A Method to Identify and Correct Problematic Software Activity Data: Exploiting Capacity Constraints and Data Redundancies

Qimu Zheng[1]
zheng.qm@163.com

Audris Mockus[2]
audris@utk.edu

Minghui Zhou[1]*
zhmh@pku.edu.cn

[1]School of Electronics Engineering and Computer Science, Peking University
Key Laboratory of High Confidence Software Technologies, Ministry of Education
Beijing 100871, China
[2]University of Tennessee
1520 Middle Drive Knoxville, TN 37996-2250, USA

ABSTRACT

Mining software repositories to understand and improve software development is a common approach in research and practice. The operational data obtained from these repositories often do not faithfully represent the intended aspects of software development and, therefore, may jeopardize the conclusions derived from it. We propose an approach to identify problematic values based on the constraints of software development and to correct such values using data redundancies. We investigate the approach using issue and commit data of Mozilla project. In particular, we identified problematic data in four types of events and found the fraction of problematic values to exceed 10% and rapidly rising. We found the corrected values to be 50% closer to the most accurate estimate of task completion time. Finally, we found that the models of time until fix changed substantially when data were corrected, with the corrected data providing a 20% better fit. We discuss how the approach may be generalized to other types of operational data to increase fidelity of software measurement in practice and in research.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*process metrics*

General Terms

Measurement, Human Factors

Keywords

data quality, mining software repositories, capacity constraint, data redundancy

1. INTRODUCTION

Operational support tools such as issue tracking system (ITS), version control system (VCS), mailing lists, forums, and others, are broadly adopted by software projects, and the operational data produced and consumed by these tools are crucial for the effective operation of software development. For example, Dabbish et al. [6], argue that the data in the operational systems, such as GitHub, are actively being used by developers seeking to learn, share code, and for other key tasks, such as assessing the quality and activity of software projects. Operational data are also heavily used in software engineering research, for example, in the field of mining software repositories and in empirical software engineering. However, as extensively documented in the section on related work, operational data often do not faithfully represent the intended aspects of software development and, therefore, may jeopardize the conclusions derived from it. Such problematic data affects the conclusions presented in the academic work, see, e.g., [5, 17] and leads to poor decisions in software development, see, e.g., [26] showing that Mozilla had over 21% rate of mistaken product assignments affecting the quality of software and increasing effort and lead times.

ITS, for example, contains activities of individuals initiating and completing software project tasks. Virtually every ITS produces reports on issue resolution. That information is commonly used in practice to measure progress and select issues to be worked on. For example, the time until fix is obtained as the duration of time between issue creation and resolution date recorded in the ITS. The values recorded in the ITS, however, tend to strongly vary with the practices used by projects and individuals. For example, an individual may write a script to clean ITS by closing a large number of dormant issues. Such cleanup would produce questionable fix dates for the issues involved and an exceptionally high productivity for the individual recorded as the resolver.

In this study we propose a method to identify and correct such problematic activity data. The first premise of this study is that physical constraints tend to bound what could be accomplished by any individual or group over a fixed period of time. For example, an individual may be able to complete only a limited number of tasks in any given time interval. The mere existence of such constraints provides information that can help us identify erroneous data that

*Corresponding author.

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
ACM. 978-1-4503-3675-8/15/08...
http://dx.doi.org/10.1145/2786805.2786866

violate them. The bounds on actor productivity, even if unknown, can be estimated from the operational data as described below.

The second premise is that the operational data tend to be highly redundant with numerous types of events that could be used to measure the quantity of interest. This redundancy, as in the information theory, can help us to correct erroneous data using redundant events.

We, therefore, use the existence of real constraints in software development to identify problematic values and to correct such values by selecting redundant observations that are more likely to be correct and evaluate our approach on Mozilla ITS and VCS. In particular, we answer the following research questions.

RQ1 Is it possible to identify and fix incorrect task completion dates in software development? We propose to use a Poisson distribution for individuals' productivity to identify likely inaccurate task completion dates based on issue resolution events and replace such problematic data with redundant events associated with the same task.

RQ2 What are the specific mechanisms in Mozilla that result in erroneous issue fix completion dates? Manual analysis of 200 issues with erroneous dates suggests three common mechanisms: issues tracked in other system, long-dormant issues, and issues linked to committed patches in the VCS tend to be batch-fixed.

RQ3 Can redundancies in operational data provide alternative task-completion dates? We used alternative observations of the last comment posted for an issue to correct the issue resolution dates and authors.

RQ4 Are corrected data more accurate than uncorrected data? By comparing corrected and uncorrected values to the last commit date in the VCS we found corrected values to be 50% more accurate.

RQ5 How redundant are the alternative observations? We found that when the primary observations are incorrect the redundant observations are also more likely to be incorrect, but, despite that, they provide ample redundancy with more than 90% of them being non-problematic.

RQ6 Does the data correction matter in common models of time until fix? We fit a model commonly found in the literature and found both its structure and the fit to change substantially after data correction.

RQ7 Can the approach be generalized to other types of events and tasks? We identify likely incorrect observations for four types of events and discuss how to apply the approach more broadly.

The rest of the paper is organized as follows. We describe the related work in Section 2. We propose a method to identify and correct problematic values in Section 3, and apply the approach to correct issue fix dates in Section 4. We elaborate on the limitations in Section 5 and discuss how to generalize the method to other types of operational data in Section 6.

2. RELATED WORK

As operational data in software development has been increasingly used in research and practice, the concerns about its accuracy and completeness have been drawing an increasing attention. We consider related work from two aspects: the extensive use of ITS data to conduct analysis, and research on operational data quality.

Being one of the most important software repositories, ITS has been widely mined to measure a variety of aspects in software development. Some of the studies use the number of issues to measure effort or performance, e.g., using the number of completed issues to measure team effort [23] and productivity [8], or using the number of defects fixed to measure individual performance [7, 26]. Other studies focus on improving issue resolution practices. For example, mining the history to recommend right issues for people [1, 28], and comparing the similarity between issue reports to detect duplicate reports [22]. Bug fixing is an important topic with a substantial amount of literature. Common questions include: which bugs get fixed [11]? how long it takes to fix a bug [18]? Guo et al. [11] performed an empirical study to characterize factors that affect which bugs get fixed in Windows Vista and Windows 7. Kim et al. [18] studied the bug-fix time of files in ArgoUML and PostgreSQL by identifying when bugs are introduced and when they are fixed. However, as highlighted by this study, the above metrics derived from operational data may suffer from the variation of how people practice software development.

