Empirical Software Engineering

Deriving a Usage-Independent Software Quality Metric

--Manuscript Draft--

Manuscript Number:	EMSE-D-19-00023R1
Full Title:	Deriving a Usage-Independent Software Quality Metric
Article Type:	SI: PROMISE 2018
Keywords:	Software Quality; Software Usage; Software Faults; Bayesian Networks; NPM packages

Corresponding Author:	Tapajit Dey University of Tennessee, Knoxville Knoxville, TN UNITED STATES
Corresponding Author Secondary Information:	
Corresponding Author's Institution:	University of Tennessee, Knoxville
Corresponding Author's Secondary Institution:	
First Author:	Tapajit Dey
First Author Secondary Information:	
Order of Authors:	Tapajit Dey
	Audris Mockus
Order of Authors Secondary Information:	

Funding Information:	National Science Foundation (1633437)	Dr. Audris Mockus

Abstract:	Context: The extent of post-release use of software affects the number of faults, thus biasing quality metrics and adversely affecting associated decisions. The proprietary nature of usage data limited deeper exploration of this subject in the past. Objective: To determine how software faults and software use are related and how, based on that, an accurate, usage-independent quality measure can be designed. Method: Via Google Analytics we measure new users, usage intensity, usage frequency, exceptions, and release date and duration for complex proprietary mobile applications for Android and iOS. We utilize Linear Regression, Bayesian Network, and Random Forest models to explain the interrelationships and to derive the usage independent release quality measure. To increase external validity, we also investigate usage (downloads) and quality (number of issues) for 4430 popular NPM packages. Results: We found the number of new users to be the primary factor determining the number of exceptions, and found no direct link between the intensity and frequency of software usage and software faults. Crashes increased with the power of 1.02-1.04 of new user for the Android app and power of 1.6 for the iOS app. Release quality expressed as crashes per user was found to be independent of other usage-related predictors, thus serving as a usage independent measure of software quality. Usage also affected quality in NPM, where downloads were strongly associated with numbers of issues. Unlike in mobile case where exceptions per user decrease over time, for 45.8\% of the NPM packages the number of issues per download increase. Conclusions: We expect our result and our proposed quality measure will help gauge release quality of a software more accurately and inspire further research in this area.

Dear Sir/Madam,

We have carefully read the report of the editors and the reviewers regarding our article titled “Deriving a Usage-Independent Software Quality Metric”, and have extensively revised the paper to address the points they have raised.
We would like to thank the editors and the reviewers for their comments. We think the manuscript has improved greatly as a result of the reviewers’ constructive criticisms. We hope our latest response is satisfactory to you.
With this letter, we provide a document containing sentences form each report (*in italic*) that *pose specific questions or criticize our work*. Below these, we have added our own comments, and describe how each have been addressed in the paper. The annotations (e.g. S-3.3, pg 21) indicates that, e.g. section 3.3 on page 21 has been modified or extended.
Since the paper has been reorganized and extensive changes were made, we start by providing a summary of the updates to the paper, and then address the specific questions by the reviewers.

Summary of update:

- We have added three specific research questions that our paper it trying to answer, in accordance with the suggestion of Reviewer 1.
- The Sections of the paper have been reorganized as per the suggestion of Reviewer 1. A new “**Motivation**” section (S2) has been added. All sections related to data description and preprocessing have been consolidated into one section titled “**Data Description**” (S3). A “**Methodological Overview**” section (S4) has been added that discuss the methodologies used in the paper. Separate subsections have been added in the “**Results**” section (S5) that address each research question. The section “Implication of our Findings” has been repurposed into “**Discussion**” section (S6). The “Comparison with published results” section has been made into a subsection under the “**Related Works**” section (S7). The “**Limitations**” (S8) and “**Conclusion**” (S9) sections have also been modified according to the reviews.
- An Appendix has been provided with the article that discusses the relationship between exceptions and a code complexity measure (Lines of Code - LOC). We added this as an Appendix, since we do not believe this analysis is essential for addressing the research questions we posed.
- We have added timeline plots showing the difference in trends for the mobile applications when the number of exceptions is directly used as a predictor vs. when it is normalized by the number of users. Similar plots have been added for 4 popular NPM packages.

Editor Comments

Thank you for your submission to EMSE. Based on the reviewers' feedback, we recommend "Major Revision" for your manuscript.

All the reviewers agree that the paper is generally easy-to-read, it deals with a relevant topic for the readers of this journal, and the difference from the previously published PROMISE paper is satisfying.
However, they also highlight that the paper needs to be improved regarding the following issues:
a) threats to the internal, construct and conclusion validity of the findings (all reviewers), b) missing control for potential confounding factors, and unclear implications of the findings (Reviewer 1), c) small/limited sample size (Reviewers 2 and 3), d) discussion on other internal quality variables (Reviewer 2), e) validity of applied techniques and tests --tests for normality, default use of R-package (Reviewer 3).

We appreciate the editors' and the reviewers' positive and constructive comments. As shown below in this letter and in the revised manuscript, we have we have substantially revised it to address all the points raised by the reviewers.

> **0)** We would like to address a common question raised by the reviewers: *the absence of any code complexity related measures in the paper*.

We believe this question was raised in response to us not being entirely clear about the research goals that are addressed in the article. We have added the research questions (**S-1**) we are addressing in this article and listed the motivations (**S-2**) for undertaking this problem. We believe these should be able to clarify why having code complexity related measures is not essential for the purpose of this work. To reiterate, the overarching goal of this paper is not to produce the best model of predicting or explaining exceptions nor to design the "best possible" (but impractical) quality metric. Our main objective here is to illustrate the interdependence of usage and observed number of crashes, and advocate the necessity for normalizing the observed number of software faults (like crashes) by the extent of usage to meaningfully compare quality of different software releases. Furthermore, the objective of the proposed measure is to highlight quality after adjusting for the effects of usage on exceptions. In other words, as intended, the proposed measure should reflect the impact of code, process, and coordination complexity on failures (see, e.g, Cataldo et al, 2009[7]). These aspects ensure that a development team can utilize the quality measure to take appropriate action for quality improvement instead of being mislead by the effects of usage. As our findings show, the usage explains a large amount of variation in the number of observed exceptions which, conceptually, is easy to understand as the obvious mechanisms of how usage may translate into exceptions exist. While, similarly, it is clear why more complex code may contain defects, the relationship between latent defects and failures is much more complex, since not all defects may result in failures.
The software under consideration being a closed-source commercial software, it was not easy to obtain information on the source code and even more difficult to associate what exact code was used in each release of the software. However, we were able to obtain a significant portion of the commit log and used that information to investigate if the adding of the number of lines of code as a measure for code complexity affects the number of exceptions. The result of the analysis is presented in **Appendix I.**

The concern about possible confounding by the code complexity variables are addressed in **S-2**.

Reviewer 1 Comments

1. *Moreover, when studying NPM packages, the authors find that the number of issues per download increases over time.*

   The exact statement in the **abstract and S-5.3.2, (pg 1, 28)**, “the value of the quality variable (number of issues per download) increases with time for almost half of the packages (2030 out of 4430 packages, 45.8%)” In other words, for less than half of the packages *the number of issues per download increases over time.*

2. *In the first place, a very few details on how the model was built are reported: what type of logistic regression model is used? Were the assumptions made on the underlying data by the model verified? What is the selected family type (e.g., "binomial" or "quasi-binomial")? Why? How was the model implemented? All these questions would deserve an answer to enable/ease replicability.*

   We used Ordinary Least Squares (OLS) regression method for the linear regression, the information has been added to **S-4, S-5 (pg 13, 17)**. The model was implemented using the “lm” function in R. Furthermore, we have checked the validity of the underlying assumptions for using an OLS estimator and performed usual diagnostic checks of the fitted model, which is discussed in **S-5.1.1 (pg 18)**.

3. *The second major problem with the statistical modeling is the lack of control variables. Besides the independent variables gathered from Section 4, there might be additional factors explaining the number of exceptions: as an example, the complexity of source code may play a role. Unfortunately, the authors do not take into account those control variables: this produces a serious threat to conclusion validity. Thus, I would strongly recommend to (i) reason on the possible factors impacting the number of exceptions and (ii) control for them in the statistical model.*

   We tried to addressed this with our answer to **point 0)**. As noted in the response to 0), our goal is to produce a measure that is directly affected by code and process complexity, so that the development team can take appropriate remediation action, instead of being mislead by the effects of usage on the number of exceptions. We do provide the analysis including code complexity covariates presented in the Appendix. The key question for which a software project needs to know an answer is whether or not a specific release has better quality than the prior release. We believe that we demonstrate that it is important to adjust exceptions for usage in order to answer that question. The specific action (e.g., refactoring complex code, doing better code inspections, improving testing) pending on the circumstances can then be taken by the

development team.

4. *When employing Random Forest, the authors run a 10-fold cross validation. By definition, this strategy randomly partitions the set of data to create training and test sets: such a randomness may bias the results, as it can create specific combinations of training/test sets leading to under- or over-estimate the importance of the exploited variables. To account for this aspect, a robustness analysis should be performed: the authors can run the validation multiple times (e.g., through a 10 times 10-fold cross validation) to assess how stable the results are over different runs. This would reinforce the validity of the conclusions given.*

   We thank the reviewer for the suggestion. Keeping this the suggestions by other reviewers in mind, we decided to employ a 10 times 2-fold cross validation because the sample of releases was quite small (11 in one case) and doing 10-fold cross-validation would, in such a case, predict a single observation. **S-4, S-5** updated.

5. *The structure of the paper should be substantially improved. In the first place, the paper contains several different analyses: a methodological overview that summarizes them would be highly beneficial for the reader. In the second place, the paper does not contain any research question, and this makes the reading experience really hard. Please, explicitly state what are the goals of the paper and the specific research questions that the paper wants to address. Third, the use of sections looks strange to me: in particular, I would recommend a more cohesive way of grouping related argumentations. For instance, Sections 2, 3, and 4 are all related to the dataset and would be better to have them in one section. Similarly, in the other sections (e.g., Sections 5 and 6), it is unclear what is part of the methodology and what is part of the results. I would strongly suggest to have a section for each research question with two subsections for methodology and results.*

   We thank the reviewer for the suggestion. The paper has been reorganized accordingly as described in the beginning of the response.

6. *In Section 9 the authors examine the results achieved on the NPM packages. While they may be interesting, I had hard times in understanding the real link between this study and the previous one. I think the paper should be more explicit in making clear what are the relations among the different sections of the paper and on the overall goal of the study. I believe that the paper can be strongly improved in this sense. Perhaps, the methodological overview and the addition of the research questions could help in reaching this point.*

   We have added a research question addressing this point, (**S-1, pg 4**) and added further clarifications (**S-5.3**) addressing this point.

7. *The implications of the study are unclear. Section 10 is called "Implication of our Findings" but, unfortunately, no implications are discussed. This section indeed further discusses the results achieved, but does not provide any insight on how to make them actionable and what are the next steps to do by the research community. Thus, I believe that the section would require major changes to make it meaningful.*

   The title of the section was corrected. We have also added the two major implications at the end of that section (**S-6, pg 31**), and further discussed the point in the Conclusion (**S-9, pg 35,36**).

Reviewer 2 Comments

1. *The study was only on 2 mobile apps (which actually had crash reports - the NPM dataset relied on issue reports). The number of releases were also very small. I'd suggest the authors to extend their dataset with more apps.*

   Obtaining a crash report dataset with actual number of users, like the one we have proved extremely difficult as we investigated numerous options. A normal crash report dataset, like the one for Ubuntu linux distribution, does not contain any information on the actual number of users, making a study like this impossible. As we mentioned in limitations (**S-8, pg 34**), we had no control over the release cycle, and we were unable to get any more data for these two mobile applications as well. As a result, we were unable to implement this suggestion.

2. *Also, it would be worth exploring if issue reports contain crash reports.*

   This is a good suggestion, and we'd like to investigate it further in later studies. However, the goal for including the study on NPM was to examine if our idea is extensible to other scenarios besides crash reports. Keeping that goal in mind, we chose to use the number of issues for NPM (since we are not aware of a crash report database for NPM). Our results indicate that even a measure that is not strictly a measure of software failures like crashes, is dependent on a usage parameter like the number of downloads for 99.2% of the packages examined, demonstrating the some extent of the generalizability of our idea of the need to adjust numbers of crashes by the number of users.

3. *The study also looked at only a few variables. How about internal variables (e.g. code complexity, lines of code, etc.)? These were known to correlate with software quality too. If they were included, would they have an impact on the correlation between the number of users and the quality outcome? Those internal variables are quite standard in the literature so I think such a study would be much needed.*

   The prediction models appear to have a poor predictive performance (as mentioned in the paper, but the specific results in terms of precision, recall and F-measure). This may

be a major concern, suggesting that the number of users may not be a good predictor.

We tried to address these points in our answer to point 0) and further in Section 2.

4. *Minor comments:*
   Fig 3: what does p=0 mean?
   Page 18 - the paragraph at the end of section 6.2: what is the conclusion here?
   Page 20 - "To obtain empirical ...": this sentence is quite confusing
   Page 30 - subsection ???

   All of these points have been addressed in the paper. (**pg 18, 21, 22, 34**)

Reviewer 3 Comments

1. *The derived measure "Quality" is a function of two existing measures. It is the number of exceptions standardized by the number of new users. For this measure, the authors did not collect the data in a different manner, nor did they use a formula that semantically reflects different perspectives (e.g. process or product side)*
   Response
   As stated in the paper, our objective was to obtain an actionable and easy-to-use measure that can more accurately compare quality of the release than simply the count of failures. We are not aware of how to measure software quality more directly. For example, issue reports are also influenced by usage as we show for the NPM packages. Some of the previous approaches attempted to model defects through code complexity, process complexity, and social complexity e.g, [7], so these and similar methods could be employed by the project to set up more specific action plans. Our point is that the more involved models for the most part do not adjust for the extent of usage with rare exceptions (e.g., observations by Caper Jones [26] and the work on Interval Quality [43], and Customer Quality [20]).

2. *The differences between the behavior of the new derived measure and other measures have not been well-investigated. The authors ignored potential product measures (e.g. Measures of code complexity) and process measures (e.g. mean time between failures).*

   As noted in the response to point 0), the goals of this study is not to investigate what product or process measures affect exceptions (as the topic has vast literature at least in regards to defects), but to correct the exception counts for the extent of usage. As stated in the paper, our objective was to obtain an actionable and easy-to-use measure that can more accurately compare quality of the releases.
   We hope that the added research questions and Motivation Section clarify that it would not be necessary for us to achieve our goals.

3. *It is difficult to know whether this derived measure is better than the other measures in literature, as there is no sign for a baseline to compare with in this work.*

   We have added the timeline plots comparing the trends (last point in the update summary) that addresses the concerns about how this measure is different. We also added an explanation of the other similar measures used in the literature.

4. *Random forest is a wrapper-based FS technique, and is not the best ranker. The authors might consider using a hybrid-based ranker instead (e.g. AutoSpearman)*

   Thank you for the suggestion. The AutoSpearman paper mentions that it handles the consistency issue in ranking, and indeed there is some variation in the relative ranking of the variables for different iterations of Random Forest, but the number of new users has consistently been ranked as the most important variable in different iterations of the Random Forest runs, and the usage intensity and frequency has consistently been ranked low. To establish robustness we are using the result of different models to reach our conclusion, and similar results were obtained by different models, so we do not feel it is essential to use the best available ranker for this task.

5. *The authors heavily relied on a minimal set of measures without explaining eg the reason behind including these measures and excluding others.*

   We have added an explanation for that in **S-2**, The reason for dropping some of the observed measures in discussed in detail in **S-3.1.3.**

6. *Section 7 "The comparison with published results" is inadequate. The authors i) ignored several aspects in the comparative studies and ii) did not reproduce or replicate experiments to highlight the differences and similarities.*

   We have added an explanation about the reason for including that section (**S-7.1**). The reason for highlighting the results of these studies it to compare the results of those studies and our study (please also see the issue 3 above), which we found was showing similar trends, for the purpose of complementing the lack of an extensive dataset in our study. There are important differences, however. First, other studies have not used data for mobile applications, hence we needed to replicate results using different measures (i.e., measures available from Google Analytics and similar data collection suites). Second, while defect data is widely available for open source projects, failure and, especially, usage data are not absent. Any type of data for commercial projects is confidential and can not be used for replication.

7. *The authors used a linear regression model. However, tests for normality (e.g. the Kolmogorov-Smirnov (K-S) or ShapiroñWilk test) have not been considered.*

   We have added explanation about validating the assumptions for using OLS estimators

(**S-5.1.1, pg 18**).

8. *The validity of the regression analysis is questionable as the number of subjects per variable is small. For iOS dataset, the authors used a sample size of only 11 observations for 6 then 4 variables.*

   We agree with the comment. That is the reason why we use other models to validate our findings. Since a similar result was reported by all the three different approaches we took, that increases the confidence in our result.

9. *The construction process of Bayesian networks was a tool-driven experience. The authors applied and experimented most, if not all, of the algorithms included in the applied R-package, without a proper explanation on why they used those algorithms. The authors applied a continuous BN structure search and at the same time they applied a discrete version)*

   We have added the reason for considering those methods in **S-4.1, pg 15**. However, given that we performed an extensive simulation study to choose the best performing method and used it, we have no reason to believe a tool driven method would, in any way, be less accurate than a model constructed by hand. In fact, by using this approach, we ensured an BN model construction process free from personal biases of the researchers.

