used with care to automatically correct issues regarding these fields. Most of these approaches are based on heuristics and should not be used as replacement for careful manual inspection. Manual inspection and quality assurance is key and should be conducted, at least on a significant sample.

It is possible to automatically correct misleading bug report fields.

6.4 Mapping Bug Data

As discussed, there exist many research studies and approaches targeting bug reports as standalone artifacts. Bug reports are key to software maintenance and therefore of great interest to software developers and managers. But in many software projects and companies, bug-tracking systems are separated from version control systems (VCS) and thus do not allow immediate reasoning about code quality and those artifacts that have shown to be defect prone. Thus, we need to relate bug reports with code changes. Once we are able to identify which code changes were made in order to fix which issue, we will be able to reason about code quality in detail, for example, identifying the most defect-prone source artifacts.

Although mapping bug reports to code changes is a common task when mining version archives, there exist surprisingly few fundamentally different mapping strategies and even less studies investigating the correctness of these mapping strategies and their impact on recommendation and prediction models based on these mappings. Recently, researchers investigated whether natural language processing and linguistic data can be used to improve existing mapping techniques. We discuss these approaches in Sect. 6.4.6. However, many mapping strategies are based on regular expressions searching for issue report references in version control commit messages and using a set of negative filters (e.g., bug to be closed and fixed or the bug report not to be marked as resolved before the code changes were committed) to eliminate obvious false positive mappings. But before discussing the consequences and issues regarding strategies to map bug reports and code changes, this section starts with an overview on how to map reported bugs to code changes.

6.4.1 Relating Bugs to Code Changes

Relating bug reports and code changes means finding references between artifacts contained in two separate archives. Fischer et al. [21] and Čubranić et al. [17] were among the first to search for references between code changes in VCSs and bug reports contained in issue-tracking systems. Their approach is straightforward. Commit messages of code changes may contain references to bug reports (e.g., "Fixes bug 7478" or simply "fixing #2367") that are easily found using *regular expressions*. The result is a set of pairs of bug reports and code changes for which the commit message of a code change is suspected to contain a reference of the corresponding bug report. But this set might also contain *false positive* relations. While regular expressions are excellent to detect patterns of text, they *ignore the context.* Thus, a regular expression like `(bug|issue|fix):?\s*#?\s?(\d+)` will match a commit message like "fixing copyright issue: 2002 → 2003." Clearly, the bug ID 2002 matched by the regular expression is referencing a date but not the bug report with the ID 2002. Thus, most bug data analysts apply a set of filters to prevent such false positives to be added to the final association set. Common filters are as follows:

Creation Order. The bug report should have been created before the code change was applied.

Resolution Order. The code change should be committed before the report was marked as "Resolved."

Authors. The person who commits the change should be the same person who marks the report "Resolved" (there exist many development processes in which the fix requires review and the reviewer marks the report as "Resolved").

Software Version. Filters might also consider the affected versions mentioned in the bug reports and the branch(es) where the code change was applied.

Bug ID Ranges. Certain bug report ID ranges are more likely to introduce false positives as others (e.g., small bug IDs are likely to reference a year, date, build, or a line number instead of a report ID). Ignoring such references can be simple but requires software projects with a considerably higher number of reported bug reports than the chosen ignore-threshold or bug-tracking systems with a starting bug id above the threshold. The alternative is to mark these references for manual inspection—a very accurate but also very time-consuming process.

The used regular expressions and filters highly depend on the individual software project and the underlying development process. There may exist clear commit message formatting rules or none. Different projects use different bug report processes. For example, Jira bug reports usually start with a project identifier followed by a number (e.g., JRUBY-2002); this very simple difference can eliminate many false-positive mappings such as the confusion between year numbers and bug report IDs. Depending on these processes, their setups, and the development process, regular expressions and false positive filters should be changed, added, or removed. The overall approach described is illustrated in Fig. 6.3.

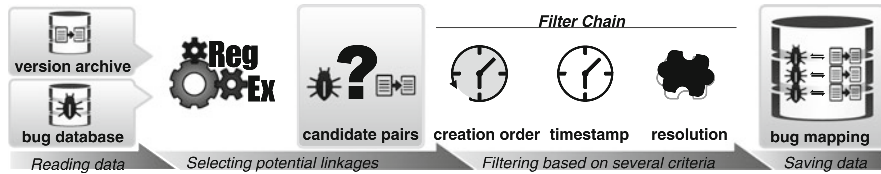

Fig. 6.3 Mapping bug reports to code changes using regular expressions

Mapping bugs to changes requires filtering, which is largely project-specific.

6.4.2 Relating Bugs to Code Artifacts

The result of the above-described mapping process is a set of pairs of bug reports associated with code changes. Each pair suggests that the bug report has been fixed or at least targeted in the associated code change. Assuming that our bug mapping strategy is perfect and thus introduces no false positives (see Sect. 6.4.3), we can use the pairs of bugs and changes to reason about code quality. Each code change touches a number of code artifacts. Mapping the fixed bug reports associated with code changes to those artifacts changed by the change, we can identify those bug reports whose resolution required a code artifact to be modified. Vice versa, for each code artifact we can list the bug reports that caused a change in the artifact. Similarly, we can also identify which developer applied changes required to fix which bug reports or VCS branches in which bug reports were fixed.

The bug report aggregation strategy works if the association between bug reports and code changes assigns each bug report to exactly one code change. But fixing bug reports can be a complicated and time-consuming task. It is common that there exist multiple code changes whose commit message claims to fix a bug report. Looking at commit messages only, it remains undecidable which code change contributes to the overall fix. The only thing we know is that the current last code change is likely to be the last fix and thus likely to be part of the final bug fix. But the contribution of the previous code changes associated to the same bug report remains undecidable. There are multiple possible contributions of these code changes: (a) the code change is part of the final fix but was incomplete, (b) the changes applied were completely reverted by later changes, or (c) the changes applied were partially reverted, thus the code change partially contributed to the final fix. Depending on the actual contribution of the individual code changes, we would have to consider a code change when aggregating quality data or not. To illustrate this, consider the following example (also shown in Fig. 6.4): There exist three code changes (CC_1, CC_2, CC_3) whose commit messages state that bug report `#123` has been fixed. Code change CC_1 gets completely reverted by CC_2 that also applies a patch in $File_B$ that will be part of the initial fix. The changes applied by CC_2 to $File_C$ are again overwritten by changes applied in CC_3. Looking at the individual code changes,

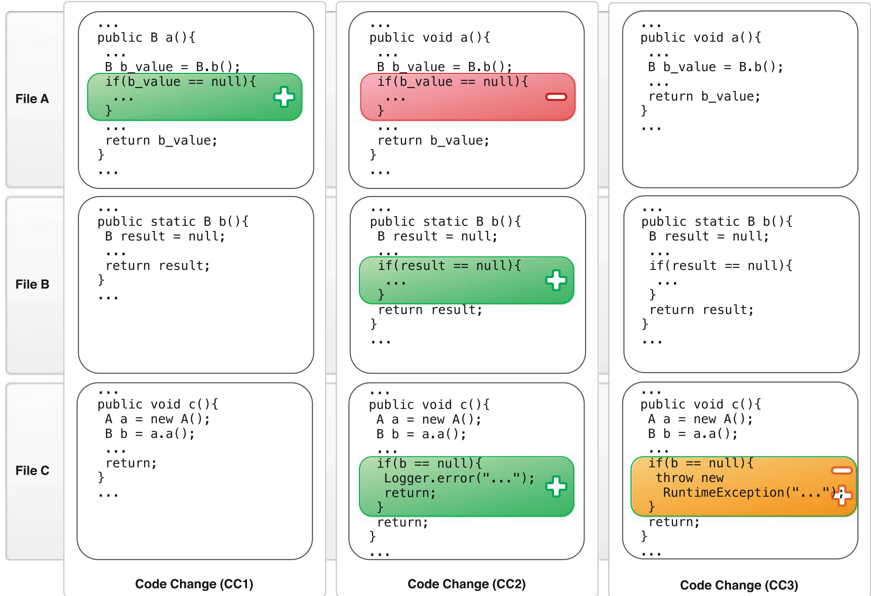

Fig. 6.4 Example of code changes overwriting each other. Which code change contributed to the overall fix?

it is clear that CC_1 does not contribute to the final fix whereas CC_2 and CC_3 do contribute. Thus, the bug report `#123` required changes to the files $File_B$ and $File_C$ but not to $File_A$ although $File_A$ was changed twice.

Many studies and mining approaches do not consider source code analysis and patch aggregation when counting the number of applied fixes per source artifact (e.g., source files of binaries). Two frequently used heuristics are to use either the last code change only or to count the *distinct* number of bugs per changed code artifact. But both heuristics can be dangerous. The rationale behind using the last code change only is that the last change is very likely to contribute to the fix and thus proposes a low risk. But this rational assumes that the last applied change does not revert or clean up earlier unnecessary changes. In these cases, choosing the last code change selects exactly those code parts that should be considered as not changed. Counting the *distinct* number of bug reports per changed code artifact considers all applied changes but still does not identify code changes reverting earlier changes or cleaning up code. For code artifacts that are fixed and cleaned up, each bug ID is counted only once. But code artifacts being changes and later reverted will still be falsely associated to bug reports and thus considered being fixed, although the aggregated patch applied no semantical difference to the artifact. Consequently, there exists no heuristic to aggregate bug reports over a set of code changes without introducing mapping bias. The only safe way is to apply source code analysis and to aggregate the individual patches to create a final summarizing patch and to use this summary patch to detect changed source artifacts.

