Operational Data are Missing, Incorrect, and Decontextualized

Audris Mockus

January 12, 2016

1 Background

Big data is upon us and the data scientist is the hottest profession. In software engineering, analyzing software development data from social networks, issue trackers, or version control systems is proliferating.

Data science is defined as generalizable extraction of knowledge from data [2]. While this definition sounds quite profound, how exactly does data science differs from traditional science? Traditional science extract knowledge [1]By knowledge here we mean a useful model} from experimental data. Data science, on the other hand, claims to be able to extract knowledge from all data and, in fact, many of the success stories of data science rely on Operational Data (OD) [6]: data left as traces from operational support tools and then integrated with more traditional data. With many human activities increasingly transacted partially or entirely within the digital domain, voluminous traces of operational data wait to be explored, understood, and used. It is quite tempting to apply a variety of statistical and machine learning techniques that have been refined on experimental data to OD, but pitfalls await as OD is not experimental data. The following three examples illustrate the principal drawbacks of operational data: missing observations, incorrect data, and unidentified context.

2 Examples

2.1 Missing data

During World War II, to minimize bomber losses to enemy fire, researchers from the Center for Naval Analyses had conducted a study of the damage

[1]{

done to aircraft that had returned from missions, and had recommended that armor be added to the areas that showed the most damage. Statistician Wald pointed out that only aircraft that had survived were observed and proposed to reinforce the areas with no damage, since planes hit in such areas would be lost. We will revisit this example later, but the key lesson is that many of the events observed in software development, such as defects, are contingent on numerous factors, such as the extent of software use and the ability of users to report issues, that are typically not observed. More defects reported by users, consequently, yet counter-intuitively, tend to indicate better quality software.

2.2 How to Augment?

The simplest way to measure the defectiveness of software as perceived by end users is to normalize the number of user reported defects by the extent of use as, for example, described in [3]. The extent of use may be more easily and readily available for common software delivered as a service. Other types of events may need similar normalization. For example, a code commit, or a joining of a new participant, should be considered in context of what is not observed, such as commits that could have been made and the pool of potentially interested and capable participants who chose not to join. A discussion below suggests techniques that could be used to augment the operational data with critical missing pieces.

2.3 Incorrect data

In a large telecommunication equipment company with a large customer support business, the reliability of systems is critical because technicians need to be dispatched at great expense to repair the broken parts. When the flash memory cards were started to be used instead of CDs for system installation and operation, it was noticed that service technicians tend to replace a large number of flash cards: a number tens of times larger than would have been expected based on the flash card manufacturer specifications. A natural reaction would have been to confront the manufacturer and demand compensation for not meeting the specs. However, the reliability obtained based on the number of flash cards replaces is obviously wrong for someone understanding how service technicians work. Many of the problems in complex systems are transient or hard to reproduce and replacing the flash card is the simplest step a technician can do. If the system operates normally after the replacement, technicians do not have time or tools to prove that the culprit was indeed a failed flash card. While the services operational data clearly indicates how many flash cards were replaced because of failure, this data is incorrect if used to assess the reliability of the flash card. Virtually all of operational data in software engineering is similarly incorrect. For example, a developer may fix a bug and declare it to be so, but often they fix another bug or cause the symptom of the bug to disappear under certain conditions. It is not unusual to see the same bug being fixed many times over many years until, if ever, the ultimate fix is implemented.

2.4 How to Correct?

The operational data record can not be taken at face value. It can only get meaning from the understanding of what the actor though (and did not think), knew (and did not know), and did (and did not do) at the time the event was recorded. Fortunately, the behavior of individuals and groups have regularities. Identifying such regularities and using them to fix problematic data can be done as described in, for example, [12].