The problems with operational data have been an increasing focus of attention. For example, Bird et al. [5] studied the linkages between VCS and ITS and their effects on research result. They investigated historical data from several software projects and found strong evidence of systematic bias. The biases found by Bird et al. were later verified by Thanh et al. [25] for commercial projects. Kim et al.[19] proposed approaches to deal with the unlinked bugs. They measured the impact of noise in the data and proposed an classification algorithm for identifying such noise. Bachmann et al. [2] presented tools for reverse-engineering link data. Their tool enables users to quickly find and examine relevant changes, and annotate them as desired.

Errors in issue tracking data and their effects on defect prediction models are investigated in [24] and [12]. Their findings show that issue report classifications are unreliable. Due to the noise in issue tracking data, files tend to be wrongly marked to be error-prone and wrongly predicted to be error-prone. The noise in code commit data is studied in, e.g., [13], which investigated whether a single commit includes several tasks. The use of constraints in commercial projects was exploited in, for example, [10], where an algorithm that distributes monthly developer effort to individual tasks is evaluated. The accuracy of a triage activity in ITS is evaluated in [27]. A method that compares an intermediate value of an attribute in ITS to its final value is used to estimate the odds that a product attribute in the ITS is incorrect. This method is used to illustrate how the incorrect product attribute sends the issue to the wrong product team leading to wasted effort and issue resolution delays [26]. Various research challenges associated with operational data are outlined in, for example, [20].

A comprehensive review of outlier detection is presented by Hodge and Austin [14]. Our approach of using a mixture model is similar to approaches proposed by Yamanishi et al. [29]. The proposed approach incorporates actual data generation mechanism in operational data within software development context to construct more accurate models that are based on the understanding of the processes generating the data. This makes it possible to identify the outliers more precisely. Furthermore, the outlier detection work typically assumes that the outliers will be removed from the sample or accommodated (as in robust statistical methods). Removing or ignoring outliers would, unfortunately, bias the sample and analysis results as we show in this study. We, therefore, unlike in the prior work, focus on providing a theoretical basis for the methods used to identify and correct problematic observations in operational data.

3. METHOD

In this section we attempt to answer RQ1 by relying on the premise that constraints may serve as an information source and that operational data are highly redundant. While the idea of using constraints as an information source is very general, we discuss a specific application of it to identify problematic values in activity streams represented by different types of events in software operational data.

It is common for operational data to include activity streams: sequences of events that represent actions. Posts to a mailing list, status changes to an issue, or commits to a version control system represent such event streams. Considering the complexity of software development process and the variation of operational support tools, an event in one system is often associated with events in other systems. For example, before closing an issue in ITS, a code commit to resolve the issue may be submitted in VCS, and the code review discussion may have been through the mailing list. These additional events in any of the associated operational support tools may provide redundant events in case the primary events are identified as problematic.

In software mining, certain events from activity streams are used to estimate the task completion time. In cases when the task represents a nontrivial amount of effort as, for example, in code changes or in issue fixes, it is reasonable to expect that an individual will have a limited capacity to perform such tasks and the completion times (if accurate) can not happen simultaneously or be very close to each other for two distinct and nontrivial tasks. However, this constraint, while enforced in the real world, often does not hold in the operational data because the event timestamps are imperfect representations of task completion times. More specifically, the events representing the task completion date may contain a date far from the actual task completion date or the event may be associated with a person who was not involved in the task. Such inaccuracies may jeopardize conclusion drawn from the analysis relying on these events. The existence of constraints in the physical world can help us identify such clearly erroneous data by assuming or estimating productivity bounds of actors. We describe an approach to determine such constraints below.

Lets denote the operational data event time that we use to approximate the completion time of task i by actor j as t_{ij}. Assuming the productivity of actor j as p_j and the difficulty of task i as e_i we would have the duration between the events be $t_{ij} - t_{i-1,j} \geq \frac{e_i}{p_j}$ if the tasks are done in sequence[1]. The formula shows that the duration between such events can not approach zero unless the effort for the task approaches zero ($e_i \to 0$) or the productivity of the actor tends to infinity ($p_j \to \infty$). For simplicity assuming an exponential distribution of the task completion inter-arrival times, the count of events over any specific interval of time would follow a Poisson distribution with the intensity defined by some function of the ratio $\frac{e_i}{p_j}$ for the events occurring during that interval. We, therefore, expect the count of task completions to follow a Poisson distribution[2]. If we observe extreme outliers in these counts, it is reasonable to suspect that the data may be incorrect. We can use various approaches to detect outliers, for example by assuming that the operational data values come from a mixture distribution: with probability θ the observation is accurate (and comes from a Poisson distribution parameterized by λ_{good}) and with $1-\theta$ the observation is inaccurate (and comes from a Poisson distribution parameterized by λ_{bad}).

To get some idea of what λ_{good} would look, we need to gauge the distribution of $\frac{e_i}{p_j}$ based on the understanding of real world of actors and their tasks. To gauge the distribution of θ and λ_{bad}, we need to investigate the mechanisms by which the operational data get corrupted (i.e., do not reflect the intended task completion times). Such investigation may also reveal more precisely the distributions of normal and erroneous completion times. If the direct access to the actors and the ability to understand what happens in reality is limited, we can still rely on the statistical properties of the observed distribution and use mixture models like described above to estimate the probability of erroneous observation θ and the λ_{good} and λ_{bad}.

For simplicity, lets assume that $\lambda_{bad} >> \lambda_{good}$[3]. The following simple rule can be used to identify erroneous values:

$$isProblematic(n_j) = \begin{cases} 1 & if \quad \sum_i I(t_{ij} \in d) \geq cut \\ 0 & if \quad \sum_i I(t_{ij} \in d) < cut \end{cases} \qquad (1)$$

where $\sum_i I(t_{ij} \in d)$ is the number of task completions by developer j during day d and *cut* is a large enough number to ensure that non-problematic observations are highly unlikely to be marked as erroneous.

Unfortunately, once the erroneous data are identified, we often can not simply discard it: the subsequent analysis frequently assumes that data are complete. For example, if we want to know the number of issues fixed by each developer and understand what affects their performance, discarding problematic data would make the analysis biased. Removing incorrect data would, for example, affect the estimates of each developer's performance by a varying amount.

