Patterns for making sense of bug data / Patterns for cleaning and analyzing bug data

Rodrigo Souza Christina Chavez Roberto Bittencourt
Software Engineering Labs
Department of Computer Science - IM
Universidade Federal da Bahia (UFBA), Brazil
rodrigo@dcc.ufba.br, flach@dcc.ufba.br, roberto@uefs.br

***Abstract*—TODO [1]**

***Index Terms*—TODO; TODO; TODO; TODO**

I. Introduction

Someone said that data analysis is easy, the hard part is cleaning the data. This is true also of bug databases. You deal with data that may be incomplete, inconsistent, or just plain wrong [cite Bird, Aranda etc.]. Without proper guidance, it is easy to get lost. We hope to provide some hints for those adventurous enough to analyse bug data, by suggesting some filtering, and also general analyses that support more specific analyses.

Conventions: lines beginning with > denote R code.

II. Data

The R snippets refer to data extracted from Bugzilla databases. Other databases may have different schemas, so the scripts may need to be adapted.

Tables: bugs_activity (we'll call it changes), bugs, longdescs (we'll call it comments) Let's assume that we have imported such tables as a R data frame.

> head(events)

...

To ensure reproducibility, the full source code for this paper, together with data sets, is available at ...

III. Patterns

The format we use is:

-

Write a short story referencing the patterns, that shows how one typically uses them

IV. Pattern Cluster: Gather

A. Grab the Release Dates

Problem

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 tasks may be performed based on the proximity of a release date. For example, verification and validation tasks are expected before a release, and users that are not developers only report bugs for a version after it has been released.

It should be easy for a data analyst 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

There are common solutions to this problem: one involves a version control system, and the other one, the project's website.

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[1] 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. [2] to discover release dates for the project Eclipse/Platform.

Related Patterns

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

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

V. Patterns Cluster: Clean

A. Look Out For Mass Edits

Problem

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

Context

Changes to a bug report often the result of an effort made by developers to triage, fix or verify a bug. Such efforts take time. It is expected, for example, that a bug status is only changed to `VERIFIED` after a developer spends some time creating test scenarios, running the software, inspecting the source code changes etc.

However, on some bug data sets, one can find hundreds or thousands of bugs that are changed by the same developer within minutes or hours. These are mass edits and should not be interpreted as a burst of productivity. For example, if a developer changes the status of a thousand bug reports `VERIFIED` within a few hours, it does not mean that he verified a thousand bugs — that would be humanly impossible. More often than not, it just means that the repository needed some cleanup, by marking old bugs as `VERIFIED` to help developers keep track of current problems.

When analyzing bug data, mass edits should be identified and removed from the data set, as they may distort the results.

Solution

Mass edits 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 edit.

There at least three solutions to find mass edits: two based on number of changes over time, and one based on developer comments. Before applying a solution, a type of change must be chosen. In the following examples, it is assumed that we are interested in mass verifications, i.e., mass edits that change a bug status to `VERIFIED`, but the solutions apply to other types of change as well.

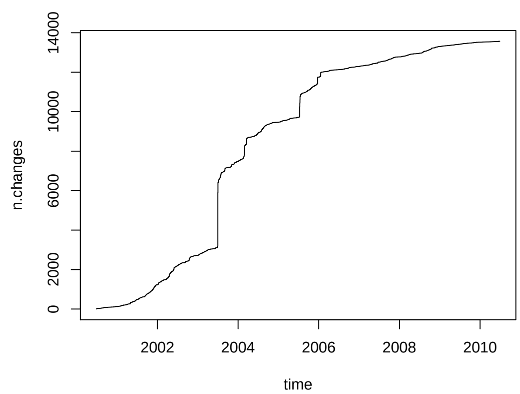
Solution 1: plot the accumulated number of verifications over time and look for steep ascents. The first solution is more visual, exploratory. Select only the changes that update the bug status to verified, along with the time of the change. Sort the changes by time and then plot the accumulated number 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 edit.

Here is the R code. Assume that `changes` is the list of changes to bug reports in NetBeans/Platform.

Center figure

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

Solution 2: count the number of changes per user per day. Then, sort by number of changes, from highest to lowest, and select pairs (user, date) with a high number of changes. The changes that match the selected pairs (user, date) are likely to be mass edits.

Here is the R code for the solution. Assume that `ver` holds the data that was computed in the previous solution. The counts are computed using the `count` function from the package `plyr` and assgined to the variable `cnt`. Here, the highest six counts are shown.

```
> library(plyr)
> ver$date <- as.Date(ver$time)
> cnt <- count(ver, c("date", "user"))
> cnt <- cnt[order(cnt$freq, decreasing=T), ]
> head(cnt)
```

date	user	freq
2003-07-01	17822	2706
2003-07-02	17822	531
2005-07-12	182	444
2005-12-20	182	313
2004-02-27	182	199
2005-07-13	182	182

Solution 3: count the number of changes per user per day with the same comment. Same as before, except that the comment is taken into account. For optimization purposes, the comment text can be replaced by its MD5 hash (or another hash function).

