Modeling Relationship between Post-Release Faults and Usage in Mobile Software

Tapajit Dey
University of Tennessee, Knoxville
Knoxville, Tennessee
tdey2@vols.utk.edu

Audris Mockus
University of Tennessee, Knoxville
Knoxville, Tennessee
audris@utk.edu

ABSTRACT

Background: The way post-release usage of a software affects the number of faults experienced by users is scarcely explored due to the proprietary nature of such data. The commonly used quality measure of post-release faults may, therefore, reflect usage instead of the quality of the software development process. Aim: To determine how software faults and software use are related in a post-deployment scenario and, based on that, derive post-deployment quality measure that reflects developers' performance more accurately. Method: We analyze Google Analytics data counting daily new users, visits, time-on-site, visits per user, and release start date and duration for 169 releases of a complex communication application for Android OS. We utilize Linear Regression, Bayesian Network, and Random Forest models to explain the interrelationships and to derive release quality measure that is relatively stable with respect to variations in software usage. Results: We found the number of new users and release start date to be the determining factors for the number of exceptions, and found no direct link between the intensity and frequency of software usage and software faults. Furthermore, the relative increase in the number of crashes was found to be stably associated with a power of 1.3 relative increase in the number of new users. Based on the findings we propose a release quality measure: number of crashes per user for a release of the software, which was seen to be independent of any other usage variables, providing us with a usage independent measure of software quality. 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.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; *Search-based software engineering*; • **Computing methodologies →** *Classification and regression trees*; *Bayesian network models*; *Learning linear models*;

KEYWORDS

Software Quality, Software Usage, Software Faults, Bayesian Networks, Linear Regression, Random Forest

ACM Reference Format:
Tapajit Dey and Audris Mockus. 2018. Modeling Relationship between Post-Release Faults and Usage in Mobile Software. In *The 14th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE'18), October 10, 2018, Oulu, Finland.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3273934.3273941

1 INTRODUCTION

Common sense dictates that a software with few defects has better quality than a software with numerous defects. However, the number of discovered software faults increases with the number of users, as observed in [12, 15]. Since software faults are manifestations of underlying defects[1], this, in turn, dramatically affects the interpretation of post-release defects. In the extreme, no users implies zero reported software faults for a release, independent of software quality. This fact is often not considered in industry or in empirical studies (although few studies do note that [7, 9]) and may lead to misguided quality improvement efforts and/or misguided developer performance metrics.

A possible reason for this oversight is the difficulty of tracking usage even though defects and crashes reported by users are often carefully tracked in larger software projects. As a result, little is known of how the extent of software use affects manifestation of software defects in a post-release scenario. This is due in part to the scarcity of rich post-deployment datasets which is likely caused partly by the proprietary nature of such data and the difficulty in obtaining and sharing it even for the software development teams in these proprietary projects since the deployment is typically managed by a different team within the organization. Without such data, however, it becomes exceedingly difficult to interpret customer reported defects since releases with more customers tend to have more defects reported against them despite their higher quality [12, 28]. Moreover, some crucial metrics, for example the number of users for a specific release, are virtually impossible to measure without a usage monitoring system and many traditional software-as-a-product systems do not or can not have such capability.

We conducted our study by looking into 169 releases of a proprietary mobile software from the telecommunication domain for Android OS. The data was retrieved from Google Analytics platform which contained information related to software usage, some release specific information, as well as the number of application crashes. Our **first aim** in this study is to model the relationship among post-deployment variables, specifically, finding the relationships among variables describing different aspects of software usage and software crashes (manifestations of underlying defects).

[1]Although the relationship is not very well understood [9]

Our **second aim** is to use that model to construct a quality measure for the software releases that is independent of software usage, giving us the ability to compare the quality of software releases more accurately.

We used three usage related variables: number of users, usage intensity (average duration of software use per user), and usage frequency (average number of sessions per user), along with two variables describing attributes of the particular release: 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.

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. Then we used a Bayesian Network (BN) model to discover the interrelationship between the variables. Finally, we ran a random forest (RF) model to identify the variable importances for predicting the number of exceptions. All analyses in this study was done in R [39].

We found that the number of new users and the release date (representing the evolution of software development practices and increasing functionality) were the most significant factors needed to explain the exceptions experienced by end users of software. Our findings also suggest that the frequency and intensity of usage have little impact on the number of exceptions.