Mapping code changes to bug reports is a crucial but also error-prone task. It is very important to adjust mapping strategies to individual projects and to verify the correctness of the strategy.

6.4.3 Mapping Bias

Although the described process of relating bug reports and code changes using regular expressions is frequently used in state-of-the-art research studies, there exists concerning evidence that such a simplistic process causes *mapping bias*—an inclination of mapping only bug reports and code changes that fulfill certain criteria. Even in realistic scenarios, data quality is low and some values are even missing [45]. This fact is confirmed by Liebchen and Shepperd [38] who surveyed hundreds of empirical software engineering papers to assess how studies manage data quality issues. Their result is alarming: "[. . .] only 23 out of the many hundreds of studies assessed, explicitly considered data quality" [38] and the issue of noise and missing data is not only limited to studies on relating bug data to code changes but also occurs in software engineering effort and cost estimation [48, 61]. In this section, we discuss error propagations and mapping limitations (the list is not complete) and their impact on quality datasets.

Unmapped Bug Reports

The first problem of any mapping strategy is that it will not find code changes that are bug fixes but state *no references to a bug report* or which state *references in an unrecognized format.* Using regular expressions to detect explicit bug report references (e.g., "1234" or "XSTR-1234") will not cover text references such as using the bug report title as commit message or phrasing the solution of a problem described in the bug report. Thus, regular expressions are too limited to cover all possible reference styles.

Bird et al. [12] showed that a selective mapping strategy, such as using only regular expressions, introduces mapping bias. The mapping strategy determines which bug reports and code changes get mapped and thus selects only those code changes and bug reports that reference each other using a specific manner. Bug reports and code changes using different, not handled, reference styles will be ignored. In their study, Bird et al. showed that certain bug types are over-represented leading to a biased quality dataset that "[. . .] threatens the effectiveness of processes that rely on biased datasets to build prediction models and the generalizable of hypotheses tested on biased data." This effect is not limited to open-source projects but also present in industrial setups enforcing strict development guidelines [54]. It seems clear that bug data analysts should act to reduce the amount of noise and bias introduced by mapping strategies. Possible solutions to this problem can be applied from two different sides of the problem: as pre- or post-processing steps.

Dealing with the problem as post-processing steps requires statistical methods to deal with the noise already introduced. Kim et al. [33] introduced a noise detection and elimination algorithm that eliminates data instances likely to be noise instances. Similarly, Cartwright et al. [16] used simple data imputation techniques to deal with the problem of missing data in software engineering datasets.

Dealing with the noise and bias problem from the other side, as preprocessing step, should be seen as a two-track challenge. Bug data analysts have to come up with less restrictive mapping strategies (see Sect. 6.4.6) that should be combined. On the other hand, software repository analysts have to deal with the data created by others. Empty commit messages or not-existing references cannot be overcome, no matter which strategy will be used. Thus, we also need better tool support allowing software developers to link artifacts with each other [14] and allowing repository analysts to create more reliable and less biased quality datasets.

Mismatched Timestamps

In Sect. 6.4.1, we also used a filtering pipeline to remove false-positive mappings such as bug reports closed before the fix was applied, or code changes applied before the bug report was filed. Some filters use timestamps to determine the order in which code changes and bug reports have been created, applied, and resolved. But timestamps should be handled with care, especially if we compare timestamps recorded by different systems and possibly on different machines and even time zones. A slight offset between timestamps in VCSs and the issue repositories can be fatal. A good example is the OpenBravo project. Their setup of VCS and issue management system showed a timestamp offset of multiple hours over a period of time. The effect is that when mining *OpenBravo* and mapping bug reports to code changes, the described filtering mechanism eliminated many true positive bug mappings. The problem is that many bug reports got created just hours before the actual fix was applied. But the time offset between both servers caused the creation time-stamp of the bug report to appear as being after the commit time-stamp of the actual, manually verified bug fix. Possible solutions would be to allow a certain time gap. But what is a good value for such a time gap? And should this time gap be applied to the complete dataset or only to a particular subset?

Similar problems occur when using timestamps to decide if a bug fix candidate was applied after the referenced bug report was recreated but before the bug was as marked as resolved. Kim and Whitehead showed that “bug fixes times in buggy files range from 100–200 days” [32]. Thus, using a time-based filtering mechanism might be of little help. A time period of 200 days is long and, in active software projects, we can expect many unrelated code changes to be submitted during such long time periods.

Unmapped bug reports and mismatched time stamps can introduce bias in bug data.

6.4.4 *Error Propagation: Misclassified Bug Reports*

Section 6.3 already covered the issue of bug report reliability. In this paragraph, this discussion is continued since unreliable bug reports contribute to mapping bias. More specific, this section discusses the impact of misclassified issue reports when mapping issue reports to source code changes.

Herzig et al. [26] used the mapping strategy described in Sect. 6.4.1 to map bug reports to code changes, once including incorrectly classified bug report and once excluding these noise instances. The authors reported that for all five investigated open source projects, the percentage of misclassified bug reports that could be mapped to code changes and thus to code files lies above 20 %. Thus, more than 20 % of code changes marked as bug fix should not be marked as such since the bug report associated contained no bug description. Going one step further and counting the distinct number of bug reports fixed in a source file (see Sect. 6.4.2), the authors reported that on average 39 % of those source files being marked as having at least one bug never had a reported bug.

To give some more details on the differences between original and classified bug counts, Fig. 6.5 shows stacked bar plots displaying the distribution of bug count differences among source files. Each stacked bar contains intervals reflecting the difference between the original number of bug fixes (*num_original_bugs*) and the number of classified bug fixes (*num_classified_bugs*). A positive difference indicates that the number of defects fixed in the corresponding 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 with large bug count differences. The number of files for which more bugs could have been found is marginal.

Misclassified reports can impact the bug count of source files and wrongly mark bug-free source files as being bug prone.

6.4.5 *Impact of Tangled Changes*

The last important mapping bias source are simultaneously applied code changes that serve multiple development tasks (e.g., fixing a bug and cleaning up code or fixing a bug while implementing a new feature). We call these changes *tangled*. The problem is that it is hard for bug data analysts, mapping bug reports to code changes, to determine which code artifact changed in order to resolve either task. Which code artifacts were changed to fix the bug report and which code artifacts were changed to implement the new feature?

Kawrykow and Robillard [30] investigated over 24,000 code changes of seven open-source projects and showed that up to 15 % of method updates were due to *non-essential differences*. Later, Herzig and Zeller [25] manually inspected and

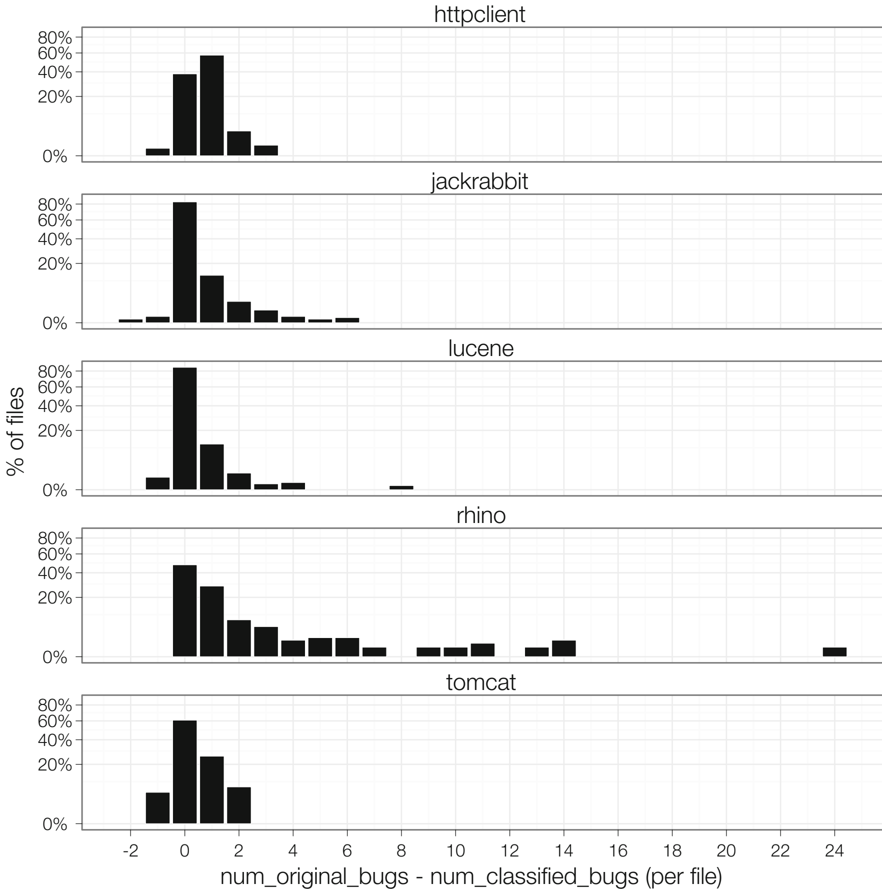

Fig. 6.5 Histograms showing the difference between the original number of bug fixes (*num_original_bugs*) and the number of classified bug fixes (*num_classified_bugs*) and their frequencies across all five projects

classified more than 7,000 code changes from five open-source projects and found that between 6 % and 15 % of all code changes, which contained references to at least one issue report, are tangled. Independently, Kawrykow and Robillard [30] and Herzig and Zeller [25] developed algorithms to separate tangled code changes from each other. The algorithm proposed by Kawrykow and Robillard identified non-essential changes allowing bug data analysts to map bug-fixing code changes only to essentially changed source artifacts. The algorithms proposed by Herzig and Zeller and a similar algorithm proposed by Herzig and Zeller [29] aim to untangle any multi-purpose code change into so-called code change partitions—subsets of applied code changes. Each such change partition contains those change operations likely belonging together. Thus, different change partitions are likely to contain change operations addressing different change purposes.

