Tracking the distribution of bugs across software release versions

A thesis submitted to the University of Dublin, Trinity College
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Department of Statistics, Trinity College Dublin

November 2015

Seán Ó Ríordáin

Declaration

Seán Ó Ríordáin

Dated: November 9, 2015

Abstract

Real software systems always contain bugs and the question on every release manager's mind coming up to a release centres around how many undiscovered bugs there still remain. This work looks at one model, (Goel and Okumoto, 1979), which tries to answer this question and extends previous work to try to borrow strength from previous release versions to help answer this question in a more rational manner.

Acknowledgements

First of all, I would like to express my utmost gratitude to my supervisor Simon Wilson. For more than four years, he has been very helpful and provided invaluable guidance to me.

I am also very thankful to all the wonderful people in our department, in particular Myra, Eamonn and John who have offered support and statistical guidance. Thanks must also go to those who took part in various long discussions in the grad room including Tiep, Louis, Susi, Arnab, Thinh, Cristina, Shuaiwei, Donnacha, Shane, Gernot, Arthur, Angela, Brett, Jason, Rozenn and everyone else in TCD, also to all my friends and family. Thanks also to Stephanie and Kate for their support. I would also like to thank my two sisters who have been particularly supportive over the last few months.

I am indebted to my parents, Róisín Fant Ó Ríordáin and Fionnbharr Ó Ríordáin for their long-lasting encouragement and support, I dedicate this thesis to them.

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie).

Seán Ó Ríordáin

Trinity College Dublin

November 2015

Contents

List of Tables

List of Figures

Glossary

Confusion Matrix is a square contingency table which allows the clear visualization of the performance of a classification prediction algorithm. Typically the class predicted is labelled on the left of the matrix and the actual class is labelled across the top of the matrix and the labels are typically given in the same order from the top and from the left. If an algorithm is perfect for a given class then the predicted count in a given row will be in the corresponding column. Associated terminology include *True Positive (TP)*, *True Negative (TN)*, *False Positive (FP)*, *False Negative (FN)*, *accuracy*, *Kappa*. According to Kuhn and Johnson (2013, p.254), a confusion matrix is '. . . a simple cross-tabulation of the observed and predicted classes for the data.'. 48

Ergodicity An ergodic Markov chain will be memory-less for a sufficiently long timescales MacKay (2003). A discrete space Markov chain 'is ergodic if all its states are ergodic' and 'a state is ergodic if it is aperiodic, recurrent and non-null'(Murphy, 2012) and Murphy then goes on to state his theorem 17.2.2 'Every irreducible (singly connected), ergodic Markov chain has a limiting distribution, which is equal to π, its unique stationary distibution'. . 40

Gelman's $\hat{R}$ (Gelman et al., 2013; Gelman and Rubin, 1992) is used as an indicator of the convergence of MCMC chains and works by calculating the variance within and between chains. A value of $\hat{R}$ which is close to 1.0 indicates likely convergence. It is closely related to Gelman's method for calculating the effective chain length. $n_{\text{effective}}$. Note that Gelman has changed the exact definition of how $\hat{R}$ is calculated over recent years, and this description is taken from Gelman et al. (2013, Section 11.4, page 284). Given a set of m Markov chains each of length n, which we denote θ_{ij} for $i = 1, \ldots, n$ and $j = 1, \ldots, m$, we estimate the between sequence variance B and the within sequence variance, W:

$$B = \frac{n}{m-1}\sum_{j=1}^{m}(\bar{\theta}_{.j} - \bar{\theta}_{..})^2 \quad where$$
$$\bar{\theta}_{.j} = \frac{1}{n}\sum_{i=1}^{n}\theta_{ij}$$
$$\bar{\theta}_{..} = \frac{1}{m}\sum_{j=1}^{m}\bar{\theta}_{.j}$$
$$W = \frac{1}{m}\sum_{j=1}^{m}s_j^2 \quad where$$
$$s_j^2 = \frac{1}{n-1}\sum_{i=1}^{n}(\theta_{ij} - \bar{\theta}_{.j})^2$$

Then we can estimate the weighted marginal posterior variance of the estimand, and then finally we calculate $\hat{R}$:

$$\widehat{\mathrm{var}}^{+}(\theta|\mathrm{Data}) = \frac{n-1}{n}W + \frac{1}{n}B$$
$$\hat{R} = \sqrt{\frac{\widehat{\mathrm{var}}^{+}(\theta|\mathrm{Data})}{W}}$$

. 62

Gini Coefficient The Gini Coefficient (Gini, 1912; Ceriani and Verme, 2012) is widely used to describe income inequality, and in a society where everybody earns exactly the same amount, then the Gini coefficient will be zero and where all income is earned by a single person, then it will have a value of one. The particular version we use[1] is calculated as follows: Given a pre-sorted vector X of the values of interest with length N, and X_i denotes the i^{th} element of the sorted vector X, then we calculate as: $\frac{1}{N}\left[\frac{2\sum_{i=1}^{N} ix_i}{\sum_{i=1}^{N} x_i} - (N+1)\right]$. 26

IEEE double a computer number format which is widely used to hold real valued numerical quantities and is defined by an IEEE standard IEEE Task P754 (2008) and is held in 8 bytes. The R language R Core Team (2015)'numeric' type is defined to be an IEEE double [2] . 99

Kappa attempts to highlight the real gain in the performance of the classifier compared to a classifier which was completely random, range is zero to one. If O

[1]R package `'ineq' function 'Gini()', (Zeileis, 2014)`

[2]`https://en.wikipedia.org/wiki/Double-precision_floating-point_format`

is the observed accuracy and E is the expected accuracy based on the marginal totals of the confusion matrix, then Kappa = $\frac{O-E}{1-E}$, (Kuhn and Johnson, 2013). Kappa can take on values between -1 and $+1$ and zero means that there is no match while -1 means perfect disagreement and $+1$ means perfect agreement between the observed and the predicted classes. . 49, 88