Based on the result of our model, we propose that a quality metric of average number of exceptions experienced by end users would be most suitable in this scenario. The result of a model with this quality variable was found to be dependent only on the release date, and not on any usage related variables.

The novelty of our work involves the study of the relationship between usage and exceptions and the application of BNs to explain relationships among highly correlated predictors in a mobile development context. We also found that the number of users (representing usage) is the most important predictor of defects, so the software development quality measured as exceptions would be misleading. We replicated the finding in [12, 28] that pointed out that post-release defects are also a misleading measure of process quality and also that the normalization of defects by the number of users provides a more meaningful process quality measure. Furthermore, we hope that our proposed approach of adjusting the defects, exceptions, or other quality indicators for the factors that affect them, but are not relevant to software process quality, could be applied more widely.

In summary, we aim to increase the understanding of software quality in post-deployment scenario in mobile software development by finding the relationship between post-release faults and software usage using different modeling techniques to confront issues related to the observational nature of the data and high correlations among software development and deployment measures. We also propose a usage independent software release quality measure, which is derived from the result of our analysis.

The rest of the paper is organized as follows: In Section 2, we present the background information on the software and the data. In Section 3, we describe the details of the preprocessing steps. In Section 4, we present the different modeling approaches and the results and a comparison of our result and already published result is presented in Section 5. Our proposed quality measure and the results of the corresponding analysis is presented in Section 6 The interpretation of the result is discussed in Section 7. In Section 8, we talk about the possible limitations. In Section 9, we list some related works, and finally in Section 10, we discuss the implications of our study and conclude.

2 BACKGROUND

In this section we provide background information related to the mobile software product and the data obtained from Google Analytics reflecting how the application was used.

2.1 The Software

The software chosen for this study was Avaya Communicator for Android, currently known as Avaya Equinox®. It integrates the Android device 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.

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 [5].

2.2 The Data

The post-deployment data for this application was 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 Android mobile application investigated in this study.

We obtained usage data for Pre-GA (General Availability), Experimental, Development, and GA release versions for the application from the Google Analytics platform. However, since we are interested in modeling the post-deployment quality of the software as experienced by real users, we focus only on the GA release data, because for this software only the GA releases are made available to the end users.

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

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†

version, operating system version, mobile device brand, category, and model.

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 of the application, Android OS version, and type of device, provided sufficiently fine granularity for our analysis.

3 DATA PREPROCESSING

This section contains the data cleaning, transformation, and variable construction steps undertaken prior to applying the different modeling methods on the data.

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.

Aggregating data to per-release granularity: Since our aim is to model the post-deployment 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. We dropped eight of them 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 the first attempt for the team to deploy mobile software on Android OS (although a different team has already produced analogous product for iOS earlier). As such, not everything was well documented and also was rapidly evolving over time and no record of the exact release dates for most of the releases was available. We did manage to get release dates for some of releases from Google Play Store, but not all the release dates were available. For the releases with dates available on Google Play 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 modification of variables: In the final iteration for cleaning the data, we modified the measure for the duration of a release slightly. We did this because although most of the usage for a release is within a certain time period, a few dates of last use were more than a year since previous use, and we didn't want the duration for the release to be influenced by these outliers; so instead of the time between last and first day of usage of a release, we took the difference between the date when 90th quantile of the total number of new users for the release was achieved, and the first date as the effective duration of a release. Our choice was later validated by the first iteration of LR modeling, when we kept the full duration (last-first date) and our chosen measure of duration. The results showed that our chosen measure of duration was more important compared to the full duration. Also, since we found the total number of new users and new visits to be highly correlated (Spearman correlation 0.75), we focus on visits per user instead of the total number of new visits. For the same reason, we normalized the "Time.On.Site" variable to measure time spent on site per user.

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 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

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)

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.

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 distribution, so we took logarithm of them. Final distribution of the variables in available in our GitHub repository:
https://github.com/tapjdey/release_qual_model.

4 ANALYSIS: EXPLAINING EXCEPTIONS

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.