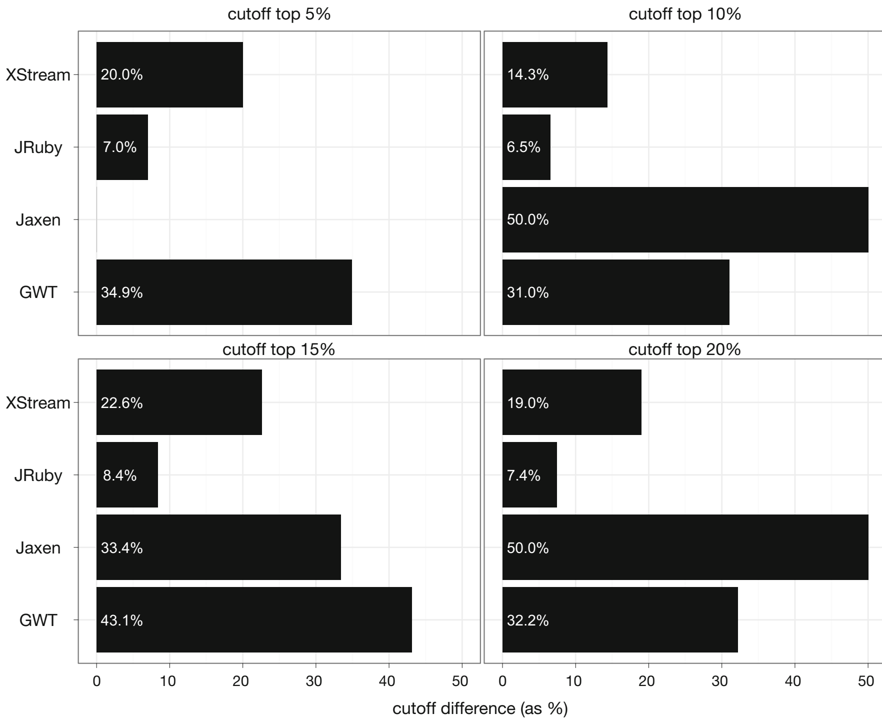

Fig. 6.6 The impact of tangled changes on bug-counting models represented by the percentage of the most defect-prone files that do not belong to this category because they were falsely associated with bug reports

But what is the impact of tangled changes on bug-counting models counting the number of distinct bug reports per source file? Herzig [24] showed this impact by using the untangling algorithm proposed by Herzig and Zeller [25]. He generated two datasets associating source files with the distinct number of bug reports whose resolution required the corresponding source file to change. One dataset contained the original bug fix count as discussed in Sect. 6.4.2. For the second bug count set, Herzig associated bug reports only to those code artifacts modified by change operations located in bug-fixing change partitions. He observed that between 10 % and 38 % of all source files were assigned a different bug count. Between 2 % and 7 % of files originally associated with at least one bug report had no bug associated after untangling. The impact on those files with the most bug counts is even worse. Herzig sorted source files decreasingly once by their original number of associated reports and once by their number of associated bug reports after untangling. He then used the symmetric difference between the two sets containing the corresponding top x% most defect-prone files. The results show that between 6 % and 50 % of the most defect-prone files do not belong to this category because they were falsely associated with bug reports. The detailed comparison results for all four open-source projects can be found in Fig. 6.6. Furthermore, Herzig showed that the Spearman

rank correlations between the files remaining in the intersections of original and classified most defect-prone entities tends to be low: between 0.1 and 1 (median: 0.5; harmonic mean: 0.38).

Tangled changes can severely impact bug count models by associating bug fixes to files that never had a bug.

6.4.6 Alternative Mapping Approaches

Using regular expressions as the only strategy to find references between bug reports and code changes is clearly not good enough and is empirically proven to introduce mapping bias—the subset of bug reports that can be linked to bug-fixing code changes using regular expressions is not representative for the overall set of bug reports being resolved in a software project [12].

In recent years, more and more alternative approaches of finding such artifact references have been developed; Thomas [63] showed that there exists a trend in using topic modeling to trace source code and bug reports. Topic modeling is used to trace code artifacts and bug reports [5] and to specifically search for source files that may be related to bug reports [55]. Wu et al. [66] manually inspected explicit links between bug reports and change logs and extracted characteristic features of bug reports and code changes linking each other. Based on these features, the authors developed a prototype that "[...] automatically learns criteria of features from explicit links to recover missing links." Wu et al. also evaluated the impact of recovered links on software maintainability and defect prediction models and report that ReLink yields significantly better prediction accuracy when compared to traditional linking heuristics. Any mapping strategy linking bug reports and code changes that is not relying on developers to explicitly mention bug report identifiers when committing bug fixes or mentioning code change revisions when closing or resolving bug reports will help bridge the gap between those bug reports that can be linked to code changes and those that cannot be linked. Using natural language processing and topic modeling, we can rely on the individual software repository artifacts themselves.

The alternative to find links between bug reports and code changes retroactive during the mining processes are development environments that link artifacts already during the development process. Many commercial bug tracking, version control, and code review environments and tools follow this strategy. This does not only results in much more precise datasets that can be used to build recommendation systems, but also provides more detailed development information for actual developers, allowing them to instantly switch between development tools and thus navigate fluently through related artifacts. Prominent examples are the commercial tool series from Atlassian, or tools that support an automated mapping between code changes and bug reports based on usage data collections (e.g., Mylyn [31],

Palantír [60], Hipikat [17], Jazz, or Team Foundation Server). Systems like *Team Foundation Server* allow developers to attach related work items (e.g., bug reports) to code changes (and vice versa) using drag and drop. Thus, the developer does not have to manually add links to bug reports in her commit message but rather selects proposed artifacts (based on heuristics) or simply selects these artifacts from a given list of assigned tasks. Although such embedded mapping tools come with their own challenges (dummy bug reports created to commit or incomplete links between artifacts), these environments improve the linkage precision and number of artifacts that can be linked at all dramatically. Although such systems are very common in industry, they are rarely used in open-source projects. Thus, selecting the right project also requires investigating which tools and processes are used in the project.

There exist many approaches that may help to reduce bias in bug data. Using programming environments integrating version control systems and issue management systems can significantly improve mapping accuracy.

6.5 Predicting Bugs

Knowing where bugs were fixed can be helpful and allows to review why artifacts were bug prone and which processes or circumstances led to these issues. Even more important, it allows to learn from these reviewed issues and to learn for future development. One way of leveraging past bugs is to estimate and predict future issues. Such bug prediction models have become popular in research and been adopted in industry. The number of publications on bug prediction models is too large to allow an extensive review of all approaches and findings. For a detailed review on different fault prediction studies in software engineering, we recommend the systematic literature review conducted by Hall et al. [23]. The authors provide answers to research questions: "How does context affect fault prediction?" "Which independent variables should be included in fault prediction models?" "Which modeling techniques perform best when used in fault prediction?" [23].

The goal of this section is to explain how to turn your own mined historic data into a bug prediction model. Along this path, analogous to the previous sections of this chapter, we discuss issues and pitfalls when training bug-prediction models.

6.5.1 Relating Bugs and Code Features

One application of *defect prediction models* is to support decisions on how to *allocate quality assurance resources*—for instance, which components to focus upon during reviewing and testing. The models can help by predicting the *number*

Table 6.3 Overall defect prediction model accuracy using different software measures on Windows Vista [adapted with permission from 50, 52]

Model	Precision	Recall
Change Bursts [52]	91.1%	92.0%
Organizational Structure [51]	86.2%	84.0%
Code Churn [49]	78.6%	79.9%
Code Complexity [41]	79.3%	66.0%
Social network measures [13]	76.9%	70.5%
Dependencies [68]	74.4%	69.9%
Test Coverage [47]	83.8%	54.4%
Pre-Release Defects	73.8%	62.9%

and sometimes the *location* of defects to be fixed in near future. This works because defects are not equally distributed across the code base; therefore, defect prediction models try to locate hot-spots in the system that are more defect prone than others.

Given a set of code artifacts, such a prediction model returns *risk factors* that indicate:

- The likelihood that a given artifact contains software defects (classification)
- Even more precisely, a number of expected defects to be found within the code artifact (prediction)

Most defect prediction models are based on *product metrics* (e.g., for each module, its domain or its code complexity) and *process metrics* (e.g., for each artifact, past defects found, or past changes applied). The model *correlates* these metrics with defect likelihood and can then be used to check new code artifacts expressed by their corresponding software metrics.

Over the years, researchers and engineers proposed hundreds of code metrics that can be used to build defect prediction models. The approach is always the same. The software metrics contains *meta-information* about each individual software artifact (e.g., lines of code per source file or number of authors that changed a source file) that describes code properties separating defect-prone code artifacts from artifacts that are not. The type of meta information can be arbitrary and can also describe process information (who developed the code how) or describe the dependencies between individual code artifacts (e.g., using call graphs). Table 6.3 summarizes the predictive power of post-release defect prediction models for Windows Vista categorized by the type of software metrics the models are based on. The differences in precision and recall measures show that the chosen set of software metrics heavily influences the prediction performance of the corresponding prediction model. Also note that these are numbers for the Microsoft's Windows Vista software product only. Switching to different software products in Microsoft or outside Microsoft might lead to different prediction performances and might also result in different rankings.

