Software Reliability for Agile Testing

van Driel, Willem; Bikker, Jan Willem; Tijink, Matthijs; Di Bucchianico, A

DOI
10.3390/math8050791

Publication date
2020

Document Version
Final published version

Published in
Mathematics

Citation (APA)
van Driel, W., Bikker, J. W., Tijink, M., & Di Bucchianico, A. (2020). Software Reliability for Agile Testing. *Mathematics*, *8*(5), [791]. https://doi.org/10.3390/math8050791

mathematics

MDPI

Article

Software Reliability for Agile Testing

Willem Dirk van Driel [1,*,†], Jan Willem Bikker [2,†] and Matthijs Tijink [2,†] and Alessandro Di Bucchianico [3,†]

[1] Signify and Delft University of Technology, 5656 AE Eindhoven, The Netherlands
[2] CQM (Consultants in Quantitative Methods), 5616 RM Eindhoven, The Netherlands; JanWillem.Bikker@cqm.nl (J.W.B.); Matthijs.Tijink@cqm.nl (M.T.)
[3] Department of Mathematics and Computer Science, Eindhoven University of Technology, 5612 AZ Eindhoven, The Netherlands; a.d.bucchianico@tue.nl
* Correspondence: willem.van.driel@signify.com
† These authors contributed equally to this work.

Received: 24 March 2020; Accepted: 8 May 2020; Published: 14 May 2020

Abstract: It is known that quantitative measures for the reliability of software systems can be derived from software reliability models, and, as such, support the product development process. Over the past four decades, research activities in this area have been performed. As a result, many software reliability models have been proposed. It was shown that, once these models reach a certain level of convergence, it can enable the developer to release the software and stop software testing accordingly. Criteria to determine the optimal testing time include the number of remaining errors, failure rate, reliability requirements, or total system cost. In this paper, we present our results in predicting the reliability of software for agile testing environments. We seek to model this way of working by extending the Jelinski–Moranda model to a "stack" of feature-specific models, assuming that the bugs are labeled with the features they belong to. In order to demonstrate the extended model, two use cases are presented. The questions to be answered in these two cases are: how many software bugs remain in the software and should one decide to stop testing the software?

Keywords: software reliability; agile software testing; Jelinski–Moranda model; Goel–Okumoto model

1. Introduction

Digitization and connectivity of lighting systems has seen an exponentially increasing impact in the last years within the lighting industry [1,2]. The impact is far beyond the impact on single products, but extends to an ever larger amount of connected systems. This is rapidly changing the lighting industry, as more intelligent interfaces with the technical environment and with different kinds of users are being built-in by using more and different kinds of sensors, (wireless) communication, and different kinds of interacting or interfacing devices. Examples are depicted in Figure 1.

The trend towards controlled and connected systems also implies that other components will start playing an equal role in the reliability of such systems. Here, reliability needs to be complimented with availability and other modeling approaches are to be considered [3]. In the lighting industry, there is a strong focus on hardware reliability, including going from component reliability to system reliability; however, in controlled and connected systems, software plays a much more prominent role than in sophisticated "single" products, such as color-adjustable lamps at home, streetlights, UV sterilization lights, and similar products. In these systems, availability is more strongly determined by software reliability than by hardware reliability [3]. In a previous study, the reliability of software was evaluated using the Goel–Okumoto reliability growth model [4]. The results of that first attempt was not satisfactory, as convergence was not met in most cases [4]. It is known that different models

can produce very different answers when assessing software reliability in the future [5]. A significant amount of research has been performed in the area of reliability growth and software reliability, that considers the process of finding (and repairing) bugs in existing software, essentially during a test phase. The following references give a comprehensive overview [6–11]. An important class of software reliability growth models is known as general order statistics (GOS) models [12,13]. The special case in which the order statistics come from an exponential distribution is known as the Jelinski–Moranda model [14]. The main assumption for this class of models is that the times between failures of a software system can be defined as the differences between two consecutive order statistics. It is assumed that the initial number of failures, denoted by a, is unknown but fixed and finite. A typical assumption is that the development of the software has finished, except for the bugs that have to be detected and repaired [5,8,15]. The software reliability models then answer questions such as: what is the number of remaining bugs and how many would we find if we spend a specified number of additional weeks of testing, see [16] for an industrial experience point of view and [17] for a methodological point of view. In a more recent study Rana et al. [18] demonstrated the use of eight different software reliability growth models that were evaluated on eleven large projects. Prior classification of the expected shape was proven to improve the software reliability prediction. In many software development companies, software is developed in a cadence of sprints resulting in biweekly releases in the so-called scaled agile framework (SAFe) [19]. This means there is a second reason why bugs are found, apart from finding them by doing tests, namely, new bugs are introduced because new features are added to the software continuously.

Figure 1. The growing population with increased urbanization results in the need to focus on energy efficiency and sustainability, thereby increasing digitization and rapidly evolving technologies containing software.

In this paper, we introduce attempt at modeling this way of working by extending the Jelinski–Moranda model to a "stack" of feature-specific models, assuming that the bugs are labeled with the feature they belong to. The feature-specific model parameters can be considered as random effects, so that differences between features are modeled as well. In order to demonstrate the extended model, two use cases are presented. Here, we modeled the software testing phase to get a detailed sense of the software maturity. Once software is deemed mature enough by the organization, it is released to the end-users. The new, operational use of the software is different from the testing phase, and this phase was not modeled. The questions to be answered in the two cases are: how many software bugs remain in the software and should one decide to stop testing the software [20,21]? We built upon the mathematical model that describes the number of bugs detected in every time interval (sprint), specified per software feature. We derived a way to evaluate the likelihood function, which is presented in the next section on estimation. We set out with a model with only one feature, which is a variant of the Jelinski–Moranda model, but adapted for the counts per sprint; we needed expressions for conditional probabilities based on recent history, where only the cumulative counts turn out to be important. We extended the results to multiple features, where we shifted the time axis

as different software features are completed at different times. We conclude by describing how all ingredients are combined to the likelihood function. In the next sections, we first introduce in Section 2 the mathematical derivations of our approach. In Section 3 we present the results for two use cases. The paper ends with conclusions in Section 4. Here we also present suggestions for future work.

2. Mathematical Derivations

In this section we state the mathematical foundation of our approach, see Table 1. We start with a subsection in which we state the notation that we use in this paper, and then continue with two subsections on the distribution of bug detection times that we need to derive the likelihood function. In a subsequent subsection we present a Bayesian approach to our feature-specific model.

2.1. Notation

Capital letters are used for random quantities, small letters for constants, or model parameters. We follow the setup as is explained in the GQS setting, see Table 1. A software tool has a bugs, which are detected at time T_i after testing starts at time 0. As usual in industrial practice, these times are not observed or recorded directly. Instead, we only observe number of bugs detected in time intervals. This kind of data is known as grouped data, which is a form of interval censoring. For now, we do not distinguish between different features.

