Patterns for cleaning and analyzing 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 development process and the quality of the product being developed. Such data, however, is often incomplete and inaccurate, and thus must be analyzed with care. In this paper, we present patterns to help both novice and experienced data scientists to extract useful information from bug data, while avoiding common pitfalls.**

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

I. Introduction

R code in the patterns should go to example section.

Bug tracking systems store bug reports for a software project and keep track of all progress and discussion for each bug. Therefore, they provide information not just about the defects on the software, but also about the development process. This is why bug reports are such an invaluable source of information for data scientists studying software.

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, 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 to help users and developers keep track of current bug reports [4].

fall into common mistakes

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 from cleaning up bug data up to deriving meaningful conclusions from bug reports.

Each solution is presented in a structured form called pattern. A pattern contains 7 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; (6) *examples* of the pattern in use; (7) *related patterns*.

II. Data and Code

Each pattern is illustrated by examples when necessary. We show how to analyze Bugzilla[1] data using R, a programming language for data analysis[2]. The examples use data from the NetBeans/Platform project, made available in the Mining Challenge of 2011[3]. All the data and code used in the examples is available online[4].

In the examples, we use three tables (i.e., R data frames) to store bug data. Such tables are based on Bugzilla's database schema, except some tables and columns are renamed for clarity.

The `bugs` table contains general information about bug reports, which are identified by unique numbers (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`).

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 `comments` table contains comments that *users* added to a *bug report* at some point in *time*.

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* at some point in *time*. (In this context, user denotes a user of the bug tracking system, which can be a developer or a final user.)

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 *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

[1]Bugzilla is a popular tracking system, available at http://www.bugzilla.org/
[2]Available at http://www.r-project.org/
[3]http://2011.msrconf.org/msr-challenge.html
[4]https://github.com/rodrigorgs/dapse13-bugpatterns

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.

is based no Bugzilla's database schema, Data from other bts should be similar.
The R snippets refer to data extracted from Bugzilla databases. Other databases may have different schemas, so the scripts may need to be adapted.
> head(events)
...
To ensure reproducibility, the full source code for this paper, together with data sets, is available at ...
Conventions: lines beginning with > denote R code.

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

III. 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 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[5] 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.

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

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. [4] 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.

IV. 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. 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 updates 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 issues.

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.

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

Solution

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 updates that change a bug status to VERIFIED, but the solutions apply to other types of change as well.

Solution 1. 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 update.

Solution 2. Count how many times a day a user has changed a field in a bug report and used the same comment text. In other words, 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, and also more subjective. The second solution is numeric and takes into account the users and their comments, but is more computationally intensive.

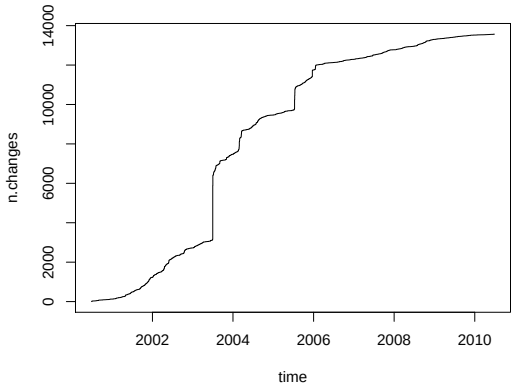
Examples

For example, Souza et al. [4] used the first 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 updates and consequently discarded.

Solution 1. 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 updates.

Solution 2. 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)
> 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

Related Patterns

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

V. Don't Be Confused

Problem

Assess the association between two categorical variables while considering variables that 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.

This is hard to describe!

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

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

VI. 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.

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].

VII. Choose a Suitable Threshold

Problem

Find a threshold for a metric so that data is separated into two distinct groups.

Context

Metrics are often used to help classify a data point into one of two groups. For example, they may help to determine if a data point is an outlier or not. When the mapping between data and groups is known for a subset of the data, then standard data mining techniques, such as cross-validation, can be leveraged to yield a threshold. The problem is more difficult if there is no information about the composition of the groups.

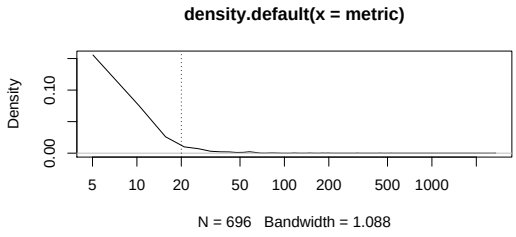
Solution

In fact, this is a hard, highly context-dependant problem, and it is related to finding outliers. Usually, the distribution of a metric is analyzed by means of a density plot, a box plot, or a bean plot (that mixes concepts from the later two). For heavily skewed data, logarithmic scales are used.