MAR Missing At Random, frequently used in reference to the imputation of missing data. This refers to the case where data is missing at random in a way which is independent of the covariates that we are interested in, which contrasts with MNAR and MCAR [3]. xxiii, 40

MCAR Missing Completely At Random, frequently used in reference to the imputation of missing data and sometimes called 'uniform non-response'. In this case there is no pattern to the missingness at all. Contrast with MAR and MNAR [4]. xxiii, 40

MNAR Missing Not At Random, frequently used in reference to the imputation of missing data. In this case the data that is missing is correlated with the covariates of interest and often occurs in political surveys where people at the top and bottom of the income scale do not respond. Contrast with MAR and MCAR [5]. xxiii, 40

UAT is when the customer tests software before formal acceptance that it is to specification. 8

[3] `www.missingdata.org.uk` comprehensively describes the difference between these mechanisms in the article 'Missingness Mechanisms'

[4] `www.missingdata.org.uk` comprehensively describes the difference between these mechanisms in the article 'Missingness Mechanisms'

[5] `www.missingdata.org.uk` comprehensively describes the difference between these mechanisms in the article 'Missingness Mechanisms'

Chapter 1

Introduction

This chapter outlines the rest of the thesis and summarises the main contributions.

This research is about the modelling of the number of bugs in software systems. The terms 'bug', 'fault' and 'error' will be used interchangeably, though in some literature they have very specific and distinct meanings.

The dataset used in this research is from Open Source Software (OSS) and was extracted from open sources of data and have been made available to other researchers as part of this research.

1.1 Statistical motivation

Software is becoming increasingly complex and managing software projects is getting more and more difficult. Bugs in software are now considered a given and rather than trying to eliminate them, the goal is around their management. We extend previous work by looking at how information across multiple versions can be used for inference on future versions which we do using a hierarchical Non-Homogeneous Poisson model combined with a data imputation model for labelling bugs which have no version.

1.2 Outline of the thesis and contributions

This thesis is divided into following chapters:

Chapter 2: Background Information

This chapter gives some background into software development practices and open source software in particular. Software bug databases are discussed and the dataset used as the motivation for this work is described. In particular a new dataset, Firefox-2013, is introduced.

Chapter 3: Statistical Theory

This chapter reviews statistical inference and the Bayesian approach. Secondly, we briefly outline the theory of Monte Carlo methods and Markov chain Monte Carlo (MCMC). Thirdly, we review classification, and the distinction between generative and declarative models. Finally we discuss models of software reliability.

Chapter 4: Goel-Okumoto

This chapter reviews the Goel-Okumoto model and looks to extend this work to software projects where there are multiple related versions of the same software released one after the other.

Chapter 5: Semi-Supervised Classification

Chapter 5 introduces the version model in a standalone form and shows that it is surprisingly useful. This model imputes missing version labels in the bug database.

Chapter 6: Combined model

Chapter 6 presents the combined model and the results. The combined model takes the version model which imputes labels and uses the Goel-Okumoto model on top of this.

Chapter 7: Case Study

Chapter 6 presents a case study based on the Firefox dataset and looks at the optimal release period by maximizing the expected utility.

Chapter 8: Discussion

Chapter 8 discusses the results and discusses future work.

Chapter 9: Conclusions

The last chapter briefly concludes the thesis.

1.3 Research Contributions

The following are the main contributions described in this thesis:

1. A new dataset, Firefox-2013, has been published for the community researching software reliability.
2. Combining the information from successive versions can lead to better estimates of parameters.
3. Using the information in the unlabelled bugs to augment the labelled bugs to improve modelling.
4. A fast implementation of the above.

Chapter 2

Background Information

2.1 Introduction

This chapter informally outlines some background information for this thesis and is intended as a very short overview of software development, bugs and some of the issues involved. Those already familiar with the domain can skip forward in this chapter to Section 2.4 where there is a description of the dataset used later.

2.2 Background Information on Software Development and Open Source

2.2.1 Software Development

Software development concerns the production of software whether in a commercial or non-commercial setting, and the processes and procedures surrounding it. Software engineering is famously defined by Boehm (1976) in his seminal paper at which time software was already worth some $20 billion in the USA.

> 'Software engineering is the means by which we attempt to produce all of this software in a way that is both cost-effective and reliable enough to deserve our trust.'

While some have taken to referring to 'software engineering', Lutz et al. (2014) describe the first undergraduate degree in software engineering (as distinct from Computer Science) in the United States which started in 1996 in the Rochester Institute of

Technology (RIT). As a discipline software development is still very young compared to other professional engineering disciplines such as civil engineering which can demonstrate large scale projects requiring the coordination of huge numbers of people - think of the Roman aqueducts about 2,000 years ago, the Egyptian pyramids about 4,000 years ago or Newgrange in the valley of the Boyne about 5,000 years ago.

Professional software development was in quite some turmoil in the early 1970s when Knuth (1974) wrote his paper summarizing the arguments for structured programming, which at the time was quite controversial. In that paper Knuth claimed that 'premature optimization is the root of all evil', and by that he meant that programmers regularly try to write the 'best' code (using their definition of best) instead of trying to write code that is reliable and maintainable. Software engineering as a discipline still has considerable difficulties coordinating more than a handful of people at a time. Bugs can easily be introduced when there are communications problems in a team or between teams on a project. Many countries now have mandatory certification of professional engineers, but not for software development. In safety critical areas such as nuclear safety, air traffic control systems, motor vehicle braking systems and medical devices, there has been considerable progress made in moving towards a strict and refined development process, but in many small IT companies and in Open Source, the process is considerably more *ad hoc*. There is currently work going on at the International Organization for Standardization (ISO) relating to software for medical devices, led by Dr. Fergal Mc Caffery of Dundalk Institute of Technology (DKIT), and International Electrotechnical Commission (2014) was published in May 2014 and a number of other documents are in train.