Once the errors are identified, we, therefore, need to correct the data. To do that we propose to use redundant observations from the event streams that are associated with the same task. As we mentioned above, the redundant observations could come from other systems used in software development. The redundant observations should be able to avoid the problems that the original data have while being a reasonable approximation of the needed measures. For example, the redundant observations should provide redundancy, i.e., they should not be uniformly problematic in cases when the original observations are problematic. The choice of redundant observations should also be based on the

[1] While the actors in software development tend to engage in several tasks at a time, this is often caused by the need to wait on input from other parties or tasks of higher priority interrupting tasks of lower priority.

[2] Albeit of varying intensity that depends on the difficulty of the tasks conducted during that interval. See Section 4.3.

[3] Evaluation described in Section 4.2 suggests this to be a reasonable assumption.

understanding of the problems in the original data and of the nature of the desired measures. It is not unusual to have numerous redundant observations and measures in software mining. For example, there are numerous events in ITS, such as issue status change, comment, report, attachment, and numerous other events with timestamps. Version control system and mailing list may contain events associated with the same issue. Developers may also use other means, such as twitter, to announce that they have managed to fix a problem. As with the identification of problematic observation, we may be able to select redundant observations based on available data. For example, any measure (defined as a function of the redundant observations) that well approximates non-problematic primary observations may be selected as a suitable alternative.

The identification of erroneous data and the correction process can be described as follows.

1. Gather events from available sources and link them by using tasks and individuals involved in these events.
2. Choose the primary event type to represent the desired task completion times.
3. Choose a set of redundant event types that approximate the desired task completion times and provide redundancy (are at least some times non-problematic when the primary event is problematic).
4. Obtain event times t_{ik} for task i and event type k ($k = 1$ represents the primary event and $k \neq 1$ represents redundant events). Use the distribution of t_{ik} for each k to identify problematic values[4]. Denote the identification method as $isProblematic(t_{ik})$. $isProblematic(t_{ik})$ should return the likelihood that the observed value t_{ik} is incorrect.
5. For task i, obtain values of $isProblematic(t_{ik})$ for each redundant observation type k.
6. Choose from the alternative observations via the following rule:

$$correct(t_i) = \begin{cases} \arg(\min_{k>1}(isProblematic(t_{ik})))\ if \\ \quad isProblematic(t_{i1}) \\ t_{i1}\ if\ !isProblematic(t_{i1}) \end{cases} \tag{2}$$

   Where the alternative k that has the lowest likelihood of being inaccurate is chosen to represent the task completion time.

4. AN ILLUSTRATION USING MOZILLA DATA

In this section we illustrate how to apply the approach described in Section 3 to identify and correct problematic issue fix events. We introduce Mozilla data used in this study in Section 4.1. In Section 4.2 we investigate the error mechanisms in Mozilla data, in particular, how the problems in the data can be detected and what mechanisms cause these abnormalities. Based on that understanding, we describe how we identify problematic issue fix dates of Mozilla in Section 4.3 and how we correct such values in Section 4.4. We investigate the extent of data redundancy in Section 4.6. We compare the accuracy and the impact on model fit for uncorrected and corrected values in Section 4.5 and Section 4.7.

Table 1: Quantiles of Productivity

10%	60%	70%	80%	90%	100%
1	1	2	2	3	209

4.1 Mozilla Data

Mozilla uses Bugzilla as the issue tracking system (ITS). We use the official Bugzilla dump provided by the Mozilla community (in January 2013) to conduct our analysis[5]. The data includes 774809 issues reported to Mozilla from September 1994 to January 2013. It records all activities conducted on these issues: from the time somebody reported an issue until the time somebody closed it (it also may remain open at the time when the Bugzilla dump was created). For each issue a sequence of events take place: issue is created, assigned, submitted, tested, and resolved[6]. The issue may also be reassigned, its attributes changed, comments, debugging traces, etc. added. Each such event has an associate date, time, the type of action, and the person performing the action.

Mozilla uses Mercurial version control system (VCS) to manage code. We cloned code bases of all products in Mozilla community and gather all available code commits in Feb 2014. The extracted commit logs include the code committer, commit time, changed files and revision information of every commit.

4.2 Investigation of Error Mechanism

In this section we discuss quantitative and qualitative approaches to discover the mechanisms by which erroneous data can get introduced to answer RQ2. In particular, we discuss how to determine if there may be a violation of productivity bound. We also describe manual inspection of a sample of problematic events to identify probable mechanisms by which errors are introduced.

4.2.1 The Existence of Problematic Data

We use issue resolution events in Bugzilla to calculate individual's productivity by counting the number of issues she fixed each day. More specifically, we count the number of events in which she changed the resolution of an issue to FIXED.

Overall we obtain 145654 positive observations (each observation is a person-day, i.e., the number of issues fixed by an individual in one of her days active in the project), and Table 1 shows the quantiles of these counts. Based on the assumption of physical constraints on individual's productivity (an individual can only accomplish a certain amount of tasks in a certain time unit, i.e., a day here), we are expecting a relatively low number of fixes per active day with a median of 1 and 90-th percentile of 3 fixes per day. However, as shown in Table 2, the Bugzilla records an extremely high number of fixes for some individual/day combinations, suggesting there is a violation of the assumption of limited productivity.

It is important to note that a large fraction of all issues are fixed during these most-productive days. For example, the

[4] For simplicity of description we talk about a single individual j_{ik} and omit the index j_{ik} from the discussion here.

[5] This Bugzilla dump provided by the Mozilla community has a higher quality than other snapshots retrieved on-line by people [30].

[6] Each "RESOLVED" issue has a resolution, e.g., FIXED, DUPLICATE. As the name suggests, FIXED means that the issue is fixed, DUPLICATE means that the reported issue is a duplicate of another issue.

Table 2: Exceptionally "Productive" Individuals (Based on Issue Fix Events)

Date	UserID	count
2012-04-03	392439	209
2009-02-16	14534	165
2009-09-16	69426	148
2009-02-20	14534	148
2010-05-11	91159	110
2012-12-22	302720	108
2011-01-04	374575	106
2009-02-19	14534	103
2009-02-28	14534	100
2012-05-04	373476	99

top 1% of most productive person/days fixed 33549 issues, counting for 12% of all the issues fixed.

We also calculate the number of person-days during each month when one person fixes ten or more issues on a single day. As the "Batch fixes" curve in Figure 1 shows, there is a significant increase in the number of outliers in recent years.

The kinds of charts presented in this section can be used to illustrate the extent of problems associated with any event type. While in this section we consider issue resolution events, in the later section we present the quality of last comment events and last VCS commit events.