To allow machine-learning algorithms to learn which metrics correlate most with defect-prone source files, the dataset to train defect prediction models (and also to check their result and measure accuracy) requires a response or dependent variable

Table 6.4 Excerpt of an example metrics set combining network dependency metrics as described by Zimmermann and Nagappan [68] and the number of distinct bugs fixed per source file

filename	size	sizeOut	sizeIn	density	...	numBugs
optimizer/ClassCompiler.java	12	11	2	0.2651	...	1
optimizer/Codegen.java	30	29	2	0.1678	...	37
JavaAdapter.java	23	22	3	0.2094	...	11
ast/AstRoot.java	11	7	6	0.4	...	0
Parser.java	73	71	5	0.0778	...	33
ast/FunctionNode.java	20	7	17	0.2710	...	1
IRFactory.java	69	67	3	0.0837	...	23
CompilerEnvirons.java	14	5	11	0.2197	...	3
ObjToIntMap.java	15	3	13	0.2667	...	0
ast/ScriptNode.java	24	10	18	0.2536	...	0
ScriptRuntime.java	98	51	72	0.0842	...	41
IdFunctionCall.java	9	4	7	0.375	...	0
Scriptable.java	122	2	121	0.0529	...	0
IdFunctionObject.java	37	8	32	0.1876	...	0
Context.java	148	46	130	0.0484	...	19
ast/XmlString.java	6	4	3	0.4	...	0
ast/NodeVisitor.java	54	2	54	0.0527	...	0
ast/XmlFragment.java	8	4	5	0.3928	...	0
ast/AstNode.java	67	9	64	0.0961	...	1

The path prefix "/org/mozilla/javascript/" has been elided from each

that adds quality-related information per code artifact. Using the approach described in Sect. 6.4.2, we know which code changes fixed bug reports in which source files. Thus, we can count the distinct number of bugs fixed per source file and use this *bug count* as quality measurement—the more bugs were fixed, the lower the code quality. Source files without bug fixes have a *bug count* of zero.

The resulting dataset is a table-like data structure that associates each code artifact with a set of explanatory variables (metrics) and a dependent variable (number of recorded bug fixes). Table 6.4 shows an example dataset for the open-source project Rhino using network dependency metrics as described by Zimmermann and Nagappan [68] and the *bug count* metric as described earlier. The chosen code dependency network metric set is used exemplary and can be replaced or extended by any other metric set that can be collected for source files. For more details on the individual metrics, we refer the reader to the original dataset description [68]. The complete sample dataset as shown in Fig. 6.4 is available as a comma-separated text file (CSV) for download from http://rsse.org/book/c06/sampleset.csv.

In the next section, we use this dataset to model the relationship between the dependent variable and the explanatory variables using machine learners.

Relating software, history, or process metrics with bug fixes allows accurate bug prediction models to be built.

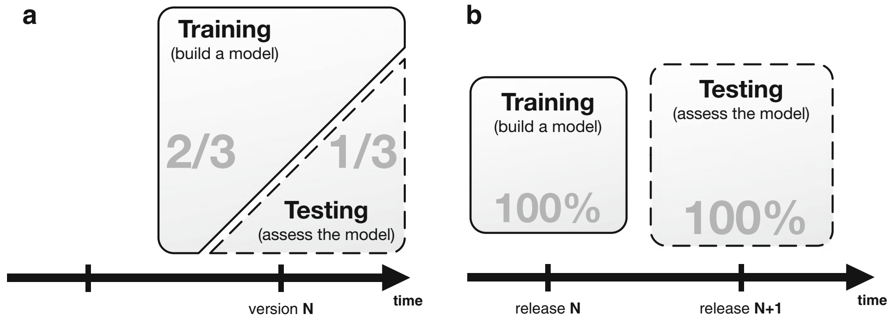

Fig. 6.7 (**a**) Random sampling or stratified random sampling splits one snapshot of a software project into 2/3 training and 1/3 testing parts. Performing multiple splits (holdouts) and reporting mean accuracy measures reduces sampling errors. (**b**) Using two releases or versions of one or different project histories is closest to what can be deployed in the real world where past project data is used to identify defect-prone entities in on-going or future releases

6.5.2 Training Prediction Models

To build and evaluate a bug prediction model, one needs a training and a testing set. Figure 6.7 shows two common approaches to train and test bug prediction models. Randomly splitting a single dataset into two subsets is frequently used if only one revision of a software project is available. The single dataset is split into a training set (usually containing two-thirds of the original set's artifacts) and into a testing set (see Fig. 6.7a). The intersection of the training and testing set is empty while the union of training and testing data matches the original dataset. Sampling datasets includes fuzziness: a single random sample can produce good results although the prediction model performs poorly on average. Thus, sampling is often combined with repeated holdout setups. Instead of splitting once, the dataset gets repeatedly split into training and testing subsets and for each cross-validation or holdout precision, recall, and accuracy are recorded. These measures correspond to the mean values over the corresponding set of performance holdouts.

The alternative of splitting one revision of a software project apart is to use two revisions of the software code base (see Fig. 6.7b). This method is commonly used to train and test prediction models based on releases. The earlier release serves as a training set while the other, later revision, is used to test the prediction model. Models are trained on revisions of different software projects. These forward or cross-release prediction setups are closest to what can be deployed in the real world where past project data is used to identify bug-prone entities in ongoing or future releases.

The training data will then be passed to a machine-learning algorithm (e.g., support vector machine). The resulting model will then accept new instances and returns a predicted value. Prediction models can either be trained as *classification* models or *regression* models. Classification models usually associate instances with a category (bug-prone or not bug-prone) while regression models predict the exact number of bugs to be expected in the corresponding code artifact. There are

many statistical tools and algorithms on how to actually train machine learners for bug prediction purposes and there exist many different machine learners that can be used to predict bugs. Different models may assume different preconditions on the operation dataset and the predictive power of the models not only depends on the used metric sets but also on the machine learner used to predict bugs. In Chap. 3, Menzies [43] discusses this topic in more detail.

6.5.3 From Prediction to Recommendation

Many prediction models and their predicted values can be interpreted as recommendations. The predicted values estimate future events or future situations. This knowledge can be used to take action to support or work against a predicted trend or a predicted result. Bug prediction models predict the expected number of bugs to be fixed in code artifacts. Thus, the prediction results of these models can be used to determine those artifacts that should be tested or reviewed more carefully. Turning a prediction model into a recommendation system usually requires an interpretation of the predicted values and insights that allow to draw possible consequences for the software project.

Discussion the transformation between prediction to recommendation systems goes beyond the content of this chapter but will be discussed in later chapters.

Just as data quality, the interpretation and consequences of predictor and recommendation models should be constantly questioned.

6.6 Hands-On: Mining Bug Repositories

After discussing the foundations (and perils!) of mining, let us now provide some hands-on experience. This section focuses on mining issue repositories, and the next one will focus on how to predict future bugs.

To mine issue repositories, we use the open-source, general purpose mining framework Mozkito. It provides the necessary extraction and parsing functionality required to bring bug reports into a uniform yet powerful format. Out of the box, Mozkito supports the open-source bug-tracking systems Bugzilla, Jira, Google Project Hosting, and others. Adding a new or customized connector requires the user to implement one interface.

The API of the uniform bug data model is shown in Fig. 6.8 as a UML class diagram. The user can decide whether to operate on an SQL database or to use Java objects and the Mozkito framework. The bug data model contains the most common bug report fields including attachments, discussions, and bug report history. For each bug report mined, Mozkito persists exactly one `Report` object in the database that can later be restored (see Step 3) and used for analysis purposes.

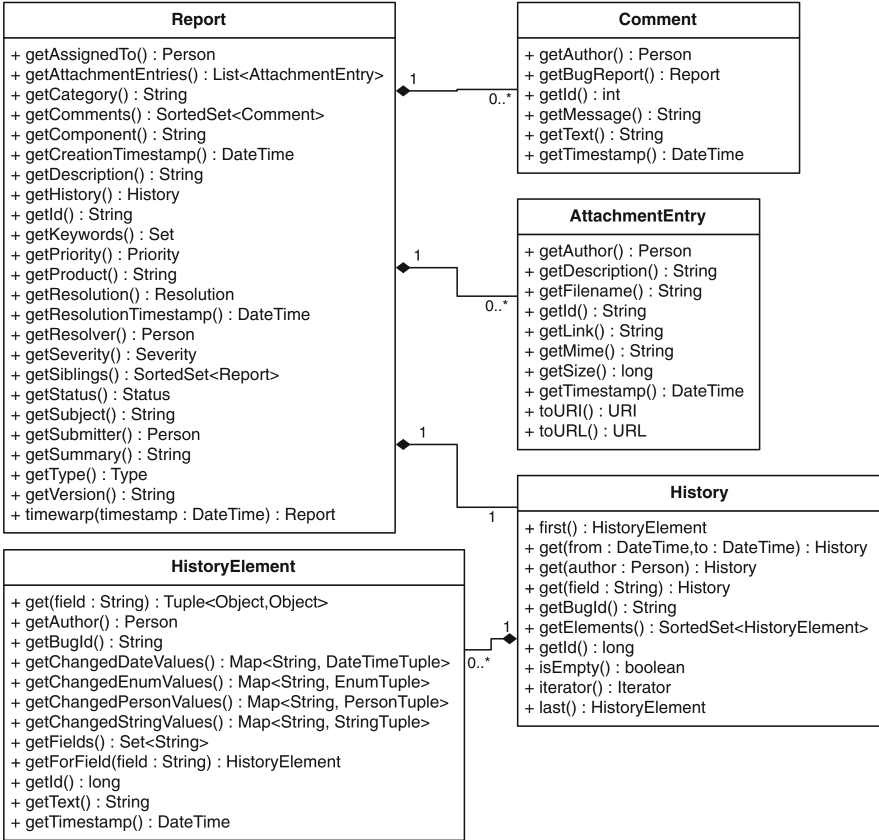

Fig. 6.8 Mozkito bug report model. The UML diagram lists only the most important methods