2.5 Decontextualized data

Some interesting metabolic theories, for example, alometric scaling [9], postulate a relationship between the body size and the heart rate. Measures of hart rate and sizes of animals are observed from tiny mice to huge elephants and a relationship is found to be $\frac{3}{4}$ power law. Obviously, comparing animals of such diverse sizes is certainly an interesting intellectual activity and, in this particular case, it may reveal some fundamental aspects of animal metabolism. Similarly in software development, peoples productivity, source code files, and software projects vary greatly in size with scales that differ by much more than the size of mice vs elephant. These artifacts of disparate sizes are recorded the same in the operational support tools: a file with 10M lines is treated the same as a file with one line or as a binary file where the lines of code are no longer meaningful. Unlike in the example of a grand unifying metabolism theory noted above, it often makes absolutely no sense to treat these disparate files, defects, projects, and other artifacts as being the same. The only aspect that unifies them is that they are recorded the same way by operational support tools, but otherwise have no inherent relationship that is worth theorizing about.

2.6 How to Identify Context

A basic understanding on the types of activities (fixing, branching) and roles (developer, tester, builder) in conjunction with how these activities and roles may be reflected in the specific event patterns can provide a way to contextualize events. More generally, the example illustrates that stratifying operational data into different groups is a problem-specific exercise.

3 A Life of a Defect

Software defects are of concern to developers and users. The next few paragraphs will draw attention to identifying missing data and adjusting the analysis accordingly, identifying inaccurate values and correcting them, and in segmenting the the events into "mice" and "Elephants" in the context of software defects.

If we observe a defect fixed in a file it is instructive to think what had to happen for this to occur. First, the event is predicated on someone running, testing, or building the software. For vast majority of FLOSS project repositories hosted on major forges such as github, that premise is not likely. We should not be surprised that most projects do not have any fixes or, even more extremely, claim that these projects are more error-free than projects with bug fixes. Even when this premise is satisfied, the user has to be motivated and capable of enough to report the issue and do it in a way that allows developers to reproduce and fix it [13]. Once the issue is reported, the fix is predicated on developers willing to pay attention to it and having spare time to do it as well as the issue being important enough to be worth the effort needed to fix it. This reasoning suggests that fixed issues depend on existence of experienced user base and active development community. For example, the issues that end up being fixed may not be the ones that inexperienced user encounter or the chances of them being fixed may depend on how busy the developer community may be at a particular point in time.

In addition to the factors noted above, the issues would not "get fixed" if the developer does not note the issue in a commit message [1]. Different developers and different types of issues are likely to result in different chances of the issue ID being noted. Unfortunately these are just a small list of problems related to missing data in a single domain: the count of fixed issues.

For the same domain lets see how issues may be "incorrect". For example, an important part of the issue is the affected component of the system: it is often very difficult for issue reporters to get it right [11, 10]. The fix date for an issue may not accurately represent its actual fix date [12]. Finally, the issue description may be often incorrect. An extreme example involved highly reliable software where under mysterious conditions certain table was filling up too fast, causing the system to restart. Via simple search of past fixes I found a fix describing exactly the same problem that was delivered to a major customer six months earlier. Celebration? Alas, even though the fix mentioned the right table, it was actually a fix for a different table and was unrelated to the problem at hand. Why was the description incorrect? It was written by the issue reporter and, even though, developer has fixed it, there was no compelling reason to change the description. Analysis of serious defect related to a synchronization issue revealed fixes spanning seven years [8] all claiming to have fixed the issues for the problem to reappear again. Basic techniques to use natural constraints to identify and correct some errors in issue data are described in [12].

As noted above, an issue reported by one person may not be an issues for another person. This maxim holds even stronger when comparing distinct projects. It is, therefore surprising to think that a defect discovered and fixed for, for example, flight control software, would be in any way similar to a layout issue associated for a specific JavaScript framework. Similarly, an issue in Bugzilla used to track code inspection results is probably quite unlike an issues use to report a security vulnerability. In both of these cases the same or similar operational support tool (issue tracker) is used, but the fact that all trackable items in an issue tracker are "issues", does not provide a mandate to put them into the same category and analyze deep relationships as in the case of the metabolic theory.