10. *In the simulation, the authors created 1000 different datasets out of a small amount of observations (e.g. 173 and 11.)*

    A simulated dataset is different from a bootstrapped dataset, and we used a custom BN model for creating the simulated dataset. The conditional parameters were calculated using our original data, but other than that, our data had no other role in the creation of the simulated dataset. We used the “rbn” function from the “bnlearn” package for creating the dataset. We invite the reviewer to check the manual page for that function: http://www.bnlearn.com/documentation/man/rbn.html

11. *The authors did not investigate or consider the false positives that may occur by constructing a belief network.*

    The belief network was created by averaging bootstrap results with a custom threshold. However, we performed no additional checks. This point has been added to the “Limitations” section.

12. *The 10-fold cross validation has been applied to a dataset that consists of only 4 variables and 11 observations. Also, I could not know whether the authors split the datasets before or after applying this technique.*

Keeping the suggestions from other reviewers in mind, we decided to perform a 10 times 2-fold cross validation instead (See answer to R1: Q4).
For the second point, we have given the link to the public GitHub repository that includes the code we used for the study.

13. *The selection criteria behind Choosing LR, BN and RF*

    Explanation added in **S4, pg 13.**

    a. We started by using the OLS estimator since it is one of the simplest modeling methods that gives good models in a lot of situations and the result is easy to interpret.
    b. However, we were unsure about the accuracy of the result due to the presence to moderate to high correlation between some of the predictor variables (e.g. the Release Date and Release Duration variables had a correlation of -0.88 for the iOS application data). Therefore, we decided to use Bayesian Network (BN) for modeling the interrelationship among these variables, since the accuracy of this model is unaffected by the presence of high correlation among the predictors. Variables with high correlation simply appear as connected nodes in the final model. Since the use of Bayesian Network models is not very common in this context and it is one of the main contributions of this paper, we discuss BN models in greater detail later in this section.
    c. The third and last modeling approach we used is Random Forest regression method. Random Forest is one of the best off-the-shelf models that work well with almost all types of data and it is easy to get the relative importance of the predictor variables from a fitted model. These two factors led us to use Random Forest regression as a third modeling technique to identify the most impactful predictors explaining the number of exceptions. Another reason for using the Random Forest model is that it is a predictive model, while the other two are explanatory models and they serve different purposes. We wanted to examine the effect of post-deployment measures for the purpose of prediction and observing which variables play an important role.

14. *Choosing these specific 6 measures.*

    Explanation added in **S-3 (pg 6), S-3.1.3**.

15. *Choosing the datasets.*

    Explanation added in **S-1, S-3, S-3.2**. Given the lack of publicly available datasets that has both the post-release software failure (like defects or crashes) numbers and software usage measures, having access to this dataset gave us an opportunity to perform a study like this and investigate the interrelationship between the number of

software failures and software usage.

16. *Inadequate justification on the aggregation process used to combine/sum the observations collected from the commercial datasets.*

    Explanation added in **S-3.1.3 (pg 8).**
    We had two main reasons for aggregating the data:
    a. The goals of our study are concerned with identifying the relationship between exceptions and other post-deployment variables for different releases, and defining a quality measure to compare the qualities of different releases. Therefore, having the measures aggregated at per-release granularity is essential.
    b. We would have been able to take a time-series based approach and still work out our goals if the releases were cleanly separated in time, i.e. if there were no overlap between releases. Unfortunately, we observed from the data that users continue to use one release long after the subsequent releases are available, and there is no clear pattern about how long a release is used. Therefore, we had to aggregate the data to a per-release level to be able to achieve the goals of this study.

17. *Why do we need to build a predictor (i.e. RF)?*

    Explanation added in **S-4 (pg 13), S-9 (pg 36).** Also, see answer to Q-13.

18. *The authors used 4430 popular NPM packages. How did they measure popularity?*

    By the number of downloads, it has been mentioned in the paper. (**S 3.2, pg 11**)

19. *The methods that were employed in constructing the belief networks are broad.*

    That is by design. We wanted to perform the simulation study on a broad range to techniques to identify the best performing heuristic method in this context (i.e., for the software failure data).

Title Page containing ALL Author Contact Info.

Deriving a Usage-Independent Software Quality Metric

Tapajit Dey
Graduate Student
University of Tennessee, Knoxville
Min H. Kao Building, Room 619
1520 Middle Drive
Knoxville, Tennessee, USA - 37996
Email: tdey2@vols.utk.edu
ORCID: 0000-0002-1379-8539

Audris Mockus
Ericsson-Harlan Mills Chair Professor
University of Tennessee, Knoxville
Min H. Kao Building, Room 613
1520 Middle Drive
Knoxville, Tennessee, USA - 37996
Email: audris@utk.edu
Office Phone: 865-974-2265

Conflict of interest disclosure:
Funding: This work was supported by the National Science Foundation under Grant No. 1633437.

Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Deriving a Usage-Independent Software Quality Metric

Tapajit Dey · Audris Mockus

Received: date / Accepted: date

Abstract Context: The extent of post-release use of software affects the number of faults, thus biasing quality metrics and adversely affecting associated decisions. The proprietary nature of usage data limited deeper exploration of this subject in the past. Objective: To determine how software faults and software use are related and how, based on that, an accurate, usage-independent quality measure can be designed. Method: Via Google Analytics we measure new users, usage intensity, usage frequency, exceptions, and release date and duration for complex proprietary mobile applications for Android and iOS. We utilize Linear Regression, Bayesian Network, and Random Forest models to explain the interrelationships and to derive the usage independent release quality measure. To increase external validity, we also investigate usage (downloads) and quality (number of issues) for 4430 popular NPM packages. Results: We found the number of new users to be the primary factor determining the number of exceptions, and found no direct link between the intensity and frequency of software usage and software faults. Crashes increased with the power of 1.02-1.04 of new user for the Android app and power of 1.6 for the iOS app. Release quality expressed as crashes per user was found to be independent of other usage-related predictors, thus serving as a usage independent measure of software quality. Usage also affected quality in NPM, where downloads were strongly associated with numbers of issues. Unlike in mobile case where exceptions per user decrease over time, for 45.8% of the NPM packages the number of issues per download increase. Conclusions: We expect our result and our proposed quality measure will help gauge release quality of a software more accurately and inspire further research in this area.

Keywords Software Quality · Software Usage · Software Faults · Bayesian Networks · NPM packages

Tapajit Dey · Audris Mockus
Department of Electrical Engineering and Computer Science
University of Tennessee, Knoxville
Knoxville, Tennessee, USA
E-mail: tdey2@vols.utk.edu, E-mail: audris@utk.edu

1 Introduction

Improving quality of software is one of the objectives of software engineering. "Software Quality" has been defined in various ways, however, in this paper we take a narrow focus on the manifestation of software defects as crashes observed from the perspective of the users. Observing a software crash, generally speaking, is a manifestation of low quality of the software to a user. Thus, it seems intuitive to measure the quality of software[1] by counting the number of crashes, with more crashes being associated with lower quality. If, for example, we compare two different softwares or two different releases of a software, we first calculate the number of crashes for each and then compare these numbers. However, software with more users tends to see more crashes [12,21,45] as each user may exercise it differently. In an extreme case, a software or a release with no users will have no crashes, regardless of its quality. This interdependence of software usage volume and crashes experienced is typically not considered in quality measurement in industry or in empirical studies (although few studies do note that [15,17]). Ignoring this relationship, however, would misguide quality improvement efforts (avoid quality improvements for releases/ software with low usage) and/or misguided developer performance metrics (reward developers of low-usage products). This analogy can also be extended for software defects (bugs), and by extension for issues raised against a software, since software crashes are manifestations of underlying defects and [27,45,21] observed that the number of discovered software defects increases with the number of users, although the relationship between crashes and defects is not very well understood [17].

One possible reason for this oversight is the scarcity of reliable usage data. While the number of defects and crashes reported by users are carefully tracked by most large scale projects (*e.g.* Mozilla Firefox, Ubuntu etc.), tracking the variables related to usage, *e.g.* the number of users, intensity of usage etc. is almost impossible without a reliable monitoring system. Such a system is rarely used by open-source software and even many traditional software-as-a-product systems do not or can not have such capability. Moreover, even when such a dataset is available, it is almost always proprietary, so obtaining and sharing it, even for the software development teams in these proprietary projects, is difficult since the deployment is typically managed by a different team within the organization. Without such data, however, it becomes exceedingly difficult to interpret the quality of a software from the customer reported crashes/defects alone due to the interdependence of usage and crashes/defects [12,21,45]. The overarching goal of this study is to advocate the necessity of taking the usage aspect into consideration while comparing the qualities of different releases/softwares and to illustrate one possible way of doing so.

We were able to obtain the usage data for some mobile applications developed by Avaya, *viz.* Avaya Communicator for Android (currently known as Avaya Equinox®) and Avaya one-X®Mobile SIP for iOS. The usage data was obtained from Google Analytics. For analyzing the usage data for these applications, we used three usage related variables: number of users, usage intensity (average duration of software use per user), and usage frequency (average number of times the app was used by a user), along with two variables describing attributes of the particular re-

[1] in fact, we mean to measure one aspect of the quality of software

lease: release date and effective duration of the release, measured by how long the release continued to have new users, and looked at how these variables affect the number of exceptions *i.e.* application crashes. Thus, the **first research question** we are addressing in this paper is about modeling the relationships between these different post-deployment variables, specifically, finding the relationships among variables describing different aspects of software usage and software crashes (which are manifestations of underlying defects). Once again, we are not trying to build the best model explaining the number of exceptions, but we focus on discovering the relationship between exceptions and other post-deployment measures.

Since the quality of a software/release measured by the number of defects/crashes is often misleading due to its dependence on usage, as we mentioned before, our **second research question** is about constructing a usage-independent measure of quality, which is an actionable and easy-to-use measure (instead of the "best possible", but impractical quality metric), and one that can more accurately compare quality of the release from a user's perspective than simply the count of failures. In other words, as intended, the proposed measure should reflect the impact of code, process, and coordination complexity on failures (see, e.g. [7]). These aspects ensure that a development team can utilize the quality measure to take appropriate action for quality improvement instead of being mislead by the effects of usage.

The methodology we employed for addressing these research questions is as follows: After the usual data cleaning and variable construction stages, we first applied a linear regression (LR) model to identify the significant predictors for the number of exceptions. However, due to the moderate to high correlation among the predictors, the result of the LR model couldn't be completely trusted. Then we used a Bayesian Network (BN) model to discover the interrelationship between the variables. The BN model was generated by using structure search algorithms, and the search method was chosen based on the result of a simulation study. We have presented the detailed result of the simulation study and hope that other practitioners willing to use BN structure search methods in their work might find it useful. Finally, we ran a random forest (RF) model to identify the importance of the variables for predicting the number of exceptions. All analyses in this study was done in R [58]. We found that the frequency and intensity of usage have little impact on the number of exceptions, but the number of users does have a significant impact. Thus, we establish the interdependence between the number of crashes and usage, specifically, the number of users, and finally propose a quality metric that is independent of usage, which would enable us to compare the qualities of different softwares and/or different releases of a software more accurately.

In our previous work [12], we only analyzed the General Availability releases for Avaya Communicator for Android. Since all of the data was from Google Analytics, we had the same set of variables for all the Avaya softwares. We found similar results from that study as well, with the number of new users being the most important factor affecting the number of exceptions. Furthermore, we proposed a quality metric of average number of exceptions experienced by end users (so lower means better) and found it to be independent of other usage metrics.

In this study, we have added the analysis of the development version of Avaya Communicator for Android and the General Availability releases of Avaya one-X®Mobile SIP iOS Client. Although we collected data for several other apps, those were dropped due to having too few releases/ exceptions/ users to give a reliable result. We employed some new data correction steps for correcting the observed

number of users and visits (Section 3.1.3). We used the same three modeling techniques and the same quality measure and found the result to be very similar for all cases considered. We also added a timeline showing how the perceived quality of the releases vary with time for different releases.

To increase the external validity of these findings, we also consider the relationship between downloads, a measure of usage, and number of issues, a measure similar to the number of crashes or bugs, for a rather different scenario of NPM packages. Node Package Manager (NPM) is the package manager for node.js, an open-source, cross-platform JavaScript run-time environment. Since we do not have the number of crashes for these packages, we looked at the number of issues reported for these packages instead. Our **third research question** is about exploring the external validity of the research, specifically focusing on finding out if our hypothesis that usage related measures, like the number of users, affect the software failure related measures, like the number of exceptions, which we found to be true for the Avaya mobile applications, holds true for a different software ecosystem like NPM for a different set of measures for usage and software failure, measured by the number of downloads and reported issues respectively. We also want to explore how the quality of these NPM packages, measured by the number of issues per download, vary with time.

We chose to look at 4430 NPM packages that had more than 10,000 monthly downloads and a GitHub page with issues enabled. From the analysis of NPM packages we found that for only 36 out of 4430 packages (0.8%) the number of daily downloads is not a significant predictor (p-value > 0.5) of the number of issues on that day.

The primary contributions include an observed strong relationship between the number of exceptions (crashes) and usage by analyzing two different mobile softwares (three different versions) and an actionable and easy-to-use quality metric that gives a more complete and reliable measure of quality. The wider implications of these findings suggest the potentially serious problems in existing quality metrics and predictions. Specifically, the organizational goals for a software project quality should take into account usage, and software defect predictors could be improved substantially if the population of users would be taken into account and could be predicted. Moreover, the analysis of the NPM packages established the extendibility of the the concept, thus opening the possibility of wider application of the approach. We have also presented the detailed result of the simulation study we conducted to choose the best performing BN structure search algorithms, which we believe will of of use to practitioners willing to use BN structure search methods in their work. To increase external validity, we investigated usage-quality relationship in a different context: the number of downloads and issues of 4430 NPM packages. We found the number of downloads to be a significant predictor of number of issues for most (99.2%) of them. Finally, we have presented a timeline showing how the perceived quality changes with time during the different releases for the mobile applications and during the life of the NPM packages, and discussed the trends we found.

The rest of the paper is organized as follows: We discuss the research questions and the motivation for the study in Section 2. In Section 3, we describe the data used in our study, providing details of the data source, data collection process, and detailed data preprocessing steps. In section 4, we provide a overview of the methodology we used in the study, which includes details of the simulation

study we performed for choosing the best performing BN structure search method that was used in subsequent analysis. In Section 5, we present the answers to the research questions of our study, which includes the models describing the inter-relationship between exceptions and other post-deployment variables, the quality measure we proposed and models showing its independence of other usage measures, and the results of the analysis of the NPM packages. We discuss various aspects of our result in Section 6. In Section 7, we discuss various works in related topics. Finally, we discuss the limitations of our study in Section 8 and conclude the paper in Section 9.

2 Motivation

We hypothesize that the observed number of software failures, measured by the number of defects, crashes, and/or issues reported against it, depend on a number of internal as well as external factors. Mathematically, $OSF = f(v_i, v_e)$, where OSF = observed software failures, $f(.)$ is some function, v_i is a set of internal factors, and v_e is a set of external factors. The internal factors represent the set of factors that are artifacts of the software development process, and includes product parameters like size of the software and code complexity, process parameters like change entropy (distribution of changes with a component), human factors like the number of authors involved, number of reviewers with expertise, and code review parameters like number of reviews, number of reviewers etc. The impact of of these parameters are well known and have been used in multiple past studies [7,38,37,59, 34]. However, the number of **observed** software failures also depend on external factors that are not part of the software development process. Hypothetically, such parameters include the number of users using the product, the intensity of usage (for how long they use it), and frequency of usage (how frequently they use it). The effect of these parameters is not very known or studied. However, when the measures related to software failure are collected, these are always the **observed** measures, not the **actual** measures. It is impossible to know the **actual** number of bugs in a given software for instance.

However, as we mentioned earlier, there are serious problems related to using the observed measures of failures for comparing the qualities of different softwares and releases, since the observed number of low defects/crashes could be a result of low usage, and not better quality. Therefore, to be able to compare the actual quality of different softwares/releases that is a function of only the internal factors, we need a measure that is free from the influence of external factors like usage, i.e. for such a quality measure Q, $Q = g(v_i)$, where $g(.)$ is some function.

It is worth mentioning that, hypothetically, the internal and external factors should be independent of each other, at least for the general availability releases of closed-source softwares, since most of the users of such releases are not involved in the software development process, thus the external usage factors should be unaffected by software design artifacts. Therefore, none of the internal or external factors should act as a confounding variable, since a confounder affects both the dependent and independent variables in a causal relationship. This means that not accounting for the external factors would not systematically misclassify the perceived quality of all releases/softwares under consideration, but would randomly misclassify some of them. Such random effects are much harder to detect, and could be one of the reasons why this topic hasn't gained as much attention, since such

misclassification errors could have wrongly been attributed to measurement errors or just random effects. However, accounting for the effect of usage would have enabled the developers to correctly compare the qualities of these releases/softwares and explain many of these apparent random misclassification errors.

The (hypothetical) independence of external and internal factors have implications on the design of this study as well. Since we are using closed-source commercial mobile applications for our study, we could not get detailed information about the actual source code. However, we could argue that if we can establish the interdependence between the usage (external) factors and the observed number of crashes, the validity of such a relationship is unlikely to be affected by the presence of unobserved internal factors.

We performed a small study, the result of which is reported in Appendix I, adding a variable representing the number of lines of code in our dataset, which is an internal factor, and found it to be independent of any of the external factors representing usage. The result of the study supports our hypothesis that the internal and external factors are indeed independent of each other.

Finally, the goal of this study is not to make the best possible model for software crashes (exceptions), nor to design the "best possible" (but impractical) quality metric, but to investigate the interrelationship between software usage and software faults (crashes, issues) and deriving an actionable and easy-to-use Quality measure that is independent of software usage. While we want to accurately represent the interrelationship between exceptions and other observed variables, we do not need the best model explaining exceptions. While having information on the internal artifacts would certainly have improved our model, not having them does not invalidate our conclusion and is enough for the purpose of this study.