We chose LR for the simplicity, robustness and ease of interpretation. To better understand interrelationships among variable (since LR is not applicable for sets of highly correlated predictors) we used BN models. Finally, to establish the predictive capabilities of our models we used RF, which is known as one of the best Machine Learning classifiers. That way we could both obtain the most insight and also to validate our findings through the use of radically different approaches.

4.1 Linear Regression Model

We first used a linear regression model to discover the significant variables affecting the number of exceptions. The output of the fitted model is shown in Table 3.The model resulted in a decent fit, given the sample size of 169, with adjusted R^2 of 0.435, and the variables "New.Users" and "Release.Date" were the only significant variables, with both coefficients related to the variables being positive, indicating more users result in more exceptions and, for this software, later releases had more exceptions. It can also be seen that the variables representing usage intensity and duration came out to be not significant, indicating absence of a strong relationship between the variables.

Table 3: Summary Result of LR model for "Exceptions"

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-230.5082	71.2493	-3.24	0.0015
Release.Date	23.6401	7.3496	3.22	0.0016
New.Users	0.6556	0.1628	4.03	0.0001
Usage.Intensity	-0.0093	0.0530	-0.17	0.8615
Release.Duration	0.1327	0.1484	0.89	0.3725
Usage.Frequency	0.0903	0.2287	0.39	0.6935

However, due to the presence of medium-high correlation between the variables, the variance inflation factor (VIF) associated with "New.Users" was 5.57 , which, according to [23], is a *"cause for concern"* due to the "tolerance" (inverse of VIF) being less than 0.2. To address this issue, we decided to use a Bayesian Network model, which is unaffected by the multicollinearity problem.

4.2 Bayesian Network Model

Bayesian Network [21, 44] 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[2]. Bayesian Networks models can be useful in the context of Software Engineering research [9] 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 [2, 24, 48, 49], 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 [21].

4.2.1 Bayesian Network Model construction: Despite the promises of BNs, they tend to be quite sensitive to data, and operational data, is often problematic [26, 51]. 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 [3], 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 [43]. 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.

[2]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.

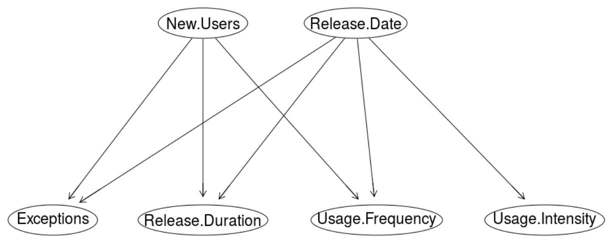

Figure 1: Custom model used for Simulation Study

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:

- *Greedy search algorithms* [33, 43]
- *Hybrid algorithms* [33, 43]
- *Posterior maximization* using *deal* package in R [43].
- *Simulated Annealing* using *catnet* package in R [43] .
- *PC Algorithm* using *pcalg* package in R [43] .
- *MAP (maximum a posteriori estimation) Bayesian Model Averaging* [33, 43]

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 [45, 46]. BIC score is used for discrete data while the Gaussian equivalent of BIC (bic-g) score is used for continuous data.

The results, structure and 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 [10], 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 [10] 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.

Simulation Study:
We performed the simulation study by first creating a random BN (see Figure 1) 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 to generate values for the coefficients for each edge. This model was used in our simulation study going forward. We created 1000 different datasets (it is possible to generate both continuous and discrete data) from the BN structure in Figure 1, and applied the different structure search algorithms (both continuous and discrete versions, where available) listed above. Our performance metric is finding how many times can the different algorithms recover the underlying structure from the simulated data.

The result of the simulation study had the following findings:

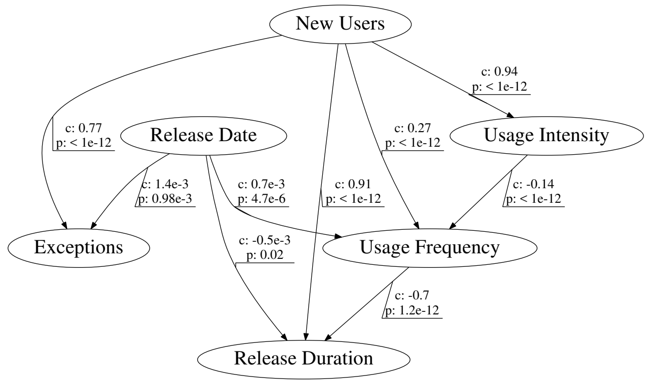

Figure 2: Final BN Model (with c: coefficients after fitting the transformed, but unscaled data, p: p-value for the link)

- Log-transformed and scaled 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 algorithm outperformed all others both in terms of accuracy and runtime, being able to recover the underlying structure 76% of the times and making no more than one error 88% of times.

