Patterns for Extracting High Level Information from Bug Reports

Rodrigo Souza*†, Christina Chavez*‡ and Roberto Bittencourt*§
*Dept. of Computer Science, Federal University of Bahia, Brazil
†Data Processing Center, Federal University of Bahia, Brazil
‡Fraunhofer Project Center, Federal University of Bahia, Brazil
§Dept. of Exact Sciences, State University of Feira de Santana, Brazil
Email: rodrigo@dcc.ufba.br, flach@dcc.ufba.br, roberto@uefs.br

***Abstract*—Bug reports provide insight about the quality of an evolving software and about its development process. Such data, however, is often incomplete and inaccurate, and thus should be cleaned before analysis. In this paper, we present patterns that help both novice and experienced data scientists to discard invalid bug data that could lead to wrong conclusions.**

***Index Terms*—data analysis, mining software repositories, patterns, bugs.**

I. Introduction

Bug tracking systems store bug reports for a software project and keep record of discussion and progress changes for each bug. This data can be used not only to assess quality attributes of the software, but also to reason about its development process. Such richness of information makes bug reports an invaluable source for data scientists interested in software engineering.

However, bug tracking systems often contain data that is inaccurate, incomplete [1], or biased [2]. For example, changing the status of a bug report to `VERIFIED` usually means that, after a resolution was found to the bug, some kind of software verification (e.g., source code inspection, testing) was performed and the resolution was considered appropriate. Sometimes, however, old bug reports are marked as verified just to help users and developers keep track of current bug reports [3].

Without proper guidance, it is easy to overlook pitfalls in the data and draw wrong conclusions. In this paper, we provide best practices and step-by-step solutions to recurring problems related to cleaning bug data.

II. Data Set

Each pattern contains an *Examples* section with code snippets showing how to apply the pattern on real data. The snippets are written in R, a programming language for data analysis[1]. The data used are bug reports from the project NetBeans/Platform, made available for the 2011 edition of the MSR Mining Challenge[2].

[1]http://www.r-project.org/

[2]http://2011.msrconf.org/msr-challenge.html

TABLE I
Sample of `bugs` table, hour info omitted.

bug	severity	priority	creation.time	modif.time
397	normal	P4	1998-07-31	2008-12-23
427	normal	P4	1998-08-08	2008-12-23
479	normal	P2	1998-08-19	2008-12-23

TABLE II
Sample of `comments` table, hour info omitted.

bug	comment.md5	user	time
397	bafe80ea4ba2b73b6883243d8718ae3b	1887	2000-11-01
427	64d38f33f8848525a81b415406d064df	46	1998-08-08
427	264e61d606f0a4c0c8d477de00fba347	124	2000-07-25

The NetBeans project uses Bugzilla[3] as its bug tracking system, which stores all data in a MySQL database. The source code presented here refers to database tables and columns used by Bugzilla, but it should work with any bug tracking system with minor changes.

Although the full NetBeans data set contains 57 database tables, in this paper only three are used: `bugs`, `changes` (originally, `bugs_activity`), and `comments` (originally, `longdescs`). Notice that some tables and columns were renamed for clarity purposes.

The `bugs` table contains general information about each bug report, which is identified by a unique number (column `bug`). Each bug report has a `severity`, a `priority`, and two timestamps: the time of creation (`creation.time`), and the time of the last modification (`modif.time`). Table I shows a sample of the `bugs` table.

The `comments` table contains comments that each `user` added to a `bug` report at some point in `time`. To reduce the file size, the comment text was replaced by its MD5 hash (column `comment.md5`). With high probability, two comments are represented by the same hash number if and only if they contain the same text. Table II shows a sample of the `comments` table.

[3]http://www.bugzilla.org/

TABLE III
SAMPLE OF `changes` TABLE, HOUR INFO OMITTED.

bug	user	time	field	new.value
427	17822	2009-10-30	resolution	WONTFIX
500	182	2002-04-12	bug_status	CLOSED
755	182	2002-01-11	bug_status	REOPENED

The `changes` table contains all modifications users made on bug reports over time. This includes changes in priority, status, resolution, or any other field in a bug report. Each row contains the `new value` of a `field` that was modified by a `user`[4] at some point in `time`. Table III shows a sample of the `bugs` table.