4.2.2 Experiment to identify causes of problematic data

In this section we investigate issues where extremely high numbers of fixes on a single day by a single individual are recorded. Specifically, we sample 200 issues that are fixed in top 20 most productive person-days shown in Table 2. We manually check the comments of these issues and try to find out the reasons why these individuals can achieve such high productivity. According to our investigation, we find three possible reasons for these outliers.

Development Process Tracked By Other System.

One possible reason for the outliers is that some issues are tracked in other system. Specifically, in Mozilla, the development process of many issues is tracked by an agile development system called Pivotal Tracker. When the issue is marked as ACCEPTED in Pivotal Tracker, it indicates the completion of fixing activities. At the same time, a comment suggesting that the work on the issue has been accepted is automatically created in Bugzilla, but the issue status is not immediately changed. Instead, the issue status in Bugzilla is changed manually to FIXED some time after it is accepted in Pivotal Tracker. In this case, there is no more work on the issue after the issue is accepted in Pivotal Tracker. Therefore, the timestamp of the last comment before the issue is marked as fixed in Bugzilla, i.e. the timestamp of the comment created by Pivotal Tracker, is a more accurate estimation of issue fix date.

On 2012-04-03, Bugzilla recorded that the person with ID 392439 fixed 209 issues. It turns out that most of these issues are tracked by Pivotal Tracker and they have been accepted in that system. The person (ID 392439) simply changed the resolution of these resolved issues to FIXED. So the logins and timestamps that mark the issues as fixed recorded in the ITS system do not represent actual fixers or fix dates for those issues.

Dormant issues.

Sometimes when an issue receives no attention for a long time, Bugzilla administrators will decide to re-evaluate the issue and, possibly, change the state to fixed. In this case, the real working process stops at the time when the last comment (before the issue is marked as fixed) is created. Therefore, the timestamp of the last comment before the issue is marked as fixed is a more accurate estimation of issue fix date.

On 2009-02-16, Bugzilla recorded that the person with ID 14534 changed the resolution of 165 issues to FIXED. When we check these issues, we find that these issues have not been touched for years. This person was actually re-evaluating the issues and changing the status to FIXED.

Closing issues with committed patches.

In some cases, a patch is first checked into the VCS and the committer leaves a link in the associated issue of Bugzilla pointing to this code commit. After that, someone (not the committer) adds another comment noting the fact that the issue was fixed (with the same link pointing to the code commit) and then mark the issue as FIXED. In this case, the timestamp of the prior to the last comment appears to be closer to the real fix date.

On 2009-09-16, Bugzilla recorded that the person with ID 69426 marked 148 issues as FIXED. These issues are fixed with a link pointing to a commit in the VCS. The patches for these issues were first checked in, and later these issues were marked as fixed with a link pointing to those patches. Again, the logins and timestamps that are recorded to mark the issues as fixed do not represent the actual fixer and fix date for those issues.

In summary we found three mechanisms that led to incorrect date or actor. In all of these cases, a better proxy for task completion time appears to be the date of the last comment or of the last code commit date associated with the issue. Following the method described in Section 3, we use the last comment as the alternative event to perform correction described in the next sections.

4.3 Method to Identify Problematic Data

In this section we elaborate on implementing the problematic data identification method introduced in Section 3 for issue fix dates in Mozilla.

We model the number of issues fixed by an individual each day using a mixed Poisson distribution. As discussed in Section 3, the intensity of the task completion events may vary depending on the task difficulty, therefore each day d would produce a Poisson random variable with a different intensity λ_d. Our observations, therefore, would represent a mixture of Poisson random variables with different intensity parameters λ_d. Furthermore, we do not consider inactive days[7], obtaining a zero-truncated distribution. A commonly used mixing distribution for the intensity of Poisson random variables is Gamma distribution. The resulting mixture distribution is a zero-truncated negative binomial distribution.

We use R function ZANBI from package gamlss.dist to match quantiles of the observed person/day fix counts to the truncated negative binomial distribution and found that parameters $\mu = 0.64$ and $\sigma = 1$ (with $\nu = 1e - 7$ to en-

[7] Many participants in open source project, such as Mozilla, may not work full-time on the project and there are many other tasks apart from fixing issues, so counting days with zero fixes would also include days when the individual may not have spent any effort on fixing issues.

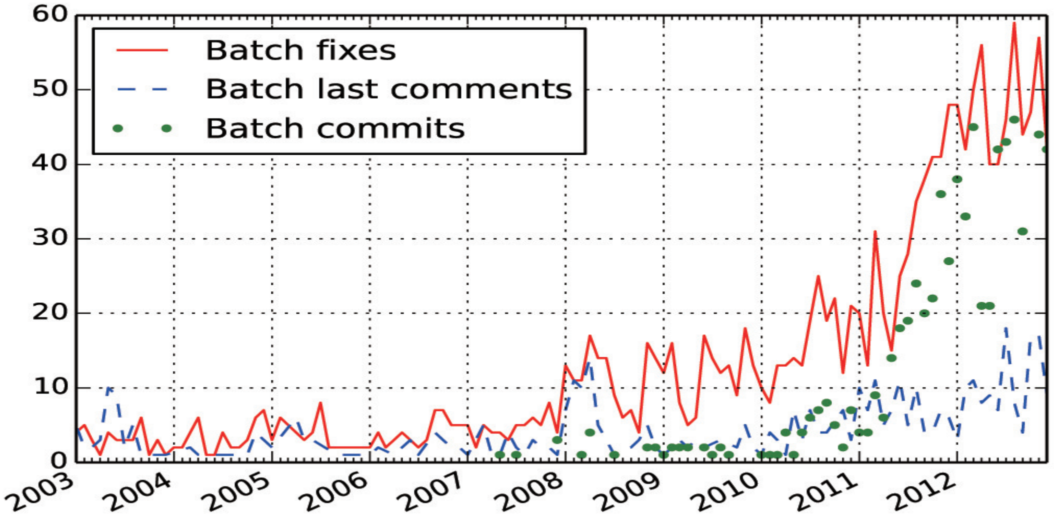

Figure 1: Batch-fix/batch-last-comment/batch-code-commit in each month

able zero-truncation) matched closely the 65-th and 85-th quantiles (corresponding to counts of two and three, respectively). Based on these parameters for the truncated negative binomial distribution, the probability of observing a count of 10 or larger is less than $1e-4$[8]. We, therefore, use 10 as the bound to determine physical productivity, i.e., we consider all events, where an individual fixes more than 10 issues in one day, as problematic. Using this approach we identify 36748 of 774809 observations as problematic.

