2nd International Conference on Advanced Research Methods and Analytics (CARMA2018)
Universitat Politècnica de València, València, 2018
DOI: http://dx.doi.org/10.4995/CARMA2018.2018.8331

Fishing for Errors in an Ocean Rather than a Pond

Wilson, John [a] and Te'eni, Dov [b]
[a]Ivey Business School, University of Western Ontario, London N6G ON1, Canada, [b]Coller School of Management, Tel Aviv University, Israel.

Abstract

In the internet age, a proliferation of services appear on the web. Errors in using the internet service or app are dynamically introduced as new devices/interfaces/software are produced and are found to be incompatible with an app that is perfectly good for other devices. The number of users who can detect various errors changes dynamically: for instance, there may be new adopters of the software over time. It may also happen that an old user might upgrade and thus run into new incompatibility errors. Allowing new users and errors to enter dynamically poses considerable modeling and estimation difficulties. In the era of Big Data, methods for dynamically updating as new observations arise are important. Traditional models for detecting errors have generally assumed a finite number of errors. We provide a general model that allows for a procedure for finding maximum likelihood estimators of key parameters where the number of errors and the number of users can change.

Keywords: *Errors in software apps; Big Data; reliability; software development and testing.*

1. Introduction

The internet and mobile have changed the way that software is distributed and used. Cloud computing, open source software and continuous connectivity, in particular, allow for the operation and connection of many services, many applications, many devices and many users. The advent of the Internet of Things will magnify the criticality of securing uninterrupted operation and connectivity. As early as the 1990s, there was a recognized need for research in the economics of software development and maintenance (e.g. Banker et al. 1998 and Chan et al.1994). Issues such as the timing of software releases, the development and management of interoperability, and the allocation of resources to testing are critical to management. Once software is released, it is important to have models that track errors as software is being used. Real-time detection of errors encountered by users is often the norm. There is a need, therefore, for research that builds on the new realities of the software industry and that leads to practical tools.

We concentrate on the probability of errors in software, which must be a parameter of any managerial model and can directly affect managerial decisions such as when to stop software testing and how to it. We are interested in the behavior of errors over time because software management is dynamic.

Finding errors is like fishing and open computer systems are analogous to a pond linked to the ocean. Our proposed model describes fishing in an ocean rather than the pond of previous models. In a pond, the rate of catching fish depends on how many are left in it. When the pond is opened to an ocean, waves bring in new fish and will not find all the fish when confronted with a practically infinite stream of fish coming in from the ocean. The potential number of failures due to communication software, printers, operating systems and I/O devices is practically infinite. In the new context of open systems, we distinguish between *errors of content* and *errors of incompatibility*. The former are code contained in the system that is incorrect with respect to its specification (e.g., an incorrect loop or a select construct that does not cover all required cases). The latter are code that is incompatible with conditions external to the specified system, e.g., problems in working parallel to new versions of other packages.

In this paper, our focus is the interaction between the system and its environment: hence the notion of incompatibilities. This shift has already occurred in industry. For example, Mercury Interactive, a software testing company, realized back in 1998 that current software, unlike the past, cannot be contained in a single system (see Forbes, 1998). A similar shift is needed in the operations research modeling of error detection, for instance by extending extant models to include different patterns of error behavior and detection that break the assumptions of instant removal of detected errors and of no new sources of errors (e.g., Gaudoin, 1999; Yang et al., 2016). Another example is the distinct behavior of performance errors that occur after release (Zaman et al., 2012).

A *failure* is an unexpected result of a program execution. Failures are the external phenomena that the user experiences. An *error* (or fault) is incorrect code that, under certain conditions, will produce a failure. Errors are hidden from the user but are perceived as the cause of failures. Therefore the more failures experienced, the more errors assumed to exist in the software. A failure is related to a specific error, called a *detected error*. Several failures may be related to the same error, in which case there would only be one detected error. In reliability growth models, for instance, the number of errors is supposed fixed at the time a prototype is produced and the goal is to systematically eliminate them. (See, e.g., Heydari and Sullivan, K. M. 2017).

Open systems are related one to another. An error may be the result of a combination of conditions in two distinct applications that is incompatible with the code. Errors of incompatibility are practically endless and cannot be determined as a function of the code alone. This is different to traditional models.