The *status* field is used to track the progress of the bug fixing activity. A bug report is created with status `NEW` or `UNCONFIRMED`. Then, its status may be changed to `ASSIGNED`, to denote that a user has taken responsibility on the bug. After that, the bug is `RESOLVED`, then optionally `VERIFIED` by the quality assurance team, and finally `CLOSED` when the next software release comes out. If, after resolving the bug, someone finds that the resolution was not appropriate, the status is changed to `REOPENED`.

All the data and code used in this paper is online[5].

III. NOT EVERYONE IS A PROGRAMMER

Problem

Find the quality engineers on a team, if there is any.

Context

Developers tend to assume specific roles in the process of tracking bugs. While many developers participate by fixing bugs, quality engineers usually take bug fixes and verify if they are appropriate. Making the distiction between quality engineers and other developers is important when studying the influence of human factors in the handling of bugs.

Solution

Analyze each developer's activity in the bug tracking system, such as status and resolution changes. In particular, count how many times each developer has...

- ... changed the status to `VERIFIED` (number of verifications);
- ... changed the resolution to `FIXED` (number of fixes).

Then, compute the ratio between verifications and fixes for each developer (add 1 to the number of fixes to avoid division by zero). If such ratio is greater than some threshold (e.g., 5 or 10), it suggests that the developer is specialized in verifications. Select all such developers and compute the total number of verifications performed by them, compared to the total number of verifications in the project. If they perform a great part of the verifications in the project (e.g., more than 50%), then the project has a quality engineering team, formed by the that developers.

Discussion

It is a common mistake to use the absolute number of verifications to determine if a developer is a quality engineer. In some projects, however, developers that fixes bugs also mark them as `VERIFIED`. Because of that, the ratio between verifications and fixes is a better indicator.

Developers can change roles over time. If this is the case, consider analyzing a shorter period of time. Even better, use sliding windows, i.e., analyze multiple consecutive short periods over time.

Examples

```
> resolution <- subset(changes, field == 'resolution')
> status <- subset(changes, field == 'bug_status')
> t1 <- table(resolution$user, resolution$new.value)
> t2 <- table(status$user, status$new.value)
> t <- merge(as.data.frame.matrix(t1),
+            as.data.frame.matrix(t2),
+            by="row.names")
> t$qe <- t$VERIFIED / (1 + t$FIXED) > 10
> total <- sum(t$VERIFIED)
> part <- sum(t$VERIFIED[t$qe]) / total
> if (part < 0.5)
+   t$qe <- FALSE
> num.qe.developers <- sum(t$qe)
```

Related Patterns

What threshold to use to distinguish between quality engineer and other developers? What percentage of verifications a group of developers has to perform for them to be considered a quality engineering team? There are no magic numbers; as always, be sure to [Choose a Suitable Threshold].

IV. TESTING PHASE

Problem

Identify testing phases in the software development life cycle.

Context

Before a new version is shipped to final users, it is common to test new implemented features and bug fixes. Sometimes most of the testing effort is concentrated on a well-defined testing phase, that precedes the next release. In a bug tracking system, testing efforts are recorded as bug status changes, from `RESOLVED` to `VERIFIED`. Testing phases, therefore, should appear as a relatively large number of verifications comprised in a relatively short period.

Solution

Solution 1. Plot the accumulated number of verifications over time using a line chart. If you know the software release dates, highlight them in the chart with vertical lines. The chart is monotonically increasing, but some portions may exhibit a steeper ascent, that stands for periods with a high verification activity. Such periods may be testing phases, particularly if they precede a release date.

[4] In this context, user denotes a user of the bug tracking system, which can be either a developer or a final user.

[5] https://github.com/rodrigorgs/dapse13-bugpatterns

Solution 2. Count the number of verifications per day (i.e., changes that set a bug status to `VERIFIED`). If, for a particular day, the number is above some threshold (e.g., 5 verifications), then mark that day as testing phase candidate.

Group together testing phase candidates that are at most a few days apart from each other (e.g., 3 days, to account for holidays and weekends). Count the total number of verifications in each group. If the number is above some threshold (e.g., 20 verifications), then that group of days comprise a testing phase.

Discussion

The first solution is suitable for visual exploration of the data. If the data set is too large, however, it becomes difficult to visualize. The second solution is more scalable and deterministic, but relies on a careful choice of thresholds.