4.4 The Choice of Redundant Observations for Data Correction

Once the problematic data are identified, we try to answer RQ3 by selecting redundant observations for issue fix dates.

The choice of redundant observations is based on the available data and on the investigations described in Sections 3 and 4.2. We found two types of events containing redundant information related to the fix date: the last comment and the last commit. Figure 1 shows the differences among redundant observations. Note that the curves "Batch fixes" and "Batch last comments" present the number of person-days per month where a single developer is associated with ten or more issue resolution and last-comment events, respectively. "Batch commits" counts person-days with 160 or more commits in VCS (see Section 6 for the identification of problematic commits). Since we use the last commit date to evaluate the performance of the correction technique (later in Section4.5), here we only use the last comment as the redundant event. The last comment event itself may be problematic as described in the method section. Table 3 summarizes problematic values obtained for the last comment events. We can observe that while it also has some clearly problematic counts, only the top two person/day combinations exceed 100, while nine such combinations exceed 100 in the issue fix events shown in Table 2. Whats more important, none of the actors in the two tables overlap, suggesting that problematic observations based on the resolution change events t_{i1} do not always coincide with problematic observations from the last comment event t_{i2}. We, therefore, would expect that $isProblematic_{t_{i2}}$ to be small for cases when $isProblematic_{t_i1}$ is high, thus providing reasonable levels of redundancy. Based on the analysis described in Section 4.3, the negative binomial distribution for the last comment event has $\mu = 0.4$ and $\sigma = 1$, with the probability to observe the count of ten or more such events per person-day being less than $1e-5$.

Table 3: Exceptionally "Productive" Individuals (Based on Last Comment Events)

Date	UserID	count
2003-06-12	23402	320
2000-12	14534	165
1999-03-05	3819	83
2001-08-10	23402	76
1999-02-03	3853	75
2005-06-01	422	67
2003-06-20	23402	61
2002-01-25	15368	59
2012-07-10	404027	52
2000-03-30	12352	52

4.5 Evaluation of Accuracy

About 16% of the issues are fixed with a link pointing to the commit in the VCS. The timestamp of the commit is more likely to reflect the completion of the fix activities as there are no subsequent modifications to the code in relation to the specific fix. To answer RQ4, we evaluate the accuracy of corrected data by comparing the timestamp with or without the proposed correction to the timestamp in the VCS.

We apply our correction method on the issues that can be linked to code commits and calculate quantiles of two metrics to evaluate the accuracy of uncorrected data and corrected data relative to timestamps in the VCS. We calculate both absolute errors and relative errors of uncorrected data and corrected data, where

$$\text{absolute error} = |\text{timestamp} - \text{vcs timestamp}|$$

[8] We follow Johnson's [16] recommendation to use a higher p-value for statistical evidence instead of the commonly used value of 0.05, because using the latter value often leads to unreproducible results [30]. Here we use $1e-4$.

Table 4: Absolute Error

Quantile	Uncorrected	Corrected
0.50	0 days 07:17:13	0 days 01:08:17
0.75	1 days 00:16:33	0 days 11:03:00
0.80	1 days 08:52:50	0 days 21:21:03
0.90	5 days 21:59:42	4 days 12:40:42
0.99	75 days 03:43:39	72 days 11:18:15

Table 5: Relative Error

Quantile	Uncorrected	Corrected
0.50	0.020502	0.007296
0.75	0.210515	0.077663
0.80	0.369903	0.154409
0.90	1.650358	0.850241
0.99	148.281840	73.326035

and

$$\text{relative error} = \frac{|\text{timestamp} - \text{vcs timestamp}|}{\text{vcs timestamp} - \text{issue creation time}}$$

Table 4 and 5 show that the corrected values have lower relative or absolute errors (are about 50% more accurate) than the uncorrected ones.

4.6 Investigation of Data Redundancies

Although we get a more accurate estimation of the task completion time as discussed in Section 4.5, a better understanding of the redundant data is needed. In this section, we investigate the extent of data redundancy to answer RQ5.

We use comment activities in Bugzilla to calculate individual's productivity by counting the number of last comments she made (the number of comments she made that are the last comment of an issue) each day. The number of last comments per day per person, like the number of issues fixed, should not exceed a certain level. We consider the last comment date as problematic if the commenter made 10 or more last comments per day.

Among the 230830 issues, 7887 issues (3.4%) are identified to have problematic last comment dates. Among the 36748 issues whose fix dates are (identified as) problematic, 3587 (9.8%) have problematic last comments. The result shows that the redundant observations are more likely to be problematic if the primary observations are problematic (Fisher test p-val$< 1e-5$). Despite that, they provide ample redundancy with more than 90% observations not marked as problematic when the primary events are problematic.

4.7 Impact on Model of Issue Resolution Interval

We further evaluate the approach by modeling time until fix using corrected and uncorrected data. While we already established that the errors cover a substantial portion of all data and are relatively large, the statistical models tend to be quite robust and may not change substantially even if the data were to be corrected. If we observe substantial differences in the model after data correction, this would suggest a positive answer to RQ6.

We fit two models to conduct the evaluation. First, we use a variety of predictors reported in the literature to explain the time until fix and report how the results change after correction. Second, we replicate an existing study that reported four predictors to be statistically significant.

The observations for the models are issues resolved with the resolution FIXED and with their fix dates identified as problematic by the approach described in Section 4.3. We use this subset to focus on revealing the potential problems resulting from uncorrected data. For all problematic issue fix dates, the redundant observations, i.e. the timestamps of the last comment, were available and were used in data correction as shown in Equation 2 in Section 3. We also remove issues created after 2012-04-28, as many of them may still be open at the time Mozilla dump was created. Finally we remove outlier issues that take more than one year to fix. After these filters are applied, we have 24459 issues for the regression models.