Final BN 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.

To obtain the final BN model we ran the Bootstrapped Hill-Climbing search 10 times using distinct random seeds on the continuous version of the data, based on the result of our simulation study, and we choose the structure that appeared most frequently. The resultant BN model is shown in Figure 2. It resulted from the bootstrapped greedy hill-climbing search in 9 out of 10 runs. 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 the threshold values found to be optimal for each method from the bootstrap runs (that can be obtained by `attr(boot.object, "threshold")` command).

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 4 we have shown the result from one bootstrap run of the HC method for all the edges we have in our final model. 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 [37], 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

Table 4: Example bootstrap result

from	to	strength	direction
New.Users	Exceptions	1.00	0.94
Release.Date	Exceptions	1.00	0.83
New.Users	Release.Duration	1.00	0.71
Release.Duration	New.Users	1.00	0.29
Exceptions	Release.Date	1.00	0.17
Exceptions	New.Users	1.00	0.06
Usage.Intensity	Usage.Frequency	1.00	0.69
Usage.Frequency	Usage.Intensity	1.00	0.31
New.Users	Usage.Frequency	1.00	0.74
Usage.Frequency	New.Users	1.00	0.26
Release.Date	Release.Duration	0.96	0.70
Release.Duration	Release.Date	0.96	0.30
Release.Date	Usage.Frequency	0.88	0.77
Usage.Frequency	Release.Date	0.88	0.22
New.Users	Usage.Intensity	0.85	0.62
Usage.Intensity	New.Users	0.85	0.38
Usage.Frequency	Release.Duration	0.84	0.53
Release.Duration	Usage.Frequency	0.84	0.47

is a property of the topological ordering of nodes in a DAG, and affects the total probability distribution of the variables.

Based on the results from bootstrap, we chose the model shown in Figure 2 as our final BN model. This model was fitted to the unscaled data, and the resulting coefficient of each link is also shown in the figure. The p-value for each link was calculated 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). Note that, as mentioned before, the Release.Date variable, which is represented by the difference in days between the start date of a release and the Unix epoch time (1970-01-01), was not log-transformed, but the other variables were.

By looking at the p-values for the links, we can say that except the link from "Release.Date" to "Release.Duration", all others are statistically significant. Links having a negative coefficient indicate a negative relationship between the parent and the child node. We also verified that for all the runs that resulted in the model in Figure 2, up to a threshold of 0.35, the model remains the same, and the variables affecting the "Exceptions" variable remain the same for a threshold as low as 0.2.

The performance of explanatory models is evaluated by the fraction of deviance explained by the model. Our model explains 44% of variation in "Exceptions" (adjusted R^2 value of the model).

4.3 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, as shown in Figure 3, indicates that "Release.Date" and "New.Users" are the two most important variables. We ran a 10-fold cross-validation exercise with "Exceptions" as the response variable, the resultant R^2 varied between 0.09 and 0.75 (mean: 0.47, standard deviation: 0.2). The high standard deviation is likely caused

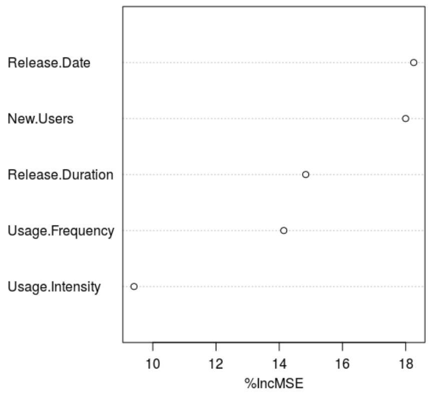

Figure 3: Variable Importance Plot of RF model for "Exceptions"

by our relatively small sample size of 169. This result reinforces the result we got from earlier analyses.

