Patterns for cleaning bug data

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 its development process. Such data, however, is often incomplete and inaccurate, and thus should be cleaned before any 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 [1], biased [2], or incomplete [3]. 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 (source code inspection, testing etc.) 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 [4].

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 *Example* 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 the bug reports from the project NetBeans/Platform, made available for the 2011 edition of the Mining Challenge[2].

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

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.

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 I
Sample of the `bugs` table. The hour part of the times is ommitted.

The `comments` table contains comments that each `user` added to a `bug` report at some point in `time`. To reduce the file size, the text for the comment 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.

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

TABLE II
Sample of the `comments` table. The hour part of the time is ommitted.

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

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

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

The `changes` table contains all modifications made by users on bug reports over time. This includes changes in priority, status, resolution, and virtually 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.

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

TABLE III
SAMPLE OF THE `changes` TABLE. THE HOUR PART OF THE TIME IS OMMITTED.

The *status* field is used to keep track of 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 `CLOSED`, after 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 forms of resolving 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 available online[5].

III. Look Out For Mass Updates

Problem

Discover changes to bug reports that were the result of a mass update.

Context

Changes to a bug report are often the result of an effort made by developers to triage, fix or verify a bug. Sometimes, however, hundreds or thousands of bug reports are changed almost simultaneously. Such changes are not caused by a burst of productivity; instead, they are the result of a mass update, often done with the purpose of cleaning up the bug tracking system.

Mass updates can be motivated by a policy change. In Eclipse Modeling Framework, for instance, developers decided that bug reports containing fixes that were already published on their website should have the status `VERIFIED`. A mass status update was needed to make previous bug reports conform to the new policy.

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

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

Mass updates are characterized by a large number of changes of the *same type* (e.g., marking a bug report as `VERIFIED` or changing a target milestone) made by a single developer in a short period of time. Often such changes are accompanied by a comment that is the same for all changes in the mass update.

Solution

First, choose the type of change that you wish to analyze (e.g., changing a bug status to `VERIFIED`). Then apply one of the following solutions.

Solution 1. Select only the changes of the chosen type, along with the time of the change. Sort the changes by time and then plot the accumulated number of changes over time as a line chart.

The line is always increasing, but periods with many changes will stand out as steep ascents. Examine your vertical axis to assess whether such ascents represent a large number of changes (e.g., thousands). If this is the case, then it is likely that the changes were caused by a mass update.

Solution 2. Select only the changes of the chosen type, along with the date of the change, the user who made the change and the comment. Then, for each triple $\langle date, user, comment \rangle$, count the number of occurrences in the data set. Then, sort the triples by number of occurences. Triples with highest frequencies are good candidates for mass updates. You may analyse the comment text to see if they refer to cleanup, policy change or mass update in general.

Discussion

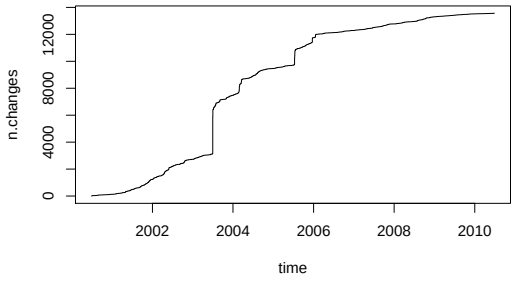
The first solution to find mass updates is more visual, but less accurate, as it does not take into account the user who made the changes. It is useful for a quick exploratory assessment of mass updates. The second solution is numeric and takes into account the users and their comments, but is more computationally intensive.

Examples

For example, Souza et al. [4] used the first solution (see Figure 2 in their paper) and a variation of the second without taking comments into account to detect mass verifications (i.e., changes in which the status is set to `VERIFIED`). They discarded all changes that were part of a mass update which updated at least 50 bug reports.

Solution 1. Here is a sample R code to plot the accumulated number of verifications over time.

```
> ver <- subset(changes, field == "bug_status"
+   & new.value == "VERIFIED")
> ver <- ver[order(ver$time), ]
> ver$n.changes <- 1:nrow(ver)
> with(ver, plot(n.changes ~ time, type="l"))
```



In this chart, some line segments are almost vertical (e.g., the line between 2002 and 2004). Such segments mark dates when there were mass updates.