Table 1. Symbols used in the Jelinski–Moranda model for grouped data.

Symbol	Random	Domain	Meaning
a	No	$\mathbb{N}$	Initial number of bugs
b	No	$\mathbb{R}^+$	Rate of bug detection
T_i	Yes	$\mathbb{R}^+$	Time at which bug i is found
L	No	$\mathbb{N}$	Number of intervals
ℓ_i	No	$\mathbb{R}^+$	End time of interval i
N_i	Yes	$\mathbb{N}$	Number of bugs found in the time interval $[\ell_{i-1}, \ell_i]$
C_i	Yes	$\mathbb{N}$	Cumulative number of bugs detected until ℓ_i, $C_i = N_1 + \cdots + N_i$

Throughout, we assume that the T_i's are independent and identically distributed. Throughout, we assume that the T_i's are independent and identically distributed (further denoted by i.i.d). Identically distributed implies that are defects are equally hard to find, even though works has been done to relax this. We realize the impact of this assumption. The Jelinski–Moranda [14] model additionally assumes the T_i's are exponentially distributed with rate b, so all $T_i \sim \text{Exp}(b)$. We explicitly mention when we make the same assumption. In our setting, we assume that we only observe counts N_i in contiguous time periods with start and end points $0 = \ell_0 < \ell_1 < \ell_2 < \cdots$, i.e., $N_i = \sum_{j=1}^{a} \mathbf{1}(\ell_{i-1} < T_j \leq \ell_i)$.

2.2. Multinomial Distribution of Bug Counts

As the times T_j are independent, the joint probability distribution of increments in numbers of bugs over a finite set of observation intervals is given by a multinomial distribution. Consider the event E_{ij} that $\ell_{j-1} \leq T_i \leq \ell_j$. Then for each bug i, exactly one of the mutually exclusive events E_{ij} occurs. Considering that bug i is found at time T_i, the probability that event E_{ij} occurs is given by $F(\ell_j) - F(\ell_{j-1})$, which is the same for each bug i. In view of the independence, we have the settings of a multinomial distribution [22]. We give the full probability distribution in a few different ways that are used later on. Let $m \leq L$ and let T denote a random variable with the same distribution as the T_i's. Then we have

$$\begin{aligned} P(N_1 = k_1, \ldots, N_m = k_m) &= \binom{a}{k_1, \ldots, k_m} \prod_{i=1}^{m} [F(\ell_i) - F(\ell_{i-1})]^{k_i} \, [1 - F(\ell_m)]^{a-(k_1+\cdots+k_m)} \\ &= \binom{a}{k_1, \ldots, k_m} \prod_{i=1}^{m} [P(\ell_{i-1} < T \leq \ell_i)]^{k_i} \, [P(\ell_m < T)]^{a-(k_1+\cdots k_m)} \\ &= \binom{a}{k_1, \ldots, k_m} \prod_{i=1}^{m+1} [P(\ell_{i-1} < T \leq \ell_i)]^{k_i} \end{aligned}$$

where in the last step we set $\ell_{m+1} = \infty, F(\ell_{m+1}) = 1$, and $k_{m+1} = a - (k_1 + \cdots + k_m)$ for notational convenience (see Figure 2). The expression before the product is the multinomial coefficient, a generalization of the binomial coefficient.

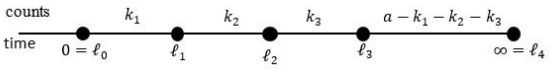

Figure 2. Representation of the time axis.

2.3. Conditioning on History

In what follows, we study the joint probability distribution conditional on the history up to some point ℓ_m. We will need this to combine the different sprint. It turns out it is sufficient to summarize the entire history prior to ℓ_m by the total bug count at this time point. If we assume exponentially distributed bug detection times T_i, it appears that a restarting property holds. At time point ℓ_m, a certain number of bugs remains in the system. The conditional joint probability distribution is again multinomial and corresponds to the above result after proper bookkeeping: shifting the time axis so that ℓ_m corresponds to the new 0, and work with the remaining number of bugs at this time point. This is summarized in the following two results.

In the notation above, suppose the bug detection times T_i are i.i.d. with cumulative probability function F. Writing the definition of conditional probability and cancelling common terms in the numerator and denominator, we obtain that

$$\begin{aligned} &P(N_{m+s} = k_{m+s}, \ldots, N_{m+1} = k_{m+1} \mid N_m = k_m, \ldots, N_1 = k_1) \\ &= P(N_{m+s} = k_{m+s}, \ldots, N_{m+1} = k_{m+1} \mid C_m = k_1 + \cdots + k_m). \end{aligned}$$

Moreover, substitution of our expression for the joint distribution of bug counts, it is easy to see that this conditional probability is equal to

$$\binom{a - k_1 - \cdots - k_m}{k_{m+1}, \ldots, k_{m+s}} \frac{\prod_{i=1}^{s+1} \left(F(\ell_{m+i}) - F(\ell_{m+i-1})\right)^{k_{m+i}}}{\left(1 - F(\ell_m)\right)^{a-(k_1+\cdots k_m)}}$$

where we set $\ell_{m+s+1} = \infty$, $F(\ell_{m+s+1}) = 1$, and $k_{m+s+1} = a - (k_1 + \cdots + k_{m+s})$ for notational convenience.

We will also need the specific form of the likelihood under the additional assumption that T_j follows an exponential distribution. Although we only use it for one sprint forward into time, a more general result holds for s sprints into the future given in the following result. In the notation above, suppose the i.i.d. bug detection times T_i follow an exponential distribution as in the Jelinski–Moranda setting. We see that the joint probability distribution condition on history until time ℓ_m is the same as the multinomial distribution with quantities as if we start over.

$$\begin{aligned} &P(N_{m+s} = k_{m+s}, \ldots, N_{m+1} = k_{m+1} \mid N_m = k_m, \ldots, N_1 = k_1) = \\ &\binom{a - k_1 - \cdots - k_m}{k_{m+1}, \ldots, k_{m+s}} \left[\prod_{i=1}^{s+1} P(\ell_{m+i-1} \leq T \leq \ell_{m+i} \mid T > \ell_m)^{k_{m+i}} \right] \end{aligned}$$

where as before we set $\ell_{m+s+1} = \infty$, $F(\ell_{m+s+1}) = 1$, and $k_{m+s+1} = a - (k_1 + \cdots + k_{m+s})$ for notional convenience. The conditional probabilities are given by:

$$P(\ell_{m+i-1} \leq T \leq \ell_{m+i} \mid T > \ell_m) = e^{-b(\ell_{m+i-1} - \ell_m)} - e^{-b(\ell_{m+i} - \ell_m)}$$

which are the same as the unconditional probabilities if we would shift the time axis so that ℓ_m would lie at zero. The history until ℓ_m only enters the formulas via the sum of counts $k_1 + \cdots + k_m$.