4 What to do?

The three examples point out key differences between operational data and experimental data. In order to apply the wealth of techniques developed for experimental data we first need to bring operational data to the quality standards associated with experimental data. It is helpful to think about OD as precise but tricky-to-use measurement apparatus. As with any precise instruments that need extensive tuning and calibration, opportunities for misuse abound. Having a clear understanding of how OD came to be and developing practices on how to use it effectively are essential. Unlike instruments measuring natural phenomena, this apparatus works on traces left by operational support tools, and, as the activities involving these tools change and the tools evolve, the measurement apparatus will have to be updated or the measurements will loose accuracy.

The examples suggest approaches to spend "the 98% of the effort spent that goes into data preparation and data cleaning activities that precede data analysis." While it is impossible to describe all possible traps that await an eager explorer of the operational data, there are a number of steps that can (and should) be taken to address some of the issues noted in various publications, for example, [5, 4].

Typically, the first step is to understand how the operational data get recorded. The best approach is to select a sample of actors to observe them doing their work and then compare to what has been recorded. If it is not possible to observe the action directly, the actors can be identified from the operational data and asked about the recorded events and our interpretations of what they mean. Operational data that can not be subject to validation can not be trusted for any downstream analysis.

Fortunately, narrowing the domain to software engineering in general and focusing on common developer actions such as code commits and issue handling can bring the necessary understanding on how to interpret the recorded events, how to model behavior in order to identify and correct inaccuracies, and how to separate events by context. Unfortunately, none of these tasks is trivial, for example, separating defects by priority inferred from the number of users affected [7] or a data-driven way to add context as done in, for example, {ZMKZ14}. This is what makes operational data such an interesting area to play with and make discoveries.

5 References

References

[1] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. The missing links: bugs and bug-fix commits. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 97–106. ACM, 2010.

[2] Vasant Dhar. Data science and prediction. *Commun. ACM*, 56(12):64–73, December 2013.

[3] R. Hackbarth, A. Mockus, J. Palframan, and R. Sethi. Customer quality improvement of software systems. *Software, IEEE*, (99):1–1, 2015.

[4] Audris Mockus. Software support tools and experimental work. In V Basili and et al, editors, *Empirical Software Engineering Issues: Critical Assessments and Future Directions*, volume LNCS 4336, pages 91–99. Springer, 2007.

[5] Audris Mockus. Missing data in software engineering. In J. Singer et al., editor, *Guide to Advanced Empirical Software Engineering*, pages 185–200. Springer-Verlag, 2008.

[6] Audris Mockus. Engineering big data solutions. In *ICSE'14 FOSE*, 2014.

[7] Audris Mockus, Roy T. Fielding, and James Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):1–38, July 2002.

[8] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. High-impact defects: a study of breakage and surprise defects. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 300–310, New York, NY, USA, 2011. ACM.

[9] Geoffrey B. West, James H. Brown, and Brian J. Enquist. A general model for the origin of allometric scaling laws in biology. *Science*, 276(5309):122–126, 1997.

[10] Jialiang Xie, Qimu Zhengand, Minghui Zhou, and Audris Mockus. Product assignment recommender. In *ICSE'14 Demonstrations*, 2014.

[11] Jialiang Xie, Minghui Zhou, and Audris Mockus. Impact of triage: a study of Mozilla and Gnome. In *ESEM'13*, 2013.

[12] Qimu Zheng, Audris Mockus, and Minghui Zhou. A method to identify and correct problematic software activity data: Exploiting capacity constraints and data redundancies. In *ESEC/FSE'15*, Bergamo, Italy, 2015. ACM.

[13] Minghui Zhou and Audris Mockus. What make valuable contributors: Willingness and opportunity in oss community. *IEEE Transations on Software Engineering*, 2013. submitted.