5 COMPARISON WITH PUBLISHED RESULTS

Since our study was based on only one system, we decided to compare our findings with already reported results. 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 [15], 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). The study published in [28] 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 less than what would have been predicted by our model (1.78). Although we were unable to do a direct comparison to another mobile application, 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.

6 APPLICATION OF OUR MODEL: A DERIVED MEASURE OF QUALITY

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 [25]. 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

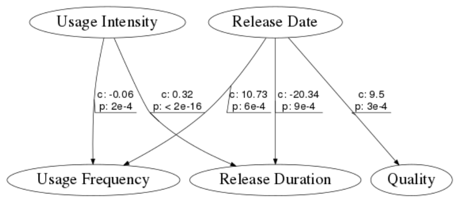

Figure 4: Bayesian Network Model for "Quality" (with c: coefficients after fitting the transformed, but unscaled data, p: p-value for the link)

releases of identical quality. To obtain empirical validation of this postulated mechanistic relationship we rely on our models, all of which show the number of software exceptions to be dependent on the number of users and on software release date. 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 34.5 (mean: 1.103, median: 0, standard deviation: 4.15).

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, depends on "Release.Date". The LR model (Table 5) suggest that "Release.Duration" is also a significant predictor of the quality variable. The BN model(Figure 4), 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-fold cross-validation with the RF model (Variable Importance plot in Figure 5) indicates that the "Release.Date" variable is much more important compared to others, and the two usage related variables are of much lower importance. The R^2 value for the LR model was 0.1271 in this case, and for the 10-fold cross-validation (RF model) it varies between 0.006 and 0.544 (mean: 0.262, standard deviation: 0.174).

Table 5: Summary Result of LR Model for "Quality"

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-136.7113	29.2217	-4.68	0.0000
Release.Date	14.0977	3.0144	4.68	0.0000
Usage.Intensity	-0.0071	0.0205	-0.35	0.7279
Release.Duration	0.0983	0.0367	2.68	0.0080
Usage.Frequency	-0.0594	0.0719	-0.83	0.4102

The results from these analyses clearly indicate that the quality measure defined by the number of exceptions per user is independent of software usage, and, therefore, suitable for comparing the quality of software development process among different releases of a software.

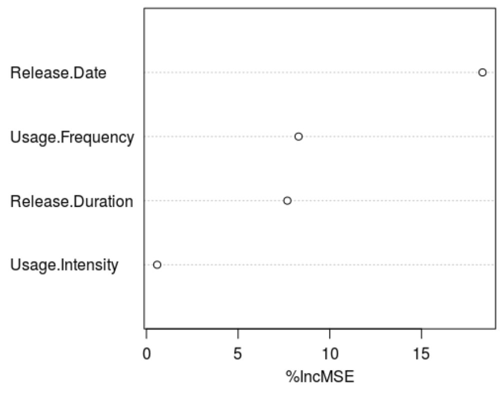

Figure 5: Variable Importance plot from the Random Forest Model for Quality Variable

7 INSIGHTS FROM THE MODEL

The graph in Figure 2 makes it evident that the number of new users is the most important variable in explaining various post-release variables. The p-values for the links suggest that the influence of this variable on the number of exceptions, frequency of usage, and duration of the release are statistically significant. The graph also indicates that more new users for a release indicate more exceptions being found for the software, a higher frequency of visits, more time being spent on site per user *i.e.* higher intensity of usage, and also 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 [12, 28, 32] that consider post-release defects for a completely different server software system. More users leading to a higher frequency and intensity of usage might indicate higher levels of satisfaction with the release.

The release date also affects no. of exceptions, usage frequency, and release duration; although it has much lower influence, as can be observed by looking at the coefficients. It provides some insight on how this software has evolved. The durations of the releases have become shorter, which could indicate that the users are moving to newer releases more rapidly. Even after compensating for the effect the number of users have on the number of exceptions, the number of exceptions are increasing with time. This may indicate that the software is becoming more complex with time, which is consistent with a rapid growth of functionality and the size of associated code base, although we have no explicit evidence to support our speculation. We also notice that the visits per user, which measures usage frequency, increases with time, which could indicate the software is getting more useful.