Prior studies [9, 4, 15, 21] have used various issue report attributes for predicting time until fix. Developer reputation, issue severity, number of assignees, attachments and dependencies are used by Bhattacharya et al [4]. The number of comments is used by Hooimeijer et al [15]. Issue priority, issue severity are used by Giger et al [9]. To replicate previous work on predicting time until fix, we fit the following multivariate linear regression model:

$$\begin{aligned} \ln(days+1) \sim \text{severity} + \ln(attachments+1) \\ + reputation + \ln(assignee+1) \\ + \ln(depends+1) + priority + late \\ + \ln(comments+1) + resolver + last\_commenter \end{aligned} \quad (3)$$

The response variable is the natural logarithm of time until fix measured in days. The predictors include:

severity: the issue severity recorded in Bugzilla. The issue severity shows the impact of an issue with seven levels: blocker, critical, major, normal, minor, trivial and enhancement. More impactful issues are more likely to be resolved sooner, so we expect that as severity decreases the issue resolution time progressively increases as well, with larger estimated coefficients.

attachments: the number of attachments associated with the issue. More attachments may indicate a better documentation for the problem resulting in a shorter time until fix.

depends: the number of other issues that the issue depends on. If an issue depends on other issues, the processing of the issue may be blocked until other issues are resolved. Such blocking should increase the time until fix.

assignee: the number of assignee changes associated with the issue. An increase in the number of assignee changes should increase the issue-fix time, as observed in, e.g., [3].

priority: the issue priority recorded in Bugzilla. This metric indicates the priority of the issue, including six levels: – (unassigned), P1, P2, P3, P4 and P5. Higher priority should be associated with shorter time until fix.

reputation: reputation of the reporter, which is defined as:

$$reputation = \frac{\#\text{issues fixed reported by the person}}{\#\text{issues reported by the person} + 1},$$

Issues reported by reputable people tend to take less time to fix.

late: whether the issue is reported in the later period:

$$late = \begin{cases} 1 & if \quad \text{create date} > \text{2010-09-27} \\ 0 & otherwise \end{cases} \quad (4)$$

This variable is used to describe how issue-fix time may vary over time.

comments: the number of comments associated with the issue. The number of comments reflects the attention that the issue receives, which should affect the time until fix. More comments may indicate complicated discussion or difficulty replicating the issue, resulting in longer time until fix.

resolver: whether the reporter closed the issue herself. The fact that the reporter resolved the issue herself may indicate that she created the issue once she came up with a fix. We expect such issues to take less time to fix as part of the work may have occurred before the issue was reported.

last_commenter: the author of the last comment before the issue is marked as FIXED. This variable is used to adjust for the variations in practices (random effects) among participants resolving the issue.

The values of predictors *attachments, depends, assignee, comments* were highly skewed in our dataset, we therefore take the natural logarithm of these predictors.

Table 6: Regression With Uncorrected Data

	Estimate	$p - value$
(Intercept)	4.91	0.00
critical	0.39	0.00
major	0.64	0.00
normal	0.80	0.00
minor	1.02	0.00
trivial	0.75	0.00
enhancement	1.23	0.00
$\ln(attachments + 1)$	-0.16	0.00
$\ln(depends + 1)$	0.62	0.00
$\ln(assignee + 1)$	0.32	0.00
reputation	-1.04	0.00
P1	-0.22	0.00
P2	0.08	0.11
P3	0.32	0.00
P4	0.52	0.00
P5	1.33	0.00
$\ln(comments + 1)$	0.54	0.00
resolver	-0.22	0.00
late	-0.72	0.00

Regression results are shown in Tables 6 and 7. The second column shows the estimated coefficients for the predictors and the third column shows the p-values. The values of coefficients are the estimated effects that different predictors have on the issue fix time measured by days. For example, as shown in Table 7, the regression model predicts that a decrease in the reputation by 50% from the median reputation of .58 will result in an increase of time until fix by 16%: $(exp(-0.52 * .58 * .5 + 0.52 * .58) = 1.16)$.

The positive coefficients for *depends*, *assignee*, *comments*, the negative coefficients for *reputation* and *resolver*, and the coefficients for different priority and severity levels match our expectation of the model as discussed above.

Note that the negative coefficient for *attachments* suggests that an increase in the number of attachments will reduce the time until fix, supporting our hypothesis that many attachments may make the issue easier to reproduce and, therefore, to fix. Also, the severity coefficients are more reasonably ordered for the corrected data than for the uncorrected data, with the order *blocker*, *critical*, *major*, *normal*, *minor*, *trivial*, and *enhancement* taking increasingly longer times to resolve. For uncorrected data we see trivial severity issues taking less time to resolve than minor or normal severity issues: a questionable result.

Table 7: Regression With Corrected Data

	Estimate	$p - value$
(Intercept)	-2.23	0.02
critical	0.28	0.01
major	0.43	0.00
normal	0.60	0.00
minor	0.75	0.00
trivial	0.75	0.00
enhancement	1.12	0.00
$ln(attachments + 1)$	-0.12	0.00
$ln(depends + 1)$	0.41	0.00
$ln(assignee + 1)$	0.45	0.00
reputation	-0.52	0.00
P1	-0.09	0.05
P2	0.20	0.00
P3	0.43	0.00
P4	0.49	0.00
P5	0.85	0.00
$ln(comments + 1)$	1.08	0.00
resolver	-0.21	0.00
late	-0.20	0.00

Table 9: Comparison of Statistical Significance

Predictor	Original	Uncorrected	Corrected
Assignee	Significant	+/Significant	+/Significant
Severity	Significant	+/Significant	+/Significant
Depends	Significant	+/Significant	+/Significant
Attachments	Significant	−/Significant	+/Significant

We perform the diagnostics on the regression with corrected data, and the quantile-quantile plot shown in Figure 2 suggests approximate normality of the residuals. The residual plot (not provided for the lack of space) does not indicate non-homogeneous variance.

The correction of data makes a substantial difference when modeling the time until fix. After applying the correction, the coefficients for four variables switch between being significant and not being significant (for p-value= 0.01), implying that the correction of data changes the model substantially. The adjusted R-square increases from 0.381 to 0.452, suggesting the corrected model fits about 20% better than the uncorrected one.

For comparison, Table 8 shows the published models of time until fix that are replicated in this study. Only *reputation* predictor becomes statistically significant in our model: the findings for the remaining predictors are replicated. However, it's difficult to replicate the studies that did not report the effects of the predictors. The effects of *attachements* and *comments* reported by [15] are in contrast to findings in our model, perhaps because of different practices of projects studied there.

