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
rodrigorgs@dcc.ufba.br, flach@dcc.ufba.br, roberto@uefs.br

***Abstract*—TODO 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)
...
```

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?

Context

releases are important events. They help to determine phases in the software lifecycle. The activity in a project that is near release date may be different from the activity in regular periods. For example, in Eclipse/Platform, verifications are much more frequent just before a release.

Solution

project websites do not usually keep info about all the previous releases. Fetch previous versions of the sites using The Internet Archive, archive.org. Other solution is look for tags in the version control system.

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

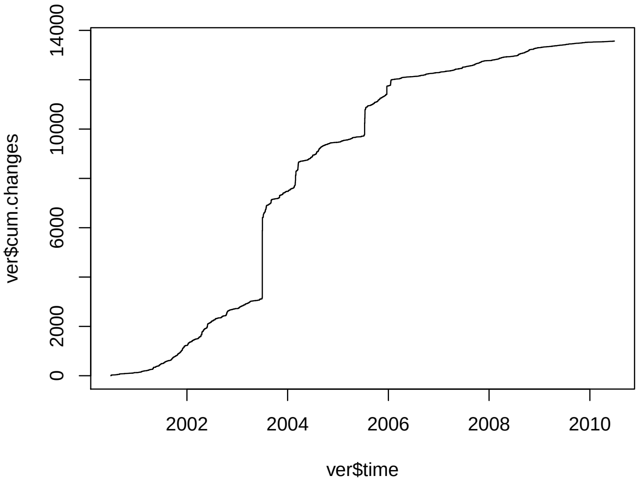
Mass edits are characterized by a large number of changes of the same type (e.g., marking a bug report as VERIFIED or changing a target milestone) made by a single developer in a short period of time. Often such changes are accompanied by a comment that is the same for all changes in the mass edit.

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.

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



(2) Choose a type of change you are interested

Construct a contingency table to count how many changes

(3) Perform a hash of the comments (this is a optimization to avoid working with the full comments). Compute a contingency table to count, for each hash, the number of occurences per day, per developer. You can sort the hashes by

(please note that there is a chance of hash collision, but it is unlikely, so such event, if it occurs, is probably sporadic and will not affect the analyses)

```
> #library(digest)
> #comments$hash <- sapply(comments$text, digest) # or sel
> #comments$date <- as.Date(comments$time)
> #
> #tab <- count(comments, c("hash", "date"))
> #tab <- tab[order(tab$freq, decreasing=TRUE), ]
> #head(tab)
```

Discussion

Examples

For example, at MSR 2012, Souza et al. [2] used the first two methods. See Figure XXX in their paper for a sample visualization of

Related Patterns

Be sure to Choose a SuitableThreshold. It is also a good idea to ReadTheFineMessages

B. 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.

C. Choose a Suitable Threshold

D. Don't be confused

Problem

confusion variables affect outcomes.

Context

Solution

use mosaic plots. use correlation matrices. Consequences: mosaic plots are easy to read with two variables, barely readable with 3 or 4, but start to become confusing with 5 or more.

E. Meet the teams / Know your subjects

Problem

How to discover developers' roles?

Context

Some developers are triagers, other are "fixers", there are also QA experts. When studying human factors, it is important to know that developers are not all equal and have different roles.

Solution

Analyze their activity in the bug tracking system, specially the status changes. Count the relative number of activities. Plot distribution of ratios. After discovering a group of developers with a specific role, see if the group is responsible for most of that activity in the project.

Related Patterns

Be sure to 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