3 Data Description

For this study, we looked into two different types of software, which are vastly different in nature. The primary focus of the study is on the commercial software developed by Avaya for mobile applications, which are from the telecommunication domain. We chose the Avaya mobile applications because we had access to the actual post-deployment measures for these. We hypothesized that the number of users is the most important measure of usage, but, as we mentioned earlier, obtaining the actual number of users of a software post-deployment is extremely difficult without a dedicated monitoring tool, and such data is often proprietary. Therefore, having access to the actual usage measures gave us a golden opportunity to study the relationship between software usage and software crashes. However, using this data also had some limitations, *e.g.* we could not choose the variables being measured, the duration of the measurements, or the applications for which the data is being collected. Being a commercial software, the source code is essentially closed-source, which makes it difficult for us to conduct a through investigation with all variables of interest in one model.

For external validation of the theory developed using this data, we looked into the NPM packages, which are open-source JavaScript packages used in web-development. We used NPM for our study because (1) it is one of the largest open-source communities, which makes it a good candidate to be investigated, and (2) it collects the number of downloads for the packages, which is a far better

measure than other usage measures like the number of stars or forks of GitHub projects.

In this section, we provide some details about the software being studied, discuss the data source, describe the data, and give details about the data preprocessing steps.

3.1 Data on Mobile Applications developed by Avaya

3.1.1 Software description

One of the software chosen for this study was Avaya Communicator for Android (currently known as Avaya Equinox®). It integrates the mobile devices of the users with their office Avaya Aura®communications environment and delivers mobile voice and video VoIP calling, cellular call integration, rich conferencing, instant messaging, presence, visual voicemail, corporate directory access and enterprise call logs[2].

Another software we studied was the Avaya one-X®Mobile SIP for iOS, which provides mobile communications for the iPhone, iPod touch, and iPad through a wireless-enabled SIP Avaya Aura®environment combining enterprise features with the convenience of a mobile endpoint for users on the go. The Avaya one-X Mobile®SIP for iOS appears as an end point in the Aura®environment[3].

Avaya is developing large, complex, real-time software systems that are embedded and standalone products. Development and testing are spread through 10 to 13 time zones in the North America, USA, Europe and Asia. R&D department employed many virtual collaboration tools such as JIRA, Git, WIKIs and Crucible. Development teams use Scrum-like development methodologies with a typical 4-week sprint. We consider a 15+ year old software component, the so-called Spark engine. As a software platform, Spark provides a consistent set of signaling platform functionalities to a variety of Avaya telephone product applications, including those of third parties. Spark is a client platform that provides signaling manager, session manager, media manager, audio manager, and video manager. The codebase involves more than 200K files and, over all forks, over 4M commits. The Android software chosen for this study is a fork of the Spark codebase. A more in-depth description of the development process is provided in [13].

3.1.2 Data Description: Source

The post-deployment data for the mobile applications were obtained from the Google Analytics platform. Google Analytics is a web analytics service offered by Google that tracks and reports website traffic. It is now one of the most widely used web analytics services on the internet. In addition to traditional web applications it also allows tracking of mobile applications. To do that, the producer of a mobile application needs to set up an account and instrument their mobile application to send certain events to Google Analytics. Notably, it works for the mobile applications investigated in this study.

[2] `https://support.avaya.com/products/P1574/avaya-equinox-for-android`

[3] `https://support.avaya.com/products/P0949/avaya-onex-mobile-sip-for-ios`

Table 1 Measures available in the Original Data

Application Release Version	No. of exceptions†
Operating System version in the user's device	Date of record entry
No. of fatal exceptions†	No. of new visits†
No. of visits†	Time on site†
Details on user's mobile device: brand, category(mobile or tablet) and model	No. of new users†
No. of total users†	Sessions per user†

We collected data for a number of mobile applications developed by Avaya from Google Analytics, but some of the datasets turned out to be unusable for this study, for reasons ranging from very low volume of collected data (*e.g.* Avaya Communicator for Android - Experimental Releases) to zero recorded exceptions making an analysis impractical (*e.g.* Avaya One-X®ScsCommander). The following datasets were found usable:

- Avaya Communicator for Android - General Availability and Development versions.
- Avaya one-X®Mobile SIP for iOS - General Availability versions.

The data was collected between December 2013 and May 2016, although the exact time varies across the applications. Although we are primarily focused on the General Availability (GA) versions, since only these versions are available for end-users, we also decided to look into the development version for Avaya Communicator for Android, since we have detailed data available for these versions and we wanted to see if it shows a different characteristics from the GA versions.

The original data obtained from Google Analytics had measures for the variables listed in Table 1, aggregated at a per-day granularity, meaning that each entry in the original data table contained the measures for the numerical variables (marked with a † symbol in the table) for each unique combination of date, application release version, operating system version, mobile device brand, category, and model. We had the same set of variable for all the applications listed above. As we mentioned earlier, we had no role in selecting which variables to measure, and we received the data "as-is".

It is important to note that Google Analytics releases only aggregate data even to developers of the application and limits the number of REST API calls, so one can not, for example, retrieve usage data for every calendar second or get exact time of the events. The daily counts split by release version of the application, OS version, and type of device, provided sufficiently fine granularity for our analysis.

3.1.3 Data Preprocessing

This section contains the data cleaning, transformation, and variable construction steps undertaken prior to the application of the different modeling methods. The major preprocessing step is aggregating the measures to a per-release granularity. We had two main reasons for aggregating the data:

1. The goals of our study are concerned with identifying the relationship between exceptions and other post-deployment variables for different releases, and defining a quality measure to compare the qualities of different releases. Therefore, having the measures aggregated at per-release granularity is essential.

2. We would have been able to take a time-series based approach and still work out our goals if the releases were cleanly separated in time, *i.e.* if there were no overlap between releases. Unfortunately, we observed from the data that users continue to use one release long after the subsequent releases are available, and there is no clear pattern about how long a release is used. Therefore, we had to aggregate the data to a per-release level to be able to achieve the goals of this study.

The preprocessing steps we took are discussed below:

Removal of variables before aggregation: Upon initial investigation into the data, we found that no. of exceptions and no. of fatal exceptions were exactly the same, as recorded by Google Analytics, so we removed the no. of fatal exceptions from the dataset. Only fatal exceptions were recorded for this application, *i.e.*, crashes that require a complete restart of the mobile application and, potentially, may affect the operating system itself. This is not surprising since the bulk of the functionality for the application was written in C++ and called from Android Java applications via Native Interface. We did not consider the variables related to mobile device details and Android operating system versions because the application, as noted above, was primarily written in C++ and the user interface aspects that vary greatest among devices and versions of OS were not likely to have influence. To validate that assumption we investigated and found no correlation of exceptions with either variable.

Data correction: This additional prepossessing step was not a part of our previous study [12]. We found during careful inspection of the data that some of the releases had non-zero number of users but zero new users in the dataset. This obviously hints at some part of the data being missing. So, as a data correction step, we modified the number of new users so that in the chronological order of the data, the cumulative number of new users is never less than the number of users for a day.

Aggregating data to per-release granularity: We had some missing values in the data, however, most of the missing data was about the mobile devices and since we didn't use them in our analysis, we got rid of that problem by simply dropping the variables. Since our aim is to model the quality of the different releases, we aggregated the data to a per-release granularity, from the the original data that was recorded in per-day granularity. The raw data contained 177 different GA releases and 25 development releases for the Avaya communicator for Android and 11 GA releases for the Avaya mobile SIP for iOS. We dropped 4 GA releases for the Avaya communicator for Android from further consideration because a significant portion of observations were missing. The result of aggregation, however, was two new variables: start date (first day for which we have a record for that release) of a release, and end date (last date for which we have a record for that release) of a release, which in turn helped create another variable: duration of a release. We did not to keep the end date in the final table, since duration and start date can be used to compute the end date.

Verifying the correctness of Release date: The original data involves only the usage aspects and the version information of the software. The project under consideration was relatively new and it was one of the early attempts for the team to deploy mobile software on Android and iOS. As such, not everything was well documented and also was rapidly evolving over time and no record of the exact

Table 2 Measures in the Aggregated Data Table

Release variable - Start Date for the release (Release.Date)	*Release variable* - Effective Duration of the release (Release.Duration)
Post-Release defects - Total No. of exceptions (Exceptions)	*Usage variable* - Average time on site per user (Usage.Intensity)
Usage variable -Total number of new users (New.Users)	*Usage variable* - No. of visits per user (Usage.Frequency)

release dates for most of the releases was available. We did manage to get release dates for some of releases from Google Play Store/ Apple App Store, but not all the release dates were available. For the releases with dates available on Google Play Store/ Apple App Store, the official release dates from Avaya records, and the start dates obtained from the data were either very close or exactly the same, so we do not have a reason to doubt the dates obtained from the data.

Removal of variables post aggregation: The numerical variables were aggregated to give a sum for each variable. Upon further inspection, we found the number of users, new users, visits, and new visits to be highly correlated. In the second iteration, we removed the variable "sessions per user", because aggregating it directly is meaningless, and we were not sure how it was originally calculated by Google Analytics (was it a mean or a median? were new users or total users counted?). We also removed the "total users" and "total visits", because while summing up the new users/visits for each day gives an accurate measurement of the total number of new users/visits for a release, it is not guaranteed that summing up total users/visits does the same due to possible double counting the number of users/visits.

Final list of variables: Keeping the goal of our study in mind, the variables we have after the initial cleaning steps give us necessary information for a model of post-release defects and software usage. In our list of variables, we have the total number of exceptions *i.e.* post-release defects. As for measures related to software usage, we have the total number of new users; the "Time.On.Site" variable, normalized by the number of users of a release, provides a measure for the temporal intensity of usage per user; and the number of visits per user is a measure for the frequency of usage. We also have two variables related to each individual release: the start date *i.e.* the release date gives a measure for the calender time of each release, and is useful in gaining insight about if the number of post-release defects and software usage vary with time, and the duration of a release, which could have an effect on the number of exceptions and the number of new users, since these variables were not normalized with duration. Since we only have a limited amount of data, we restricted ourselves to use only these six variables. Our final aggregated data table had the measures listed in Table 2, with the corresponding variable names we used in the model enclosed in brackets.

To reiterate what we mentioned earlier, we had no control over which variables to measure during data collection, however, while the set of variables we obtained are not exhaustive, we believe the three usage related variables: number of new users, usage intensity, and usage frequency adequately capture and report how much usage a software is getting.

Log-transformation of variables: The release date was converted from the Date format to numeric format, which resulted in the values for the release date variable being represented by the difference in days from Unix time (counted from 1970-01-01). We found that all of the variables under consideration had a long-tailed

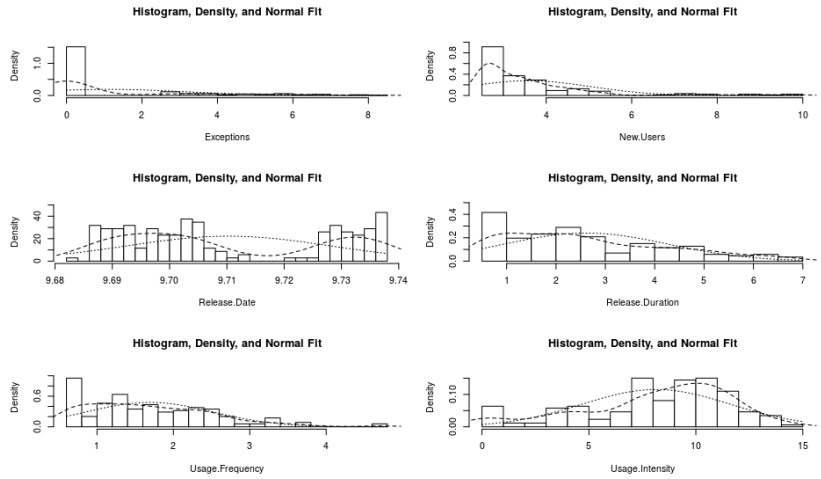

Fig. 1 Distribution of the variables after transformation: GA releases of Avaya communicator for Android

distribution, so we took logarithm of them. The distribution of the variables of GA releases of Avaya communicator for Android is shown in Figure 1. The distribution of the variables of other applications is available in our GitHub repository: `https://github.com/tapjdey/release_qual_model`.

3.2 The NPM Packages

The two notable points about the NPM data are:
(1). We limited our study to only the more popular packages in the NPM ecosystem, since a vast majority of NPM packages have a very limited number of users, and can not truly be considered a candidate for a study exploring the relationship between software usage and software faults/exceptions/issues. As mentioned earlier, we defined the criteria that a package should have more than 10,000 downloads per month (this is the criteria for popularity we defined, because, according to [69], automated downloads are expected to be around 50 per day, or 1500 per month, and packages with over 10K downloads should, therefore, not be noticeably impacted by downloads by automated sources.) and a GitHub page with issues enabled to be a part of our study. Under this condition, we ended up with 4430 NPM packages which is roughly 0.5% of the entire NPM ecosystem (which had around 800,000 packages at the time of data collection).
(2). The NPM data we obtained was relatively clean, and since we only used this data to extend the external validity of our study, we did not employ any preprocessing steps on this data.

3.2.1 The NPM Ecosystem

Node Package Manager or NPM is one of the most active and dynamic software ecosystems at present. It hosted more than 800,000 packages at the time of data collection, and have more than doubled in size in past couple of years (in

January 2017, NPM reportedly hosted around 350,000 packages [11]). The popularity of NPM packages have, accordingly, skyrocketed as well. According to [70], "JavaScript is getting more popular all the time, and NPM is being adopted by an ever greater percentage of the JavaScript community." About 75% of all JavaScript developers used NPM, with about 10 million users, in January 2018, according to [70]. Therefore, NPM is an excellent candidate for this study. Moreover, since they track the number of downloads of all packages in the ecosystem, which, in spite of essentially being a mix of downloads by users, bots, and mirror servers, as explained in [69], is the closest measure of usage we could find for open-source projects, and is a far better measure than, *e.g.* number of stars of a GitHub repository which was used in studies like [4] as a measure of popularity, which is little different from usage as we measure it.

3.2.2 NPM data: Defining collection parameters

We found 4430 projects which had more than 10,000 monthly downloads since January 2018 and also had public GitHub repositories with nonzero number of issues. We collected the number of downloads and the total number of issues for all these packages from 2015-03-01 to 2018-08-31. However, we did not conduct a release by release comparison for these packages, because the release durations vary by a lot for most packages. Since the recorded number of downloads is a mix of downloads by human and non-human users, a release by release comparison would not give a reliable picture of the effect of actual usage by human users on the number of issues. However, the number of downloads by bots are relatively stable and vary only with time [69], so controlling for the date (time variable) would eliminate the spurious effects of downloads by bots. So, we decided to focus on the entire packages instead of releases of the packages, and measured the effect daily downloads have on number of issues of that package on that day after controlling for the calender date.

3.2.3 NPM data: Data collection

We used the API provided by NPM for collecting daily downloads of the 4430 NPM packages we studied. (The API documentation is available in: *https://github.com/npm/registry/blob/master/docs/download-counts.md*).

To obtain the metadata information for every package in NPM, we wrote a "follower" script, as described in *https://github.com/npm/registry/blob/master/docs/follower.md*. The output contained the metadata information for all releases of all packages in NPM. From this we extracted the URL of GitHub repositories of the packages. Some NPM packages do not have a valid GitHub URL, so those were dropped from subsequent analysis, as per the criteria we defined. Using the Rest API provided by GitHub we collected information on the issues for all these NPM packages. Finally, we used the issue creation dates to construct a dataset of the total number of issues per day. We used the total number of issues instead of the number of open issues because we are interested in the number of issues encountered by the users of the packages. Whether an issue is resolved or not depends on a number of factors, *e.g.* the number of developers, the responsiveness of the developers, the number of packages managed by each developer, the complexity of the problem; most of

which are unrelated to usage, so we decided using the total number of issues is a much more reasonable option.

Our final dataset for the NPM packages contained the number of recorded downloads per day and the total number of issues reported for that package until that day. The date range was from 2015-03-01 to 2018-08-31.

4 Methodological Overview

In this section we describe the methodology we followed in this paper. We employed three different modeling techniques for finding out the relationship among the post-release variables: (1) The Linear Regression (Ordinary Least Squares - OLS) method, (2) Bayesian structure search method, and (3) Random Forest Regression method. Since we primarily focus on finding which variables are have the most impact on the number of exceptions, we used it as the response variable for the OLS and Random Forest regression models.

We started by using the OLS estimator since it is one of the simplest modeling methods that gives good models in a lot of situations and the result is easy to interpret. However, we were unsure about the accuracy of the result due to the presence to moderate to high correlation between some of the predictor variables (*e.g.* the Release Date and Release Duration variables had a correlation of -0.88 for the iOS application data). Therefore, we decided to use Bayesian Network (BN) for modeling the interrelationship among these variables, since the accuracy of this model is unaffected by the presence of high correlation among the predictors. Variables with high correlation simply appear as connected nodes in the final model. Since the use of Bayesian Network models is not very common in this context and it is one of the main contributions of this paper, we discuss BN models in greater detail later in this section. The third and last modeling approach we used is Random Forest regression method. Random Forest is one of the best off-the-shelf models that work well with almost all types of data and generally does and it is easy to get the relative importance of the predictor variables from a fitted model. These two factors led us to use Random Forest regression as a third modeling technique to identify the most impactful predictors explaining the number of exceptions. To find the best fitting Random Forest model, we performed a grid search using the "tune" function of the "e1071" [40] R package to find the best model parameters "ntree": the number of tress to grow, and "mtry": the number of variables randomly sampled as candidates at each split. Since the sample size of datasets are limited, we used 10 times 2 fold cross-validation as the tuning method. Another reason for using the Random Forest model is that it is a predictive model, while the other two are explanatory models and they serve different purposes [65]. We wanted to examine the effect of post-deployment measures for the purpose of prediction and observing which variables play an important role.