Here is the R code. Assume that we have a `comments` data frame. First, merge it with `ver` to select only comments related to verifications. Then, do the counting as usual. Here, the 6 records with higher counts are shown.

```
> library(plyr)
> 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

Discussion

The first solution to find mass edits is more visual, but less accurate, as it does not take into account the user who made the changes, and also more subjective. The second solution is numeric and takes into account the users, but does not distinguish regular changes and mass edits if they were made by the same user in the same day. The third solution addresses this problem, but is more computationally intensive, as it involves user's comments.

Examples

For example, Souza et al. [2] used the first two solutions. They used the first solution to build the chart shown in their paper's Figure 2. Then, they used the second solution to find change bursts with more than 50 changes by the same user on the same day. Such changes were considered mass edits and consequently discarded.

Related Patterns

How many changes in a day are considered normal and how many indicate mass edits? Be sure to [Choose a Suitable Threshold]. After finding mass edit candidates, it is a good idea to [Read the Fine Comments] to gain more confidence that the changes resulted from a mass edit.

B. Don't Be Confused

Problem

Assess the association between two categorical variables while considering variables tha may mask the association. / Discover categorical variables that mask the associaton between two other categorical variables.

Context

Measuring the associaton between two categorical variables gives some evidence of a causal relationship between them. However, an apparent association between two variables may be caused by a third variable that influences both. Or, on the contrary, a third variable may mask an existing association between two variables. Such variable is called a confounding variable.

Solution

Use mosaic plots. Mosaic plots are a visualization of contingency tables and help visualize the relationship between two or more categorical variables—although it can become hard to visualize with many variables.

A mosaic plot represents each cell in a contingency table by a rectangle with area proportional to the cell count.

This is hard to describe!

For categorical variables, use mosaic plots, stratified by possible confounding variables.

Tests of independence: fisher test and chi-squared test.

For continuous variables, use correlation matrices.

Discussion

Mosaic plots are easy to read with two variables, barely readable with 3 or 4, but start to become confusing with 5 or more. Mosaic matrices can be easier to visualize.

Examples

Use mosaic plots. use correlation matrices.

Related Patterns

C. Meet the teams / Know your subjects

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.

```
> resolutions <- subset(changes, field == 'resolution')
> status <- subset(changes, field == 'bug_status')
> t1 <- table(resolutions$user, resolutions$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$VERIFIED[t$verifier]) / total
> if (part < 0.5)
+   t$qe <- FALSE
> sum(t$qe)
[1] 0
```

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.

Related Patterns

What threshold to use to distinguish between quality engineer and other developers? What percentage of verifications a group of developers have 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].

D. *Old Wine Tastes Better*

(aka Let them mature / give 'em some time)

Problem

how to avoid analysing bugs that are not accurate

Context

it may be the case that a bug is later discovered to be invalid, or a fix is reopened, but you may not know this and consider the bug as valid or fixed, because you do not have enough history.

Solution

Discard newer bugs, that do not have enough history. Plot the distribution of bug durations (from open to last activity, or from open to a specific activity that you are interested as an outcome variable). (in my case, I discovered that more than 90% are reopened within 1 year, so it is probably safe to discard bugs that are newer than 1 year).

Related Patterns

Be sure to Choose a Suitable Threshold.

E. *Choose a Suitable Threshold*

F. *Count 'em all*

Problem

How to know if your variables are associated?

Solution

count the cases. Built a contingency table. After that you can use Fisher's exact test to assess association.

Related Patterns

[Classify] your data (make sure your [data is classified])

VI. Pattern Cluster: Analyze

A. *Read the Fine Messages*

(aka Read the Fine Comments)

Problem

How to know if a specific status change really represents what you think it represents?

Context

a specific status change may have different meanings on different projects. For example, when a bug is marked as FIXED, does it mean that a patch was submitted? That the fix is available in the version control system?

Solution

don't be afraid and read the comments of a few cases in which that status change occurs. Specially comments of bugs that are exceptional in one way or another.

Related Patterns

it is more productive to read comments that are related. You may want to [Classify] your data before .

B. *Know Your Project*

Problem

Context

when you do not know your project, you run the risk of making wrong assumptions about the data.

Solution

read developer documentation on the website. You may find guidelines, well defined roles and processes. Seek material in developer conferences.

Examples

eclipsecon, the talk "eclipse way". NetBeans docs talk about quality engineers.

C. *Know their aliases/names*

Problem

how to discover the multiple identities of a developer?

Solution

use the methods proposed by Bird et al. in Mining Email Social

D. *Eavesdropping*

(aka Be a regular reader/listener)

Problem

how to detect what is not explicit in the bug metadata?

Solution

do a text search. Maybe use regular patterns.

Examples

verification techniques reported by developers.

Related Patterns

read the fine comments.

E. *This Case is Classified*

(aka Classify your Cases / Make your cases classified / Separate the wheat from the chaff)

Problem

How to avoid analysing cases that you are not interested in?

Context

Solution

create scripts to automate the classification and enrich your data with derived observations.

Related Patterns

after classifying the cases, [Read the Fine Comments] of each class and verify if the classification is adequate

F. *Triangulate / Check*

Check your inferences by reading a sample of the data.

G. *Explore*

Read some data to get the idiosyncrasies of each project. See Read the Fine Comments, Know Your Project, Eavesdropping.

H. *Choose a suitable time grain*

day? week? month? release?

I. *Future Perfect*

When making predictions based on temporal data, you usually train your models on the available data, and then asks the question: how would the model perform on future data. The problem, of course, is that future data does not exist yet. A sensible approach to overcome this problem is to split your data set in two parts: earlier data and more recent data. The second part, recent data, represents the future from the point of view of an observer that only knows the earlier data. This way, you can train your model on the first part of the data and test it on the second part.

J. *Estimate time-related metrics*

- tempo de correcao - tempo de verificacao - tempo de latencia - etc.

Voce nao tem os dados exatos, tem que fazer simplificacoes, definir janelas razoaveis, definir intervalos de interesse.

References

[1] 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.

[2] 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