To replicate the results on the same dataset, we use the model presented in [4]. Predictors used in this model include the number of assignees, issue severity, issue dependencies, and the number of attachments. We fit the model with uncorrected and corrected data with results shown in Table 9. All four predictors are significant replicating the original results. However, the sign of the coefficient for *attachments* (which were not reported in the original study) flip from negative to positive after applying our data correction method,

Table 8: Prior Work on Time Until Fix

Paper	Studied Projects	Predictor	Effect on time until fix	Significance
[4]	Chrome, Eclipse, Firefox Seamonkey, Thunderbird	assignee	NA	Yes
		severity	NA	Yes
		attachments	NA	Yes
		dependencies	NA	Yes
		reputation	NA	Low correlation
[15]	Firefox	attachments	+	Yes
		comments	−	Yes
		severity	Higher sev indicates shorter time	Yes
		reputation	NA	No
[21]	Eclipse	priority	NA	NA
		severity	NA	NA
		dependencies	NA	NA
		comments	NA	NA
[9]	Eclipse, Mozilla, Gnome	priority	NA	Differ in projects
		severity	NA	Differ in projects
		comments	NA	Differ in projects

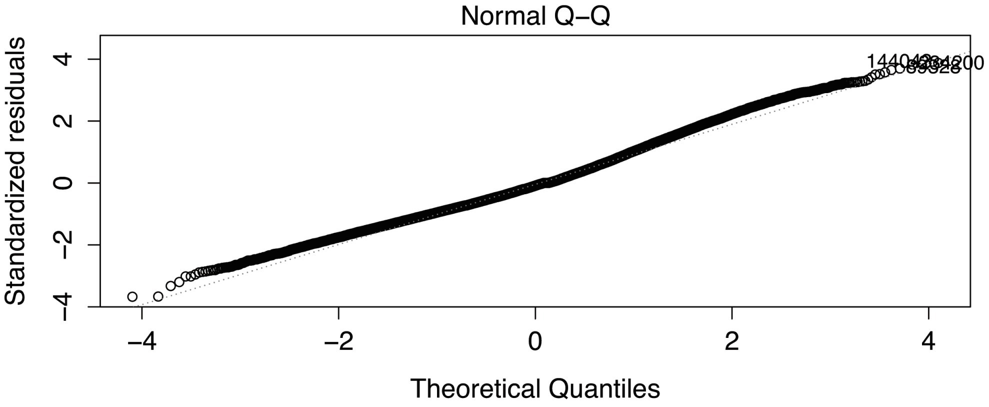

Figure 2: Quantile-quantile plot of regression residuals

indicating an opposite effect of this predictor. Furthermore, the R-square increases from 0.02 to 0.11 after applying the correction method, suggesting a much better fit of the corrected data. Study by Hooimeijer et al. [15], however, shows that the presence of *attachments* is correlated with a longer time until fix.

5. LIMITATIONS

Our approach has a number of limitations. We discuss issues related to the ability to apply our method, limitation related to the identification of incorrect values, and other potential issues.

General Method.

In this study we use physical constraints, e.g., individual productivity bound, that exist in software development to identify problematic values and to correct such values by using redundant observations that are more likely to be correct. However, not all data may have such physical bounds, or the analysts may not be aware of the bounds even if they exist. As noted in the section describing the method, such bounds may be estimable from the observations directly. Furthermore, not all events may have redundant counterparts in operational data.

Method For Identifying Problematic Data.

We discussed the method of identifying problematic issue fix dates in Section 4.3. However, the cut-off bound is set based on the likelihood of observing problematic data. Although this is good enough for identifying major outliers, it may fail to detect minor outliers in our dataset. Moreover, a universal bound for all users in all periods of time may not be optimal. Using a mixture model described in Section 3 may help identify the bounds more precisely and, potentially, suggest other distributions as being more appropriate than the truncated negative binomial distribution we used. Besides, we only consider a single event type and a single task: issue fix. Other tasks also require effort and may be done by the same individual. Using that additional information may help further tighten the capacity constraints resulting in an even more accurate identification of problematic data.

Insights for Practice.

Despite the dramatic differences between the models trained with corrected and uncorrected data, practical predictability of the two models have not been tested. Further empirical investigation is needed to fully assess the differences between models trained with corrected/uncorrected data.

Table 10: Exceptionally "Productive" Individuals (Based on Code Commit Events)

Date	UserID	count
2013-03-21	Bobby Holley	1160
2013-08-22	Ms2ger	1029
2013-02-25	Gregory Szorc	1024
2014-01-27	B2G Bumper Bot	998
2012-08-04	Ms2ger	991
2013-07-24	Ms2ger	986
2013-01-08	ffxbld	981
2011-07-21	ffxbld	964
2013-08-06	ffxbld	945
2013-02-20	ffxbld	907

Table 11: Exceptionally "Productive" Individuals (Based on Issue Report Events)

Date	UserID	count
2012-10-1	452624	542
1999-11-22	4415	277
2011-6-24	12809	116
2009-12-16	24572	110
2012-1-27	148348	93
2012-10-12	384312	90
2011-12-14	24572	87
2010-10-13	164048	87
2012-6-1	24572	86
2000-7-8	41	86

Finally, our method only corrects task completion time after identifying problematic task completion time and individuals. A careful selection of redundant data is needed for correcting problematic task completion individuals.

6. GENERALIZATIONS

In this section we investigate RQ7, in particular, we discuss how the method may apply to event streams other than ITS event streams and how the redundant observations may be chosen from an event stream associated with another system.

If an individual performs a task that takes nonzero effort we can use an event corresponding to the time of task completion recorded in operational data as a proxy. An arbitrary event (or a function calculating a metric from a single or multiple events) recorded in operational data could be chosen, with the best proxy depending on the type of task, the practices of using tools, and the nature of the operational data. For example, the issue reporting is also a task that requires nontrivial effort. The task completion time in issue reporting is the date when the issue is created (if we ignore subsequent interactions with developers or triagers). We can look at issue reporting event stream and determine problematic values as well. The number of issues reported per day per person, like the number of issues fixed, should not exceed a certain level. An extremely high number of issue reports may indicate that the reported issues were imported from other ITS and, therefore, the recorded reporters and recorded timestamps may be unreliable. In Mozilla community, based on this constraint, we count the number of issues reported by each user on each day, and the ten most productive person-days are shown in Table 11.

While we argued that the last commit date better represents issue fix completion, it is also not free from problems. In particular, we count the number of code commits by each person on each day and present the results in Table 10. It shows that code commit data from VCS also violates the capacity constraints. The truncated negative binomial distribution with parameters $\mu = 10$ and $\sigma = 2$ matches the observed quantiles of the commit date reasonably well. In Figure 1 we used the daily commit count of 160 as a cutoff representing tail probability of less than $1e-4$ for *Batchcommits*.