The negative link between the average time on site (intensity) and visits per user (frequency) can be explained by the nature of this software. As a communicator software it may have a type of users who remain connected for longer durations, and, therefore, have fewer visits (new sessions); while another type of users might just turn it on whenever required, *e.g.* to make a call, and turn it off immediately afterwards. In fact, users who on an average remain connected for longer durations can not have too many visits since the total possible usage time is bounded by calendar time. This may explain this connection and also provide another insight that for the users of this particular software, the total usage time perhaps doesn't vary a lot.

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 OS 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. 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 [12, 28, 31] 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 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. We also found the incidence of exceptions to be increasing over time.

We found that the **quality** metric depends on "Release.Date" (quality seems to decrease for later releases) variable. The LR analysis suggested "Release.Duration" as another significant variable. Indeed, the number of exceptions per new user (or the chances that a user would experience a crash) are low for this software, as indicated by the coefficient 0.77 for the link between users and exceptions (Figure 2). We do not observe this relationship in other models, however. A study on more software products would be needed to establish if the the feedback (better quality leads to longer usage) holds. We saw that the number of exceptions increase for later releases for this software even after accounting for the increase in number of users, so, it appears that this product had lower quality of the development process for later releases. It may have been caused by the move to release more frequently or other factors, but it is something the development team should be aware of and take the most appropriate action. 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 can be used as a software development quality metric to compare quality of different releases and to quantifying the impact of variables representing the development process of a release.

8 LIMITATIONS

In terms of modeling aspects, there are some limitations related to all the different approaches. The LR model results could have been affected by multicollinearity problem, as was indicated by the VIF value. The RF model, on the other hand, was used for 10 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 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 [43]. 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 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 project under consideration was 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.

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 [28]. Unfortunately, due to the nature of parallel development for multiple releases and products noted in subsection 2.1, it was virtually impossible to separate the changes that would only affect the specific release on the Android platform. There might be other unobserved variables driving some relationships, but not explored in this study.

Our model is obtained on a single mobile application 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.

9 RELATED WORK

Although software quality has always been a common topic in software engineering [1, 20], 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. [4, 41]). There have been a number of works on predicting and improving the software quality as well (e.g. [16, 27, 30, 50]). Comparatively, studies about post-deployment quality and dynamics have been less frequent [18, 22]. However, a number of studies have looked at the aspects of software quality metrics, especially the quality perceived by the customers, e.g., [12, 26, 28, 32, 40]. A notable non-academic work involves a study of mobile app monitoring company's (Crittercism) data [11]. The author of the news article found it necessary to normalize crash data by the number of launches. Finally, an empirical investigation between release frequency and quality on Mozilla Firefox has been investigated in [19].

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. [6, 9], while the earliest publications utilizing BNs we could find [14] constructed search of the structure based on the statistical significance of partial correlations in the context of modeling delays in globally distributed development. [38, 47] considered the application of Bayesian networks to prediction of effort, [8, 34, 35] used Bayesian networks to predict defects, and [36] 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 [29, 32] as a measure of software quality. Hackbarth et al. [13] 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 [17, 20, 42] 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 advancements proposed in this paper over the published work are focused on two primary areas: (1) study of the relationship between software faults and usage using **post-release** data in mobile application context, and (2) proposing a usage independent exception-based software quality metric based on our models.

10 CONCLUSIONS

From the practical perspective we have established that the extent of use has very strong relationship with the number of exceptions for a large mobile application. Counting exceptions, 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 exceptions by usage based on the Bayesian Network models. 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 be collected. Consequently, not adjusting software development measures for usage should not be considered as an excusable practice.

From the 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. Both models indicate that there are only two variables that are related to exceptions in this data: number of users and release date which we used as a proxy for the release quality. 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, it could not be done. If, instead, we considered exceptions during different periods of a release, that would have allowed us to introduce such categorical variable and interpret the estimated coefficient for each release as release quality (higher number meaning lower quality).

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. 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 [12, 15, 32]. It would, therefore, be advisable to incorporate forecasts of usage into defect prediction models to increase their accuracy.

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 freely 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.

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.

REFERENCES

[1] Barry W Boehm, John R Brown, and Mlity Lipow. 1976. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press, 592–605.

[2] Lionel C Briand, Jürgen Wüst, John W Daly, and D Victor Porter. 2000. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of systems and software* 51, 3 (2000), 245–273.