For the implementation later on, we only use the special case $s = 1$, which gives the following result. $P(N_{m+1} = k_{m+1} \mid N_m = k_m, \ldots, N_1 = k_1)$ is given by a formula as in a binomial distribution, where the conditioning on history can be summarized by $C_m = k_1 + \cdots + k_m$. The probability density is equal to:

$$\begin{aligned} P(N_{m+1} = k_{m+1} \mid N_m = k_m, \ldots, N_1 = k_1) &= P(N_{m+1} = k_{m+1} \mid C_m = k_1 + \cdots + k_m) \\ &= \binom{a - k_1 - \cdots - k_m}{k_{m+1}} \left(1 - e^{-b(\ell_{m+1} - \ell_m)}\right)^{k_{m+1}} \cdot \\ &= \left(e^{-b(\ell_{m+1} - \ell_m)} - 1\right)^{a - (k_1 + \cdots + k_{m+1})} \end{aligned}$$

We now set out to show that these binomial densities can be combined in a product over the sprints to form the full joint probability, which is used in the implementation to get an expression for the likelihood. In the setting above, assuming exponentially distributed bug detection times, the joint probability mass function of the counts $(N_1, \ldots, N_m)$ is given by the product over the sprints of the densities of the conditional binomial expressions given above:

$$P(N_1 = k_1, \ldots, N_m = k_m) = \prod_{i=1}^{m} P(N_i = k_i \mid C_{i-1} = k_1 + \cdots k_{i-1})$$

This can be seen as follows. The event $(N_m = k_m, \ldots, N_1 = k_1)$ can be rewritten into its cumulative count version $(C_m = c_m, \ldots, C_1 = c_1)$ with $c_i = c_{i-1} + k_i$ and $c_1 = k_1$. We use the chain rule for probabilities, e.g., for $m = 3$ we have $P(C_3 = c_3, C_2 = c_2, C_1 = c_1) = P(C_3 = 3 \mid C_2 = c_2, C_1 = c_1) \cdot P(C_2 = c_2 \mid C_1 = c_1) \cdot P(C_1 = c_1)$. Here, the history-by-total-count result allows us to simplify the event on which we condition to only the most recent cumulative count: $P(C_3 = c_3 \mid C_2 = c_2, C_1 = c_1) = P(C_3 = c_3 \mid C_2 = c_2)$.

2.4. Extension to Multiple Features

As explained in the introduction, the situation of interest consists not just of one fixed software implementation but of a growing software implementation, where bugs are labeled to which feature they belong. A feature here means a certain functionality in the software. For example, open or close a window in the user interface. Or, in the case of a connected lighting application, dimming the light levels. A new feature in an agile environment may be added every so-called sprint. Here, a sprint means 2–3 weeks of software development. We assume that exactly one feature is added in every time interval between ℓ_i and ℓ_{i+1}, although this assumption is not essential and is merely done to keep notation simple. Ahead, we sketch how the general case of features being finished at arbitrary times can be incorporated. Thus we have features $f = 0, \ldots, L-1$ where each feature f is finished at time point ℓ_f and subject to testing from that point in time onward. We assume that the test intensity for each feature is constant over time, even though new features might be added that could take up testing capacity of the older features. Each feature has a_f bugs in total which is considered fixed for now. The a_f bugs of this feature are found at times $T_{f,1}, \ldots, T_{f,a_f}$ that are shifted exponential distributions that start at time point ℓ_f, so that $T_{f,i} - \ell_f$ is exponentially distribution with parameter b_f. The CDF is $F_{T_{f,i}}(t) = \exp(-b(t - \ell_f))$ for $t \geq \ell_f$ and 0 for $t < \ell_f$. The time points ℓ_i are common to all features and do not get an additional index for f. This way, we arrive at a description of bugs

labeled for different features. Following the above, it is stated for unshifted exponential distributions, although one can imagine a version for shifted (replacing $e^{-b\ell_i}$ with $e^{-b(\ell_i-\ell_f)}$) and keeping proper track of what happens before time ℓ_f (e.g., by defining $0^0 = 1$ in the product). Instead of generalizing these results, we take a different path which is also closer to the software interpretation, by simply shifting the time axis. Note that we assumed that feature f is finished at time point l_f. For the case where new features do not coincide exactly with sprints, say that feature f is finished at time $l_t(f)$, where t is increasing. The shifts are defined accordingly, $l'_m = l_m - l_t(f)$ and $N(f+1)$ is replaced by $N(t(f)+1)$. We assume that bugs can be ascribed to a single feature in this model, and that detection happens independently between the features, even though in some respects one would expect connections between different features, or even bugs on the interface between two features. These are not explicitly covered in the current model.

To set up notation for shifting the time axis, we fix the feature f and drop it from notation of the new symbols, and add an apostrophe to each symbol to indicate it is the shifted time or event. Let $m \geq f$. Define $T'_i = T_{f,i} - \ell_f$, and $\ell'_m = \ell_m - \ell_f$, which describes the shifted time axis. We also need to translate the history event. Define $N'_i = N_{i+f}$, the number of bugs found in interval $[\ell_{i+f-1}, \ell_{i+f}]$, so that N'_1 corresponds to the first interval on our shifted time axis, $[\ell_f, \ell_{f+1}]$. Consider the event $(N'_1 = k'_1, \ldots N'_m = k'_m)$. Rewriting from relative to absolute time gives $(N_{f+1} = k_{f+1}, \ldots N_{f+m} = k_{f+m})$ with $k_{i+f} = k'_i$ to match the indexing. The only possible realizations for $N_1, \ldots, N_{f-1}$ are 0 due to the shifted exponential distribution, so that the event corresponds to $(N_1 = 0, \ldots, N_f = 0, N_{f+1} = k_{f+1}, \ldots N_{f+m} = k_{f+m})$ and has equal probability. We use the conditional binomial restarting result from above and note that the shifted version of the problem satisfies the condition.

$$\begin{aligned} P(N'_{m+1} = k'_{m+1} \mid N'_m = k'_m, \ldots, N'_1 = k'_1) &= \binom{a - k'_1 - \cdots - k'_m}{k'_{m+1}} \left(1 - e^{-b(\ell'_{m+1} - \ell'_m)}\right)^{k'_{m+1}} \cdot \\ &= \left(e^{-b(\ell'_{m+1} - \ell'_m)} - 1\right)^{a - (k'_1 + \cdots + k'_{m+1})} \end{aligned}$$

This results in an expression for the probability density function of the number of bugs found for feature f, conditional on what happened just before. These can then be combined to form the full probability density function on the shifted time axis, which subsequently is the same event as the one with leading 0 counts on the unshifted time axis. We re-introduce the feature index f and note that the above is equal to $P(N_{f,m+1} = k_{f,m+1} \mid C_{fm} = k_{f1} + \cdots + k_{fm})$ where the symbols $N_{f,i}$ and $C_{f,i}$ are the feature-specific counterparts of N_i and C_i. We assume that all bug discovery times $T_{f,i}$ are independent, which implies that the random variables $N_{f,i}$ and $N_{g,j}$ are independent as well for any i, j and features $f \neq g$. We summarize this as follows. In generalizing the setup to multiple features f as described above, at time point ℓ_m, features $0, \ldots, m$ have been added to the software and have observed counts up to that time point. The probability density of the counts up to ℓ_m is found by taking a product over the features f of the feature-specific probabilities. These are given by a product of binomial distributions of time-shifted versions. Viewing the density function as the likelihood function, the parameters are the a_f and b_f for $f = 0, \ldots, m$.