6.6.1 Step 1: Getting Mozkito

Mozkito is an open-source mining framework. For download and installing instructions, refer to the Mozkito website. To mine issue repositories, we use the Mozkito *issues* module. Once the Mozkito issues module is built (see Mozkito website for instructions), the Mozkito folder

`mozkito-tools/mozkito-issues/target/`

contains the executable jar file that can be used to mine issue repositories (referred to as mozkito-issues.jar for the sake of brevity):

`mozkito-issues-`<*version*>`-jar-with-dependencies.jar`[1]

[1] Replace <*version*> with the downloaded version number of Mozkito.

```
 1: #### DATABASE PROPERTIES ####
 2: database.host=localhost
 3: database.driver=org.postgresql.Driver
 4: database.name=moskito_rhino
 5: database.user=<USER>
 6: database.password=<PASSWD>
 7: ############################
 8: #### BUG TRACKER PROPERTIES ####
 9: tracker.type=BUGZILLA
10: bugzilla.overviewURI=https://bugzilla.mozilla.org/buglist.cgi
    ?product=Rhino
11: tracker.uri=https://bugzilla.mozilla.org
```

Fig. 6.9 Mozkito-Issues configuration to mine the publicly available issue management system for the Mozilla product Rhino

6.6.2 Step 2: Mining an Issue Repository

To demonstrate how to use Mozkito-Issues to mine issue repositories, we will mine the publicly available issue management system of Mozilla and focus on project Rhino—a Javascript engine written in Java. The restriction to project Rhino is for demonstration purposes, only.

Mozkito-Issues can be configured using JVM arguments. To get a list of all available Mozkito arguments (required arguments will be marked) execute:

```
java -Dhelp -jar mozkito-issues.jar.
```

The configuration of Mozkito-Issues depends on the target bug-tracking system and the issue management system URL. Mozilla uses the bug-tracking system Bugzilla that can be accessed using the issue management system URL: https://bugzilla.mozilla.org. Figure 6.9 summarizes the used Mozkito arguments as Mozkito configuration file (`<config_file>`). To let Mozkito use the configuration file, simply start Mozkito specifying the *config* JVM argument:

```
java -Dconfig=<config_file> -jar mozkito-issues.jar.
```

Line 10 of the configuration (Fig. 6.9) specifies the target product (in our case Rhino). The configuration file also contains the required database connection properties that will be used to persist the uniform data format. The listed configuration requires a PostgreSQL database running on localhost. Most likely, these settings need to be adapted to fit a given environment (e.g., MySQL and different host name).

Depending on the size and speed of the issue repository, it may take several hours for Mozkito to fetch and mine all reports found in the target bug-tracking system. Once the mining process completed, the specified database should be populated with persisted `Report` instances (see Fig. 6.8), one for each bug report found in the mined bug-tracking system. Bug reports requiring additional permissions or that cause parsing errors will be dropped during the mining process.

```java
public int analysis() {
  int result = 0;

  /*
   * please use the Mozkito documentation to see
   * how to create PersistenceUtil instances
   */
  final PersistenceUtil persistenceUtil =
   this.databaseArguments.getValue();

  final Criteria<Report> loadCriteria =
   persistenceUtil.createCriteria(Report.class);

  final List<Report> allReports = persistenceUtil.load(
      loadCriteria);
  for (final Report report : allReports) {
   final History reportHistory = report.getHistory();

    // we are only interested in HistoryElements
    // changing the report type
   final History reportTypeHistory = reportHistory.get("type");
   if (!reportTypeHistory.isEmpty()) {
     ++result;
    }
  }
  return result;
}
```

Fig. 6.10 Sample source code analyzing the history of issue reports counting the number of reports for which at least one history entry changing the report type can be found

6.6.3 Step 3: Analyzing Bug Reports in Java

Once the content of the target issue management system is persisted, we can use Mozkito to analyze the mined issue reports. Figure 6.10 shows Java source code that loads the mined issue reports from the database and analyzes the report's history. The purpose of the program is to investigate for how many issue reports the report *type* was at least once changed. In other words, we want to analyze how many issue reports were filed as bug reports but resolved as feature or improvement request, or vice versa.

The `PersistenceUtil` class (line 8 in Fig. 6.10) of Mozkito can be used to load persisted objects from the database into your program.[2] Once we load the `Report` instances, we iterate over them (line 15) and check for the report history concerning the report type (lines 16 and 20). If this history of modifications applied to the report type is not empty (line 21), we find a report whose report type is changed at least once. We discussed the result of this particular analysis in Sect. 6.3.

[2]Please see the Mozkito documentation on how to create such a `PersistenceUtil` instance.

The presented sample code demonstrates how easy bug report analysis can be once we transformed the bug-tracking content into a uniform, persisted data format. You can use the code snippet presented in Fig. 6.10 as a blueprint to create your own issue report analysis.

6.6.4 Relating Bugs to Changes

As discussed in Sect. 6.4.1, there exist multiple strategies and approaches to map bug reports to code changes correctly and exhaustively. The general mining tool Mozkito ships with a number of state-of-the-art mapping techniques to associate bug reports with corresponding code changes. Discussing all strategies supported by and their possible combinations would exceed the scope of this chapter. Instead, this section explains how to use Mozkito to use the most common and simplest stand-alone strategy to efficiently map bug reports to code changes using regular expressions. The wiki pages of Mozkito provide a more detailed overview of built-in mapping strategies and instructions on how to perform these mappings. Please note that this mining step requires a mined VCS. Please read the corresponding wiki page (https://wiki.mozkito.org/x/FgAz) on how to mine VCSs using Mozkito.

Mozkito allows the user to combine multiple *mapping engines*. Each engine can be seen as a voter returning a confidence value for each pair of bug report and code change. The confidence value corresponds to the likelihood that the provided bug and change should be mapped to each other. To aggregate the different confidence values, we use a veto-strategy[3]—if the confidence value of one engine is below a certain threshold, the pair of report and change are not mapped to each other. In our case, we want to limit Mozkito to use the following engines:

Regular Expression Engine. To search for explicit bug report references in commit messages.

Report Type Engine. To consider only bug reports to be mapped (our goal is to map bugs to code changes).

Completed Order Engine. To allow only a pair of associated reports and code changes for which the code change was applied before the report was marked as resolved.

Creation Order Engine. To allow only a pair of associated reports and code changes for which the issue report was filed before the code change was applied.

Timestamp Engine. To enforce that the associated report must be marked as resolved at most one day after the code change was committed.

To configure Mozkito to use exactly this set of engines, we have to add the following line to our already existing Mozkito configuration file (`<config_file>`):

[3]There exist more aggregation strategies. Please see the Mozkito manual for more details.

```
1.0 "({match}bug:?\\s*#?##ID##)" CASE_INSENSITIVE
1.0 "({match}bugs?:?\\s*(#?\\p{Digit}+,)*#?##ID##)"\
CASE_INSENSITIVE
1.0 "({match}https?://bugzilla.mozilla.org/show_bug.cgi\
\\?id=##ID##)" CASE_INSENSITIVE
1.0 "({match}#\\s?##ID##)" CASE_INSENSITIVE
1.0 "({match}BZ\\s?:?\\s*##ID##)" CASE_INSENSITIVE
1.0 "({match}fix\\s?:?\\s*##ID##)" CASE_INSENSITIVE
1.0 "({match}fixing\\s?:?\\s*##ID##)" CASE_INSENSITIVE
-100.0 "({match}test cases for") CASE_INSENSITIVE
-100.0 "({match} revert fix for") CASE_INSENSITIVE
```

Fig. 6.11 Sample `<REGEX_FILE>` specifying the regular expressions to be used to find bug report reference candidates in commit messages. Note that backslash characters must be escaped

```
mappings.engines.enabled=[RegexEngine, ReportTypeEngine, \
CompletedOrderEngine, CreationOrderEngine, TimestampEngine]

mappings.engines.reportType.type=BUG
mappings.engines.timestamp.interval="+1d 00h 00m 00s"
mappings.engines.RegexEngine.config=<REGEX_FILE>
```

The referenced regular expression file (`<REGEX_FILE>`) should contain the project-specific regular expressions Mozkito will use to match possible bug report references. The regular expression file can specify one expression per line including a confidence value to be returned if the regular expression matches (first number in line) and a specification whether Mozkito should treat the expression case sensitive or not. Mozkito iterates through all regular expressions in the `<REGEX_FILE>` from top to bottom and stops as soon as one regular expression matches. A typical regular expression file that can also be used for our Rhino project is shown in Fig. 6.11.