Historically the development of software traditionally went through a number of separate and distinct phases: requirements gathering, design, coding, testing, release and maintenance. Crucially, the duration of each phase of the project was dependent on the functionality that was promised. Software was produced in a so called 'Waterfall' methodology, i.e. that everything was specified in advance, the customer signed off on massive requirements documents and then a year or two later they would receive a delivery of the final product. Since the majority of these huge projects failed[1], there has since been a movement to reduce the size of each 'deliverable'. In many cases

[1]Depending on which reference you look at, some say more than 90% of these large projects fail to meet the initial expectations.

the requirements had gone out of date by the time the software was delivered. Quite simply, it is easier and less risky to deliver a few small projects on time and on budget than a single big project.

An extension of the waterfall model was the 'V-model' where the left hand side of the 'V' comprised of the traditional development process, the software was 'released' at the base of the 'V' and the right hand side of the 'V' was the rest of the lifetime of the software.

A further extension to the 'V-model' is the 'W-model' where the development life cycle is in the left hand side of the 'W' and the testing life cycle is in the right most half of the 'W', but there is still a clear separation between development and testing.

Agile programming was first expounded by Schwaber (1997) who described the 'Scrum' (Sutherland, 2012) methodology, which uses fixed short duration 'sprints' of usually one month between releases. This is in marked contrast to the waterfall model where the phase duration was variable. The Scrum methodology is used widely and in particular by groups working on web time, including Google, Yahoo! and Microsoft. At around the same time Beck (1999) was developing the 'eXtreme Programming' methodology (XP) and he said that deliverables should involve very small change sets and Beck advocated having only one feature per release, hence eXtreme Programming. This is particularly well suited to cloud delivered services as the provider can rollout new features as and when they see fit. Facebook are proud that their software engineers are pushed to rollout new features to the public in their first week at work which would be completely unheard of in older software companies[2].

In agile methodologies, the two halves of the 'W' are super-imposed so that there is a test phase closely associated with each stage of development. As the software requirements are written they are immediately passed to the test team who write the testing documents against them, so they are written simultaneously and collaboratively instead. There is an agile methodology called *Test Driven Development* (TDD), which is gaining ground, where the tests are written by the developer before the code. TDD encourages the developer to mentally better explore the edge cases where bugs are more likely to occur.

[2]https://www.facebook.com/video/video.php?v=10150411360573109

Traditionally, the code was created and finished by the development team and bundled up and passed over to a test team. Often known as 'throwing the code over the wall'. The test team looked for bugs and documented these bugs. The development team then fixed these bugs and created a new build for the test team. There is often a lot of antagonism between the development and test teams, where the former say the latter are slowing down the development process by finding bugs and the later retort that it would all be much quicker if development did not put bugs in their code. This process was iterated until the production manager decided to release a build to the customer. At this point, the customer's User Acceptance Testing (UAT) team started doing their testing and they sent their bugs back to the company whose test team verified the bugs and documented them, and the development team fixed the bugs and so the cycle went. Note also that a non-trivial proportion of fixes to bugs will themselves also introduce bugs, some very significant indeed like the famous 'heart-bleed' bug in OpenSSL which was introduced during a bug fix and caused significant disruption to secure websites in the spring of 2014[3]. The 'traditional' process for software development is still widely taught and used.

van Vliet (2000) discusses the general principles of software engineering. O'Regan (2002) discusses specifics of software quality.

2.2.2 Open Source Software

Open Source is a movement which promotes the free distribution of products, most notably software. Many well known software systems are Open Source, including the Linux Kernel[4], Mozilla Firefox[5], Mozilla Thunderbird[6] and the Apache webserver[7].

Fitzgerald (2006); Raymond (1999); Feller and Fitzgerald (2002) go into some considerable detail about Open Source[8] software which is developed in a non-traditional fashion, though these books are a bit dated now and Open Source development methodologies move quite quickly compared to those in many large commercial organizations which do not understand the threat that their businesses are under. The source code

[3]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160
[4]http://kernel.org
[5]http://www.mozilla.org/en-US/firefox/fx/
[6]https://www.mozilla.org/en-US/thunderbird/
[7]http://httpd.apache.org/
[8]http://opensource.org/

is available for anybody on the Internet to download at any time, and anybody can write useful code which can be incorporated into the main trunk of code. Many coders will cooperate with others to build bigger useful components, but sometimes complete outsiders will submit code which is incorporated. Big projects like Firefox will have dedicated testers, but most of the bugs will be logged by people who are not associated directly with the project.

The development process methodology in Open Source projects can vary greatly from extremely structured to extremely unstructured. As a general rule the larger and older the project the more structured the development process. Linux has been in development in public since Linus Torvalds made an announcement in the autumn of 1991 on the comp.os.minix newsgroup. Initially anybody who wanted to make a change or an addition to the project sent a 'patch' to Linus Torvalds and he merged it into his code. Today there is a hierarchical structure and Torvalds has a number of lieutenants who manage subsystems, and below there is another layer of people who manage smaller subsystems (Corbet et al., 2015). However, Torvalds is still the overall lead on the project.[9]

2.2.3 Mozilla Firefox Development

Mozilla Firefox is a well known Internet browser which was first released more than ten years ago in November 2004 and published by the Mozilla Corporation (2013*a*), itself a wholly owned subsidiary of the Mozilla Foundation (Mozilla Foundation, 2013). The Mozilla Foundation is a non-profit organization "dedicated to keeping the power of the Web in people's hands.", [10] The Mozilla Corporation earns revenue primarily through deals with search providers (Mozilla Corporation, 2013*b*) and employs many of the developers and testers who work on Firefox. Recent studies (International Telecommunications Union, 2013; w3counter, 2013; Stats, 2013; Wikimedia, 2013; Clicky, 2013) suggest that there were of the order of 450 million users of Firefox across a number of versions in 2012 and it has likely exceeded 500 million in 2013, as the ITU has estimated the number of Internet users at 2,749 million and Firefox usage is estimated at approximately 20%. Mozilla themselves claim that in May 2015 that 'Half a billion

[9]`http://en.wikipedia.org/wiki/History_of_Linux`
[10]`https://www.mozilla.org/en-US/mission/`

people around the world use Firefox'. [11]