The identification of problematic issue report data and code commit data show that the approach may indeed be generalizable to other types of events and tasks. Note, however, that the alternative ways to identify problematic data should also be employed. For example, many of top commit/days are associated with login $ffxbld$ (FireFox Build). Clearly, this administrative login does not make all these commits manually. These are likely to be mostly script-generated commits that should be identified prior to applying our data correction approach. In summary, based on the capacity constraint, we identify likely incorrect data for four types of events: the issue report, last comment, issue fix, and code commit.

7. CONCLUSIONS

Operational data may not always accurately reflect the phenomena of interest, thus misleading research, wasting developers' effort and time, and causing quality problems in software development. In this study we proposed an approach to identify and correct problematic event data based on the individual capacity constraints and redundancies present in operational data, and used Mozilla ITS and VCS to illustrate how the approach could be applied in practice. In particular, we found that batch-fixing is a common mechanism for erroneous data, and we used alternative observations of the last comment posted for an issue to correct the issue resolution dates. By comparing corrected and uncorrected values to the last commit date in the VCS, we found corrected values to be 50% more accurate. We replicated the model commonly found in the literature to predict time until fix and found both its structure and the fit to change substantially after data correction. We identified likely incorrect observations for other types of events, e.g., issue reports and code commits to illustrate how the approach could be generalized. It is important to note that capacity constraints are not the only constraints that exist in software development. It may be possible to exploit other constraints to identify and correct problematic data and, through that, to improve effectiveness of software development.

Detailed references, all data sets and more information may be found at: https://passion-lab.org/projects/dataquality

8. ACKNOWLEDGMENT

Thanks the National Natural Science Foundation of China Grants 61421091, 61432001 and 91118004, and the National Basic Research Program of China Grant 2015CB352200.

9. REFERENCES

[1] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, ICSE 2006, pages 361–370, New York, NY, USA, 2006. ACM.

[2] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein. The missing links: Bugs and bug-fix

commits. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2010, pages 97–106, New York, NY, USA, 2010. ACM.

[3] P. Bhattacharya and I. Neamtiu. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM 2010, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[4] P. Bhattacharya and I. Neamtiu. Bug-fix time prediction models: Can we do better? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR 2011, pages 207–210, New York, NY, USA, 2011. ACM.

[5] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: Bias in bug-fix datasets. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE 2009, pages 121–130, New York, NY, USA, 2009. ACM.

[6] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 1277–1286, 2012.

[7] K. Ehrlich and M. Cataldo. All-for-one and one-for-all?: a multi-level analysis of communication patterns and individual performance in geographically distributed software development. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, CSCW 2012, pages 945–954, New York, NY, USA, 2012. ACM.

[8] W. Fong Boh, S. A. Slaughter, and J. A. Espinosa. Learning from experience in software development: A multilevel analysis. *Manage. Sci.*, 53(8):1315–1331, Aug. 2007.

[9] E. Giger, M. Pinzger, and H. Gall. Predicting the fix time of bugs. In *Proceedings of the 2Nd International Workshop on Recommendation Systems for Software Engineering*, RSSE 2010, pages 52–56, New York, NY, USA, 2010. ACM.

[10] T. Graves and A. Mockus. Identifying productivity drivers by modeling work units using partial data. *Technometrics*, 43(2):168–179, May 2001.

[11] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE 2010, pages 495–504, New York, NY, USA, 2010. ACM.

[12] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE 2013, pages 392–401, Piscataway, NJ, USA, 2013. IEEE Press.

[13] K. Herzig and A. Zeller. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR 2013, pages 121–130, Piscataway, NJ, USA, 2013. IEEE Press.

[14] V. Hodge and J. Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126, 2004.

[15] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE 2007, pages 34–43, New York, NY, USA, 2007. ACM.

[16] V. E. Johnson. Revised standards for statistical evidence. *Proceedings of the National Academy of Sciences*, 110(48):19313–19317, 2013.

[17] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. German, and D. Damian. The promises and perils of mining github. In *MSR 2014*, pages 92–101, May 31-June 1, 2014.

[18] S. Kim and E. J. Whitehead, Jr. How long did it take to fix bugs? In *Proceedings of the 2006 international workshop on Mining software repositories*, MSR 2006, pages 173–174, New York, NY, USA, 2006. ACM.

[19] S. Kim, H. Zhang, R. Wu, and L. Gong. Dealing with noise in defect prediction. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE 2011, pages 481–490, New York, NY, USA, 2011. ACM.

[20] A. Mockus. Engineering big data solutions. In *FOSE, ICSE 2014*, Hyderabad, India, June 1–6 2014.

[21] L. D. Panjer. Predicting eclipse bug lifetimes. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 29–, Washington, DC, USA, 2007. IEEE Computer Society.

[22] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE 2007, pages 499–510, Washington, DC, USA, 2007. IEEE Computer Society.

[23] K. J. Stewart and S. Gosain. The impact of ideology on effectiveness in open source software development teams. *MIS Q.*, 30(2):291–314, June 2006.

[24] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. ichi Matsumoto. The impact of mislabelling on the performance and interpretation of defect prediction models. In *Proc. of the 37th Int'l Conf. on Software Engineering (ICSE)*, page To appear, 2015.

[25] B. A. Thanh H. D. Nguyen and A. E. Hassan. A case study of bias in bug-fix datasets. In *Working Conference on Reverse Engineering, IEEE*, 2010.

[26] J. Xie, Q. Zheng, M. Zhou, and A. Mockus. Product assignment recommender. In *ICSE'2014 Research Demonstration*, pages 556–559, Hyderabad, India, June 1–6 2014.

[27] J. Xie, M. Zhou, and A. Mockus. Impact of triage: a study of mozilla and gnome. In *ESEM 2013*, pages 247–250, Baltimore, Maryland, USA, Oct 10–11 2013.

[28] J. Xuan, H. Jiang, Z. Ren, and W. Zou. Developer prioritization in bug repositories. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE 2012, pages 25–35, Piscataway, NJ, USA, 2012. IEEE Press.

[29] K. Yamanishi, J.-I. Takeuchi, G. Williams, and P. Milne. On-line unsupervised outlier detection using finite mixtures with discounting learning algorithms. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 320–324. ACM, 2000.

[30] M. Zhou and A. Mockus. Who will stay in the floss community? modeling participant's initial behavior. *IEEE Transactions on Software Engineering*, 41(1):82–99, Jan 2015.