Once all the above-discussed lines are added to the Mozkito configuration file (`<config_file>`), the Mozkito mapping process can be started using the following command:

```
java -Dconfig=<config_file> -jar mozkito-mappings.jar
```
[4]

There exist mining infrastructures allowing immediate mining actions. Using such infrastructures eases reproduction and allows comparison to other studies.

Exporting Bug Count Per Source File

As a last step, we export the mapping between source files and bug reports into a comma-separated file that lists the distinct number of fixed bug reports per source file. To export the mapping we created above into a comma-separated bug count

[4] `mozkito-issues-<version>-jar-with-dependencies.jar`

file, we can use the built-in Mozkito tool mozkito-bugcount located in the folder `mozkito-tools/mozkito-bugcount/target/`. To export bug counts per source file, we execute the following command:

```
java -Dconfig=<config_file> -Dbugcount.granularity=file
  -Dbugcount.output=<csv_file> -jar mozkito-bugcount.jar[5]
```

where `<csv_file>` should point to a file path to which the bug count CSV file will be written.

In the next section, we use the `<csv_file>` to build a sample defect prediction model that can be used as basis for recommendation systems.

6.7 Hands-On: Predicting Bugs

After mining VCS and issue management system and after mapping bug reports with code changes, this section provides a hands-on tutorial on how to use the statistical environment and language R [57] to write a script that reads a dataset of our sample format (Fig. 6.4) as created in the previous section, performs a stratified repeated holdout sampling of the dataset, trains multiple machine learners on the training data before evaluating the prediction accuracy of each model using the evaluation measures precision, recall, and F-measure. Precision, recall, and f-measure are only one possibility to measure prediction or classification performance. Other performance measures include ROC curves [19] or even effort-aware prediction models [42].

The complete script containing all discussed R code snippets is available for download from http://rsse.org/book/c06/sample.R. The dataset we use for our example analysis below is also available for download from http://rsse.org/book/c06/sampleset.csv.

6.7.1 Step 1: Load Required Libraries

The script will use functionalities of multiple third-party libraries. The script will make heavy use of the *caret* [36] package for R. The last statement in the first R snippet below sets the initial random seed to an arbitrary value (we chose 1); this will make the shown results reproducible.

```
> rm(list = ls(all = TRUE))
> library(caret)
> library(gdata)
> library(plyr)
```

---

[5] `mozkito-bugcount-<`*version*`>-jar-with-dependencies.jar`

```
> library(reshape)
> library(R.utils)
> set.seed(1)
```

6.7.2 Step 2: Reading the Data

To load the sample dataset containing code dependency network metrics [68] and bug counts per source file (as described in Sect. 6.5.1 and shown in Fig. 6.4), we use the following snippet that reads the dataset directly over the Internet.

After execution, the variable `data` holds the dataset in a table-like data structure called `data.frame`. For the rest of the section, we assume that the column holding the dependent variables for all instances is called `numBugs`.

```
> data <- read.table(
+    http://rsse.org/book/c06/sampleset.csv,
+    header=T, row.names=1, sep=",")
```

We can now access the dataset using the `data` variable. The command below outputs the `numBugs` column for all 266 source files.

```
> data$numBugs
  [1]   1 37 11  0 33  1 23  3  0  0 41  0  0  0 19  0  0  0  1
 [20]   0  7  0  6  2  3 11  8  3 10  0  1 10  0  0  0  3  0 25
 [39]   5  1  1  0  3  0  1  3  2  2  1  0  0  0  0  2  0  0  1
 [58]   0  4  0  1  1  0  0  0 11  2  0 43  0  1  1  1  0  0  3
 [77]   0  0  0  4  1  0  2  0  2  0  0  0  0  3  0  0  0  0  0
 [96]   0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
[115]   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
[134]   0  0  1  0  0  0  0  0  7  1  0  0  0  0  0  0  1 10  4
[153]   7  2  2  4  4  5  0  0  1  4  0  0  1  0  1  3  0  0  2
[172]   1  2  0  0  0  0  0  0  0  1  0  0  0  0  9  0  1  1  4
[191]   1  0  1  1  2  0  0 12  3 16  0  1  3  1  0  6  3  2  0
[210]   3  2  0  0  0  0  0 22  1  6  1  4  1  0  6  0  6  1  0
[229]   0  0  2  1  0  0  0  1  0  0  4  2  3  0  0  0  0  0  0
[248]   0  0  1  1  0  0  0  0  1  0  2  0  0  0  0  0  0  0  1
```

6.7.3 Step 3: Splitting the Dataset

First, we split the original dataset into training and testing subsets using stratified sampling—the ratio of files being fixed at least once in the original dataset is preserved in both training and testing datasets. This makes training and testing sets more representative by reducing sampling errors.

The first two lines of the R-code below are dedicated to separate the dependent variable from the explanatory variables. This is necessary since we will use only the explanatory variables to train the prediction models. In the third line, we then modify the dependent variable (column `numBugs`) to distinguish between

code entities with bugs (“One”) and without (“Zero”). Finally, we use the method `createDataPartition` to split the original datasets into training and testing sets (see Sect. 6.5.2). The training sets contain 2/3 of the data instances while the testing sets contain the remaining 1/3 of the data instances.

```
> dataX <- data[,which(!colnames(data) %in% c("numBugs"))]
> dataY <- data[, which(colnames(data) %in% c("numBugs"))]
> dataY <- factor(ifelse(dataY > 0, "One", "Zero"))
>
> inTrain <- createDataPartition(dataY, times = 1, p = 2/3)
>
> trainX <- dataX[inTrain[[1]], ]
> trainY <- dataY[inTrain[[1]]]
> testX <- dataX[-inTrain[[1]], ]
> testY <- dataY[-inTrain[[1]]]
```

After execution, the variable `trainX` holds the explanatory variables of all training instances while the variable `trainY` holds the corresponding dependent variables. Respectively, `testX` and `textY` contain the explanatory and dependent variables of all testing instances.

6.7.4 Step 4: Prepare the Data

It is always a good idea to remove explanatory variables that will not contribute to the final prediction model. There are two cases in which an explanatory variable will not contribute to the model.

1. If the variable values across all instances have zero variance (can be considered a constant)—the function call `nearZeroVar(trainX)` returns the array of columns whose values show no significant variance:

```
> train.nzv <- nearZeroVar(trainX)
> if (length(train.nzv) > 0) {
+     trainX <- trainX[, -train.nzv]
+     testX <- testX[, -train.nzv]
+ }
```

2. If the variable is correlated with other variables and thus does not add any new information—the function `findCorrelation` searches through the correlation matrix `trainX` and returns a set of columns that should be removed in order to reduce pair-wise correlations above the provided absolute correlation cutoff (here, 0.9):

```
> trainX.corr <- cor(trainX)
> trainX.highcorr <- findCorrelation(trainX.corr, 0.9)
> if (length(trainX.highcorr) > 0) {
+     trainX <- trainX[, -trainX.highcorr]
+     testX <- testX[, -trainX.highcorr]
+ }
```

Rescale the training data using the center to minimize the effect of large values on the prediction model by scaling the data values into the value range [0,1]. To further reduce the number of explanatory variables, you may also perform a principal component analysis—a procedure to determine the minimum number of metrics that will account for the maximum variance in the data. The function `preProcess` estimates the required parameters for each operation and `predict.preProcess` is used to apply them to specific datasets:

```
> xTrans <- preProcess(trainX, method = c("center", "scale"))
> trainX <- predict(xTrans, trainX)
> testX <- predict(xTrans, testX)
```

6.7.5 *Step 5: Train the Models*

This script will use several prediction models for the experiments: Support vector machine with radial kernel (`svmRadial`), logistic regression (`multinorm`), recursive partitioning (`rpart`), k-nearest neighbor (`knn`), tree bagging (`treebag`), random forest (`rf`), and naive Bayesian classifier (`nb`). For a fuller understanding of these models, we advise the reader to refer to specialized machine-learning texts such as Menzies [43] (Chap. 3) or Witten et al. [65].

```
> models <- c("svmRadial","multinom","rpart","knn","treebag",
+ "rf","nb")
```

Each model is optimized by the caret package by training models using different parameters (please see the caret manual for more details). "The performance of held-out samples is calculated and the mean and standard deviations is summarized for each combination. The parameter combination with the optimal re-sampling statistic is chosen as the final model and the entire training set is used to fit a final model" [36]. The level of performed optimization can be set using the *tuneLength* parameter. We set this number to five:

```
> train.control <- trainControl(number=2)
> tuneLengthValue <- 5
```

Using the `train()` function, we generate prediction models (called *fit*) and store these models in the list `modelsFit` to later access them to compute the prediction performance measures precision, recall, and accuracy:

```
> modelsFit <- list()
+
+ for(model in models){
+   print(paste("training",model," ..."))
+   fit <- train(trainX, trainY, method = model,
+     tuneLength = tuneLengthValue, trControl = train.control,
+     metric = "Kappa")
+   modelsFit[[model]] <- fit
+ }
```

6.7.6 Step 6: Make the Prediction

Using the function `extractPrediction()` we let all models predict the dependent variables of the testing set `testX`:

```
> pred.values <- extractPrediction(modelsFit, testX, testY)
> pred.values <- subset(pred.values, dataType == "Test")
> pred.values.split <- split(pred.values, pred.values$object)
```