Firefox is an open source (Feller and Fitzgerald, 2002) project which derived from the Netscape browser in 1998 when it was open sourced[12]. Since 2011 Firefox has been developed using an Agile (Schwaber, 1997) methodology. Bug reports are public[13] and the source code is in a publicly accessible repository,[14]. Mozilla Firefox is relatively unusual in being released on a very short (42 day) fixed release cycle. Version 5 of Firefox, released on the 21st of June 2011, was the first of the 'Rapid-Releases' and there was a 56 day gap before version 6 was released. Subsequent versions have been released every 42 days or almost. Firefox version 18 was released on the 8th of January 2013, and version 19 was scheduled for release in the week of the 18th of February 2013, and was actually released on the 19th of February 2013. At the end of May 2015, the current release is 38 which was released on the 12th of May 2015, and according to `https://wiki.mozilla.org/RapidRelease/Calendar`, version 39 will be released on the 30th of June 2015. Note that `https://wiki.mozilla.org/RapidRelease/Calendar` appears to be the authoritative source of information on software releases by the Mozilla organization despite being a Wiki.

As an open source project, the Firefox project is managed in a transparent way across the Internet through `https://wiki.mozilla.org/Firefox`, and bugs can be reported by anybody to `https://bugzilla.mozilla.org`, which is a database for recording and managing bugs relating to the Mozilla project. These bugs often go into a triage process where duplicates are marked as such and some reports are marked as INVALID or sometimes that the report is actually an 'enhancement'.

The Blackduck (2014) project tracks the activity of Open Source projects. As of the 1st of March 2013, Firefox was made up of 8.8 million lines of code and 1.6 million comment lines. As of May 2015, Firefox was made up of 13.0 million lines of code and 2.1 million comment lines. Until early 2007, there were less than 20 contributors of code in a given month, and then in the spring of 2007 the numbers jumped so that by June 2007 there were 121 individual contributors. Since then the number of contributors has grown to more than 300 and appears to be growing still, and there

[11]`https://blog.mozilla.org/press/ataglance/`

[12]`http://en.wikipedia.org/wiki/History_of_Firefox`

[13]Some bugs deemed to be security or HR related are not visible to the public.

[14]`https://developer.mozilla.org/en-US/docs/Developer_Guide/Source_Code/Downloading_Source_Archives`

were 383 unique contributors to the Firefox project in April 2015. As of May 2015, in total there have been more than 253,000 commits made by 3,499 contributors which took an estimated 4,064 years of effort using the COCOMO model (Boehm, 1984). The number of commits in the 12 months to 2015-05-28 is 60,226 and the number of unique contributors as 1,217. A total of 28,612,890 lines were added and 20,154,184 lines were removed - or possibly just changed.

2.3 Bugs!

Bugs in software are a big problem that can cost money and lives - there are many well known examples, including the death of patients that were given overdoses of radiation therapy and banks overcharging[15]. Knight Capital was caused to loose $465 million due to a number of software bugs and a poor testing environment[16].

Software is unusual in this world in that it does not wear out like physical products, though it can still fail. Physical products can have design flaws in exactly the same way that software does, i.e. it does not do what it was supposed to do, but physical products also have the added failure mode of physical failure when they break. Since software is digital, it cannot wear out in the same way and it can be copied perfectly. However, over time large software projects can end up being configured in ways that were not originally intended and can drift into a less than perfect state. After the last official release of a product when it is in maintenance, software which is not in active development can become more awkward to maintain as often the maintainers are often less experienced developers who did not participate in the original development of the system and are not that familiar with the code base. Yin et al. (2011) estimated that between 14 and 24% of bug fixes introduce new bugs into the system. In an Irish example, the Garda Pulse system was written and maintained by staff from a large consulting multinational. An acquaintance was asked to review the work and large sections of it consisted of very poorly written code where hundreds of lines of code were copy/pasted dozens of times with only a tiny modification each time. Best practice which would be to re-factor the code and parameterize it - should a change have been required in this code, then all of cases of the copied code would have needed

[15]`http://en.wikipedia.org/wiki/List_of_software_bugs`

[16]`http://pythonsweetness.tumblr.com/post/64740079543/how-to-lose-172-222-a-second-for-45-minu`

to be changed - making it highly likely to insert a new bug while fixing an old one. In software engineering, Fowler has described 'obvious' patterns like this as having a 'smell'[17], (Fowler, 1999).

When there is a problem with software we generally call it a 'bug'. For the purposes of this thesis, a 'bug' is a mismatch between what the user expects and what the software actually does, although others have used much stricter definitions. This turns out to be quite difficult to work with, since many users will have different expectations. A user might report a bug, but the software might have been deliberately designed with a feature built in this particular way; the software developers will mark such a bug report as 'WONTFIX' and they will close the report, i.e. they have no intention of fixing this bug. This is seen on a regular basis by those subscribed to the R-HELP and R-DEVEL mailing lists relating to R (R Core Team, 2015), as users claim to have found a bug in R and in most cases, the R-Core team reply that they have not found a bug in R and that the user has misunderstood. There may be no documents saying that the software is designed in this particular way, but it is the developers who effectively own the bug database, and have the final word. Requirements and specifications can exist as a collective understanding amongst a group of developers, built through time in chats over the Internet or in the pub, and they will be adamant that this piece of software was always meant to be created in a particular way despite there being no written document to say this. Some large Open Source organizations, such as Mozilla, now employ full-time product managers whose job it is to define the expected behaviour of the product. Note that the definition of a bug that is commonly used in proprietary software development is a mismatch between the original specification and what software actually does, e.g. Musa et al. (1987), on page 8, defines a *software failure* as:

> '. . . the departure of the external results of program operation from requirements.'

However, in Open Source development there is frequently no set of written requirements or a specification as the software is not created as a "product", but by groups of developers who want to do things better or differently. It is possible that some of these developers might have sketched out some requirements but they are laying at

[17]http://martinfowler.com/bliki/CodeSmell.html

home or have been thrown in the bin. Equally, some Open Source projects, like *Perl*, create detailed and long lasting specification documents which go through long review processes[18].