2.5. Bayesian Estimation

It is well-known that the maximum likelihood estimator for a has a positive probability to be equal to infinity (see e.g., [23,24]). We use a Bayesian setup [25–28] to avoid this problem and also because it allows us to combine the bugs originating from different features. We implemented our Bayesian approach in the Stan modeling framework [29] to estimate the software reliability model for multiple features. We do not employ strong priors although that would be possible, e.g., expressing a prior belief of the degree to which added features are similar to each other in total number of bugs a_f or the speed at which bugs are found, b_f. For a feature f, given values for (a_f, b_f), the setup from above is in essence a Jelinski–Moranda model. The values (a_f, b_f) are considered random, unknown parameters, having the same role as random effects in a (non)linear mixed effects model following some distribution. In a

Bayesian context, the a_f, b_f can be considered priors with associated hyperpriors. In our setup, a_f and b_f are modeled as independent truncated normal distributions, where the truncation are at 0 to ensure positive a_f and b_f. Both distributions have a mean and standard deviation parameter, although they are not equal to the expected value and standard deviation due to the truncation. Their posterior distributions give some insight to which extent features are different in size and complexity (in terms of speed of finding bugs). The reading of input data, pre-processing the data, fitting the Stan model, and inspecting convergence and results are done using Python. The Stan website (`mc-stan.org`) states "Stan is a state-of-the-art platform for statistical modeling and high-performance statistical computation." The website offers an extensive amount of documentation and examples. The Stan language requires specification of a model in terms of different concepts which are briefly described below. The model is applied in the situation that we have observed a number of sprints with counts to which we fit the data. The key Stan model components are as follow:

- input data: the $N_{f,i}$, $C_{f,i}$, upper bounds for the indices f and i, and time point at which a feature starts;
- parameters: total bugs remaining, $R_f := a_f - C_{f,m}$, with a lower bound of 0; the b_f; the hyperpriors for the truncated normal distributions of a_f and b_f;
- transformed parameters: a_f is considered a transformed parameter, calculated from a combination of data and a model parameter, $R_f + C_{m,f}$;
- model: distributions for a_f and b_f, hyperpriors for these, and a specification of log likelihood contributions by a double for-loop over features and over sprints, where the feature starting sprints are used.

In this setup, there is a notable construction of R_f that has no direct correspondence to the mathematical setup. The reason is that Stan needs explicit, fixed bounds for the parameters in order for the numerical engine to simulate the posterior density, and the data-dependent bound $a_f \geq C_{f,m}$ is enforced by $R_f \geq 0$. It is still possible to set priors on a_f rather than the actual model parameter R_f, via a line of code inside a loop over the features that corresponds to $R_f + C_{f,m} \quad N(\mu, \sigma)$ where the truncation $a_f \geq$ is specified in the definition of a_f. For ease of use, the statistics is programmed in Python and Stan, which handles the estimation and handling of Bayesian models in a user-friendly tool.

3. Use Cases

In order to explore the abilities of the coding, two cases are used. The first one concerns publicly available software test data [30] and the second one is a real case from the lighting industry [4].

3.1. Case 1: Firefox Data

We used the Mozilla and Eclipse Defect Tracking Data set: a data set with over 200,000 reported bugs extracted from the Eclipse and Mozilla projects (47,000 and 168,000 reported bugs, respectively) [30]. Typically, in software development, features are developed in branches that are later merged with the main branch (call it a main feature). We assume here the one-to-one correspondence between branch and feature. In the case presented, there are 15 main features (or functions). Besides providing a single snapshot of a bug report, this data set also includes all the incremental modifications as performed during the lifetime of the bug report. This data set is bundled as a set of XML files. We extracted reported bugs for a number of 15 branches over a time period staring at January 2003 until May 2013. The total number of reported bugs in this time period is 1793. Figure 3 depicts the fitted response for all individual branches (or features). Figure 4 depicts the same information in a single plot. The priors have been chosen so that they are uninformative for our dataset. For instance, b is the rate of bug detection. The variable std_rate is the standard deviation of the rates between features. One divided by the rate is the time it takes before $1 - \exp(-1) = 0.63$ of all bugs are found. The standard deviation is taken as the positive half of a normal distribution with $mean = 0$ and $stdev = 1$. This is fairly large compared to our average rate. In the output the a_i's are the total

amount of software defects, or bugs for branch/feature nr 3. The b_i's are bug_finding_rate [3]. The a_i's and b_i's are both normally distributed (between features) with means, standard deviation avg_bugs, std_bugs, and avg_rate, std_rate respectively. The mean and a high percentile are interesting here—the 97.5 percentile column indicates that given the current data we believe that the number of additional bugs over so many time periods is with 97.5 percent chance at most the indicated value. The se_mean is the Monte Carlo uncertainty (the fact that in model estimation we used only limited computer time—with more computer time we can get this number down). The 2.5 and 97.5 percentiles indicate the posterior distribution of the given parameter. The n_eff gives the effective sample size from the simulation. Rhat is an indication of quality of the estimation procedure and should be close to 1.0. The estimation may give a warning in case of problems, e.g., if the data severely deviates with the model assumptions in.

See Figure 5 for diagnostic plots of the fit. Table 2 lists the average and standard deviation of predicted remaining bugs in the software. The low numbers are due to the fact that the model prediction for the number of remaining bugs is formally for the case if no further development (other than bug fixing) takes place. As in reality, new features kept being added for this case and the number of bugs increased over time.

Table 2. Predicted number of remaining bugs.