4.1 Bayesian Network Models

Bayesian Network [33, 64] is a type of Probabilistic Graphical Model (PGM), which explicitly represents the conditional dependency/independence as a directed acyclic graph where variables represent nodes and dependencies represent links,

and thus this representation can be used as a generative model[4]. Bayesian Networks models can be useful in the context of Software Engineering research [17] due to having several advantages over regression models. To be precise, regression analysis is a very simple BN where there is one directed link from each independent variable to dependent variable. BNs, therefore, can help with multicollinearity, a common problem with software engineering data [72, 68, 6, 41], that is present in our data as well, by linking independent variables.

Another variety of PGM that we did not use in this paper (details in Section 8) is the Markov random fields that represent the interrelationships between variables as undirected graphs. They differ in the set of independencies they can encode and the factorization of the distribution that they induce [33].

Bayesian Network Model construction: Despite the promises of BNs, they tend to be quite sensitive to data, and operational data, is often problematic [43, 75]. Careful preprocessing, therefore, is needed to ensure a reliable and reproducible result. Two primary ways to use BNs exist. With the first approach the graph represents dependencies obtained from domain experts. The graph may include prior distributions about the parameters of the overall model. The data is then used to calculate the posterior distribution and to make inference. The second approach puts minimal a-priori assumptions about the model and focuses on the search for the best graphical representation for a given dataset (structure learning). This is an NP-hard problem [8], but a number of different heuristic structure learning algorithms are available. Due to the lack of any strong theory connecting the variables we are considering, we decided to use the structure search method for BN model construction. Since our goal is to find a Bayesian network model for the data, we didn't examine the methods that do not result in a Directed Acyclic Graph (DAG). We found that the *bnlearn* package in R implements a wide range of BN searching methods for continuous, discrete, or a mixed set of variables and the corresponding families of scoring functions and also has a good number of examples. These methods were also shown to be able to recover the underlying network for a protein-signaling-chain (in Biology) in [63]. We, therefore, use this package for our analysis. In addition to the methods implemented in *bnlearn* package, we investigated some methods from a few other packages which can be interfaced with the *bnlearn* package.

Due to the potential inconsistencies of the BN models, we performed our modeling in two stages. First, we considered all available BN structure methods in the *bnlearn* package and ran a simulation based study to find the methods that are most accurate and then we used those methods on our data to create the final model.

Methods considered:
The different BN structure search methods we considered are listed below:

[4] A generative model specifies a joint probability distribution over all observed variables, whereas a discriminative model (like the ones obtained from regression or decision trees) provides a model only for the target variable(s) conditional on the predictor variables. Thus, while a discriminative model allows only sampling of the target variables conditional on the predictors, a generative model can be used, for example, to simulate (*i.e.* generate) values of any variable in the model, and consequently, to gain an understanding of the underlying mechanics of a system, generative models are essential.

- *Greedy Hill-Climbing search algorithms*(HC) [51,63]
- *Hybrid algorithms*(Hybrid) [51,63]
- *Posterior maximization* using *deal* package in R [63,5] .
- *Simulated Annealing* using *catnet* package in R [2,63].
- *MAP (maximum a posteriori estimation) Bayesian Model Averaging* (MAP) [51,63]

This is not an exhaustive list of all possible BN structure search methods, in fact, it is impossible to make an exhaustive list for a heuristic search method like this, however, they represent a class of popular heuristic structure search methods that are part of the "bnlearn" package, which is a popular R package that is in continuous development since 2007.

All structure search algorithms try to maximize some form of a network score. Among the various scores available, BIC score is the suitable one when the goal is to create an explanatory model from non-informative prior models [65,66]. BIC score is used for discrete data while the Gaussian equivalent of BIC (bic-g) score is used for continuous data.

The results, *i.e.* the structure and the parameters resulting from a structure search algorithm, are often noisy, meaning that different settings induce slightly different networks. To mitigate this effect we use non-parametric bootstrap model averaging method described in [18], which provides confidence level for both the existence of edge and its direction. This enables us to select a model based a confidence threshold. Authors of [18] argue that threshold is domain specific and needs to be determined for each domain. For instance, a threshold of 0.95 indicates that only the edges that appeared in more than 95% of the bootstrap optimized models were selected.

Many applications of BNs discretize the data prior to applying the structure learning methods, and in some cases where the data distribution is too skewed to fit the normality assumption, discretizing the data produces better models than using continuous data, so we considered it as a possibility as well.

Using continuous data works best when the random variables (possibly after a transformation) have Gaussian distribution. While using discrete data does not require such assumptions, obtaining the optimal discretization for a dataset is in itself an NP-hard problem [9]. Choosing a sub-optimal discretization technique may result in spurious or missed relationships, which can in turn result in incorrect dependencies being reported in the resulting model. Given the pros and cons of both types of methods, we use methods of both types for our simulation study. As we are interested in creating a generative model, we had to use a discretization method that is unsupervised. The basic problem with commonly used supervised methods (*e.g.* Chi-square, or MDLP discretization algorithms) is that they optimize discretization to improve explanatory power for a single response variable. This is not suitable for a BN structure search, because we do not know which variables will be responses (have arrows pointing to them) and which will be independent (have no incoming arrows) *a-priori*. While some research on multidimensional discretization methods exists [57], we are not aware of any that have a robust implementation.

Simulation Study:

We performed the simulation study by first creating a random BN (see Figure 2)

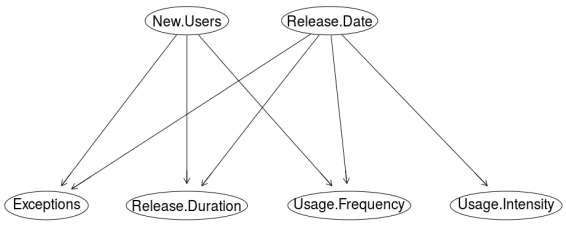

Fig. 2 Custom model used for Simulation Study

with six nodes, since we also have six variables in our final list (Table 2). For demonstration purposes we use the same variable names. We fitted this graph with our data on GA releases of Avaya communicator for Android (log-transformed and scaled) to generate values for the coefficients for each edge. This model was used in our simulation study going forward. We created 1000 different simulated datasets from the BN structure in Figure 2, and applied the different structure search algorithms (both continuous and discrete versions, where available) listed above. Our performance metric is finding how many times the different algorithms can recover the underlying structure from the simulated data.

Other than testing the methods themselves, we also tested whether or not we should discretize the data. We tried different discretization methods,as per the suggestion by [19], *viz.* equal interval, equal frequency, and k-means clustering based discretization methods from the *arules* package [23], and the Hartemink[5] discretization methods in the *bnlearn* package.

Except for the *Posterior maximization* using *deal* package, which can't be bootstrapped, all other results were bootstrapped, so we tested different thresholds in our simulation study as well. Finally, for the the Hybrid search algorithm, in which conditional independence tests are performed to restrict the search space for a subsequent greedy search, there are many restrict methods available, *viz.* gs" (Grow-Shrink), "iamb" (IAMB), "fast.iamb" (Fast-IAMB), "inter.iamb" (Inter-IAMB), "mmpc" (Max-Min Parent Children), "si.hiton.pc" (Semi- Interleaved HITON-PC), "chow.liu" (Chow-Liu), "aracne" (ARACNE) [36], and we tested all of these restrict options in our simulation study.

The reason for experimenting with these broad set of methods and options is that we wanted to perform the simulation study on a broad range to techniques to identify the best performing heuristic method in this context (*i.e.*, for the software failure data).

The result of the simulation study is shown in Table 3, which shows the fraction of times exact structures and off-by-one structures[6] were generated by each method in the simulation. The result varies with the chosen threshold, so in Table 3, we show the overall performance of the different methods which generated an exact or off-by-one structure at least once in the simulation. For the hybrid search methods, we list mention the restrict option that was used, and the '-D' suffix indicates a discretization method was used to discretize the data prior to applying a structure search method. '-D-H' indicates Hartemink discretization method and

[5] Hartemink's pairwise mutual information method[24].

[6] one extra / missing / reversed edge

Table 3 Result of Simulation Study

Method	**Exact**	**Off-by-one**
HC	0.574	0.264
MAP	0.596	0.214
Hybrid- si.hiton.pc	0.000	0.019
Hybrid- mmpc	0.000	0.016
Hybrid- gs	0.000	0.011
HC-D-F	0.000	0.010
Hybrid- iamb	0.000	0.010
Hybrid- mmpc -D-H	0.000	0.008
Hybrid- si.hiton.pc -D-H	0.000	0.008
HC-D-H	0.000	0.007
Hybrid- mmpc -D-F	0.000	0.007
Hybrid- si.hiton.pc -D-F	0.000	0.006
Hybrid- iamb -D-F	0.000	0.005
Hybrid- gs -D-F	0.000	0.004
Hybrid- gs -D-H	0.000	0.004
Hybrid- iamb -D-H	0.000	0.002

Table 4 Result of Simulation Study: Different Thresholds

Method	Threshold	Exact	Off-by-one
MAP	0.85	0.68	0.25
MAP	0.80	0.67	0.25
MAP	0.90	0.67	0.26
MAP	0.95	0.66	0.27
MAP	1.00	0.66	0.27
MAP	0.75	0.66	0.21
HC	0.65	0.63	0.23
HC	0.70	0.63	0.23
HC	0.75	0.63	0.23
HC	0.80	0.63	0.23
HC	0.85	0.62	0.24
HC	0.55	0.62	0.23
HC	0.60	0.62	0.23
MAP	0.70	0.62	0.21
HC	0.90	0.60	0.26
MAP	0.65	0.58	0.17
HC	0.95	0.57	0.29
MAP	0.60	0.43	0.14
MAP	0.55	0.33	0.11
HC	1.00	0.19	0.47

'-D-F' indicates Equal-Frequency discretization method. It is clear from the table that only HC and MAP methods can effectively reproduce the correct underlying structure around half of the times and they create more off-by-one structures than others, indicating the error rate is the lowest for these methods.

In Table 4, we show the fraction of times exact and off-by-one models were generated by HC and MAP methods, which performed the best among the methods considered, for different thresholds. It can be seen that using a moderately high threshold between 0.75 and 0.9 gives good results for both HC and MAP, while higher thresholds for HC and lower thresholds for MAP give worse results. Using the optimal threshold creates models that have more than one wrong and/or missing edge only 7-14% of the times.

The result of the simulation study had the following findings:

- Using structure search algorithms on the continuous data resulted in much more frequent recovery of the original BN structure compared to discretized data.
- Bootstrapping improves the stability of the results considerably.
- The bootstrapped Hill-Climbing search and MAP Bayesian Model Averaging algorithms outperformed all others both in terms of accuracy and runtime, being able to recover the underlying structure more than 63% of the times and making no more than one error 86% of times with optimal thresholds.

We consider this study one of the contributions of the paper, and hope that it would be useful for researchers using BN structure learning techniques.

5 Answering the Research Questions: Results and Analysis

5.1 RQ1: Modeling the relationship between Exceptions and other post-release variables

As mentioned earlier, we conducted our analysis in three stages: first, we used linear regression (LR) on the data with the number of exceptions as the response

variable; then, we used Bayesian Network (BN) modeling approach to identify the interrelationship between the variables; and finally, we used a random forest (RF) model to verify the results.

5.1.1 Linear Regression Model

We first used a linear regression model (OLS) to discover the significant variables affecting the number of exceptions. The output of the fitted model is shown in Table 5.The model resulted in a decent fit for the GA releases of Avaya Communicator for Android, given the sample size of 173, with adjusted R^2 of 0.481 (which is higher than 0.435, the adjusted R^2 reported with our previous approach [12]), and the variable "New.Users" was the most significant predictor while "Release.Date", "Usage Intensity", and "Usage Frequency" were also statistically significant. For the development releases of the same, only "New.Users" was significant, and it resulted in a good fit, with adjusted R^2 of 0.781. Finally, for the GA releases of Avaya mobile SIP for iOS, the value of R^2 was 0.874, and "New.Users", "Release.Date", and "Release.Duration" were the significant variables.

We were doubtful about the accuracy of the model since some of the predictors had a moderate to high degree of correlation. We checked the Variance Inflation Factor [39] and found it be less than 5 for all predictors, which shows the effect of multicollinearity isn't very strong. However, we also wanted to verify other assumptions behind the OLS model. We went through the list of model assumptions provided in [26], and found that our model does not satisfy the Normality assumption (assumption 6 in [26]) , *i.e.* the residuals are not normally distributed, nor does it satisfy the criteria that the variance of the errors term is constant, conditional on the all predictors (assumption 5 in [26]). However, it satisfies assumptions 1-3 (linearity, random sample, and no perfect multicollinearity assumptions) and the weaker assumption 4 (namely that: (1) E(residuals) = 0 and (2)Cov(each predictor, residuals) = 0), as mentioned in [26]. Therefore, according to [26], the OLS estimator we obtained is not BLUE (Best Linear Unbiased Estimator), but it is unbiased, and consistent. However, given the observed limitations of the OLS estimator, we decided to use other modeling approaches to verify the results reported by using this approach.

Table 5 Summary Result of LR model for "Exceptions"

	GA releases of Android			Dev releases of Android			Mobile SIP for iOS		
	Estimate	Std. Error	p-value	Estimate	Std. Error	p-value	Estimate	Std. Error	p-value
(Intercept)	-145.7827	72.2447	0.0452	-514.753	411.865	0.227	998.2528	284.0527	0.0170
New.Users	0.8172	0.1229	**0.0000**	1.08755	0.155	**1.11e-06**	1.2766	0.1969	**0.0013**
Release.Date	14.7756	7.4449	0.0488	52.884	42.383	0.227	-103.8231	29.4034	0.0167
Release.Duration	0.1361	0.1290	0.2930	-0.031	0.297	0.917	-2.0129	0.3381	0.0019
Usage.Frequency	-0.5388	0.2179	0.0144	-0.321	0.526	0.550	0.8246	0.9057	0.4043
Usage.Intensity	0.1261	0.0615	0.0419	0.025	0.183	0.892	1.0318	0.5286	0.1084

5.1.2 Bayesian Network Model

One key assumption for applying the continuous BN structure search algorithms is that the variables have a distribution close to a Gaussian distribution. To satisfy this modeling assumption, we scaled all the variables to unit scale. The variable "Exceptions" still had a long tailed distribution, but the distributions of the other variables were much closer to normal distribution.

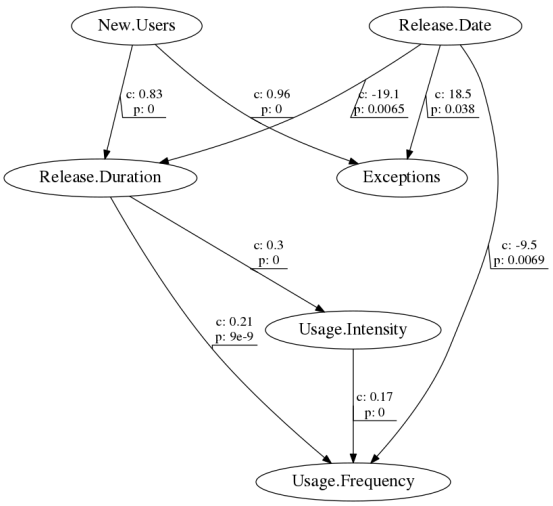

Fig. 3 Final BN Model for GA releases of Avaya Communicator for Android (with c: coefficients after fitting the transformed, but unscaled data, p: p-value for the link)

According to the result of the simulation study, we decided to use bootstrapped hill-climbing search and MAP Bayesian model averaging methods for constructing the Final BN models for our datasets and considered the model that resulted from both the methods. The resultant BN model for the GA releases of Avaya Communicator for Android is shown in Figure 3, which shows "New.Users" and "Release.Date" are parent nodes of "Exceptions". Figure 4 shows the final BN Model for Development releases of Avaya Communicator for Android, in which only "New.Users" is the parent of "Exceptions", and Figure 5 shows the final BN Model for GA releases of Avaya mobile SIP for iOS, where once again "New.Users" and "Release.Date" are parent nodes of "Exceptions". In these figures p-values $< 2e-16$ are denoted as 0.

Every bootstrap run was performed over 500 bootstrap samples, and a hill-climbing search with 100 random restarts was applied on each sample to find the best fitting network, so in essence, each resultant network was obtained by averaging 50,000 candidate networks. We used a Threshold of 0.85, as it seemed optimal from our simulation study.

The result form a bootstrap run shows the relative strength of the link and the relative confidence for the direction of the link. In Table 6 we have shown the result from one bootstrap run of the HC method for all possible edges for the GA release data of Avaya Communicator for Android. If an edge has $< 50\%$ confidence in its direction, then the edge appears in the opposite direction in our model. Although Bayesian Networks are sometimes interpreted as causal relationships [55], there are disagreements on how that should be done. We, therefore, are not interpreting these relationships as causal here. All observed links, therefore, indicate the presence of observed correlation (and are empirical in nature) and the direction is a property of the topological ordering of nodes in a DAG, and affects the total probability distribution of the variables.

