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.

Each solution is presented in a structured form called pattern. A pattern consists of 6 sections: (1) a short *name*; (2) the *problem* being solved; (3) a *context* in which the pattern can be applied; (4) one or more *solutions* to the problem; (5) a *discussion* of trade-offs and common mistakes to consider when using the pattern; and (6) *examples* of the pattern in use.

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].

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

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 two are used: `bugs` and `changes` (originally, `bugs_activity`). One table and a few 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 `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 II 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 option-

[1] http://www.r-project.org/
[2] http://2011.msrconf.org/msr-challenge.html
[3] http://www.bugzilla.org/
[4] In this context, user denotes a user of the bug tracking system, which can be either a developer or a final user.

TABLE II
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

ally `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`.

There are many ways to resolve a bug. To reflect that, when a bug status is changed to `RESOLVED`, the *resolution* field is changed either to `FIXED` (if the software was changed to solve the issue), `WORKSFORME` (if developers were not able to reproduce the problem), `DUPLICATE` (if a previous bug report describes the same problem), among other resolutions.

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 on outcomes of the software development process.

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. The absolute number is a poor indicator because, in some projects, developers that fix bugs also mark them as `VERIFIED`.

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, one at a time.

Examples

This solution was used by Souza and Chavez [3]. They used a threshold of 10 for the ratio between verifications and fixes. In the source code below, we show how to apply this solution to NetBeans/Platform:

```
> 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")
> threshold <- 10
> t$qe <- t$VERIFIED / (1 + t$FIXED) > threshold
> total <- sum(t$VERIFIED)
> proportion <- sum(t$VERIFIED[t$qe]) / total
> num.qe.developers <- sum(t$qe)
```

By running the code, we find there are 36 quality engineers, who are responsible for about 85% of the verifications.

Related Patterns

While this patterns helps locate people who concentrate quality efforts, the pattern *Testing Phase* (Section IV) helps find periods with intense testing activity.

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 features and bug fixes. In some projects, most of the testing effort is concentrated on a well-defined testing phase, that precedes the release of the next version of the software.

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. Select verifications, i.e., changes that set the bug status to `VERIFIED`. Then, 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. Although the chart is monotonically increasing, some portions may exhibit a steeper ascent, that represents a period with high verification activity. Such periods may be testing phases, particularly if they precede a release date.

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

Solution 2. Select verifications, i.e., changes that set the bug status to `VERIFIED`. Then, apply Kleinberg's algorithm [4] to verification times in order to detect bursts, i.e., periods of intense verification activity.

The algorithm is based on a Markov model and outputs a hierarchical burst structure. The first level comprises the entire period; the second level contains bursts in the period; the third level, bursts within second-level bursts, and so on. For our purposes, second-level bursts are good testing phase candidates.

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 objective, though computationally expensive.

Be suspicious if the number of verifications per day is too high (e.g., above 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, instead of 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 shown in 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.

The following R code shows how to apply the second solution, using Kleinberg's algorithm and taking second-level bursts. Then, we count the number of verifications in each burst. The `ver` variable is reused from the previous snippet of code. The first 6 bursts are shown in Table III.

```
> library(bursts)
> k <- kleinberg(unique(ver$time))
> bursts <- subset(k, level == 2)
> # Num. of verifications (total and per day average)
> bursts$count <- apply(bursts, 1, function(x)
+   sum(ver$time > as.POSIXct(x["start"])
+       & ver$time < as.POSIXct(x["end"])))
```

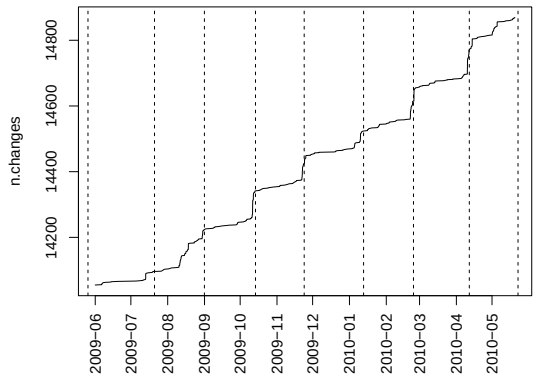

Fig. 1. Accumulated number of verifications over time.

TABLE III
Periods with intense verification activity (sample).

start	end	count	per.day
2009-07-29 09:07:15	2009-07-29 16:21:58	17	17
2009-08-26 11:07:36	2009-09-03 12:24:38	71	7
2009-09-15 02:36:27	2009-09-16 12:54:35	26	13
2009-10-26 09:47:55	2009-10-29 12:00:26	82	20
2009-12-07 13:11:33	2009-12-11 10:57:28	74	14
2010-01-20 21:38:12	2010-01-21 15:45:14	13	13

```
> days <- as.Date(bursts$end) - as.Date(bursts$start)
> days <- days + 1
> bursts$per.day <- bursts$count %/% as.numeric(days)
```

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.

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.

[4] J. Kleinberg, "Bursty and hierarchical structure in streams," in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '02. New York, NY, USA: ACM, 2002, pp. 91–101. [Online]. Available: http://doi.acm.org/10.1145/775047.775061