For example, consider the metric "number of verifications per developer and day in a bug tracking system" ("verification" means that the bug status was changed to `VERIFIED`. We would like to find a threshold such as, when the metric is greater than the threshold, it suggests that the verifications were actually just a mass verification for cleanup purposes.

It does not make sense to talk about a mass verification with just one or two verifications, so let's consider only greater values, then plot the density estimates. Because the distribution is heavily skewed, a logarithmic scale is used:

```
> metric <- metric[metric > 2]
> plot(density(metric), log="x")
> abline(v=20, lty=3)
```



A value around 20 appears to be a boundary between frequent and unfrequent values for the metric, as shown with a vertical line, and thus may be a good choice for a threshold to distinguish between usual and unusual numbers of verifications per developer and day.

Discussion

Looking for a suitable threshold can be highly subjective, and depends on the problem at hand. For example, if the problem requires an answer with high precision, it is better to choose a more restrictive threshold. Sometimes, thresholds can be inferred from natural human limits. For example, how many bugs can a developer fix in a single day? A reasonable answer can be used as a threshold to filter out invalid data.

It can be that the metric chosen is not sufficient to distinguish between the two groups. In that case, choose another metric or a combination of metrics.

Examples

Souza et al. [4] used density plots of the ratio between verifications and fixes by developers in order to find a threshold to help identify quality engineers.

Related Patterns

This pattern can be used together with several other patterns, e.g., [Mass Edits], [Know Your Subjects] etc.

VIII. Old Wine Tastes Better

aka Let them mature / give 'em some time / a long future ahead / the benefit of the doubt

Problem

Discard bug reports that are work in progress.

Context

Temporal logic: bugs that will, *eventually*, be reopened.

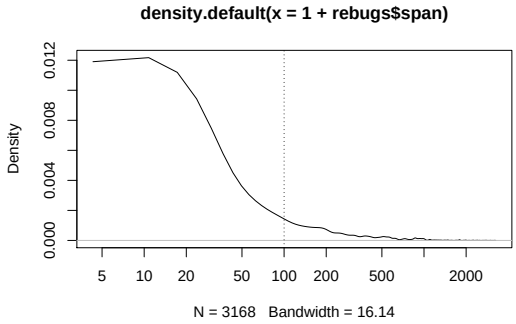
Bug reports change over time. Sometimes, one needs to classify a bug report according to the occurence of some change. For example, has a bug been fixed? Is this bug report a duplicate? A negative answer may turn into a positive one with the arrival of new data.

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 the bug to be reopened before classifying it as not reopened?

Solution

Analyze the time spans of bug reports from their creation to the outcome being studied. For example, let's consider bug reopenings as the outcome. Select only reopened bugs and use a box plot or a density plot to visualize the distribution of time spans from creation to reopening. Then choose a threshold. It should be a time span that is large enough so that most reopened bug reports are reopened within that span. R code:

```
> library(data.table)
> changes <- data.table(changes)
> reopenings <- changes[
+   field == "bug_status" & new.value == "REOPENED",
+   list(first.reopen = min(time)),
+   by=bug]
> rebugs <- merge(bugs, reopenings)
> rebugs$span <- as.numeric(
+         rebugs$first.reopen - rebugs$creation.time,
+         units="days")
> plot(density(1 + rebugs$span), log="x")
> abline(v=100, lty=3)
```



Suppose 100 days was chosen as threshold. Now, compute the life span of the bug reports, from their creation to the time of the last change in the data set. Discard all bugs whose life span is less than the threshold. Such bugs are too recent: they do not have enough data to conclude if they will be reopened or not.

```
> threshold <- 100
> present <- max(changes$time)
> bugs$life.span <- present - bugs$creation.time
> valid.bugs <- subset(bugs, life.span >= threshold)
```

After discarding recent bugs according to this criteria, assume that bugs that were not reopened will also never be reopened in the future.

Discussion

It is a common mistake to keep recent temporal data in analyses. This is equivalent to choosing a threshold of 0, i.e., even if data is limited to a few days, current information is used to predict the future.

The choice of a threshold can be based on the desired confidence that current data can be extrapolated to the future. For example, consider a confidence level of 0.95 that bugs that were not reopened within a time span will not ever be reopened. The confidence can be approximated as the probability that a bug will never reopen divided by the probability that a bug will not reopen within the given time span[6].

```
> threshold <- 100
> p.never.reopened <- 1 - (nrow(rebugs) / nrow(bugs))
> prob.not.reopened.within.timespan <-
+   ((nrow(bugs) - nrow(rebugs)) + sum(rebugs$span > threshold))
> confidence <- p.never.reopened / prob.not.reopened.within.time
```

The confidence for a threshold of 100 days is 0.97.

[6]The proof is a direct application of Bayes' theorem.

Examples

Related Patterns

Be sure to [Choose a Suitable Threshold].

IX. 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])

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