People think of failures as being catastrophic failures, but they come in all shapes and sizes from a complete crash to a bank miscalculating the interest you must repay on your loan to the trivial misspelt label in some hidden backwater of the software which nobody normally ever sees. These differences are normally labelled in a bug tracking system using the Severity variable and for Bugzilla it has the following levels: blocker, critical, major, normal, minor, trivial and enhancement. This study excludes bugs marked as enhancement.

Software can fail because of hardware faults, or in rare cases the software executable can be modified by a cosmic ray and not repaired by error correcting mechanisms inherent in many components of modern computer hardware and thus cause a failure. However, in most cases failures are due to a flaw being introduced at some stage of the production process and exists silently from before the software is ever used. Here we will concentrate on pure software failures and we will ignore failures caused by hardware or otherwise. Some literature in the domain, such as Pham (2010), describes failures in software as if they were hardware failures, i.e. that something was physically broken, which can be confusing as this is just not the case with software.

The typical life-cycle of a bug is that there is a mistake made in the design or in the coding stage, and then it can be repaired (possibly imperfectly) at any stage after that.[19] Studies have shown that the cost of a bug increases by an order of magnitude for each phase later that it is detected; the earliest of these studies are highlighted in the seminal paper on Software Engineering by Boehm (1976). For example if the cost of a defect is x when it is detected in the (first) design stage, then it is $10x$ if detected in the development stage, $100x$ if detected in the testing phase and $1,000x$ if detected after release.[20] Figure 3 in Boehm (1976) shows a plot of some data from IBM, TRW and GTE which implies a factor of about 2.5 per phase but this is clearly quite old

[18]http://perl6.org/specification/

[19]Bugs can also be due to mistakes in a configuration file and nothing to do with the software code. From the users' perspective they cannot tell that this is due to a configuration file problem and to them it is just a bug.

[20]Depending on the study the factor 10 varies from 7 upwards but the principle that the later the bug is detected is what is really important.

data and the systems developed back then were obviously much smaller in scale. One explanation of why this occurs is that when a defect occurs in a later cycle, the work in previous cycles has to be re-done, e.g. if there is a defect in the original requirements, then this will cause work in all of the following stages, i.e. design, coding and testing [21]. More recently, Kan (1995, Section 6.4) describes a number of studies which give cost ratios of between 1 and 92 and points out that the effort to track down and fix a bug that is found in the field is much higher.

While Musa et al. (1987) states that documentation defects are not bugs, we argue that documentation is part of the product as a whole and a documentation defect will cause the user to have incorrect expectations as to the way the software works, and thus in our opinion it is a bug. A thorough glossary of testing terminology has been produced by the International Software Testing Qualifications Board (ISTQB) an industry body[22]. The ISTQB defines an 'anomaly' as:

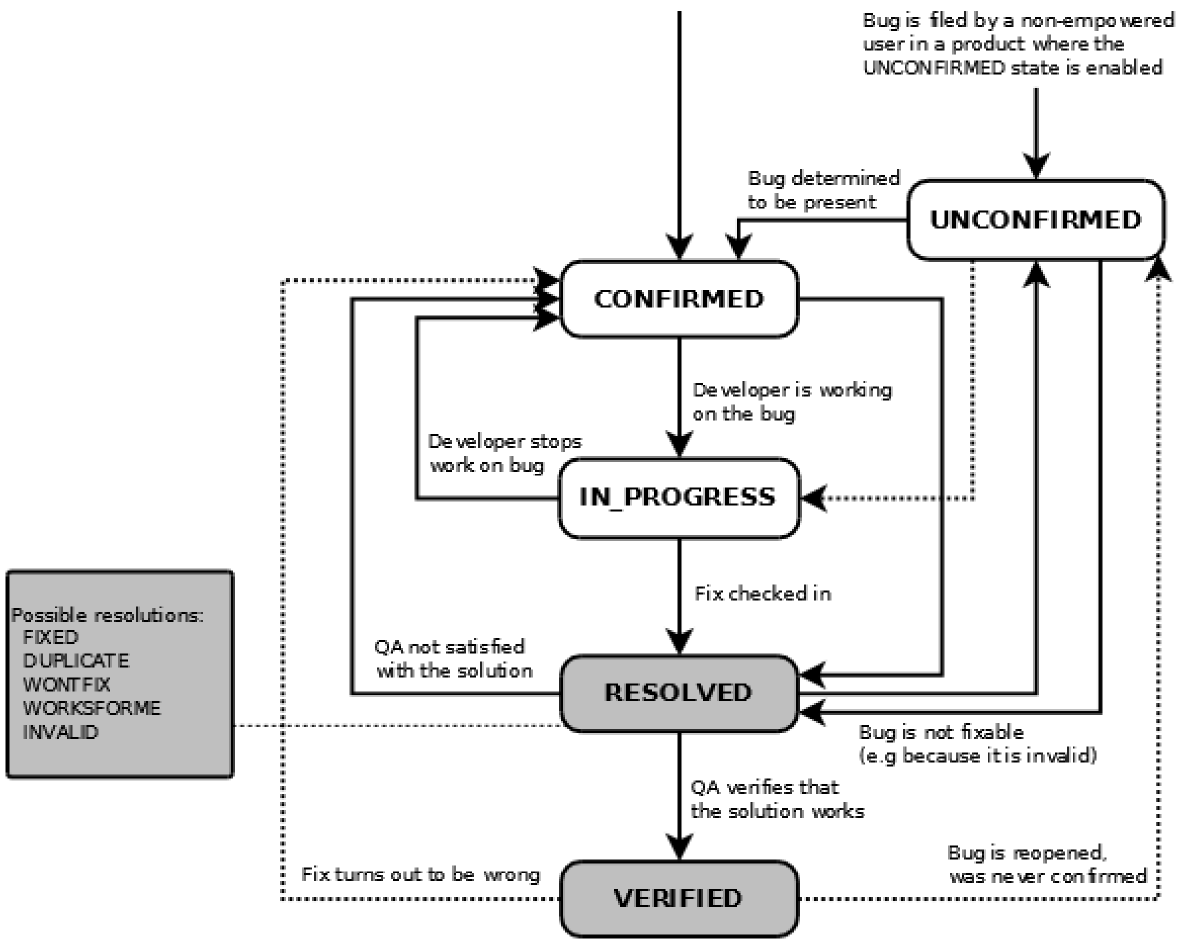
> 'Any condition that deviates from expectation based on requirements specifications, design documents, user documents, standards, etc., or from someone's perception or experience. Anomalies may be found during, but not limited to, reviewing, testing, analysis, compilation, or use of software products or applicable documentation. See Also: defect, error, fault, failure, incident, problem'