The BN models were fitted to the unscaled data, and the resulting coefficient of each link is also shown in the figures. The p-value for each link was calculated

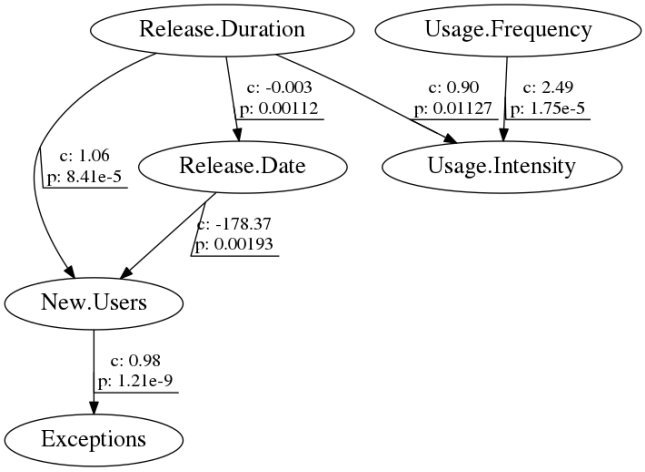

Fig. 4 Final BN Model for Development releases of Avaya Communicator for Android (with c: coefficients after fitting the transformed, but unscaled data, p: p-value for the link)

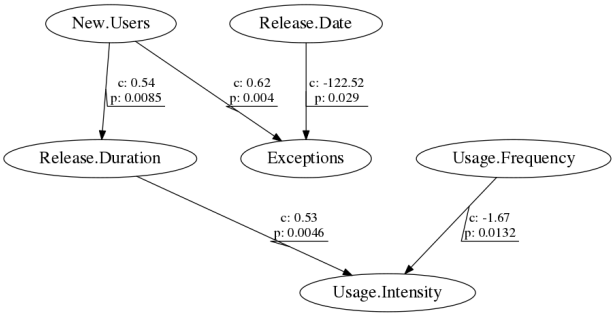

Fig. 5 Final BN Model for GA releases of Avaya mobile SIP for iOS (with c: coefficients after fitting the transformed, but unscaled data, p: p-value for the link)

from a linear model with the source nodes as predictors and the destination node as the response variable, *e.g.* the p-value for the link from "New.Users" to "Exceptions" was calculated by looking at the result of: `lm(Exceptions ~ New.Users + Release.Date)`.
We fitted the model to the transformed, but unscaled data (for easier interpretation of results).

By looking at the p-values for the links, we can say that all the links in the BN models are statistically significant. Links having a negative coefficient indicate an inverse relationship between the parent and the child node. The performance of explanatory models is evaluated by the fraction of deviance explained by the model. Our model explains 80.3% and 45.9% of the variation in "Exceptions" (adjusted R^2 value of the model) for development and GA releases for Avaya Communicator for Android respectively and 42.0% for GA releases of Avaya mobile SIP for iOS. This indicates our BN model is statistically significant, but the predictors we used could only explain around half of the variance in Exceptions.

Table 6 Example bootstrap result - GA releases of Avaya Communicator for Android

from	to	strength	direction
Exceptions	New.Users	1.00	0.34
Exceptions	Release.Date	0.86	0.47
Exceptions	Release.Duration	0.46	0.50
Exceptions	Usage.Frequency	0.75	0.78
Exceptions	Usage.Intensity	0.35	0.47
New.Users	Exceptions	1.00	0.66
New.Users	Release.Date	0.20	0.62
New.Users	Release.Duration	1.00	0.71
New.Users	Usage.Frequency	0.71	0.85
New.Users	Usage.Intensity	0.34	0.64
Release.Date	Exceptions	0.86	0.53
Release.Date	New.Users	0.20	0.38
Release.Date	Release.Duration	1.00	0.63
Release.Date	Usage.Frequency	0.97	0.82
Release.Date	Usage.Intensity	0.66	0.77
Release.Duration	Exceptions	0.46	0.50
Release.Duration	New.Users	1.00	0.29
Release.Duration	Release.Date	1.00	0.37
Release.Duration	Usage.Frequency	0.90	0.55
Release.Duration	Usage.Intensity	1.00	0.53
Usage.Frequency	Exceptions	0.75	0.22
Usage.Frequency	New.Users	0.71	0.15
Usage.Frequency	Release.Date	0.97	0.18
Usage.Frequency	Release.Duration	0.90	0.45
Usage.Frequency	Usage.Intensity	1.00	0.22
Usage.Intensity	Exceptions	0.35	0.53
Usage.Intensity	New.Users	0.34	0.36
Usage.Intensity	Release.Date	0.66	0.23
Usage.Intensity	Release.Duration	1.00	0.47
Usage.Intensity	Usage.Frequency	1.00	0.78

5.1.3 Random Forest Model

As a verification step to identify the important variables affecting the number of exceptions, we used a Random Forest model to fit the data, with "Exceptions" as the response variable. The variable importance plot for the GA release data of Avaya Communicator for Android, as shown in Figure 6, indicates that "Release.Date" and "New.Users" are the two most important variables. For the development releases of Avaya Communicator for Android, the variable importance plot is shown in Figure 7. "New.Users" is again the most important variable, followed by "Release.Duration". For the GA releases of Avaya mobile SIP for iOS, the variable importance plot again shows the number of new users is the most important variable, as can be seen from Figure 8.

The best selected model parameters derived from tuning show that the optimal models were obtained for "ntree"=600 and "mtry"=3 for all datasets. The R^2 values for these models, again obtained from 10 times 2 fold cross-validation, are 0.48, 0.56, and 0.31 for the GA and development releases of the Android application and the GA releases of the iOS application respectively. The poor performance of the iOS application is likely due to the very small sample size of the dataset. Although the overall performance of the models wasn't very good, since we had a limited number of predictors, and none of the internal factors were part of the model, this result shows that even for the purpose of prediction, the number of new users play an important role.

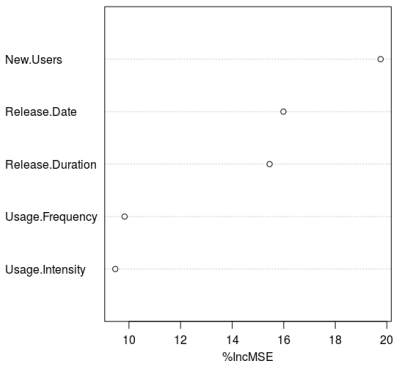

Fig. 6 Variable Importance Plot of RF model for "Exceptions" for GA release data of Avaya Communicator for Android

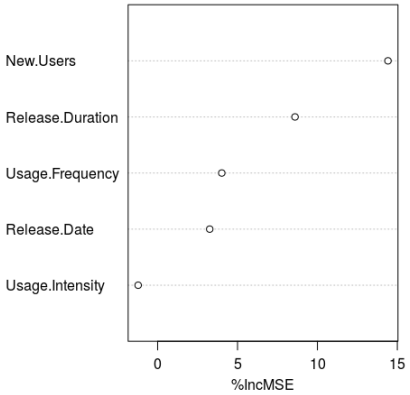

Fig. 7 Variable Importance Plot of RF model for "Exceptions" for development release data of Avaya Communicator for Android

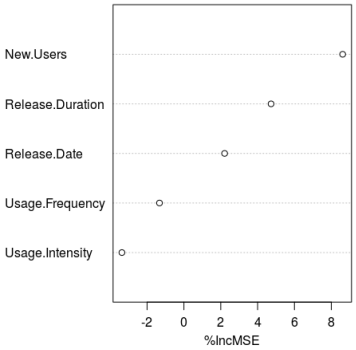

Fig. 8 Variable Importance Plot of RF model for "Exceptions" for GA releases of Avaya mobile SIP for iOS

5.2 RQ2: Deriving a usage-independent measure of Quality

5.2.1 Obtaining the Quality measure

In order to arrive at the usage independent quality measure, we follow the framework of establishing laws governing relationships among measures of software development proposed in [42]. Law is an equivalent of invariance, *i.e.* a function of measures that is constant under certain conditions. In this case we want it to be constant for releases that have the same quality. First, the law requires a plausible mechanism and second, an empirical validation. Each new user may have a different type of phone, operating system, service provider, geographic region, and usage pattern. It is reasonable to assume that some of these configurations lead to software malfunction manifested as an exception. This provides us with a plausible mechanism on how precisely more new users of one release might generate more exceptions even if we have two releases of identical quality. We rely on our mod-

els (all of which show the number of software exceptions to be dependent on the number of users and on the software release date) to obtain empirical validation of this postulated mechanistic relationship. Therefore, we arrive at the following software law that is applicable for the investigated context: the average number of exceptions experienced by each user should, therefore, be independent of usage and depend only on the qualities of a software release.

In this section we test the above evidence-based hypothesis and provide the result of an analysis with the **number of exceptions per user** as a response variable ("Quality") representing software quality. *This is actually a measure for faultiness, so a lower value of "Quality" indicates the actual quality of the software perceived by end users is better.*

The value of the "Quality" variable (not log transformed) was seen to be varying between 0 and 10.85 (mean: 0.45, median: 0, standard deviation: 1.48) for the GA release data of Avaya Communicator for Android, between 0 and 22.83 (mean: 1.12, median: 0, standard deviation: 4.55) for development versions of the same, and for the GA releases of Avaya mobile SIP for iOS it varied between 0 and 0.5 (mean: 0.0488, standard deviation: 0.15) .

5.2.2 Establishing the independence of the Quality measure and other usage related variables

Similar to the previous analysis, we applied Linear Regression, Bayesian Network search, and Random Forest modeling approaches on the dataset containing this quality measure and the remaining variables, all of which were log-transformed.

The result, as expected, shows that the quality of a software, measured by average number of faults experienced by each user, has no dependence on other usage variables. The LR model of GA releases of Avaya Communicator for Android (Table 7) suggest that other variables have some effect on the quality variable. The BN model(Figure 9), obtained with a threshold of 0.85 from a bootstrapped Hill-Climbing structure search model, indicates the "Quality" variable depends only on the "Release.Date" variable. Finally, the result of 10 times 2-fold cross-validation with the best RF model (Variable Importance plot in Figure 12) with the optimal values for "ntree"(300 in this case)) and "mtry"(1 in this case) indicates that the "Release.Date" variable is much more important compared to others, and the two usage related variables are of much lower importance.

For the development versions of Avaya Communicator for Android, all the predictor variables turned out be insignificant in both the LR (Table 7) and the BN (Figure 10) models. Even the tuned RF model gives a really bad fit in the 10 times 2-fold cross-validation as well, with a R^2 value of -0.42 (the implication of a negative value of R^2 is as explained in [50]), indicating the predictors are very poor. Still, the two usage related variables have the lowest importance in the variable importance plot as seen in Figure 13.

Finally, for the GA releases of Avaya mobile SIP for iOS, the LR model (Table 7) as well as the BN model (Figure 11) shows that release date and release duration have effect on the "Quality" variable, but the two other usage variables have no effect. We did not run RF model on this dataset owing to the very small sample size.

The results from these analyses clearly indicate that the quality measure defined by the number of exceptions per user is independent of software usage,

Table 7 Summary Result of LR Model for "Quality"

	GA releases of Android			Dev releases of Android			Mobile SIP for iOS		
	Estimate	Std. Error	p-value	Estimate	Std. Error	p-value	Estimate	Std. Error	p-value
(Intercept)	-59.6470	19.9155	0.0032	−285.0676	180.0109	0.1290	28.5606	1.1271	0.0000
Release.Date	6.1428	2.0503	0.0031	29.3616	18.5247	0.1287	-2.8833	0.1167	0.0000
Release.Duration	0.0514	0.0271	0.0600	0.0854	0.1152	0.4671	-0.0923	0.0012	0.0000
Usage.Frequency	-0.1593	0.0618	0.0108	-0.3136	0.2372	0.2011	0.0006	0.0037	0.8831
Usage.Intensity	0.0395	0.0175	0.0253	0.0595	0.0820	0.4764	0.0001	0.0022	0.9614

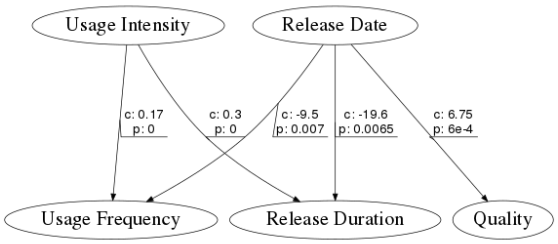

Fig. 9 Bayesian Network Model for "Quality" - GA releases of Avaya Communicator for Android (with c: coefficients after fitting the transformed, but unscaled data, p: p-value for the link)

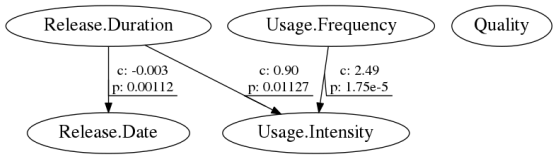

Fig. 10 Bayesian Network Model for "Quality" - Development releases of Avaya Communicator for Android (with c: coefficients after fitting the transformed, but unscaled data, p: p-value for the link)

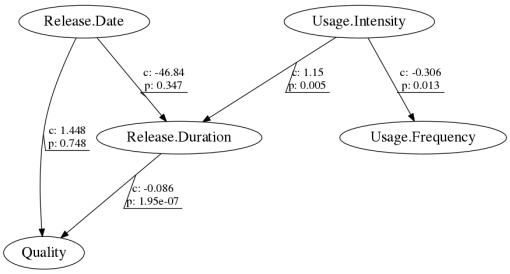

Fig. 11 Bayesian Network Model for "Quality" - GA releases of Avaya mobile SIP for iOS (with c: coefficients after fitting the transformed, but unscaled data, p: p-value for the link)

and, therefore, suitable for comparing the quality of software development process among different releases of a software.

5.2.3 Timeline of Quality

We wanted to see how the perceived quality of the releases of the different mobile applications described above change with time. As a general trend, we observe that most of the exceptions occur right after the release date. then, as the number of users keep increasing with time, the value of the quality variable drop and come to a stable value. In this paper we show only the timeline for GA releases of Avaya mobile SIP for iOS (Figure 14), since the other two softwares had a lot of releases,

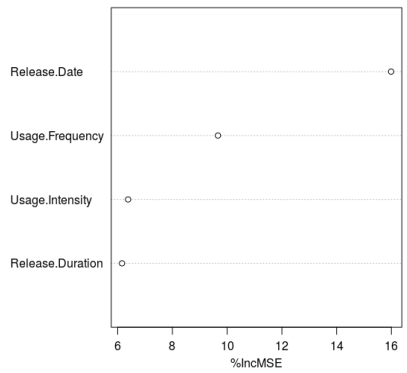

Fig. 12 Variable Importance plot from the Random Forest Model for Quality Variable - GA releases of Avaya Communicator for Android

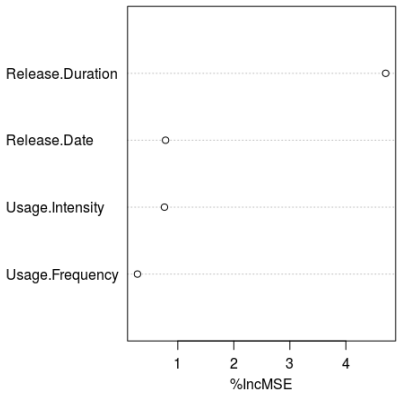

Fig. 13 Variable Importance plot from the Random Forest Model for Quality Variable - Development releases of Avaya Communicator for Android

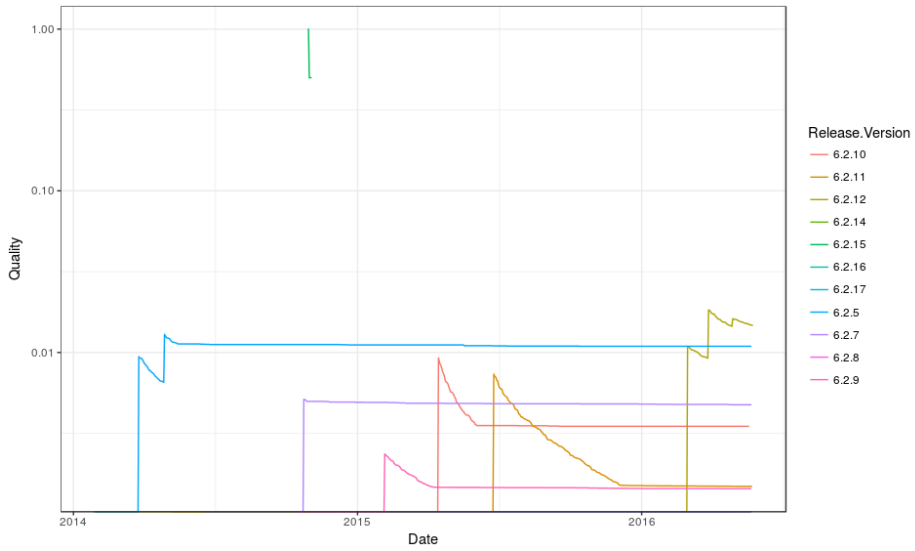

Fig. 14 Timeline for Quality Variable - GA releases of Avaya mobile SIP for iOS

making them difficult to identify from the plot. The other two are are available in our GitHub repository:
`https://github.com/tapjdey/release_qual_model`.