Be suspicious if the number of verifications for a particular day is too high (e.g., 50). Such verifications may be the result of a mass verification, i.e., multiple bug reports were simultaneously updated in order to tidy the bug tracking system. To assess if this is the case, analyze the individual verifications on that day. If most of them were performed by the same person, separated by a few seconds or minutes from each other, then they are likely the result of a mass verification, and not motivated by a testing phase.

Examples

The first solution was used by Souza and Chavez [3] (see Figure 2 in their paper). The following R code shows how to apply the solution to Eclipse/Platform. Only a subset of the data is used, otherwise testing phases would be difficult to visualize. Assume `releases$date` is a vector with release dates.

```
> ver <- subset(changes,
+         field == "bug_status"
+     & new.value == "VERIFIED")
> ver <- ver[order(ver$time), ]
> ver$n.changes <- 1:nrow(ver)
> ver <- subset(ver,
+         time >= as.POSIXct("2009-06-10")
+     & time < as.POSIXct("2010-06-09"))
> with(ver, plot(n.changes ~ time, type="l"))
> abline(v=releases$date, lty=2)
```

The result is Figure 1. Notice how verification activity (steep ascents) is concentrated just before release dates (dashed vertical lines), suggesting there are well-defined testing phases in Eclipse/Platform.

Related Patterns

It should be noted that *Not Everyone is a Programmer* (Section III): some teams have dedicated quality engineers that are responsible for testing. Well-defined testing phases are less common in such teams, because quality engineers constantly test features and bug fixes, and do not need to switch between programming and testing.

V. Grab the Release Dates

Problem

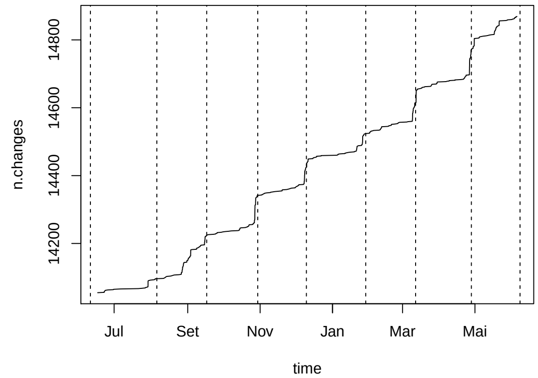

Fig. 1. Accumulated number of changes over time and release dates.

How to discover the release dates for all versions of a software?

Context

Releases are important milestones in a project, as they represent the end of a development cycle. Different development tasks may be performed based on the proximity of a release date. For example, verification and validation tasks are expected before a release, and final users only report bugs for a version after it has been released.

It should be easy for a data scientist to discover the release dates for the most recent versions of a project, as they are usually displayed on the download page. However, older release dates may not always be available.

Solution

Solution 1. Some projects tag specific revisions of the source code maintained at a version control system. Identify tags that denote specific releases, by looking at the tag name. Consider the dates of these tags as release dates.

Solution 2. The Internet Archive Wayback Machine[6] is a free service that archives copies of web pages since 1996. Find the web page that lists the latest versions, together with release dates. Then, enter the URL in the Wayback Machine to see historical versions of the page that, hopefully, contain older release dates.

Discussion

Tags in a version control system are used internally by developers, and may not represent the exact date when the version was available for download. Web pages are more accurate in this sense, but they may not list intermediate, development versions.

Examples

The second solution was used by Souza et al. [3] to discover release dates for the project Eclipse/Platform.

Related Patterns

If you [Look Out for Mass Edits], you may find other important events in a project's life, such as policy and process changes.

[6] http://archive.org/web/web.php

References

[1] J. Aranda and G. Venolia, "The secret life of bugs: Going past the errors and omissions in software repositories," in *Proc. of the 31st Int. Conf. on Soft. Engineering*, 2009, pp. 298–308.

[2] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced?: bias in bug-fix datasets," in *European Soft. Eng. Conf. and Symposium on the Foundations of Soft. Eng.*, ser. ESEC/FSE '09. ACM, 2009.

[3] R. Souza and C. Chavez, "Characterizing verification of bug fixes in two open source ides." in *MSR*, M. Lanza, M. D. Pent, and T. Xi, Eds. IEEE, 2012, pp. 70–73.