Curiously the ISTQB do not define a 'bug', but they define a defect as:

> 'A flaw in a component or system that can cause the component or system to fail to perform its required function, e.g., an incorrect statement or data definition. A defect, if encountered during execution, may cause a failure of the component or system. Synonyms: bug, fault, problem'

Our definition is supported by the seminal work of Boehm (1976) where the definition of 'software engineering' specifically includes *. . . not only computer programs, but also the associated documentation required to develop, operate, and maintain the programs* and he further goes on to emphasise this point and writes the following definition:

[21] `http://istqbexamcertification.com/what-is-the-cost-of-defects-in-software-testing/`
[22] `http://www.istqb.org/downloads/glossary.html`



Bug is filed by a non-empowered user in a product where the UNCONFIRMED state is enabled
Bug determined to be present
UNCONFIRMED
CONFIRMED
Developer is working on the bug
Developer stops work on bug
IN_PROGRESS
Fix checked in
Possible resolutions: FIXED DUPLICATE WONTFIX WORKSFORME INVALID
QA not satisfied with the solution
RESOLVED
Bug is not fixable (e.g because it is invalid)
QA verifies that the solution works
Bug is reopened, was never confirmed
Fix turns out to be wrong
VERIFIED

Fig. 2.1: The life cycle of a bug in the Bugzilla bug tracking system. Copyright The Mozilla Foundation, licensed under the MPL.

> '*Software Engineering:* The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them.'

The Bugzilla project defines the life cycle of a bug in the Bugzilla bug tracking system in their documentation (The Bugzilla Team, 2014, Section 5.4), refer also to their diagram copied here as Figure 2.1 [23].

2.3.1 Bug Databases

In many cases we do not know of the existence of a bug until somebody actually spots it and records this. Most users do not record bugs but try to ignore them and the first record is not necessarily the first sight of the bug. Further, a user might not recognize

[23]`https://bugzilla.readthedocs.org/en/latest/_images/bzLifecycle.png`

something as a bug. The exception here are so called *regression* bugs which can be automatically detected and logged because of an automated test and a framework to manage it.

It is important to note that the time and date of the failures are actually the time and date they were first recorded in a bug database, since these bugs have been around since the software was first created and there is no ‘wear and tear’ in digital software.

For the purposes of this work we are looking at the bugs that are recorded in the bug database `https://bugzilla.mozilla.org/`. On the 23rd of April 2014 the millionth bug was filed in the Mozilla bug database which includes a variety of projects, only one of which is Firefox. As of 9am on the 9th of October 2014, there were 1,080,387 bugs filed, by noon on the 28th of May 2015 there were 1,169,220 bugs in Bugzilla.

Some bugs are duplicates of previously reported bugs and eventually they will be linked to the first such bug that was actively examined and commented on by the triage or development teams and marked as a duplicate. Other bugs will be marked as ‘WONTFIX’, i.e. somebody on the project with authority has decided, that for some reason, nothing will be done about this bug and they just close it. Bugs can be perfectly valid, but if the developers are not able to reproduce it, then they cannot fix it so it might just be closed, e.g. the infamous user plea “it doesn’t work”. Other bug reports might not have enough information and will be closed. For the purposes of this work we will only be looking at bugs which have a resolution of either open, i.e. ‘—’ or ‘FIXED’. Bugs with a status of ‘UNCONFIRMED’ will be ignored as many will go on to be marked as duplicates or invalid in some way or else the person doing triage will tidy up the report to clean it up. Bugs can also occur in configuration files and example files that are provided with a software release. A consistent approach was taken to the queries used in this research to reduce bias.

Many developers do not record bugs in bug databases, but instead they either just ignore them[24] or they will just fix the bug without recording it.[25]