We develop a general framework for modelling errors that are continually created. A probabilistic model is formulated that can lead to the development of the likelihood function from which estimates can be derived. This is a complex process since at any period an error may have been introduced at any previous time. In addition, only some users can detect a new error since they are the only ones to have upgraded and thus only they can be exposed to a current error of incompatibility.

2. Model Development

The intuitive discussion is now formulated mathematically. Certain practices may blur some of the theoretical distinctions made above. For instance, a Beta site may fall between testing and production and actual reports of failures may entangle the two types of errors. The mathematical formulation ignores such difficulties and assumes, for simplicity, some additional constraints as discussed below. Moreover, one general functional form is built, which will describe both pre and post-release stages. For clarity of presentation, we use the term users to denote both units of testing and units of use, although the former relates to the pre-release stage and the latter to the post-release stage.

There are three aspects to modelling this problem: (1) The process whereby customers, internet users, arrive to use the app and perhaps cancel their subscriptions at a later time; (2) The modelling of who upgrades their software/equipment and thus may encounter new errors of incompatibility; (3) The error detection process which involves modelling the arrival of new errors into the system and the number of users who can detect them.

The description will be given for a general continuous time process. (Unfortunately, in order to capture the real time complexity of the various processes, a lot of notation and definitions are involved.) Then, the simpler case of discrete time periods will be considered.

2.1. The Arrival and Cancellation Processes

Suppose the $X(t)$ denotes the number of new customers who use the software at time t. For instance, $X(t)$ could represent the number of people who sign up at time t for an online transcription service such as that provided by *Nuance*. Over time, some subscribers will cancel and no longer use the service. Let $C(t)$ denote the time a customer who signed up at time t cancels the service. (A very large value for $C(t)$ means that the customer never cancels.)

2.2. The Upgrade Process

For an individual completely current at time t (i.e. someone who is "new" at time t or who has been using the system before time and upgrades at time t), let $U(t)$ denote the time this customer next upgrades. (A very large value for $U(t)$ means that the customer never upgrades.)

Let $Z(s,t)$ denote the number of people who were current at time s, did not upgrade between times s and t, but did upgrade at time t. The quantity $Z(s,t)$ is a function of $X(u)$, $C(u)$ and $U(u)$ for $u \leq t$.

2.3. The Error and Detection Processes

At time t, let $Y(t)$ denote the number of new errors that are introduced into the system. For instance, a new version of the iPad might be introduced at time t. A subscriber using the software with this device might ultimately encounter an incompatibility error: the software works perfectly well but there is an as yet undiscovered error when the new device is used. Prior to the introduction of the new iPad, this error did not exist.

Let $D(t)$ denote the number of users who can detect *new* errors introduced at time t. ($D(t)$ will depend on the variables $X(t)$, $C(t)$ and $U(t)$) If one assumes that users who adopted prior to time t cannot detect errors introduced at time t, then $D(t)$ is simply equal to $X(t)$, the number of new users introduced at time t. This can be a reasonable assumption if one assumes that most users will not upgrade to new technology until a fair amount of time has elapsed. However, it is not necessary to make this assumption: any upgrade pattern can be accommodated.

For an error introduced at time s, let $P(s,t)$ denote the probability that a user current at time s detects an error during period number $t+1$ given that the user has not detected it prior to this period.

2.4. Analysis for Discrete Time Periods

In this paper, we will concentrate on the special but useful case where tracking is done over discrete periods. This is often the most realistic way to proceed and gives some useful practical and theoretical results. In order to make this clear the notations $X(t)$, $C(t)$, $Y(t)$, $U(t)$) and $D(t)$ will be replaced, respectively, by X_i, C_i, Y_i, U_i and D_i, where $i = 0,1,2,3$ ... denotes the period ($i = 0$ correponds to release of the app, $i = 1$ corresponds to the end of the first period, etc. The quantity $Z(s,t)$ will be replaced by and $Z_{i,j}$ where i and $j$with $i < j$ are period numbers.