In Figure 15, 16, and 17, we show the relative trends of Exception and the Quality variable for the GA and development releases of the Android application and the GA releases of the iOS application respectively. We are not interested in the absolute values of the metrics, but the values of the metrics for a release relative to the values for other releases. We only show the releases with non-zero number of exceptions, since if the number of exceptions is zero, the the value of the quality metric is also zero. The blue dotted lines represent the releases dates of

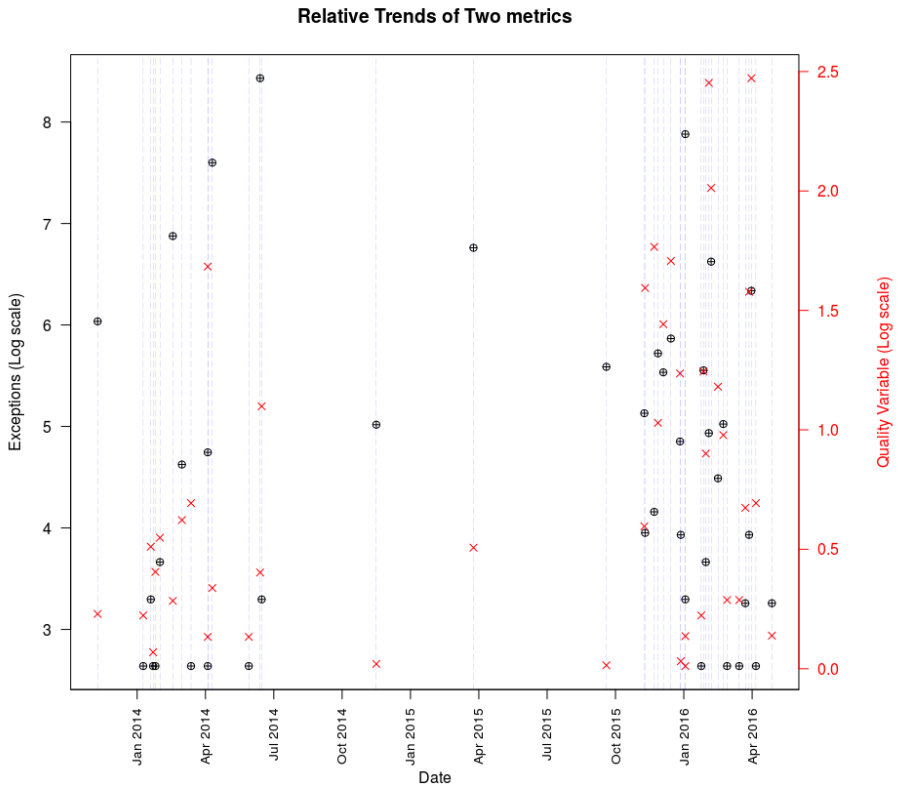

Fig. 15 Relative trends of Exception and the Quality variable - GA releases of Avaya Communicator for Android

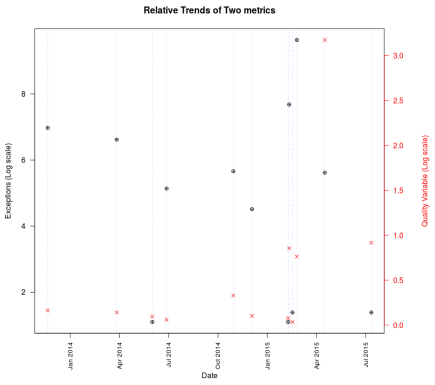

Fig. 16 Relative trends of Exception and the Quality variable - Development releases of Avaya Communicator for Android

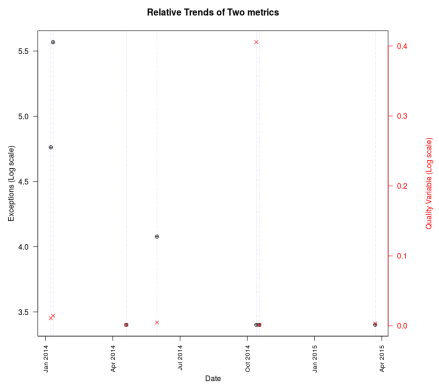

Fig. 17 Relative trends of Exception and the Quality variable - GA releases of Avaya mobile SIP for iOS

different releases, and the black marker and the red cross on the blue line represent the exceptions and the quality variable for that release respectively.

We can see that for a number of releases, Exceptions and Quality follow a similar trend, *i.e.* if the number of exceptions increase, the value of the Quality variable increases accordingly. However, there are indeed a number of cases where the number of exceptions is relatively small, but the value of Quality variable is larger than that for other releases, *e.g.* for many of the GA releases for the Android application in 2016, or vice versa, *e.g.* the release around September 2015 for the GA releases for the Android application. This indicates that if we simply keep on using the number of exceptions as the quality measure, we will misclassify (as being better or worse than other releases) a number of releases. This result supports our hypothesis that not accounting for the usage parameter will not systematically misclassify all releases, since the internal factors affecting the release are independent of the external factors, but would randomly misclassify some of them depending on how much usage a release is getting.

5.3 RQ3: Analysis of the NPM Data and Results

5.3.1 methodology

As mentioned before, for the analysis of the NPM data we focused on the timeline of the entire packages from 2015-03-01 to 2018-08-31, instead of the individual releases of a package to reduce the effect of the downloads by bots and other automated sources. Since we only had three variables (no. of issues, no. of downloads, calendar date) in our dataset, we decided to go for the simpler liner model (OLS). We had the total number of reported issues for the package as the response variable and observed how much the number of daily downloads, before and after controlling for the calender date, explained it.

We also presented the timelines for a few well-known NPM packages, showing the comparative trends of the number of issues and our proposed quality measure, defined as the number of issues per download. Since the number of downloads has a large variation, the quality measure also has a high degree of variation. So, we decided to fit a model to the quality variable, and add another line representing the fitted values of the model. We first tried to use a OLS model, but given the apparent non-linearity of the data, later decided to use a Generalized Additive Model (GAM). We did not fine tune this model, because it was only used for demonstrating the trend of the quality measure in the timeline plots.

5.3.2 Analysis and Results

We found that out of the 4430 packages, only for 36 packages the p-value of the predictor variable was more than 0.05 before adjusting for the calender date, and after adjusting for the calender date, p-value was less than 0.5 (*i.e.* the predictor was deemed significant) for all 4430 packages.

The R^2 values of the fitted models varied between 0 and 0.8637 (median: 0.4982, standard deviation: 0.0618) before adjusting for the calender date , and between 0.019 and 0.999 (median: 0.9195, standard deviation: 0.0618) after adjusting for the calender date. So, we can say that the number of daily downloads is a significant and important predictor for the number of issues encountered by the users for most of the packages and the effect is more pronounced when the effect of automated downloads is controlled by calendar date.

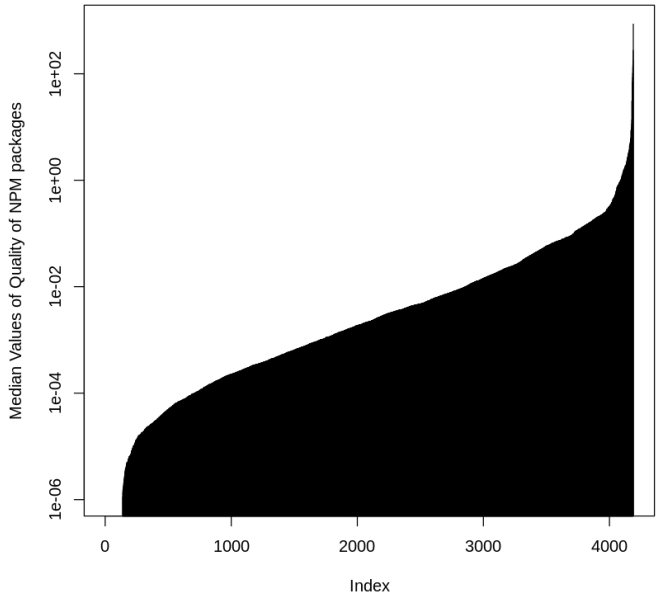

Fig. 18 Histogram of Median Values of Quality of NPM packages

Since we just established that the number of issues of an NPM package depends on the number of daily downloads, a similar quality metric of the number of issues per download should be applicable in this situation as well.

To check the quality of different packages, we looked at the minimum and median value of the quality metric for the 4430 packages. We didn't consider the absolute maximum value, since some packages had zero downloads for a few days, driving the value of the quality metric to infinity. So, we used the 90^{th} quantile value as a proxy for the maximum value. We looked at the packages for which the value of the quality metric was more than 1. The threshold was chosen because we were looking at the packages of really high values of the quality metric, and thus were of poor quality. We found that for 340 packages the 90^{th} quantile value of the quality metric was more than 1, *i.e.* they had more than 1 issue reported against them per download. The number was 100 and 3 when we looked at the median and minimum values respectively. The three packages for which the total number of issues over the number of daily downloads was more than 1 were '@ngrx/store', '@protobufjs/fetch', and '@protobufjs/inquire'. Overall, we found that the packages for which the value of the quality metric was more than 1 were mostly packages from a big project that were relatively less downloaded, *e.g.* 'babel-plugin-transform-es2015-bLOCk-scoped-functions' from babel project, 'react-scripts' from facebook-react project etc. There were a few other packages that had very few downloads during most of its life-cycle since 2015, but had an increase in popularity later on, and thus were selected in our list of packages. However, since they had very few downloads for a long time, the median or maximum value of the quality metric was more than 1 (*e.g.* 'bubleify'). For illustration, in Figure 18 we are showing the histogram of the median value of the quality metric

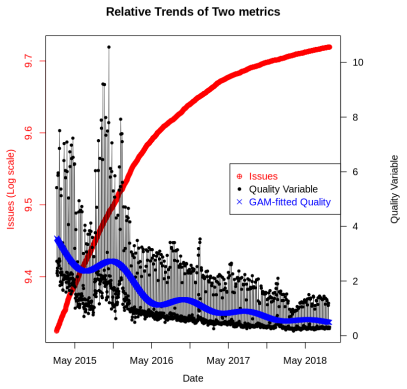

Fig. 19 Timeline for NPM package: angular

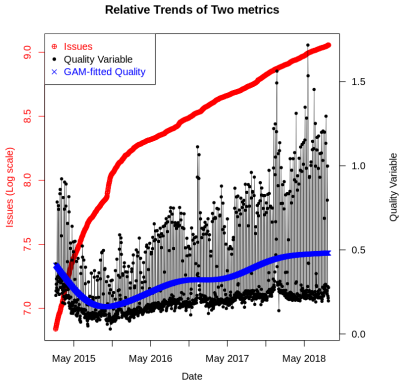

Fig. 20 Timeline for NPM package: babel

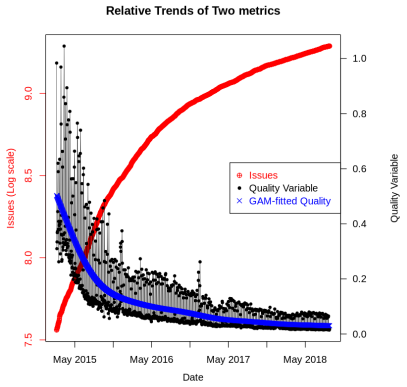

Fig. 21 Timeline for NPM package: eslint

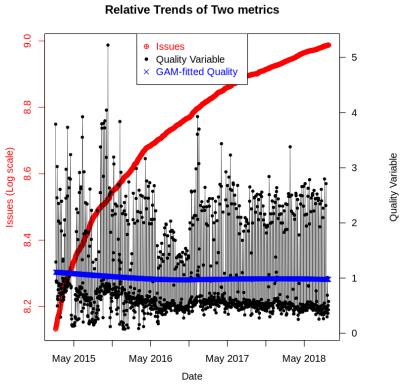

Fig. 22 Timeline for NPM package: ember-cli

for the 4430 NPM packages, which gives some idea about the overall quality distribution of the packages in the NPM ecosystem. We can see that around 75% of the packages in NPM have a median value of the quality metric less than 0.01, which mean, overall, for around 75% of the NPM packages, less than 1 in 100 regular users ever (since we are looking at the total number of issues) file an issue.

Further inspection showed that the value of the quality variable increases with time for almost half of the packages (2030 out of 4430 packages, 45.8%), unlike what we observed for the mobile applications, where for almost all of the releases of the three softwares, the value of the quality variable decreased with time.

Here we also show the timelines comparing the trend of the quality variable we defined (*i.e.* in this case, number of issues per download), along with a fitted line that was fitted using the Generalized Additive Model (GAM), and the number of issues, for a few selected well-known NPM packages for illustration. The selected NPM packages are quite popular and have a large number of issues reported against them, so so plotted the number of issues in log scale. We can see that for all the four cases, the number of issues keep increasing with a decreasing slope, but the

quality measure follows different trends for the four cases. We see that the quality measure for "angular"(Figure 19) and "eslint"(Figure 21) have a trend similar to what we saw for the mobile apps, with the value of the quality variable decreasing with time, but "babel"(Figure 20) is showing an increase in the value, followed by an initial decrease, while for "ember-cli" (Figure 22), the trend is almost constant over time. This result clearly show the necessity of normalizing the number of issues, which is a measure of software faults, by usage parameter like the number of downloads before using it as a measure for software quality.

6 Discussion

Our analysis makes it evident that the number of new users is the most important variable in explaining various post-release variables, as seen in all three of the mobile applications as well as for the NPM packages, where the number of downloads was found to be an important predictor for the number of issues. The analysis also indicates that more new users for a release indicate more exceptions being found for the software and, for the GA releases of both the apps analyzed, longer activity for the release (the duration of a release measures how long a release is actively used by users, not the time between two releases, since the releases overlap). This suggests that users may be reluctant to upgrade (or are encouraged to stay) on better-quality releases. Our findings are in agreement with findings of [45,21,49] that consider post-release defects for a completely different server software system.

The release date also affects no. of exceptions, as can be observed by looking at the coefficients. It provides some insight on how this software has evolved. Even after compensating for the effect the number of users have on the number of exceptions, the number of exceptions are increasing with time for the Android app, whereas is decreases for the iOS app. This may indicate that as the software, the OS, as well as the hardware are becoming more complex with time, which is consistent with a rapid growth of functionality and the size of associated code base. The Android app is seeing more crashes due to the variations in the devices and the OS, whereas for iOS, since the devices as well as the OS versions are tightly controlled, the users are seeing less issues, although we have no explicit evidence to support our speculation.

An interesting observation from the model is the lack of any direct relationship between exceptions and the intensity or frequency of usage. One possibility is that exceptions happen for specific Android/ iOS version/ Phone combination and the way each user is exercising application's functionality. Users for whom the application crashes must wait for the next release. This would lead to the observed phenomena where only the new users increase the number of crashes, which was observed more clearly from the timeline of crashes as well. The duration an application is used by individual users was found to have a much smaller effect on reported defects than the number of new users in prior work [21,45,48] as well. In particular, it was observed that most of the issues happen soon after deploying the release and the chances of reporting a defect for a new release drops very rapidly with time after installation.

Our models also show that some of the variance in exceptions remain unexplained using these predictors. This result is not surprising, since we did not include any internal factors like code complexity which could have an impact on the number of exceptions in our models. However, this does not undermine the

validity our result in the context of the goals of this paper because:
(1) The goal of our study is studying the relationship between exceptions and other **post-deployment** variables, and we wanted to derive a quality measure that is independent of **usage**, not one that is independent of code complexity or any other internal factor. Our proposed quality *metric is an actionable and easy-to-use* measure, and it ensures that a development team can utilize the quality measure to take appropriate action for quality improvement instead of being mislead by the effects of usage.
(2) Any of the code complexity measures is unlikely to be a confounding variable in this scenario, since a confounder is a variable that affects both the independent and dependent variables, and there is no empirical evidence in existing literature suggesting there is a a direct connection between code complexity and the extent of usage. While such a connection is hypothetically possible for open-source projects, since the users might be more willing to work with a simpler code which is easier to tweak, for a closed-source scenario such as the one we presented in this study, such a relationship is practically impossible, since the users never see the source code. The result of the analysis presented in Appendix I supports this hypothesis about the independence of the internal and external(usage) factors. Our study established the link between usage and exceptions, and we derived a usage-independent quality measure, fulfilling the goals of the study.

From the analysis of the NPM packages we observed that the number of downloads is a significant predictor for the number of issues for most of the NPM packages, and when controlled for the calendar date, which compensated for the variations in the downloads by automated sources, it was a significant predictor for all the NPM packages. So, a similar quality measure was used for this case as well. We found by looking into this metric that, overall, for around 75% of the NPM packages, less than 1 in 100 regular users ever (since we are looking at the total number of issues) file an issue. However, unlike the three mobile apps, where the value of our quality metric decreases with time for all releases, for the NPM packages the quality metric sometimes increases or remain relatively constant over time (around 45.8% of the time).

Our data, scripts, and more detailed results are available in our GitHub repository: `https://github.com/tapjdey/release_qual_model`.

> *We found that the exceptions are a result of more new users and the extent of usage does not appear to have a direct effect on the number of new users.*

Overall, none of the three models indicate that "Usage.Frequency" or "Usage.Intensity" have any effect on the "Quality" variable. We, therefore, suggest that the exceptions per new user, or a metric similar to that, can be used as a software development quality metric to objectively compare quality of different releases. The wider practical implication of this finding is twofold.

1. Our findings prove that due to the interdependence of usage and the observed number of software failures (like exceptions), *any* quality measure (like number of defects, defect density, mean time between failures) that is dependent on any of these observed number software failures would misclassify some releases being better or worse than others unless the usage aspect is taken into account.

The effect naturally would be more pronounced for softwares/releases with a large variation in usage.

2. The results of our findings also suggest that these observed number of software failures do not depend on all aspects of usage, *e.g.* we found no dependence between usage intensity or frequency and number of observed exceptions. It suggests that to make a quality measure independent of the external factors like usage, we can not just normalize it by any usage measure, *e.g.* normalizing the number of exceptions by usage intensity or frequency would not make it independent of external factors. It is important to normalize by the right measure to be able to actually make the quality measure independent of usage.