After execution, the variable `pred.values.split` holds both the real and the predicted dependent variable values. To check the predicted values for any of the used models (e.g., `svmRadial`), we can access the variable `pred.values.split` as shown in the following text. The result is a list of observed (`obs` column) and predicted (`pred` column) values for each instance in the testing dataset. The result depends on the random split and thus may vary between individual experiments.

```
> pred.values.split$svmRadial
     obs pred     model dataType    object
179  One Zero svmRadial     Test svmRadial
180 Zero Zero svmRadial     Test svmRadial
181 Zero Zero svmRadial     Test svmRadial
182 Zero Zero svmRadial     Test svmRadial
183 Zero Zero svmRadial     Test svmRadial
184 Zero Zero svmRadial     Test svmRadial
185  One Zero svmRadial     Test svmRadial
186  One  One svmRadial     Test svmRadial
187  One  One svmRadial     Test svmRadial
188 Zero  One svmRadial     Test svmRadial
189  One  One svmRadial     Test svmRadial
190  One Zero svmRadial     Test svmRadial
191  One  One svmRadial     Test svmRadial
192  One Zero svmRadial     Test svmRadial
193 Zero Zero svmRadial     Test svmRadial
```

6.7.7 Step 7: Compute Precision, Recall, and F-measure

The final part of the script computes precision, recall, and F-measure values for all models and stores these accuracy measures in a table-like data structure:

```
> getPrecision <- function(x) as.numeric(unname(x$byClass[3]))
> getRecall <- function(x) as.numeric(unname(x$byClass[1]))
> getFmeasure <- function(x, y) 2 * ((x * y)/(x + y))
>
> n.row = length(pred.values.split)
> results <- NULL
> results <- dataFrame(
+     colClasses = c(Model = "character", Precision = "double",
      Recall = "double", F.Measure = "double"), nrow = n.row)
> for (j in 1:length(pred.values.split)) {
```

```
+   conf.matrix <- confusionMatrix(pred.values.split[[j]]$pred,
+        pred.values.split[[j]]$obs, positive = "One")
+
+   precision <- getPrecision(conf.matrix)
+   if(is.na(precision)){ precision <- 0 }
+
+   recall <- getRecall(conf.matrix)
+   if(is.na(recall)){ recall <- 0 }
+
+   f.measure <- getFmeasure(precision, recall)
+   if(is.na(f.measure)){ f.measure <- 0+   }
+
+   results[j, 1] <- names(pred.values.split)[j]
+   results[j, 2:4] <- c(precision, recall, f.measure)
+ }
```

To print the prediction measures, we simply print the results table. That will print a table containing precision, recall, and F-measure values sorted by a machine-learning algorithm used for training and testing.

```
> print(results)
      Model Precision    Recall F.Measure
1       knn 0.6562500 0.5833333 0.6176471
2  multinom 0.7096774 0.6111111 0.6567164
3        nb 0.6571429 0.6388889 0.6478873
4        rf 0.7575758 0.6944444 0.7246377
5     rpart 0.5526316 0.5833333 0.5675676
6 svmRadial 0.7307692 0.5277778 0.6129032
7   treebag 0.8181818 0.7500000 0.7826087
```

The results show that using a tree bag model, we obtain a precision of 0.82, a recall of 0.75, and an F-measure of 0.78. The high precision value of 0.82 means that the tree bag model on average reports 18 % false positives—classifies code entities as having a bug although no bug was found. Similarly, the recall value of 0.75 implies that the model contains about 25 % false negatives—code entities classified as bug free but in which bugs have been fixed. Comparing this result with the overall defect prediction model accuracy measures on Windows Vista presented in Table 6.3 shows that the just-built classification model has comparable results to state-of-the-art defect prediction models (although trained and tested on a different project, using different metrics, and different granularity).

Ready-made scripts are available that predict and recommend future bugs.

6.8 Conclusion

To err is human, but to learn from the past is human too. Mining issue repositories offer several opportunities to automate this learning process, producing recommendations that can help identify present bugs and avoid future bugs. Bug data

is not without caveats, though. First, the data reflects specific users, tools, and processes, which should be identified to ensure proper interpretation of the results. Second, the data itself is frequently noisy or biased, which should also be taken into account, and where possible, reduced or eliminated. A bit of manual inspection and cross-checking can tremendously increase confidence in all automatic findings, and the future belongs to those who integrate automated tools into well-defined and systematic empirical investigations.

The central challenge of the future will be to combine both automatic and manual empirical bug analysis.

Acknowledgments We thank Sascha Just and many anonymous reviewers for their work.

References

1. Anbalagan, P., Vouk, M.: On predicting the time taken to correct bug reports in open source projects. In: Proceedings of the IEEE International Conference on Software Maintenance, pp. 523–526 (2009). doi:10.1109/ICSM.2009.5306337
2. Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., Guéhéneuc, Y.G.: Is it a bug or an enhancement?: a text-based approach to classify change requests. In: Proceedings of the IBM Centre for Advanced Studies Conference on Collaborative Research (2008). doi:10.1145/1463788.1463819
3. Anvik, J., Hiew, L., Murphy, G.C.: Who should fix this bug? In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 361–370 (2006). doi:10.1145/1134285.1134336
4. Aranda, J., Venolia, G.: The secret life of bugs: going past the errors and omissions in software repositories. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 298–308 (2009). doi:10.1109/ICSE.2009.5070530
5. Asuncion, H.U., Asuncion, A.U., Taylor, R.N.: Software traceability with topic modeling. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, vol. 1, pp. 95–104 (2010). doi:10.1145/1806799.1806817
6. Bachmann, A., Bernstein, A.: Software process data quality and characteristics: a historical view on open and closed source projects. In: Proceedings of the Joint ACM International Workshop on Principles of Software Evolution and ERCIM Workshop on Software Evolution, pp. 119–128 (2009). doi:10.1145/1595808.1595830
7. Bernstein, A., Bachmann, A.: When process data quality affects the number of bugs: correlations in software engineering datasets. In: Proceedings of the International Working Conference on Mining Software Repositories, pp. 62–71 (2010). doi:10.1109/MSR.2010.5463286
8. Bettenburg, N., Begel, A.: Deciphering the story of software development through frequent pattern mining. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 1197–1200 (2013). doi:10.1109/ICSE.2013.6606677
9. Bettenburg, N., Just, S., Schröter, A., Weiß, C., Premraj, R., Zimmermann, T.: Quality of bug reports in Eclipse. In: Proceedings of the Eclipse Technology eXchange, pp. 21–25 (2007). doi:10.1145/1328279.1328284
10. Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., Zimmermann, T.: What makes a good bug report? In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 308–318 (2008). doi:10.1145/1453101.1453146

11. Bettenburg, N., Premraj, R., Zimmermann, T.: Duplicate bug reports considered harmful ... really? In: Proceedings of the IEEE International Conference on Software Maintenance, pp. 337–345 (2008). doi:10.1109/ICSM.2008.4658082
12. Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., Devanbu, P.: Fair and balanced?: bias in bug-fix datasets. In: Proceedings of the European Software Engineering Conference/ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 121–130 (2009). doi:10.1145/1595696.1595716
13. Bird, C., Nagappan, N., Gall, H., Murphy, B., Devanbu, P.: Putting it all together: using socio-technical networks to predict failures. In: Proceedings of the International Symposium on Software Reliability Engineering, pp. 109–119 (2009). doi:10.1109/ISSRE.2009.17
14. Bird, C., Bachmann, A., Rahman, F., Bernstein, A.: LINKSTER: enabling efficient manual inspection and annotation of mined data. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 369–370 (2010). doi:10.1145/1882291.1882352
15. Breu, S., Premraj, R., Sillito, J., Zimmermann, T.: Information needs in bug reports: improving cooperation between developers and users. In: Proceedings of the ACM Conference on Computer Supported Cooperative Work, pp. 301–310 (2010). doi:10.1145/1718918.1718973
16. Cartwright, M.H., Shepperd, M.J., Song, Q.: Dealing with missing software project data. In: Proceedings of the IEEE International Symposium on Software Metrics, pp. 154–165 (2003). doi:10.1109/METRIC.2003.1232464
17. Čubranić, D., Murphy, G.C., Singer, J., Booth, K.S.: Hipikat: a project memory for software development. IEEE Trans. Software Eng. **31**(6), 446–465 (2005). doi:10.1109/TSE.2005.71
18. D'Ambros, M., Lanza, M., Robbes, R.: An extensive comparison of bug prediction approaches. In: Proceedings of the International Working Conference on Mining Software Repositories, pp. 31–41 (2010). doi:10.1109/MSR.2010.5463279
19. Davis, J., Goadrich, M.: The relationship between precision–recall and ROC curves. In: Proceedings of the International Conference on Machine Learning, pp. 233–240 (2006). doi:10.1145/1143844.1143874
20. Dhaliwal, T., Khomh, F., Zou, Y.: Classifying field crash reports for fixing bugs: a case study of Mozilla Firefox (2011). doi:10.1109/ICSM.2011.6080800
21. Fischer, M., Pinzger, M., Gall, H.: Populating a release history database from version control and bug tracking systems. In: Proceedings of the IEEE International Conference on Software Maintenance, pp. 23–32 (2003). doi:10.1109/ICSM.2003.1235403
22. Guo, P.J., Zimmermann, T., Nagappan, N., Murphy, B.: Characterizing and predicting which bugs get fixed. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, vol. 1, pp. 495–504 (2010). doi:10.1145/1806799.1806871
23. Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S.: A systematic literature review on fault prediction performance in software engineering. IEEE Trans. Software Eng. **38**(6), 1276–1304 (2012). doi:10.1109/TSE.2011.103
24. Herzig, K.: Mining and untangling change genealogies. Ph.D. thesis, Universität des Saarlandes (2013)
25. Herzig, K., Zeller, A.: The impact of tangled code changes. In: Proceedings of the International Working Conference on Mining Software Repositories, pp. 121–130 (2013)
26. Herzig, K., Just, S., Zeller, A.: It's not a bug, it's a feature: how misclassification impacts bug prediction. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 392–401 (2013). doi:10.1109/ICSE.2013.6606585
27. Hooimeijer, P., Weimer, W.: Modeling bug report quality. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 34–43 (2007). doi:10.1145/1321631.1321639
28. Jeffrey, D., Feng, M., Gupta, R.: BugFix: a learning-based tool to assist developers in fixing bugs. In: Proceedings of the IEEE International Conference on Program Comprehenension, pp. 70–79 (2009). doi:10.1109/ICPC.2009.5090029
29. Kawrykow, D.: Enabling precise interpretations of software change data. Master's thesis, McGill University (2011)