Branch	Average	Stdev
+1	4.5	3.5
+5	12.0	4.0
+10	15.2	3.8

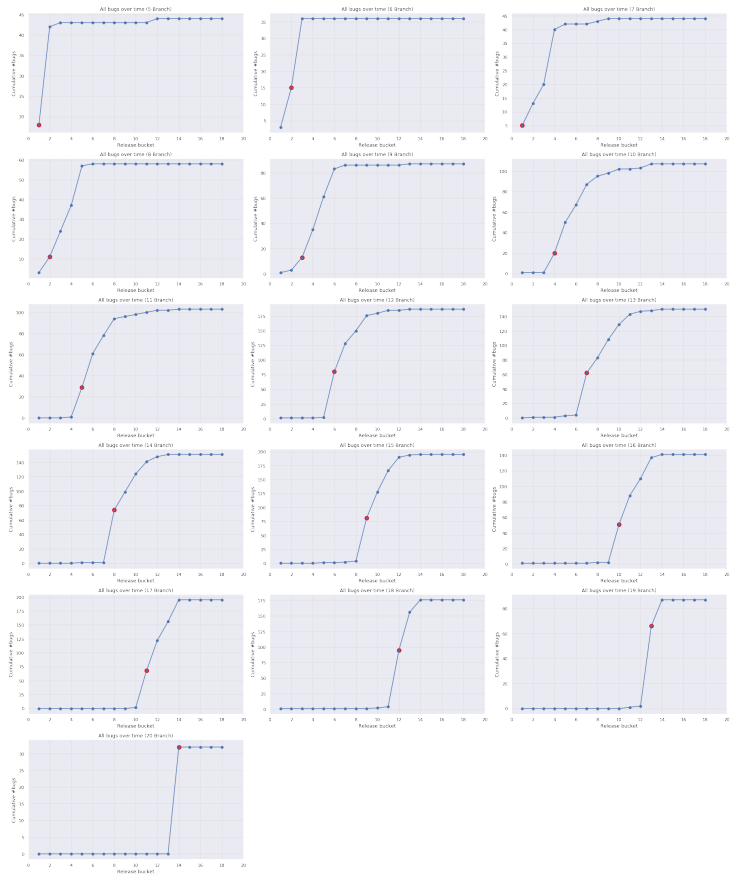

Figure 3. Fit of individual branches or features in the data set, the red dot is the first data point estimated. Here, each branch is considered as a feature which are plotted separately.

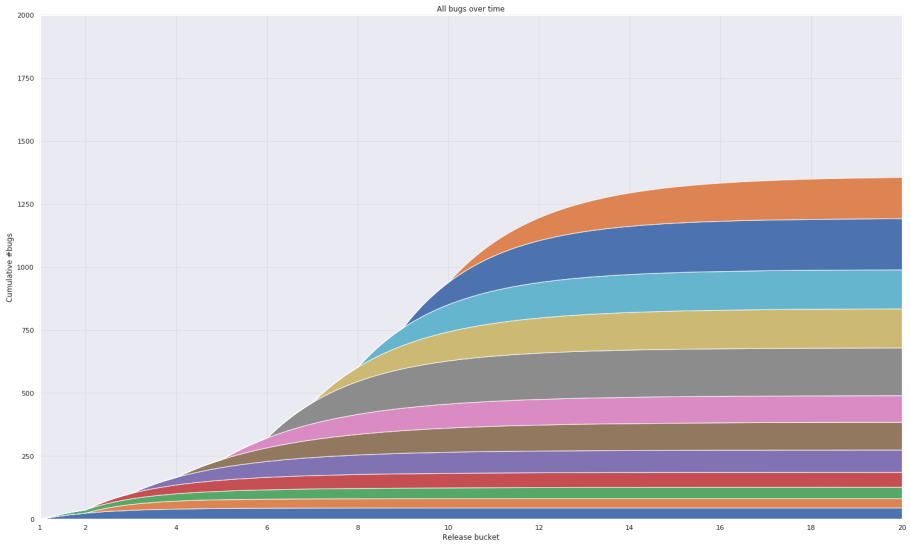

Figure 4. Fitted bugs per branch. Here, each branch is considered as a feature which are visualized with different colors in the fit.

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
a[0]	8.0e-3	4.2e-5	6.0e-3	3.1e-4	3.2e-3	6.8e-3	0.01	0.02	20000	1.0
a[1]	7.9e-3	4.2e-5	6.0e-3	3.1e-4	3.0e-3	6.6e-3	0.01	0.02	20000	1.0
a[2]	41.37	0.05	6.44	29.78	36.92	41.08	45.51	54.92	20000	1.0
a[3]	57.67	0.05	7.73	43.36	52.36	57.32	62.58	73.87	20000	1.0
a[4]	86.48	0.07	9.59	68.74	79.92	86.23	92.86	105.84	20000	1.0
a[5]	107.17	0.07	10.59	87.55	99.54	106.74	114.2	128.94	20000	1.0
a[6]	106.46	0.07	10.48	86.9	99.13	106.26	113.56	127.42	20000	1.0
a[7]	187.6	0.1	13.71	162.01	178.33	187.12	196.61	215.75	20000	1.0
a[8]	157.06	0.3	13.87	129.82	147.78	156.74	165.94	185.53	2175	1.0
a[9]	162.66	0.1	13.62	137.05	153.05	162.26	171.62	190.63	20000	1.0
a[10]	226.67	0.19	19.23	191.99	213.51	225.54	238.61	267.45	10300	1.0
a[11]	152.75	0.18	18.49	120.68	140.14	151.24	163.24	194.11	10518	1.0
a[12]	208.56	0.32	26.64	163.97	190.39	205.98	223.89	267.53	6848	1.0
a[13]	237.41	0.37	34.37	176.81	214.17	235.05	258.34	311.03	8535	1.0
a[14]	150.88	0.75	72.52	18.93	100.7	148.33	195.51	303.23	9424	1.0
a[15]	7.9e-3	4.3e-5	6.1e-3	3.0e-4	3.1e-3	6.6e-3	0.01	0.02	20000	1.0
a_common	144.76	0.24	22.22	101.83	130.39	144.34	158.99	189.52	8643	1.0
a_stddev	72.58	0.28	17.74	46.19	59.93	69.68	82.3	114.36	3950	1.0
b[0]	0.44	7.2e-4	0.07	0.28	0.4	0.44	0.48	0.59	10493	1.0
b[1]	0.44	7.1e-4	0.07	0.28	0.4	0.44	0.48	0.59	11103	1.0
b[2]	0.44	3.6e-4	0.05	0.34	0.41	0.44	0.47	0.55	20000	1.0
b[3]	0.42	4.0e-4	0.05	0.33	0.39	0.42	0.45	0.52	13299	1.0
b[4]	0.42	4.0e-4	0.04	0.33	0.39	0.42	0.45	0.5	11117	1.0
b[5]	0.39	7.6e-4	0.04	0.3	0.36	0.39	0.42	0.47	3173	1.0
b[6]	0.44	3.0e-4	0.04	0.36	0.41	0.44	0.47	0.52	20000	1.0
b[7]	0.51	1.1e-3	0.05	0.43	0.48	0.51	0.55	0.62	2136	1.0
b[8]	0.41	4.4e-4	0.04	0.32	0.38	0.41	0.44	0.49	9566	1.0
b[9]	0.48	6.9e-4	0.05	0.39	0.44	0.47	0.51	0.59	5298	1.0
b[10]	0.42	4.8e-4	0.05	0.32	0.39	0.42	0.45	0.51	9814	1.0
b[11]	0.42	6.0e-4	0.06	0.3	0.39	0.43	0.46	0.54	9937	1.0
b[12]	0.43	6.4e-4	0.06	0.3	0.39	0.43	0.46	0.55	8837	1.0
b[13]	0.47	8.2e-4	0.07	0.35	0.42	0.46	0.5	0.63	7184	1.0
b[14]	0.44	6.4e-4	0.08	0.28	0.4	0.44	0.48	0.59	13794	1.0
b[15]	0.44	7.4e-4	0.08	0.28	0.4	0.44	0.48	0.59	10380	1.0
b_common	0.44	3.4e-4	0.03	0.38	0.42	0.44	0.46	0.5	7047	1.0
b_stddev	0.06	8.9e-4	0.03	0.01	0.04	0.06	0.08	0.13	1174	1.0
asum[0]	99.04	0.07	10.05	80.15	92.26	98.69	105.54	119.78	20000	1.0
asum[1]	185.52	0.1	13.96	159.13	175.97	185.24	194.91	213.44	20000	1.0
asum[2]	292.69	0.12	17.57	259.93	280.77	292.49	304.45	327.84	20000	1.0
asum[3]	399.16	0.15	20.57	359.91	384.71	398.82	413.26	440.18	20000	1.0
asum[4]	586.76	0.17	24.39	540.25	570.25	586.31	602.83	635.77	20000	1.0
asum[5]	743.82	0.2	28.0	690.24	724.9	743.15	762.12	800.37	20000	1.0
asum[6]	906.48	0.22	31.39	846.49	885.43	905.91	927.37	969.3	20000	1.0
asum[7]	1133.1	0.34	37.21	1062.7	1107.3	1132.8	1157.4	1208.4	12198	1.0
asum[8]	1285.9	0.43	42.62	1205.5	1256.3	1284.7	1313.3	1372.7	9798	1.0
asum[9]	1494.4	0.6	52.27	1397.1	1458.4	1492.3	1527.6	1603.1	7651	1.0
asum[10]	1731.8	0.73	66.5	1609.4	1686.8	1727.4	1774.4	1871.8	8282	1.0
asum[11]	1882.7	1.16	102.96	1696.9	1812.2	1876.2	1946.7	2104.0	7892	1.0
lp__	3350.5	0.31	7.95	3335.6	3345.3	3350.1	3355.1	3368.5	651	1.0