7 Related Work

Although software quality has always been a common topic in software engineering [3,32], most of the studies have focused on pre-release data, primarily due to the developers' concern about finding the appropriate balance between the amount of testing required and the quality of software (*e.g.* [61,10]). There have been a number of works on predicting and improving the software quality as well (*e.g.* [44, 74,28,47]). Comparatively, studies about post-deployment quality and dynamics have been less frequent [35,30]. However, a number of studies have looked at the aspects of software quality metrics, especially the quality perceived by the customers, *e.g.*, [49,45,21,60,43]. A notable non-academic work involves a study of mobile app monitoring company's (Crittercism) data [20]. The author of the news article found it necessary to normalize crash data by the number of launches, similar to the approach we took in our study. Finally, an empirical investigation between release frequency and quality on Mozilla Firefox has been investigated in [31].

While Bayesian Networks have been used for software defect prediction for decades, the use of BNs for explanatory modeling in empirical software engineering is still not common despite the promise. A case for use of BNs was made by Fenton et.al. [17,14], while the earliest publications utilizing BNs we could find [25] constructed search of the structure based on the statistical significance of partial correlations in the context of modeling delays in globally distributed development. [67,56] considered the application of Bayesian networks to prediction of effort, [16,52,53] used Bayesian networks to predict defects, and [54] used BN approach for an empirical analysis of faultiness of a software. On the other hand, Bayesian structure learning is a big domain in itself with a wide range of algorithms, but its use in software engineering context is not very common.

Post-release defect density calculated as a proportion of users who experience an issue within a certain period after installing or upgrading to a new release has been proposed by [46,49] as a measure of software quality. Hackbarth et al. [22] also found the need to adjust defect counts in their proposed measure of software quality as perceived by customers. We propose a somewhat different measure of quality based on the number of exceptions per user. In general, software quality is a widely researched topic [29,32,62] etc., but in our knowledge, this is the first model-based attempt to obtain a usage independent measure of software quality and the first attempt to model exceptions in mobile applications.

The NPM ecosystem is one of the most active and dynamic JavaScript ecosystems and [71] presents its dependency structure and package popularity. [73] studies the dependency, specifically the lag in updating dependencies in various NPM

packages while [1] looked into the use of trivial packages as part of package dependencies for different NPM packages.

The advancements proposed in this paper over the published work are focused on two primary areas: (1) study of the relationship between software faults (issues for NPM packages) and usage using **post-release** data in the context of two proprietary mobile applications and 4430 popular NPM packages, and (2) proposing a usage independent exception-based software quality metric based on our models.

7.1 Comparison with published results

In this subsection we compare our findings with already reported results that studied other commercial applications. The goal of this subsection is not replicating the earlier studies, but comparing the findings of our study and those of some earlier studies. We add this section to address the limitation of our dataset having a relatively small sample of data.

Unfortunately, there aren't a lot of studies that looked into the interrelationship between software usage and software faults (defects or crashes).

Caper Jones [27] provides a number of benchmarks related to how many defects can be expected within one year of a release for products of varying size and varying number of users. According to his industry observations, the number of defects observed within a year after deployment varies very strongly with the number of users, especially for larger systems. The number of users for most of the releases we studied are very small, with a median of 7 users per release, although a few releases have more than 16,000 users. On slide 22 of of his presentation [27], Caper Jones reported that the number of defects increase 2 to 3 times for a 10 fold increase in the number of users (from 1 to 10 and 10 to 100) for a software of similar complexity (between 10,000 and 100,000 function points). However, they were looking at the number of defects, and typically the number of exceptions is larger than the number of defects, because one defect could cause crashes for multiple users (or multiple crashes for a single user).

[21] describe Customer Quality metric that divides customer-reported defect counts within three months of a server system installation by the number customers who install the system. A similar measure is discussed by [45]. The study published in [45] was done for a system with many more users (around 4,000 to 16,000),however, they reported that for a two-fold increase in the number of users the number of Modification Requests (MR tickets) increase around 1.25 times, which is more than what would have been predicted by our model ($1.02 - 1.04$) for Android apps, but less than what we have (1.6) for the iOS app.

In all three cases, the objective is to produce a quality measure that is adjusted for the extent of usage and could be used for benchmarking at least among the releases of a single product. Our proposed measure is similar to these three measures and is designed to work with mobile and web applications (where Google Analytics or similar failure data collection mechanisms are available). Although we were unable to do a direct comparison to another mobile application, due to different studies looking at different measures, these findings add more context to our result, and indicates the necessity of further studies that publish their datasets to understand the usage-fault relationship in a wider range of applications.

8 Limitations

The accuracy of our result is very much dependent on the Google Analytics data. While we do not have reasons to doubt the accuracy of the counts in Google Analytics data, we would have liked to have better definitions of how it determines "New User", "Visit", and, especially, nontrivial to aggregate quantities such as "Visits per User." Also, it is not clear if Google Analytics distorts data in any way (*e.g.*, by applying differential privacy transformations) for low counts in order to protect the privacy of the users. We do not believe it does, but we have not conducted an experiment to validate that.

Furthermore, the projects under consideration were relatively new and it was the first attempt for the team to deploy mobile software. As such, much was not well documented and was rapidly evolving over time. As mentioned earlier, we did not have the official release dates for all releases, so we put the start date of the release as the date on which the first usage was reported. However, we did verify the official dates with this reported date for the releases for which we found the release date, and they were very close, but not always exactly the same. This should not affect the overall result, given the total time scale of more than two years. The release end dates, by their nature, have to be estimated based on user activity, since there is no way to force end user to upgrade Android app. For recent releases, therefore, the end date may be censored by our data collection date, hence the duration for these releases might be underestimated.

Another limitation associated with using these commercial closed-source mobile applications is that we had no control over the release cycle or the variables being measured by Google Analytics. This limited our options for doing the analysis, sometimes severely. We had very few releases for the iOS application, and even the largest dataset of GA releases of the Android application had only 173 releases. We had a limited number of observed variables as well. However, we were unable to obtain any more data on the applications, forcing us to work with the limited data. However, we tried to increase the validity of our study by looking into three sets of releases for two applications, and used three different modeling approaches to study these datasets. The fact that we saw a strong relationship between the number of users and exceptions in all cases has led us to have confidence in the validity of our finding.

It may be possible to collect numerous additional variables that may have an impact on exceptions, for example, the number of changes to the source code made for a release as was done in [45]. Unfortunately, due to the nature of parallel development for multiple releases and products noted in subsection 3.1.1, it was virtually impossible to separate the changes that would only affect a specific release on the Android/iOS platform. To further complicate the matter, the mobile applications we studied were commercial in nature, and the source code for these were not available. However, we were able to find partial commit logs for the source code and the analysis is reported in Appendix I.

Our model is obtained on a single set of mobile applications from a specific domain, implemented via a rather complex codebase and is certainly not representative of most mobile applications that tend to be much simpler. Furthermore, mobile applications may not represent other types of software further limiting external validity of the results. However, some aspects that we see in the specific application, such as increasing number of faults with the number of users, has been

observed in rather different contexts of large-scale server software. This suggests that the model derived in the study may generalize to other domains as well.

In terms of modeling aspects, there are some limitations related to the different approaches. The RF model was used for 10 times 2 fold cross-validation, and exhibited a rather high value of standard deviation in the R^2 value, likely due to the small sample size.

While creating the BN model we did not cover all possible ways BNs can be applied to gain insight into the system. For example, we did not investigate the possible existence of any hidden node, or make an effort to formally establish the causal relationship between the nodes. We also did not investigate how the properties of one release affect the subsequent releases, nor did we investigate the presence of any feedback loops. We did not employ any measures to verify the existence/non-existence of any link that appears in the averaged bootstrapped model either.

In the simulation study, although we covered an extensive set of options, we did not try every possible combination of options for the BN structure search exercise.

We also did not use Markov Random Field analysis, which is another probabilistic graphical modeling approach. The primary reason behind choosing the BN approach was that we found an example where this method was used to successfully recover the underlying network [63]. Moreover, it is possible to interpret a BN model as a causal model, and although we did not use that interpretation in this study, our goal is to eventually establish a causal mechanism of how usage affects the number of exceptions/defects experienced by users, so we wanted to used BN from the start.

The study of the NPM packages was rather limited in scope, since we only looked at the popular (having more than 10,000 monthly downloads since January 2018) packages to reduce the effect of downloads by automated sources. We didn't look at the effects of other factors that affect the number of downloads, *e.g.* the number of dependents a package has. Although some of the issues could come from users of a dependent package, we didn't actively check the origins of the issues to verify that. We also didn't look at the releases of the packages, because of reasons mentioned before. We didn't differentiate between the types of the issues, because we just wanted to see how many times a user decided to file an issue. Overall, this study was not a direct extension of the previous work, rather, it was an extension of the concept and its application in a different domain.

9 Conclusion

From the practical perspective we have established that an external factor like the extent of use has very strong relationship with the observed number of exceptions for three large mobile applications from the telecommunication domain. Counting exceptions, or using any other quality measure dependent on an observed number exceptions, or any other software failure metric (like number of defects or issues) therefore, will not accurately measure the quality of software development process but, instead, it would strongly depend on the extent of use. In order to produce a measure that the development team can use to understand and improve quality of their software development process, we proposed to normalize the observed exceptions by usage, specifically by number of users or any related measure if it is not available. We believe that we demonstrate the importance of adjusting

exceptions for usage to produce a measure that is directly affected by code and process complexity. The specific remediation action (*e.g.* refactoring complex code, doing better code inspections, improving testing) pending on the circumstances can then be taken by the development team. Notably, a similar normalization was previously proposed in the context of post-release defects that also exhibited strong positive correlation with the number of users. As a larger proportion of applications are mobile and/or delivered as a service, the amount of usage can be relatively easily collected. Consequently, not adjusting software development measures for usage should not be considered as an excusable practice.

From theoretical perspective we provided the explanation of the relationships among post-deployment quantities using Linear Regression and Bayesian Networks. Linear Regression can be thought as a special Bayesian Network with the response node being potentially connected to each predictor node. Bayesian Networks allow for exploration of relationships among all variables and empirical determination of the relationships exhibited in a particular dataset. For all three mobile softwares analyzed, the number of users was found to be the most significant predictor with both the models. It would be preferable to have each release as a separate categorical predictor, but because for simplicity we chose to use only one observation per release.

We also established that it is possible to predict exceptions using Random Forest modeling techniques and that usage plays a key role for the accuracy of these predictions. However, the performance of the predictive model was not consistently good, since we did not have any internal factor as a predictor and, as noted above, prediction is a different task than explanation and, even though it often yields more accurate results, the prediction results may be harder to explain to developers or managers and, therefore, harder to act upon. We believe the findings do have a message for the voluminous research in defect prediction. While defects are not exceptions, usage was also found to affect post-release defects in a similar manner [27,21,49]. It would, therefore, be advisable to incorporate forecasts of usage into defect prediction models to increase their accuracy.

Our analysis of the NPM packages established that our approach is extendable to other domains as well. The study revealed that even a less accurate measure of usage like downloads, which, for NPM packages, is a mix of downloads by human users and automated sources, is an important predictor for the number of issues reported, which again is a weakly similar measure to the number of crashes or bugs. So, our approach can be applied to any situation similar to the ones we studied, even when only proxy measures for usage and crashes/ bugs are available.

We hope that this work will spur more research on software engineering aspects in post-deployment stage because, like mobile applications, modern web applications are even more reliant on usage monitoring not simply from the perspective of crash counting but also because the usability or even revenue stream from the software applications critically depends on how users behave.

From the practical perspective, we hope that any mobile or web software project can easily apply and refine the presented approach of using Google Analytics data to improve the quality of their software. Any Android OS or Apple iOS mobile application can use Google Analytics to monitor application usage and crashes, so the approach should be widely applicable. Despite that, we are not aware of any prior empirical study that would leverages Google Analytics or similar data for software quality modeling.

The result of our simulation study should also be useful for practitioners using Bayesian Network structure search techniques for choosing the best performing methods.

Finally, much more work is needed to gather additional empirical evidence of how software behaves post-deployment. It is important to note that Google Analytics data is available only for application developers, so while each project has the ability to see their app's performance, they can not see data for software created by other organizations. This can be addressed by a) projects sharing theirs post-deployment data (we have not seen examples of that); or b) publishing findings based on such data in cases such as ours, where the data itself would be impossible to release publicly since it involves numerous, often enterprise, customers who may not agree.

Acknowledgements This work was supported by the National Science Foundation (U.S.) under Grant No. 1633437.

References

1. Abdalkareem, R., Nourry, O., Wehaibi, S., Mujahid, S., Shihab, E.: Why do developers use trivial packages? an empirical case study on npm. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 385–395. ACM (2017)
2. Balov, N., Salzman, P.: catnet: Categorical Bayesian Network Inference (2016). URL `https://CRAN.R-project.org/package=catnet`. R package version 1.15.0
3. Boehm, B.W., Brown, J.R., Lipow, M.: Quantitative evaluation of software quality. In: Proceedings of the 2nd international conference on Software engineering, pp. 592–605. IEEE Computer Society Press (1976)
4. Borges, H., Hora, A., Valente, M.T.: Understanding the factors that impact the popularity of github repositories. In: Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on, pp. 334–344. IEEE (2016)
5. Bottcher, S.G., Dethlefsen., C.: deal: Learning Bayesian Networks with Mixed Variables (2013). URL `https://CRAN.R-project.org/package=deal`. R package version 1.2-37
6. Briand, L.C., Wüst, J., Daly, J.W., Porter, D.V.: Exploring the relationships between design measures and software quality in object-oriented systems. Journal of systems and software **51**(3), 245–273 (2000)
7. Cataldo, M., Mockus, A., Roberts, J.A., Herbsleb, J.D.: Software dependencies, the structure of work dependencies and their impact on failures. IEEE Transactions on Software Engineering (2009). URL `papers/multicompany.pdf`
8. Chickering, D.M.: Learning bayesian networks is np-complete. Learning from data: Artificial intelligence and statistics V **112**, 121–130 (1996)
9. Chlebus, B.S., Nguyen, S.H.: On finding optimal discretizations for two attributes. In: International Conference on Rough Sets and Current Trends in Computing, pp. 537–544. Springer (1998)
10. Dalal, S.R., Mallows, C.L.: When should one stop testing software? Journal of the American Statistical Association **83**(403), 872–879 (1988)
11. David: (2014). URL `https://developers.slashdot.org/story/17/01/14/0222245/nodejss-npm-is-now-the-largest-package-registry-in-the-world`
12. Dey, T., Mockus, A.: Modeling relationship between post-release faults and usage in mobile software. In: Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering, pp. 56–65. ACM (2018)
13. Duc, A.N., Mockus, A., Hackbarth, R., Palframan, J.: Forking and coordination in multi-platform development: a case study. In: ESEM, pp. 59:1–59:10. Torino, Italy (2014). URL `http://dl.acm.org/authorize?N14215`
14. Fenton, N., Krause, P., Neil, M.: Software measurement: Uncertainty and causal modeling. IEEE software **19**(4), 116–122 (2002)
15. Fenton, N., Neil, M., Marquez, D.: Using bayesian networks to predict software defects and reliability. Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability **222**(4), 701–712 (2008)

16. Fenton, N., Neil, M., Marsh, W., Hearty, P., Marquez, D., Krause, P., Mishra, R.: Predicting software defects in varying development lifecycles using bayesian nets. Information and Software Technology **49**(1), 32–43 (2007)
17. Fenton, N.E., Neil, M.: A critique of software defect prediction models. IEEE Transactions on software engineering **25**(5), 675–689 (1999)
18. Friedman, N., Goldszmidt, M., Wyner, A.: Data analysis with bayesian networks: A bootstrap approach. In: Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence, pp. 196–205. Morgan Kaufmann Publishers Inc. (1999)
19. Garcia, S., Luengo, J., Sáez, J.A., Lopez, V., Herrera, F.: A survey of discretization techniques: Taxonomy and empirical analysis in supervised learning. IEEE Transactions on Knowledge and Data Engineering **25**(4), 734–750 (2013)
20. Geron, T.: Do ios apps crash more than android apps? a data dive (2012). https://www.forbes.com/sites/tomiogeron/2012/02/02/does-ios-crash-more-than-android-a-data-dive
21. Hackbarth, R., Mockus, A., Palframan, J., Sethi, R.: Customer quality improvement of software systems. Software, IEEE **33**(4), 40–45 (2016). URL papers/cqm2.pdf
22. Hackbarth, R., Mockus, A., Palframan, J., Sethi, R.: Improving software quality as customers perceive it. IEEE Software **33**(4), 40–45 (2016)
23. Hahsler, M., Chelluboina, S., Hornik, K., Buchta, C.: The arules r-package ecosystem: Analyzing interesting patterns from large transaction datasets. Journal of Machine Learning Research **12**, 1977–1981 (2011). URL http://jmlr.csail.mit.edu/papers/v12/hahsler11a.html
24. Hartemink, A.J.: Principled computational methods for the validation and discovery of genetic regulatory networks. Ph.D. thesis, Massachusetts Institute of Technology (2001)
25. Herbsleb, J.D., Mockus, A.: An empirical study of speed and communication in globally-distributed software development. IEEE Transactions on Software Engineering **29**(6), 481–494 (2003). URL papers/delay.pdf
26. (https://stats.stackexchange.com/users/71364/repmat), R.: Assumptions to derive ols estimator. Cross Validated. URL https://stats.stackexchange.com/q/149111. URL:https://stats.stackexchange.com/q/149111 (version: 2015-04-30)
27. Jones, C.: Software quality in 2011: A survey of the state of the art. http://sqgne.org/presentations/2011-12/Jones-Sep-2011.pdf (2011). President, Namcook Analytics LLC, www.Namcook.com Email: Capers.Jones3@GMAILcom
28. Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N.: A large-scale empirical study of just-in-time quality assurance. IEEE Transactions on Software Engineering **39**(6), 757–773 (2013). URL http://doi.ieeecomputersociety.org/10.1109/TSE.2012.70
29. Kan, S.H.: Metrics and models in software quality engineering. Addison-Wesley Longman Publishing Co., Inc. (2002)
30. Kenny, G.Q.: Estimating defects in commercial software during operational use. IEEE Transactions on Reliability **42**(1), 107–115 (1993)
31. Khomh, F., Dhaliwal, T., Zou, Y., Adams, B.: Do faster releases improve software quality?: an empirical case study of mozilla firefox. In: Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, pp. 179–188. IEEE Press (2012)
32. Kitchenham, B., Pfleeger, S.L.: Software quality: the elusive target [special issues section]. IEEE software **13**(1), 12–21 (1996)
33. Koller, D., Friedman, N.: Probabilistic graphical models: principles and techniques. MIT press (2009)
34. Kononenko, O., Baysal, O., Guerrouj, L., Cao, Y., Godfrey, M.W.: Investigating code review quality: Do people and participation matter? In: Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on, pp. 111–120. IEEE (2015)
35. Li, P.L., Kivett, R., Zhan, Z., Jeon, S.e., Nagappan, N., Murphy, B., Ko, A.J.: Characterizing the differences between pre-and post-release versions of software. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 716–725. ACM (2011)
36. Marco Scutari: Learning bayesian networks with the bnlearn r package. Journal of Statistical Software **35**(3), 1–22 (2010). URL http://www.jstatsoft.org/v35/i03/
37. McIntosh, S., Kamei, Y., Adams, B., Hassan, A.E.: The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In: Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 192–201. ACM (2014)