[3] David Maxwell Chickering. 1996. Learning Bayesian networks is NP-complete. *Learning from data: Artificial intelligence and statistics V* 112 (1996), 121–130.

[4] Siddharta R Dalal and Collin L Mallows. 1988. When should one stop testing software? *J. Amer. Statist. Assoc.* 83, 403 (1988), 872–879.

[5] Anh Nguyen Duc, Audris Mockus, Randy Hackbarth, and John Palframan. 2014. Forking and coordination in multi-platform development: a case study. In *ESEM*. Torino, Italy, 59:1–59:10. http://dl.acm.org/authorize?N14215

[6] Norman Fenton, Paul Krause, and Martin Neil. 2002. Software measurement: Uncertainty and causal modeling. *IEEE software* 19, 4 (2002), 116–122.

[7] N Fenton, Martin Neil, and D Marquez. 2008. 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 (2008), 701–712.

[8] Norman Fenton, Martin Neil, William Marsh, Peter Hearty, David Marquez, Paul Krause, and Rajat Mishra. 2007. Predicting software defects in varying development lifecycles using Bayesian nets. *Information and Software Technology* 49, 1 (2007), 32–43.

[9] Norman E Fenton and Martin Neil. 1999. A critique of software defect prediction models. *IEEE Transactions on software engineering* 25, 5 (1999), 675–689.

[10] Nir Friedman, Moises Goldszmidt, and Abraham Wyner. 1999. Data analysis with Bayesian networks: A bootstrap approach. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 196–205.

[11] Tomio Geron. 2012. 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.

[12] R. Hackbarth, A. Mockus, J. Palframan, and R. Sethi. 2016. Customer Quality Improvement of Software Systems. *Software, IEEE* 33, 4 (2016), 40–45. papers/cqm2.pdf

[13] Randy Hackbarth, Audris Mockus, John Palframan, and Ravi Sethi. 2016. Improving software quality as customers perceive it. *IEEE Software* 33, 4 (2016), 40–45.

[14] J. D. Herbsleb and A. Mockus. 2003. An Empirical Study of Speed and Communication in Globally-Distributed Software Development. *IEEE Transactions on Software Engineering* 29, 6 (June 2003), 481–494. papers/delay.pdf

[15] Caper Jones. 2011. 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.

[16] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A Large-Scale Empirical Study of Just-In-Time Quality Assurance. *IEEE Transactions on Software Engineering* 39, 6 (2013), 757–773. http://doi.ieeecomputersociety.org/10.1109/TSE.2012.70

[17] Stephen H Kan. 2002. *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc.

[18] Garrison Q Kenny. 1993. Estimating defects in commercial software during operational use. *IEEE Transactions on Reliability* 42, 1 (1993), 107–115.

[19] Foutse Khomh, Tejinder Dhaliwal, Ying Zou, and Bram Adams. 2012. 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*. IEEE Press, 179–188.

[20] Barbara Kitchenham and Shari Lawrence Pfleeger. 1996. Software quality: the elusive target [special issues section]. *IEEE software* 13, 1 (1996), 12–21.

[21] Daphne Koller and Nir Friedman. 2009. *Probabilistic graphical models: principles and techniques*. MIT press.

[22] Paul Luo Li, Ryan Kivett, Zhiyuan Zhan, Sung-eok Jeon, Nachiappan Nagappan, Brendan Murphy, and Andrew J Ko. 2011. Characterizing the differences between pre-and post-release versions of software. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 716–725.

[23] Scott W Menard. 1995. *Applied logistic regression analysis*. Number 04; e-book.

[24] Audris Mockus. 2007. Software Support Tools and Experimental Work. In *Empirical Software Engineering Issues: Critical Assessments and Future Directions*, V Basili and et al (Eds.). Vol. LNCS 4336. Springer, 91–99. papers/SSTaEW.pdf

[25] Audris Mockus. 2013. 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.

[26] Audris Mockus. 2014. Engineering Big Data Solutions. In *ICSE'14 FOSE*. 85–99. http://dl.acm.org/authorize?N14216

[27] Audris Mockus, Randy Hackbarth, and John Palframan. 2013. 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*. 691–694. http://dl.acm.org/authorize?6845890

[28] Audris Mockus and David Weiss. 2008. Interval Quality: Relating Customer-Perceived Quality To Process Quality. In *2008 International Conference on Software Engineering*. ACM Press, Leipzig, Germany, 733–740. http://dl.acm.org/authorize?063910

[29] Audris Mockus and David Weiss. 2008. Interval quality: Relating customer-perceived quality to process quality. In *Proceedings of the 30th international conference on Software engineering*. ACM, 723–732.