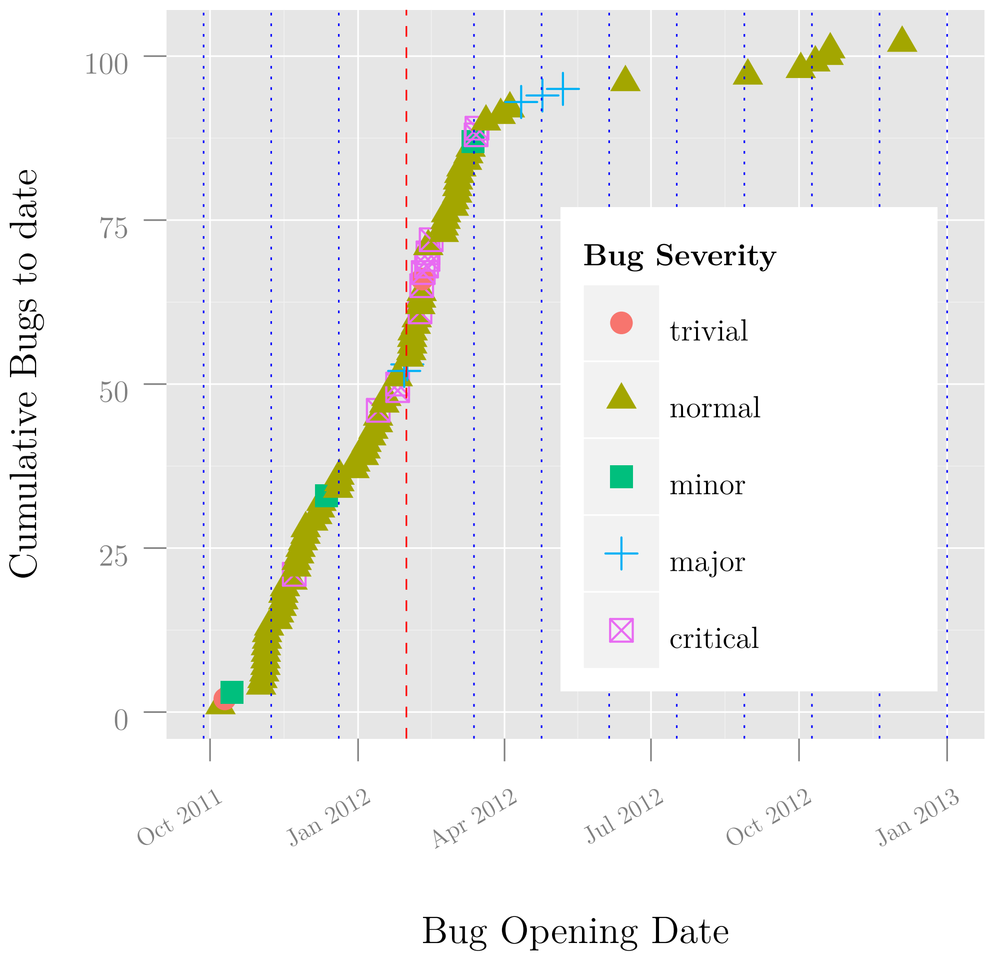


Fig. 2.2: Cumulative Number of Bugs Reported against Firefox version 10

2.3.2 Bug Data

Figure 2.2 shows the cumulative number of bugs assigned to version 10 of Firefox as extracted from Bugzilla in early February 2013. The red vertical dashed line shows the release date and the blue vertical dotted lines show the release dates of other versions of Firefox[26]. Version 10 Extended Support Release (ESR) was released to the public on the 31st of January 2012 and version 11 was released 42 days later on the 13th of March 2012. An Extended Support Release (Mozillians, 2011) is given support by the Mozilla organisation for nine release cycles (54 weeks) helping organisations which cannot roll-out new versions of Firefox every six weeks. This means that releases with security patches and other important bug fixes will continue for much longer than the normal 42 day release cycle. For our purposes, it means that reliable bug data will continue to be logged for more than a year after the original release date.

2.3.3 Open Data

Open data is an old concept, but has been recently been formalized in a number of ways including at `http://opendefinition.org/okd/` and `http://okfn.org/opendata/`.

> 'Open data is data that can be freely used, reused and redistributed by anyone - subject only, at most, to the requirement to attribute and sharealike.'
> - `OpenDefinition.org`

The advantage of open data for statistics research is that researchers from around the world can use the same set of data to test their models, thus allowing more direct comparison between models from different researchers. Until now researchers in the domain have used either proprietary data which has not been made public, or they have used the Naval Tactical Data System (NTDS) data which dates back to a paper (Jelinski and Moranda, 1972) from the early 1970s and which implies to the author that the data are from the 1960s. It includes no covariate information, and the details of its origins are classified as defence related. Discussions the authors have had with others in the domain have failed to shed any light on the issue.

[24] Thus avoiding creating work for themselves!

[25] Speaking with Prof. Brian Ripley in July 2012 regarding the R-Project, he recommended looking at the subversion check-in notes for records of bugs fixed and comparing it with the Bugzilla records.

[26] Minor security and patch releases are ignored in this document.

Curiously, while many authors cite Jelinski and Moranda (1972) as being the source of the NTDS dataset, the original and difficult to obtain paper does not actually contain the data for the NTDS dataset, and the first publication of the dataset, to the authors' knowledge, is in Goel and Okumoto (1979).

Since Jelinski and Moranda (1972)'s seminal work, many papers in the area of software reliability have used the NTDS dataset. A strength, but also a weakness of the NTDS dataset is its simplicity.

A number of datasets including NTDS are described in Pham (2010), however all of these datasets are quite simple, having only dozens of bugs at most and all of these datasets except one are restricted to only one version, the exception having two versions with 26 bugs recorded in the first and 43 bugs recorded in the second.

Papers such as Ravishanker et al. (2008) have used data sets other than NTDS or they have simulated datasets for analysis in their paper, but unfortunately these datasets are not made available publicly to other software reliability researchers.

Other bug datasets have been published, e.g. (Lamkanfi et al., 2013)[27] and (D'Ambros et al., 2010)[28], but these datasets are quite complex and intended for those who are data mining for information within bugs, e.g. looking for swear words within the text of a bug report[29].

2.4 The Firefox-2013 Dataset

Here we describe a dataset on bug discovery for the Internet browser Mozilla Firefox. This dataset has been made publicly available at `https://github.com/seanpor/Firefox-2013`. We have worked hard to make it easily accessible to researchers in software reliability and believe that it provides excellent opportunities to allow researchers to propose and evaluate a variety of software reliability models.

Figure 2.3 shows the cumulative number of bugs for each of the rapid releases of Firefox. The dashed vertical lines in the figure correspond to release dates. Table 2.1 shows the count of each bug which is labelled with a version in the dataset. We have taken all versions prior to release 5 and labelled them as 'PreRapid', and all bugs

[27] `https://github.com/ansymo/msr2013-bug_dataset`
[28] `http://bug.inf.usi.ch/`
[29] `https://github.com/ansymo/msr2013-bug_dataset/tree/master/examples/swearing`