Figure 5. Inference and diagnostics for the software reliability prediction model. Diagnostics are explained in the text and follow [31]. Here, a is total number of bugs, b is the bug finding rate and asum relates to the number of future bugs.

3.2. Case 2: Industrial Data

As a real application case, we took a connected lighting product that enables you to harness the Internet of Things to transform your building and save up to 80 percent on energy. LED luminaries with integrated sensors collect anonymous data on lighting performance and how workers use the workplace. This enables you to optimize the lighting, energy uses, cleaning, and space usage to improve efficiency, reducing energy usage, and cost. Workers can use software apps on their smartphones to book meeting rooms, navigate within the office and personalize the environment around their workstation further improving productivity and employee engagement. This smart lighting system with open API integrates seamlessly with the IT system and enables a variety of software applications to create a more intelligent work environment for both building operations managers and employees. In literature, the use of software development schemes and repositories are well described [32–34]. It is demonstrated that detailed problem handling reports will extend the space and quality of statistical analysis for software [34]. It is without doubts that detailed reports will provide significant improvement of software development processes as well as better reliability predictions [32]. In this case, the bug reports may come from different sources (implemented regression tests and tests by the team). Only bugs of sufficient severity are considered in the use case. To handle the various sources we simply took the aggregate counts per sprint as input, assuming that the total number of tests in a sprint was comparable, we get a discrete time axis that was reasonably close to both test effort and calendar time. Ticket data were fed into the code, where we distinguished tickets with severity levels S (high) and A (low). We used JIRA [35] output of bug data for 40 sprints in the period 2017 until 2019. Pick-and-mix was used for ticket severity allocation. In a cadence of two weeks, these tickets either had the allocation open or closed. Open means the issues were being solved, closed means it was solved. Recurring tickets were treated as a new open ticket which can be closed as soon as it is known to be recurrent. Ticket severity is denoted as S, A, B, C, or D. S are issues seen as a blocker that need immediate attention. A is seen as critical, B as major C and D as minor severity levels. In the case presented, we have only analyzed the closed tickets.

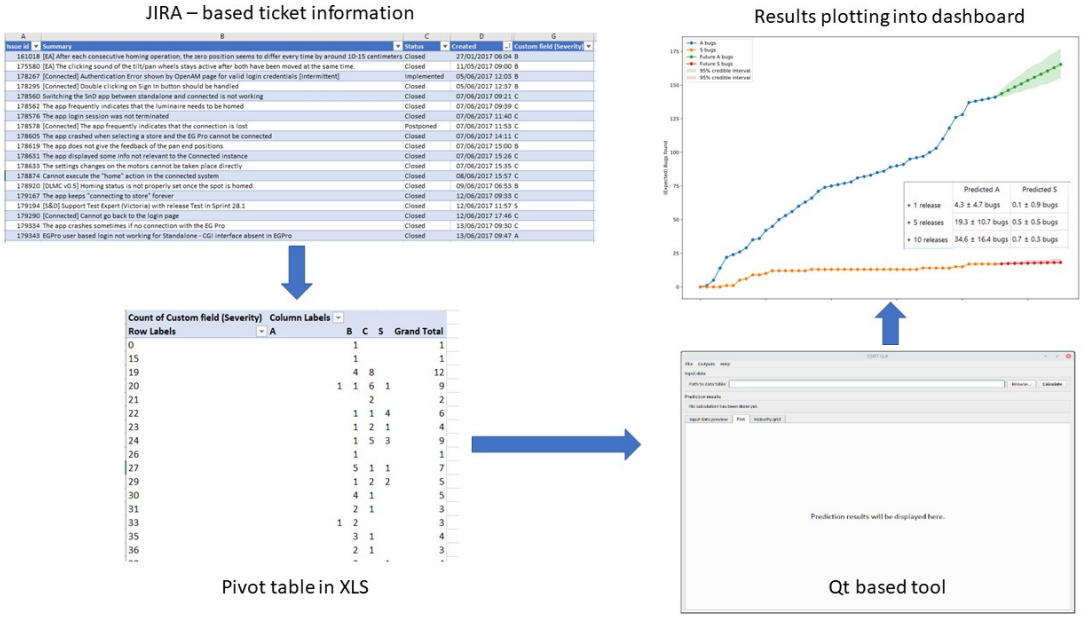

Figure 6. Flowchart for automatic generation of software reliability predictions. Details are explained in the text.