[30] Audris Mockus and David M. Weiss. 2000. Predicting Risk of Software Changes. *Bell Labs Technical Journal* 5, 2 (April–June 2000), 169–180. papers/bltj13.pdf

[31] Audris Mockus, Ping Zhang, and Paul Li. 2005. Drivers for Customer Perceived Software Quality. In *ICSE 2005*. ACM Press, St Louis, Missouri, 225–233. http://dl.acm.org/authorize?860140

[32] Audris Mockus, Ping Zhang, and Paul Luo Li. 2005. Predictors of customer perceived software quality. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. IEEE, 225–233.

[33] Radhakrishnan Nagarajan, Marco Scutari, and Sophie Lèbre. 2013. Bayesian networks in R. *Springer* 122 (2013), 125–127.

[34] Martin Neil and Norman Fenton. 1996. Predicting software quality using Bayesian belief networks. In *Proceedings of the 21st Annual Software Engineering Workshop*. NASA Goddard Space Flight Centre, 217–230.

[35] Ahmet Okutan and Olcay Taner Yıldız. 2014. Software defect prediction using Bayesian networks. *Empirical Software Engineering* 19, 1 (2014), 154–181.

[36] Ganesh J Pai and Joanne Bechta Dugan. 2007. Empirical analysis of software fault content and fault proneness using Bayesian methods. *IEEE Transactions on software Engineering* 33, 10 (2007), 675–686.

[37] Judea Pearl. 2011. Bayesian networks. *Department of Statistics, UCLA* (2011).

[38] Parag C Pendharkar, Girish H Subramanian, and James A Rodger. 2005. A probabilistic model for predicting software development effort. *IEEE Transactions on software engineering* 31, 7 (2005), 615–624.

[39] R Core Team. 2017. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. https://www.R-project.org/

[40] Pete Rotella and Sunita Chulani. 2011. Implementing quality metrics and goals at the corporate level. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 113–122.

[41] Julia Rubin and Martin Rinard. 2016. The challenges of staying together while moving fast: An exploratory study. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 982–993.

[42] Gordon Gordon Schulmeyer and James I McManus. 1992. *Handbook of software quality assurance*. Van Nostrand Reinhold Co.

[43] Marco Scutari. 2013. Learning Bayesian Networks in R, an Example in Systems Biology. (2013). http://www.bnlearn.com/about/slides/slides-useRconf13.pdf.

[44] Marco Scutari and Korbinian Strimmer. 2010. Introduction to graphical modelling. *arXiv preprint arXiv:1005.1036* (2010).

[45] Galit Shmueli. 2010. To explain or to predict? *Statistical science* (2010), 289–310.

[46] Elliott Sober. 2002. Instrumentalism, parsimony, and the Akaike framework. *Philosophy of Science* 69, S3 (2002), S112–S123.

[47] Ioannis Stamelos, Lefteris Angelis, Panagiotis Dimou, and Evaggelos Sakellaris. 2003. On the use of Bayesian belief networks for the prediction of software productivity. *Information and Software Technology* 45, 1 (2003), 51–60.

[48] Ramanath Subramanyam and Mayuram S. Krishnan. 2003. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on software engineering* 29, 4 (2003), 297–310.

[49] Ping Yu, Tarja Systa, and Hausi Muller. 2002. Predicting fault-proneness using OO metrics. An industrial case study. In *Software Maintenance and Reengineering, 2002. Proceedings. Sixth European Conference on*. IEEE, 99–107.

[50] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. 2015. Towards building a universal defect prediction model with rank transformed predictors. *Empirical Software Engineering* (2015), 1–39.

[51] Qimu Zheng, Audris Mockus, and Minghui Zhou. 2015. A Method to Identify and Correct Problematic Software Activity Data: Exploiting Capacity Constraints and Data Redundancies. In *ESEC/FSE'15*. ACM, Bergamo, Italy, 637–648. http://dl.acm.org/authorize?N14200