30. Kawrykow, D., Robillard, M.P.: Non-essential changes in version histories. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 351–360 (2011). doi:10.1145/1985793.1985842
31. Kersten, M.: Focusing knowledge work with task context. Ph.D. thesis, University of British Columbia, Vancouver (2007)
32. Kim, S., Whitehead, E.J.: How long did it take to fix bugs? In: Proceedings of the International Workshop on Mining Software Repositories, pp. 173–174 (2006). doi:10.1145/1137983.1138027
33. Kim, S., Zhang, H., Wu, R., Gong, L.: Dealing with noise in defect prediction. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 481–490 (2011). doi:10.1145/1985793.1985859
34. Kimmig, M., Monperrus, M., Mezini, M.: Querying source code with natural language. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 376–379 (2011). doi:10.1109/ASE.2011.6100076
35. Ko, A.J., Myers, B.A., Chau, D.H.: A linguistic analysis of how people describe software problems. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 127–134 (2006). doi:10.1109/VLHCC.2006.3
36. Kuhn, M.: caret: classification and regression training. Version 4.76, R package (2011). URL http://cran.r-project.org/web/packages/caret/caret.pdf. [retrieved 9 October 2013]
37. Lamkanfi, A., Demeyer, S., Soetens, Q.D., Verdonck, T.: Comparing mining algorithms for predicting the severity of a reported bug. In: Proceedings of the European Conference on Software Maintenance and Reengineering, pp. 249–258 (2011). doi:10.1109/CSMR.2011.31
38. Liebchen, G.A., Shepperd, M.: Data sets and data quality in software engineering. In: Proceedings of the International Workshop on Predictor Models in Software Engineering, pp. 39–44 (2008). doi:10.1145/1370788.1370799
39. Marks, L., Zou, Y., Hassan, A.E.: Studying the fix-time for bugs in large open source projects. In: Proceedings of the International Conference on Predictor Models in Software Engineering, pp. 11:1–11:8 (2011). doi:10.1145/2020390.2020401
40. Matter, D., Kuhn, A., Nierstrasz, O.: Assigning bug reports using a vocabulary-based expertise model of developers. In: Proceedings of the International Working Conference on Mining Software Repositories, pp. 131–140 (2009). doi:10.1109/MSR.2009.5069491
41. McCabe, T.J.: A complexity measure. IEEE Trans. Software Eng. **2**(4), 308–320 (1976). doi:10.1109/TSE.1976.233837
42. Mende, T., Koschke, R.: Effort-aware defect prediction models. In: Proceedings of the European Conference on Software Maintenance and Reengineering, pp. 107–116 (2010). doi:10.1109/CSMR.2010.18
43. Menzies, T.: Data mining: a tutorial. In: Robillard, M., Maalej, W., Walker, R.J., Zimmermann, T. (eds.) Recommendation Systems in Software Engineering. Springer, Berlin (2014)
44. Menzies, T., Marcus, A.: Automated severity assessment of software defect reports. In: Proceedings of the IEEE International Conference on Software Maintenance, pp. 346–355 (2008). doi:10.1109/ICSM.2008.4658083
45. Mockus, A.: Missing data in software engineering. In: Shull, F., Singer, J., Sjøberg, D. (eds.) Guide to Advanced Empirical Software Engineering, pp. 185–200. Springer, London (2008). doi:10.1007/978-1-84800-044-5_7
46. Mockus, A., Fielding, R.T., Herbsleb, J.D.: Two case studies of open source software development: Apache and Mozilla. ACM Trans. Software Eng. Methodol. **11**(3), 309–346 (2002). doi:10.1145/567793.567795
47. Mockus, A., Nagappan, N., Dinh-Trong, T.T.: Test coverage and post-verification defects: a multiple case study. In: Proceedings of the International Symposium on Empirical Software Engineering and Measurement, pp. 291–301 (2009). doi:10.1109/ESEM.2009.5315981
48. Myrtveit, I., Stensrud, E., Olsson, U.H.: Analyzing data sets with missing data: an empirical evaluation of imputation methods and likelihood-based methods. IEEE Trans. Software Eng. **27**(11), 999–1013 (2001). doi:10.1109/32.965340

49. Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 284–292 (2005). doi:10.1145/1062455.1062514
50. Nagappan, N., Ball, T.: Evidence-based failure prediction. In: Oram, A., Wilson, G. (eds.) Making Software: What Really works,and Why we believe it, pp. 415–434. O'Reilly Media, Sebastopol (2010)
51. Nagappan, N., Murphy, B., Basili, V.: The influence of organizational structure on software quality: an empirical case study. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 521–530 (2008). doi:10.1145/1368088.1368160
52. Nagappan, N., Zeller, A., Zimmermann, T., Herzig, K., Murphy, B.: Change bursts as defect predictors. In: Proceedings of the International Symposium on Software Reliability Engineering, pp. 309–318 (2010). doi:10.1109/ISSRE.2010.25
53. Nagwani, N.K., Verma, S.: Predicting expert developers for newly reported bugs using frequent terms similarities of bug attributes. In: Proceedings of the International Conference on ICT and Knowledge Engineering, pp. 113–117 (2012). doi:10.1109/ICTKE.2012.6152388
54. Nguyen, T.H.D., Adams, B., Hassan, A.E.: A case study of bias in bug-fix datasets. In: Proceedings of the Working Conference on Reverse Engineering, pp. 259–268 (2010). doi:10.1109/WCRE.2010.37
55. Nguyen, A.T., Nguyen, T.T., Al-Kofahi, J., Nguyen, H.V., Nguyen, T.N.: A topic-based approach for narrowing the search space of buggy files from a bug report. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 263–272 (2011). doi:10.1109/ASE.2011.6100062
56. Prifti, T., Banerjee, S., Cukic, B.: Detecting bug duplicate reports through local references. In: Proceedings of the International Conference on Predictor Models in Software Engineering, pp. 8:1–8:9 (2011). doi:10.1145/2020390.2020398
57. R Development Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna (2010)
58. Runeson, P., Alexandersson, M., Nyholm, O.: Detection of duplicate defect reports using natural language processing. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 499–510 (2007). doi:10.1109/ICSE.2007.32
59. Samuelson, W., Zeckhauser, R.: Status quo bias in decision making. J. Risk Uncertain. **1**, 7–59 (1988). doi:10.1007/BF00055564
60. Sarma, A., Noroozi, Z., van der Hoek, A.: Palantír: raising awareness among configuration management workspaces. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 444–454 (2003). doi:10.1109/ICSE.2003.1201222
61. Strike, K., El Emam, K., Madhavji, N.: Software cost estimation with incomplete data. IEEE Trans. Software Eng. **27**(10), 890–908 (2001). doi:10.1109/32.935855
62. Sun, C., Lo, D., Khoo, S.C., Jiang, J.: Towards more accurate retrieval of duplicate bug reports. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 253–262 (2011). doi:10.1109/ASE.2011.6100061
63. Thomas, S.W.: Mining software repositories using topic models. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 1138–1139 (2011). doi:10.1145/1985793.1986020
64. Wang, X., Zhang, L., Xie, T., Anvik, J., Sun, J.: An approach to detecting duplicate bug reports using natural language and execution information. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 461–470 (2008). doi:10.1145/1368088.1368151
65. Witten, I.H., Frank, E., Hall, M.A.: Data Mining: Practical Machine Learning Tools and Techniques, 3rd edn. Morgan Kaufmann, San Francisco (2011)
66. Wu, R., Zhang, H., Kim, S., Cheung, S.C.: ReLink: recovering links between bugs and changes. In: Proceedings of the European Software Engineering Conference/ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 15–25 (2011). doi:10.1145/2025113.2025120

67. Yu, L., Tsai, W.T., Zhao, W., Wu, F.: Predicting defect priority based on neural networks. In: Proceedings of the International Conference on Advanced Data Mining and Applications. Lecture Notes in Computer Science, vol. 6441, pp. 356–367 (2010). doi:10.1007/978-3-642-17313-4_35
68. Zimmermann, T., Nagappan, N.: Predicting defects using network analysis on dependency graphs. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 531–540 (2008). doi:10.1145/1368088.1368161
69. Zimmermann, T., Premraj, R., Zeller, A.: Predicting defects for Eclipse. In: Proceedings of the International Workshop on Predictor Models in Software Engineering, pp. 9:1–9:7 (2007). doi 10.1109/PROMISE.2007.10