2.5. Example: Discrete Time Periods

Assume that no one cancels a subscription. Suppose that, at the beginning of any time period, a user who is current in the prior period will upgrade with probability 0.1 (i.e. is an "early adopter") , a customer who was last current two periods ago will upgrade with probability .15, those last current three periods ago will upgrade with probability 0.3 and those current more than three four periods ago will definitely upgrade. Then the probability distribution for $\mathrm{U_t}$, the time at which a customer current at time t will upgrade is given by:

$$U_t = \begin{cases} t+1 \text{ with probability } 0.1 \\ t+2 \text{ with probability } (0.9)(0.15) = 0.135 \\ t+3 \text{ with probability } (0.9)(0.85)(0.3) = 0.2295 \\ t+4 \text{ with probability } (0.9)(0.85)(0.7) = 0.5355 \end{cases}$$

Let $B(n,p)$ denote the value of a Binomial random variable with parameters n and p. The number who can detect errors at time 0 is D_0, the initial number of subscribers. At the end of period 1, the number of people who can detect new errors at time 1 equals the number of new subscribers $X(1)$ plus the number who have upgraded from form time - $B(D_0 0.1)$, i.e

$D_1 = X_1 + B(D_0, 0.1)$.

Using a similar argument, the number of subscribers who can detect errors at any time i can be found. For instance, the values of $D(2)$ and $D(3)$ are as follows:

$D(2) = X_2 + B(D_1, 0.1) + B(D_0, (0.15)(1-0.1))$

$D_3 = X_3 + B(D_2, 0.1) + B\big(D_1, (0.15)(1-0.1)\big) + B(D_0, (0.3)(1-0.15)(1-0.1))$.

Values for $Z_{i,j}$ can also be found. $Z_{i,i+1}$ is the number of customers who were current at time i and upgrade at time $i+1$ and thus equals $B(D(D_i, 0.1)$. $Z_{i,i+2}$ is the number of people current at time i who do not upgrade at time $i+1$ but do upgrade at time $i+2$ and thus equals $B(D_i, (0.15)(1-0.1))$. Similarly, $Z_{i,i+3} = B(D_i, (0.3)(1-0.15)(1-0.1))$ and $Z_{i,i+4} = B(D_i, (1-0.3)(1-0.15)(1-0.1))$.

3. Discrete Time Periods and Constant Error Detection Probability

In this section, the problem will be simplified. We assume that $P(s,t) \equiv p$ and that $Y(t) \equiv \mu$. Ultimately, The goal is to estimate the quantities p, μ and $\upsilon \equiv X_0$ or, equivalently, D_0. (In the case of a subscription service X_0 is known but in other cases-for instance "free" software, the number of initial users may not be known.)

Consider a particular user who has the potential to discover a particular error. Then p denotes the probability that this user discovers this error during any given period. All users and all errors are assumed to be independent. (In a more general setting non-indepence may be allowed. For instance, Y_i, the number of errors introduced at time i, could have a distribution where the number if errors introduced in a given period depends on those introduced in a prior period. However, here we will focus on the simpler case of indepence which is difficult in its own right.) The p, however, may have interpretations that depend on the context: one p may be used for errors of content while a different p might be used for errors of incompatibility. Let $p(k,i)$ denote the probability that an error introduced at time k is detected for the first time during period i. This quantity can be shown to satisfy the following (proof omitted):

$p(k,i) = (1-p)^{N(k,i)}[1-(1-p)^{M(k,i)}]$,

where $N(k,i) \equiv (i-k-1)D_k + \sum_{j=k+1}^{i-2}(i-j-1)(X_j + Z_{k-1,j})$ and $M(k,i) \equiv D_k + \sum_{j=k+1}^{i-1}(X_j + Z_{k-1,j})$.

This quantity is key to writing down the likelihood. Suppose one has collected the data $x_1, \ldots, x_i$—the numbers of errors observed during each of the first i periods. Then the goal is to find the values of p, μ and υ that maximize the probability of observing this data stream. For given values of the parameters, it is necessary to construct an expression for $L(x_1, \ldots, x_i)$, the probability of observing $x_1, \ldots, x_i$. Note that

$$L(x_1, \ldots, x_i) = L(x_1)\prod_{i=1}^{n} L(\, x_i|x_{i-1}, \ldots, x_1)$$

where $L(x_1)$ is the probability of serving x_1 errors during the first period and $L(x_i|x_{i-1}, \ldots, x_1)$ is the conditional probability of observing x_i errors during period i given that $x_{i-1}, \ldots, x_1$were discovered during the previous periods. (These probabilities, of course, depend on the values of the parameters p, μ and υ.) From the expression above for $p(k,i)$ and noting that the number of errors observed in a given period is binomial with number of trials equal to the number of people who can detect an error, the above likelihood may be calculated. (It is somewhat complex since, during a given period one has to keep track of when errors were introduced and which consumers can see them.) In the worst case scenario, a grid search can be performed over the possible values for the quantities p, μ and

υ in order to find maximum likelihood estimates. For given values of the quantities p, μ and υ, the arrival of new data entails only a minor calculation to update the likelihood values. Thus in a big data context, the size of the data set does not hinder computational efficiency.