Solution 2. Here is a sample R code to count updates by user, date, and comment. First, select only verifications, and then use the `merge` operation to associate them with their respective comments. After that, do the counting as usual. Here, the 6 records with higher counts are shown.

```
> library(plyr)
> ver <- subset(changes, field == "bug_status"
+   & new.value == "VERIFIED")
> ver$date <- as.Date(ver$time)
> full <- merge(ver, comments)
> cnt <- count(full, c("date", "user",
+                      "comment.md5"))
> cnt <- cnt[order(cnt$freq, decreasing=T), ]
> head(cnt[, c("date", "user", "freq")])
```

date	user	freq
2003-07-01	17822	1703
2003-07-01	17822	972
2003-07-02	17822	447
2005-07-12	182	437
2005-12-20	182	209
2005-07-13	182	181

IV. Old Wine Tastes Better

Problem

Discard bug reports that are work in progress.

Context

Bug reports change over time. Sometimes, one needs to classify a bug report according to the eventual occurence of some change. For example, has a bug been fixed? Is this bug report a duplicate? A negative answer may turn into a positive one with the arrival of new data.

Therefore, for recent bugs, the outcome may still be unknown. For instance, when training a model to predict which bugs get reopened and which do not, it is likely that recent bugs have not been reopened—*yet*, because some of them may be in the future. The question is: how long to wait for a change to happen before assuming that it will not evenutally happen?

Solution

For the sake of clarity, assume that you're interested in classifying bug reports according to whether they are eventually reopened or not. It should be easy to adapt this solution to any other outcome.

If a bug report is reopened within the available data, classify it as reopened. If not, then consider its lifetime, from creation to the last date available in the data. If the lifetime is long enough so that reopened bugs are expected to reopen within this time (i.e., it is above some *threshold*), then classify it as non-reopened. If the lifetime is short, however, it is hard to predict whether the bug report will eventually reopen, so just discard the bug from the analysis.

So, first of all, choose a threshold. Then, compute the probability that, given that a bug report did not reopen within the threshold, it will never reopen (let's call it α, or confidence). To do that, compute the proportion of bug reports that were reopened, r. Then, compute the proportion of older bug reports (i.e., which have lifetime greater than the threshold) that were not reopened within the threshold, t. The probability α can be approximated by the quotient r/t.

Assess whether the probability α is high enough (most data scientists find 0.95 to be an acceptable value). If it is not, choose another threshold and recompute α until you are satisfied. After finding an appropriate threshold, discard bugs with lifetime shorter than the threshold.

Discussion

It is a common mistake to keep recent temporal data in analyses. This is equivalent to choosing a threshold of 0.

Examples

Here's how to apply this pattern using R to analyze bug reopening. First of all, create a data frame `data`, augmenting `bugs` with information about their first reopening and their lifetime.

```
> library(data.table)
> reopenings <- data.table(changes)[
+   field == "bug_status" & new.value == "REOPENED",
+   list(time.first.reopen = min(time)), by=bug]
> data <- merge(bugs, reopenings, all.x=T)
> data$days.to.reopen <- as.numeric(
+   data$time.first.reopen - data$creation.time,
+   units="days")
> last.time <- max(data$modif.time)
> data$lifetime <- as.numeric(
+   last.time - data$creation.time,
+   units="days")
```

In this example, we use a threshold of 42 days (6 weeks). Use it to compute α (`alpha`):

```
> threshold <- 42
> older <- subset(data, lifetime > threshold)
> r <- sum(is.na(data$days.to.reopen)) / nrow(data)
> t <- sum(is.na(older$days.to.reopen)
+   | older$days.to.reopen > threshold) / nrow(older)
> alpha <- r / t
```

In this case, $\alpha = 0.95$, which is acceptable. Therefore, you should only keep bug reports older than 42 days—which are already stored in the variable `older`—and discard the rest. In this case, only 0.7% of the bugs needed to be discarded.

References

[1] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proc. of the 2008 Conf. of the Center for Adv. Studies on Collaborative Research*, ser. CASCON '08. ACM, 2008, pp. 23:304–23:318.

[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] 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.

[4] 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. [Online]. Available: http://dblp.uni-trier.de/db/conf/msr/msr2012.html#SouzaC12