Figure 6 depicts the full flowchart of the process: from tickets to dashboard values. Actual sprint dates have an equal length for each sprint of two weeks. The total number of bugs was 270. The outcome

was produced automatically and shown in Table 3 and Figure 7. The results indicate the predicted remaining number of tickets in the weeks (or sprints) to come, if no new features would be added and only testing is performed. In the figure and table, one clearly notices the maturity of the software.

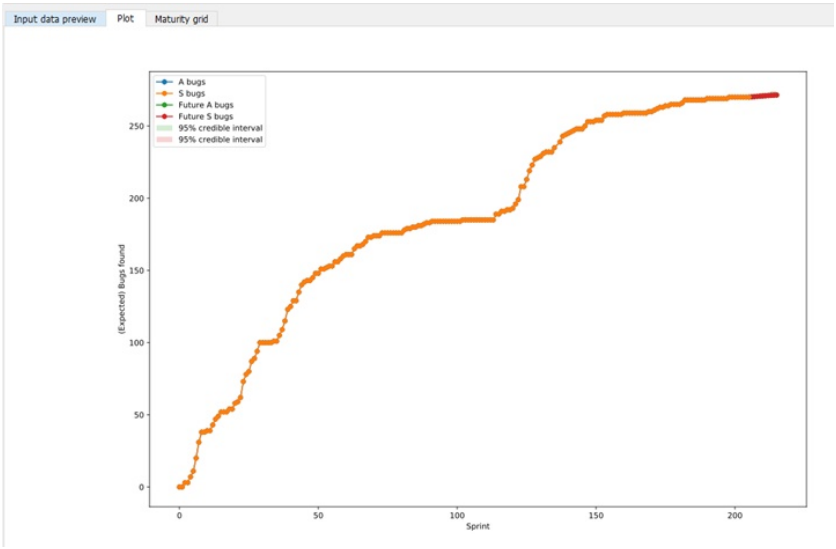

Figure 7. Fit of tickets with S-bugs in the lighting case. Horizontal axes is the time in agile sprints, in this particular case of two weeks, vertical axes shows the total number of bugs with annotation S (high level of severity).

Table 3. Predicted number of remaining bugs in the lighting case. Sprints here refers to two weeks development of the software.

Sprint	Average	Stdev
+1	0.2	0.8
+5	0.8	2.2
+10	1.5	2.6

4. Discussion

Software failures differ significantly from hardware failures. They are not caused by faulty components or wear-out due to e.g., physical environment stresses such as temperature, moisture, and vibration, software failures are caused by latent software defects. These defects were introduced in the software when it was created. However, these defects were not detected and/or removed prior to being released to the customer. In order to prevent these defects being noticed by the customer, a higher level of software reliability has to be achieved. This means to reduce the likelihood that latent defects are present in released software. Unfortunately, even with the most highly skilled software engineers following the best industry practices, the introduction of software defects is inevitable. This is due to the ever-increasing inherent complexities of the software functionality and its execution environment. Here, software reliability engineering may be helpful, a field that relates to testing and modeling of software functionality in a given environment of a particular amount of time. There is currently no method available that can guarantee a totally reliable software. In order to achieve the best possible software, a set of statistical modeling techniques are required that:

- can assess or predict the to-be-achieved reliability;
- based on the observations of software failures during testing and/or operational use.

In order to achieve these two requirements, many software reliability models have been proposed. It was shown that, once these models reach a certain level of convergence, it can enable the developer

to release the software and stop software testing accordingly. Criteria to determine the optimal testing time include the number of remaining errors, failure rate, reliability requirements, or total system cost. Typical questions that need to be addressed are:

- how many errors are still left in the software?
- what is the probability of having no failures in a given time period?
- what is the expected time until the next software failure will occur?
- what is the expected number of total software failures in a given time period?

Certainly, the question of "how many errors are left" is something completely different from "what is the expected number of errors in a given time period". One cannot estimate the first directly, but you can estimate the second. In our approach, we are content with "expected number of errors that a long testing period would yield".

In this paper, we presented an approach to predict software reliability for agile testing environments. The new approach divers from the many others in the sense that it combines features with tickets using Bayesian statistics. By doing that, a more reliable number of predicted tickets (read: software bugs) can be obtained. The advantage of this overall model by applying the JM model per feature is (1) to be able to get prediction intervals for the total number of remaining bugs across all features combined, and (2) "borrowing strength"—we saw in our test cases that a_f and b_f are relatively similar so that for a new feature even after few bugs one knows more than when you analyze features separately, which is the advantage of the overall combined analysis. The developed system software reliability approach was applied to two use cases, Mozilla and Lighting, to demonstrate how software reliability models can be used to improve the quality metrics. The new approach was made into a tool, programmed in Python. The outcome of the predictions can be used in the Quality dashboard maturity grid to enable a better judgment of whether to release the software. The strength of the software reliability approach is to be proven by more data and comparison with field return data. The outcome is satisfactory as a more reliable number of remaining tickets was calculated. As prominent advantage we note that divergence of the proposed fitting procedure is not an issue anymore in the new approach. Following is recommended for the future developments of the presented approach:

- gather more data from the software development teams;
- connect to the field quality community to gather field data of software tickets;
- using the field data determine the validity of the approach;
- make software reliability calculation part of the development process;
- automate the Python code such that ticket-feature data can be imported on-the-fly;
- include machine learning techniques and online failure prediction methods, which can be used to predict if a failure will happen 5 min from now [36];
- investigate the use of other SRGM models, including multistage ones, or those that can distinguish development and maintenance software defects [17,18];
- not focus on a specific software reliability model but rather assess forecast accuracy and then improve forecasts as was demonstrated by Zhao et al. [37];
- classify the expected shape of defect inflow prior to the prediction [18].