From the expression for $p(k,i)$ the expected number of errors detected during period i equals

$$\upsilon p(0,i) + \sum_{k=1}^{i} \mu p(k,i).$$

The variance of the number of errors detected in period i is given by

$$\upsilon p(0,i)(1 - p(0,i) + \sum_{k=1}^{i} \mu p(k,i)(1 - p(k,i).$$

Note that for given values of υ, μ and p, the above expressions are straightforward to evaluate.

From a management viewpoint, the above expressions can become effective tools. For many processes, control charts have become an important managerial tool for tracking quality, including software maintenance (Haworth, 1996). Control charts can be constructed tracking software errors: at the end of each period, compute the maximum likelihood estimate for the parameters; then compute the mean curve for the number of errors and the upper and lower control limits using the above expressions. Most applications of control charts are relatively straightforward and result in a constant central line. For software error tracking, however, the situation is more complex. For instance, the central line of a control chart based on the above expressions are not constant but its interpretation is similar to those of industrial applications. Points outside the upper and control limits indicate to the manager that the process is "out of control." This would happen, for instance, if any of the assumptions of the software error model were suddenly violated. The above expressions for mean and variance are therefore useful not only for predicting the flow of software errors but can also be used to warn a manager that the underlying marketplace is changing in an unexpected manner.

4. Conclusions

Traditionally, errors were defined within the system's boundaries. Goel (1985) suggests that "software faults can be attributed to an ignorance of the user requirements, ignorance of the rules of the computing environment, and to poor communication of software requirements between user and the programmer …" (ibid. p. 1411). This perspective focuses on mistaken code and is manifested in the quests for estimating error frequency according to various characteristics of the code. Now, with software being used by many users on the internet and mobile, errors rather than being drawn from a finite pool are constantly being introduced. In this paper, we shift the focus to the interaction between the system and its environment which leads to the notion of incompatibilities. The importance of incompatibility errors will grow with the growing impact of cloud computing and Big Data (Wang and Wu, 2016) as well as the Internet of Things (Prehofer, 2015). We formulate a robust and general model. In a Big Data setting, there is more freedom in allowing for more complex models since estimation of certain quantities (such as cancellation patterns and customer flow) is now much easier due to the sheer size of the data set. Error detection, even in a Big Data, context requires careful modelling since by design, even in large data sets, there is (and should be) a paucity of observations. We show how to calculate key quantities needed to construct the likelihood equation from which maximum likelihood estimators may be derived. A follow-up paper, rather than considering a grid search for finding these estimators will provide an algorithmic procedure that removes the need for a grid search. The important special case of discrete time periods and a constant rate of error introduction has been condidered in detail.

References

Banker, R. D., Davis, G. B., & Slaughter, S. A. (1998). Software development practices, software complexity, and software maintenance performance: A field study. *Management science*, *44*(4), 433-450.

Chan, T., Chung, S., & Ho, T. (1994). Timing of software replacement. *ICIS 1994 Proceedings*, 22.

Forbes. Shake those bugs out by A. Linsmayer. *Forbes*, May 18, 1998; 198-199.

Goel AL. Software reliability models. Assumptions, limitations and applicability. *IEEE Transactions on Software Engineering*, 1985. SE-11;(12); 1411-1423.

Gaudoin, O. (1999). Software reliability models with two debugging rates. *International Journal of Reliability, Quality and Safety Engineering*, *6*(01), 31-42.

Harworth D. A. Regression control charts to manage software maintenance. *Software Maintenance: Research and Practice*, 1996;8; 35-48.

Heydari, M., & Sullivan, K. M. (2017). An Integrated Approach to Redundancy Allocation and Test Planning for Reliability Growth. *Computers & Operations Research*.

Prehofer, C., & Chiarabini, L. (2015, July). From internet of things mashups to model-based development. In *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual* (Vol. 3, pp. 499-504). IEEE.

Wang, J., & Wu, Z. (2016). Study of the nonlinear imperfect software debugging model. *Reliability Engineering & System Safety, 153*, 180-192.

Zaman, S., Adams, B., & Hassan, A. E. (2012, June). A qualitative study on performance bugs. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*(pp. 199-208). IEEE.