38. Mcintosh, S., Kamei, Y., Adams, B., Hassan, A.E.: An empirical study of the impact of modern code review practices on software quality. Empirical Softw. Engg. **21**(5), 2146–2189 (2016). DOI 10.1007/s10664-015-9381-9. URL http://dx.doi.org/10.1007/s10664-015-9381-9
39. Menard, S.W.: Applied logistic regression analysis. 04; e-book. (1995)
40. Meyer, D., Dimitriadou, E., Hornik, K., Weingessel, A., Leisch, F.: e1071: Misc Functions of the Department of Statistics, Probability Theory Group (Formerly: E1071), TU Wien (2017). URL https://CRAN.R-project.org/package=e1071. R package version 1.6-8
41. Mockus, A.: Software support tools and experimental work. In: V. Basili, et al (eds.) Empirical Software Engineering Issues: Critical Assessments and Future Directions, vol. LNCS 4336, pp. 91–99. Springer (2007). URL papers/SSTaEW.pdf
42. Mockus, A.: Law of minor release: More bugs implies better software quality. http://mockus.org/papers/IWPSE13.pdf (2013). International Workshop on Principles of Software Evolution, St Petersburg, Russia, Aug 18-19 2013. Keynote
43. Mockus, A.: Engineering big data solutions. In: ICSE'14 FOSE, pp. 85–99 (2014). URL http://dl.acm.org/authorize?N14216
44. Mockus, A., Hackbarth, R., Palframan, J.: Risky files: An approach to focus quality improvement effort. In: 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 691–694 (2013). URL http://dl.acm.org/authorize?6845890
45. Mockus, A., Weiss, D.: Interval quality: Relating customer-perceived quality to process quality. In: 2008 International Conference on Software Engineering, pp. 733–740. ACM Press, Leipzig, Germany (2008). URL http://dl.acm.org/authorize?063910
46. Mockus, A., Weiss, D.: Interval quality: Relating customer-perceived quality to process quality. In: Proceedings of the 30th international conference on Software engineering, pp. 723–732. ACM (2008)
47. Mockus, A., Weiss, D.M.: Predicting risk of software changes. Bell Labs Technical Journal **5**(2), 169–180 (2000). URL papers/bltj13.pdf
48. Mockus, A., Zhang, P., Li, P.: Drivers for customer perceived software quality. In: ICSE 2005, pp. 225–233. ACM Press, St Louis, Missouri (2005). URL http://dl.acm.org/authorize?860140
49. Mockus, A., Zhang, P., Li, P.L.: Predictors of customer perceived software quality. In: Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on, pp. 225–233. IEEE (2005)
50. (https://stats.stackexchange.com/users/25/harvey motulsky), H.M.: When is r squared negative? Cross Validated. URL https://stats.stackexchange.com/q/12991. URL:https://stats.stackexchange.com/q/12991 (version: 2014-05-06)
51. Nagarajan, R., Scutari, M., Lèbre, S.: Bayesian networks in r. Springer **122**, 125–127 (2013)
52. Neil, M., Fenton, N.: Predicting software quality using bayesian belief networks. In: Proceedings of the 21st Annual Software Engineering Workshop, pp. 217–230. NASA Goddard Space Flight Centre (1996)
53. Okutan, A., Yıldız, O.T.: Software defect prediction using bayesian networks. Empirical Software Engineering **19**(1), 154–181 (2014)
54. Pai, G.J., Dugan, J.B.: Empirical analysis of software fault content and fault proneness using bayesian methods. IEEE Transactions on software Engineering **33**(10), 675–686 (2007)
55. Pearl, J.: Bayesian networks. Department of Statistics, UCLA (2011)
56. Pendharkar, P.C., Subramanian, G.H., Rodger, J.A.: A probabilistic model for predicting software development effort. IEEE Transactions on software engineering **31**(7), 615–624 (2005)
57. Perez, A., Larranaga, P., Inza, I.: Supervised classification with conditional gaussian networks: Increasing the structure complexity from naive bayes. International Journal of Approximate Reasoning **43**(1), 1–25 (2006)
58. R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2017). URL https://www.R-project.org/
59. Rigby, P.C., Bird, C.: Convergent contemporary software peer review practices. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 202–212. ACM (2013)
60. Rotella, P., Chulani, S.: Implementing quality metrics and goals at the corporate level. In: Proceedings of the 8th Working Conference on Mining Software Repositories, pp. 113–122. ACM (2011)

61. Rubin, J., Rinard, M.: The challenges of staying together while moving fast: An exploratory study. In: Proceedings of the 38th International Conference on Software Engineering, pp. 982–993. ACM (2016)
62. Schulmeyer, G.G., McManus, J.I.: Handbook of software quality assurance. Van Nostrand Reinhold Co. (1992)
63. Scutari, M.: Learning bayesian networks in r, an example in systems biology (2013). http://www.bnlearn.com/about/slides/slides-useRconf13.pdf
64. Scutari, M., Strimmer, K.: Introduction to graphical modelling. arXiv preprint arXiv:1005.1036 (2010)
65. Shmueli, G.: To explain or to predict? Statistical science pp. 289–310 (2010)
66. Sober, E.: Instrumentalism, parsimony, and the akaike framework. Philosophy of Science **69**(S3), S112–S123 (2002)
67. Stamelos, I., Angelis, L., Dimou, P., Sakellaris, E.: On the use of bayesian belief networks for the prediction of software productivity. Information and Software Technology **45**(1), 51–60 (2003)
68. Subramanyam, R., Krishnan, M.S.: Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. IEEE Transactions on software engineering **29**(4), 297–310 (2003)
69. Voss, L.: numeric precision matters: how npm download counts work (2014). URL https://blog.npmjs.org/post/92574016600/numeric-precision-matters-how-npm-download-counts
70. Voss, L.: The state of javascript frameworks, 2017 (2018). URL https://www.npmjs.com/npm/state-of-javascript-frameworks-2017-part-1
71. Wittern, E., Suter, P., Rajagopalan, S.: A look at the dynamics of the javascript package ecosystem. In: Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on, pp. 351–361. IEEE (2016)
72. Yu, P., Systa, T., Muller, H.: Predicting fault-proneness using oo metrics. an industrial case study. In: Software Maintenance and Reengineering, 2002. Proceedings. Sixth European Conference on, pp. 99–107. IEEE (2002)
73. Zerouali, A., Constantinou, E., Mens, T., Robles, G., González-Barahona, J.: An empirical analysis of technical lag in npm package dependencies. In: International Conference on Software Reuse, pp. 95–110. Springer (2018)
74. Zhang, F., Mockus, A., Keivanloo, I., Zou, Y.: Towards building a universal defect prediction model with rank transformed predictors. Empirical Software Engineering pp. 1–39 (2015)
75. Zheng, Q., Mockus, A., Zhou, M.: A method to identify and correct problematic software activity data: Exploiting capacity constraints and data redundancies. In: ESEC/FSE'15, pp. 637–648. ACM, Bergamo, Italy (2015). URL http://dl.acm.org/authorize?N14200

APPENDIX I: Exploring the Effect of Source Code Complexity on Exceptions for the Mobile Applications

The Source Code data for the mobile applications

As we mentioned before, the mobile applications under consideration are closed-source commercial products. Therefore, obtaining any information on the source code proved difficult. Finally, we were able to obtain a significant portion of the commit log with informations like what is the file that is updated in the commit, which module a particular commit belongs to, who is the author of the commit, if the commit is related to fixing a reported bug, the timestamp of the commit, and the number of lines of code in the updated file. However, a number of issues associated with the data made using it in a model difficult. The issues included:

– The development team worked on multiple releases simultaneously, and we had no information about which version of which file was part of which release.

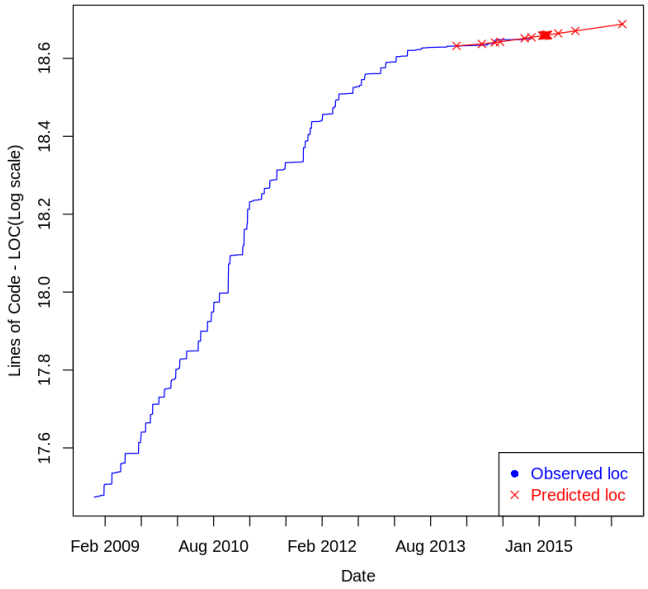

Fig. 23 Observed and predicted values of LOC for Android data

- The commit log might or might not have information on all the files that went into an application, and we had no way of verifying that.
- The date range in the received commit log was from 2009-01-02 to 2014-12-17, but the duration of the releases under consideration in this study was from 2013-09-11 to 2016-05-18. So, the later releases were not covered by the received commit log data.

However, since this is the only data we obtained about the source code, we made the following assumptions:

- We had some idea about which modules belong to which application, so we listed the modules for the mobile SIP for iOS and the Communicator for Android. We assumed all listed files in the modules belong to all the releases of the particular application, and these are the only files in the application.
- Since we had no information about which files went into into which release, we assumed the latest versions of all files in the respective modules until the release date went into that particular release.
- We assumed that number of lines of code (LOC) can be used as a proxy for the traditional complexity measures. This assumption is not unreasonable, since in case of this project these two measures are extremely closely correlated, as was confirmed by one of the authors who had close ties with the development team.

We calculated the LOC for a release as the sum of the number of lines of code of the latest versions of all files present in the relevant modules until the day of the release. As we have a number of missing data points due to the commit log not

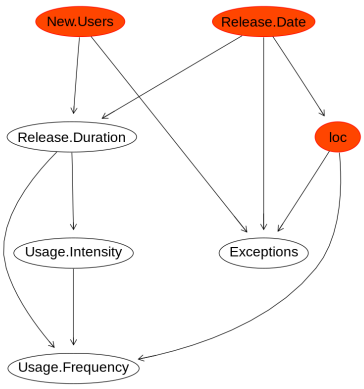

Fig. 24 BN model with LOC measure for GA releases of Avaya Communicator for Android

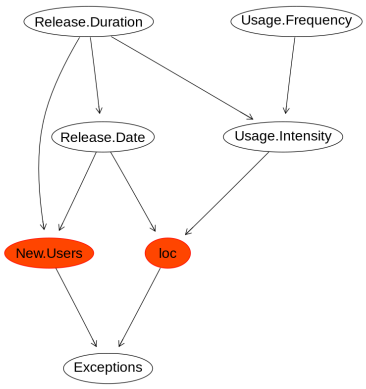

Fig. 25 BN model with LOC measure for development releases of Avaya Communicator for Android

covering the later releases, we decided to fit a model for extrapolating the value of LOC for the later releases. Although the commit log does not cover the later releases, it does cover a span of almost 6 years and the applications have commits on a regular basis. So a model with only the commit date as predictor and the LOC for the particular application as the response variable still works pretty well. We tried a number of different models for extrapolating the value of LOC: linear regression (OLS estimator), Random Forest regression, Support Vector Machine (SVM) regression, and Generalized Additive Model (GAM) regression. The GAM regression produced the best results for extrapolation. The observed and predicted LOC for the Android data against time is shown in Figure 23. The predicted values are shown for the release dates of the GA releases of the Android application. So, in the final dataset, we had the observed values of LOC where we had the data, and the rest of the values were filled with the predicted values of LOC. It is worth mentioning here that we used two different GAM models, one for the Android data, the other for the iOS data. Moreover, since we did not have any way to distinguish between the development and GA releases of the Android application, we used the same mechanism to calculate LOC for both types of releases.

Methodology

We added the LOC variable to our existing dataset for the three mobile applications. Then we fitted the models as before, except, we did not calculate the OLS estimator, since there was a very high correlation between the Release Date and LOC variables. We fitted the BN and RF models to the new dataset to examine the importance of the LOC variable and the fit of the new models.

Results

The BN models for the GA and development releases for Avaya communicator for Android and GA releases of Avaya mobile SIP for iOS are shown in Figures 24, 25, and 26 respectively, with the BN models showing a better fit with the LOC variable added, with R^2 values of 0.90, 0.73, and 0.53 respectively.

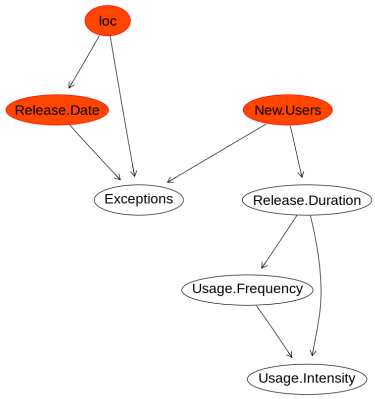

Fig. 26 BN model with LOC measure for GA releases of mobile SIP for iOS

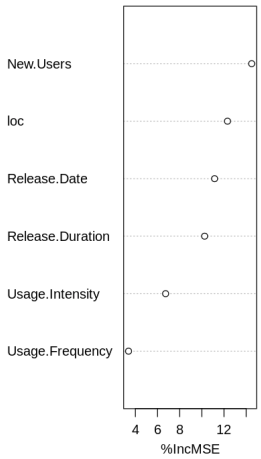

Fig. 27 Variable Importance Plot of RF model for "Exceptions" for GA releases of Avaya Communicator for Android

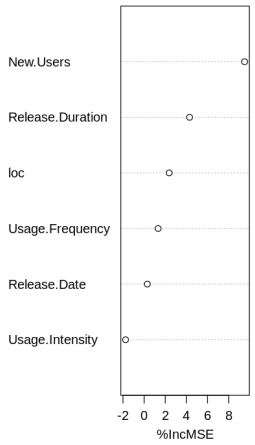

Fig. 28 Variable Importance Plot of RF model for "Exceptions" for development releases of Avaya Communicator for Android

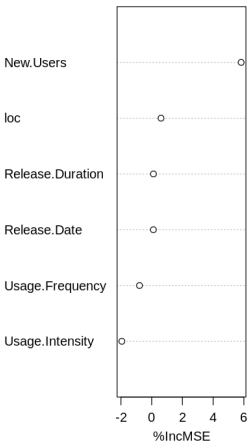

Fig. 29 Variable Importance Plot of RF model for "Exceptions" for GA releases of mobile SIP for iOS

The variable importance plots for the for the GA and development releases for Avaya communicator for Android and GA releases of Avaya mobile SIP for iOS are shown in Figures 27, 28, and 29 respectively. The R^2 values of the RF models also showed improvement, having values of 0.49, 0.61, and 0.35 respectively.

Discussion

The results of the fitted BN and RF models indicate that LOC is an important predictor for Exceptions. For the BN models for all three cases, LOC was directly

connected to Exceptions, even replacing the connection between Release Dates and Exceptions in Figure 25. For the three RF models as well, the LOC variable was shown to be an important predictor.

However, the number of new users still remained the most important predictor, as can be seen from the RF variable importance plots. Moreover, though the fit of the models improved, they did not improve a lot. Although the calculation of the LOC measure involved a number of assumptions, due to not having access to the actual source code of the software, the results show that the code complexity measure has no relationship with the number of new users, which is as we hypothesized in Section 2.

Since the goal of our study was to derive a usage-independent measure of quality, the quality metric we proposed should be independent of the other usage variables, *viz.* usage frequency and intensity. We do not need our quality metric to be independent of code complexity, because what we intend to do is objectively compare the qualities of different releases/softwares regardless of how much usage each of them had. With that view in mind, this piece of analysis is not directly related to our goal, so it was put as an appendix.