Author Contributions: W.D.v.D.: methodology, writing—original draft preparation; J.W.B.: conceptualization, project administration, and investigation; M.T.: conceptualization, writing—review and editing, software, and investigation; A.D.B.: writing—review and editing, supervision, and conceptualization. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Van Driel, W.; Fan, X. *Solid State Lighting Reliability: Components to Systems*; Springer: New York, NY, USA, 2012. [CrossRef]
2. Van Driel, W.; Fan, X.; Zhang, G. *Solid State Lighting Reliability: Components to Systems Part II*; Springer: New York, NY, USA, 2016; doi:10.1007/978-3-319-58175-0. [CrossRef]
3. Papp, Z.; Exarchakos, G. (Eds.) *Runtime Reconfiguration in Networked Embedded Systems—Design and Testing Practice*; Springer: Singapore, 2016. [CrossRef]
4. Van Driel, W.; Schuld, M.; Wijgers, R.; Kooten, W. Software reliability and its interaction with hardware reliability. In Proceedings of the 15th International Conference on Thermal, Mechanical and Multi-Physics Simulation and Experiments in Microelectronics and Microsystems (EuroSimE), Ghent, Belgium, 7–9 April 2014.
5. Abdel-Ghaly, A.A.; Chan, P.Y.; Littlewood, B. Evaluation of competing software reliability predictions. *IEEE Trans. Softw. Eng.* **1986**, *SE-12*, 950–967. [CrossRef]
6. Bendell, A.; Mellor, P. (Eds.) *Software Reliability: State of the Art Report*; Pergamon Infotech Limited: Maidenhead, UK, 1986.
7. Lyu, M. (Ed.) *Handbook of Software Reliability Engineering*; McGraw-Hill and IEEE Computer Society: New York, NY, USA, 1996.
8. Pham, H. (Ed.) *Software Reliability and Testing*; IEEE Computer Society Press: Los Alamitos, CA, USA, 1995.
9. Xie, M. Software reliability models—Past, present and future. In *Recent Advances in Reliability Theory (Bordeaux, 2000)*; Stat. Ind. Technol.; Birkhäuser Boston: Boston, MA, USA, 2000; pp. 325–340.
10. Bishop, P.; Povyakalo, A. Deriving a frequentist conservative confidence bound for probability of failure per demand for systems with different operational and test profiles. *Reliab. Eng. Syst. Safety* **2017**, *158*, 246–253. [CrossRef]
11. Adams, E. Optimizing preventive service of software products. *IBM J. Res. Dev.* **1984**, *28*, 2–14. [CrossRef]
12. Miller, D. Exponential order statistic models of software reliability growth. *IEEE Trans. Softw. Eng.* **1986**, *SE-12*, 12–24. [CrossRef]
13. Joe, H. Statistical Inference for General-Order-Statistics and Nonhomogeneous-Poisson-Process Software Reliability Models. *IEEE Trans. Software Eng.* **1989**, *15*, 1485–1490. [CrossRef]
14. Jelinski, Z.; Moranda, P. Software Reliability Research. In *Statistical Computer Performance Evaluation*; Freiberger, W., Ed.; Academic Press: Cambridge, MA, USA, 1972; pp. 465–497.
15. Xie, M.; Hong, G. Software reliability modeling, estimation and analysis. In *Advances in Reliability*; North-Holland: Amsterdam, The Netherlands, 2001; Volume 20, pp. 707–731.
16. Almering, V.; Van Genuchten, M.; Cloudt, G.; Sonnemans, P. Using Software Reliability Growth Models in Practice. *Softw. IEEE* **2007**, *24*, 82–88. [CrossRef]
17. Pham, H. (Ed.) *System Software Reliability*; Springer: London, UK, 2000. [CrossRef]
18. Rana, R.; Staron, M.; Berger, C.; Hansson, J.; Nilsson, M.; Törner, F.; Meding, W.; Höglund, C. Selecting software reliability growth models and improving their predictive accuracy using historical projects data. *J. Syst. Softw.* **2014**, *98*, 59–78. [CrossRef]
19. Xie, M.; Hong, G.; Wohlin, C. Modeling and analysis of software system reliability. In *Case Studies in Reliability and Maintenance*; Blischke, W., Murthy, D., Eds.; Wiley: New York, NY, USA, 2003; Chapter 10, pp. 233–249.
20. Dalal, S.R.; Mallows, C.L. When should one stop testing software? *J. Amer. Statist. Assoc.* **1988**, *83*, 872–879. [CrossRef]
21. Zacks, S. Sequential procedures in software reliability testing. In *Recent Advances in Life-Testing and Reliability*; CRC: Boca Raton, FL, USA, 1995; pp. 107–126.
22. Johnson, N.; Kemp, A.; Kotz, S. *Univariate Discrete Distributions*; Wiley Series in Probability and Statistics; Wiley: New York, NY, USA, 2005.
23. Blumenthal, S.; Dahiya, R. Estimation of sample size with grouped data. *J. Stat. Plan. Inference* **1995**, *44*, 95–115. [CrossRef]
24. Littlewood, B.; Verrall, J.L. Likelihood function of a debugging model for computer software reliability. *IEEE Trans. Rel.* **1981**, *30*, 145–148. [CrossRef]
25. Bai, C.G. Bayesian network based software reliability prediction with an operational profile. *J. Syst. Softw.* **2005**, *77*, 103–112. [CrossRef]

26. Basu, S.; Ebrahimi, N. Bayesian software reliability models based on martingale processes. *Technometrics* **2003**, *45*, 150–158. [CrossRef]
27. Littlewood, B.; Sofer, A. A Bayesian modification to the Jelinski-Moranda software reliability growth model. *Softw. Eng. J.* **1987**, *2*, 30–41. [CrossRef]
28. Littlewood, B. Stochastic reliability-growth: A model for fault-removal in computer-programs and hardware-designs. *IEEE Trans. Reliab.* **1981**, *30*, 313–320. [CrossRef]
29. Team, T.S.D. Stan Python Code. 2018. Available online: https://mc-stan.org/ (accessed on 15 November 2018).
30. Lamkanfi, A.; Perez, J.; Demeyer, S. The Eclipse and Mozilla Defect Tracking Dataset: a Genuine Dataset for Mining Bug Information. In Proceedings of the MSR '13 10th Working Conference on Mining Software Repositories, San Francisco, CA, USA, 18–19 May 2013.
31. Depaoli, S., V.d.S.R. Improving Transparency and Replication in Bayesian Statistics: The WAMBS-Checklist. *Psychol. Methods* **2017**, *22*, 240–261. [CrossRef] [PubMed]
32. Janczarek, P.; Sosnowski, J. Investigating software testing and maintenance reports: case study. *Inf. Softw. Technol.* **2015**, *58*, 272–288. [CrossRef]
33. Rathore, S.; Kumar, S. An empirical study of some software fault prediction techniques for the number of faults prediction. *Soft Comput.* **2017**, *21*, 7417–7434. [CrossRef]
34. Sosnowski, J.; Dobrzyński, B.; Janczarek, P. Analysing problem handling schemes in software projects. *Inf. Softw. Technol.* **2017**, *91*, 56–71. [CrossRef]
35. Atlassian. JIRA Software Description. 2020. Available online: https://www.atlassian.com/software/jira (accessed on 12 May 2020).
36. Salfner, F., L.M..M.M. A survey of online failure prediction methods. *ACM Comput. Surv.* **2010**, *42*, 12–24. [CrossRef]
37. Zhao, X.; Robu, V.; Flynn, D.; Salako, K.; Strigini, L. Assessing the Safety and Reliability of Autonomous Vehicles from Road Testing. In Proceedings of the 30th International Symposium on Software Reliability Engineering (ISSRE), Berlin, Germany, 28 October–1 November 2019.