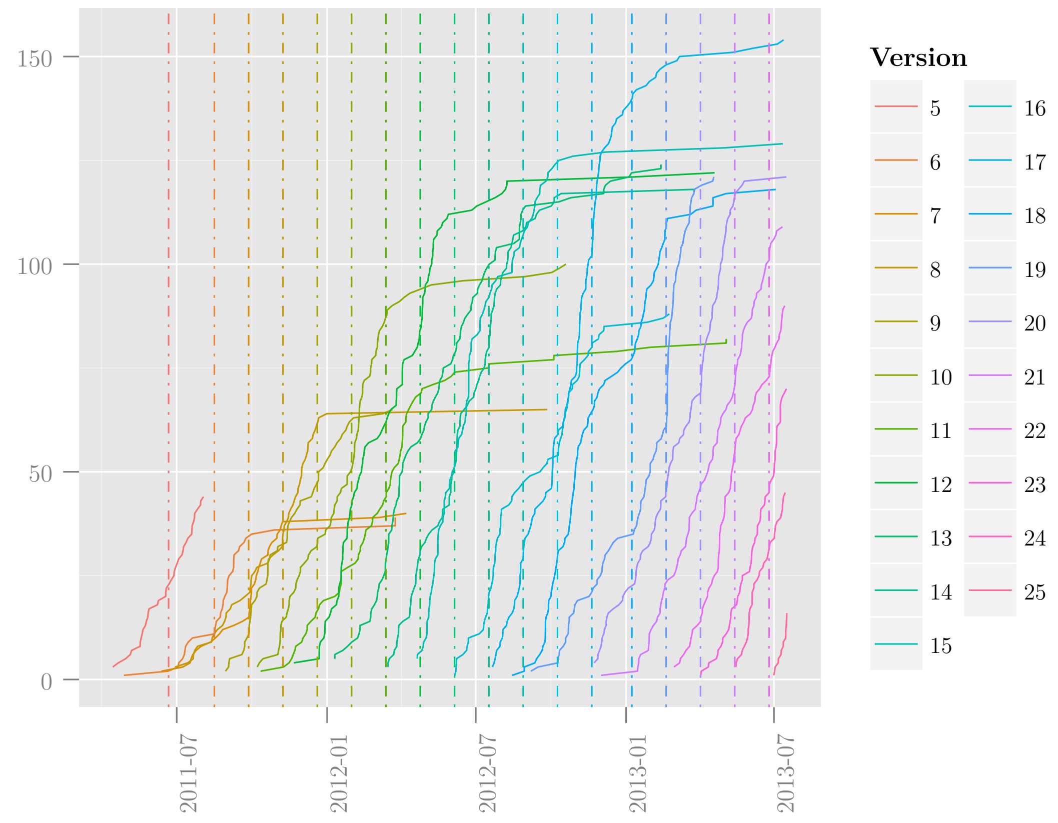


Fig. 2.3: Cumulative number of Bugs logged to Bugzilla for each individual Rapid-Release version of Firefox. The dashed vertical lines correspond to release dates in Table 2.1

marked as ‘UNSPECIFIED’ or ‘TRUNK’ have been relabelled as ‘Unknown’. As can be seen in Figure 2.3, the paths of each of the known releases are largely parallel to each other and there are often an inflection points on the release date. This inflection point is at the point where the number of users (or testers) jumps within a few days from thousands to tens or hundreds of millions and this could be interpreted in a Goel-Okumoto model (Goel and Okumoto, 1979) as having a different b, which we will discuss further in Chapter 4. Once the next version has been released, the number of users drops suddenly giving another inflection.

As can be seen in Table 2.1, most defects are recorded against TRUNK or UNSPECIFIED, which we have combined and relabelled as ‘Unknown’. Note that we do not include the many defects which have been marked as an ‘enhancement’, or rejected as one of (DUPLICATE, EXPIRED, INCOMPLETE, INVALID, WONTFIX, WORKSFORME) or it is still in the state UNCONFIRMED.

As previously mentioned, for the purposes of this work we will only be looking at bugs which have a resolution of either open, i.e. ‘—’ or ‘FIXED’. Bugs with a status of ‘UNCONFIRMED’ will be ignored as many will go on to be marked as duplicates or invalid in some way or else the person doing triage will tidy up the report to clean it up.

2.4.1 Fields in the dataset

The data are based on an ‘academic snapshot’, of the full Bugzilla database made on the 18th of July 2013. In constructing the ‘academic snapshot’, the Mozilla team have removed any bugs which might be security or HR related. Our data has been further filtered to refer to only Firefox bugs on or after 2011-01-01 which are marked as neither enhancements nor UNCONFIRMED.

The data are taken from a snapshot of `https://bugzilla.mozilla.org/` taken on the 18th of July 2013.

There is one record per bug. Note that bug reports are not perfect and may have mistakes, but they are a record of what was in the Bugzilla database on the 18th of July 2013. Note too that fields such as `bug_severity` and `priority` may be changed after the bug is originally reported - typically by those in triage or a developer who sees a trivial bug marked as high severity and high priority. Each record has the following

Release	BugCount
PreRapid	102
5	42
6	41
7	41
8	65
9	65
10	99
11	78
12	119
13	125
14	118
15	127
16	85
17	135
18	124
19	126
20	121
21	108
22	91
23	76
24	48
25	23
Unknown	8,461
	10,420

Table 2.1: The number of bugs logged to Bugzilla for each individual Rapid-Release version of Firefox.

fields

Version : The version number of Firefox associated with this bug, e.g. 14, or 'Pre-Rapid' to describe any version prior to 5, or 'Unknown' for any bug which did not have a version number associated with it.

bug_id : The original bug number in the Bugzilla database. Further details on this bug can be queried at `https://bugzilla.mozilla.org/`.

bug_severity : One of: blocker, critical, major, normal, minor, trivial. Note that these labels have an order.

bug_status : One of: ASSIGNED, NEW, REOPENED, RESOLVED, VERIFIED.

priority : One of: '–', P1, P2, P3, P4, P5. Where P1 has the highest priority and '–' means that no priority has been assigned.

creation_ts : A character string POSIX time stamp of when the bug was first inserted into the Bugzilla database, e.g. `"2011-04-13 17:19:05"`.

reporter : The ID number of the person reporting the bug from the underlying Bugzilla database - for privacy reasons instead of needlessly using email addresses.

component_id : The ID number of the component from the Bugzilla database. There are 45 components used in Firefox, some of which are shared with other Mozilla Corporation products like Thunderbird.

2.4.2 Version Covariates

This subsection describes the covariates and how they were created and extracted. There is one covariate for each *version* of the software as specified by the `Version` field. There are many bugs recorded for each version.

Referring to Section 2.4.2, for each version of Firefox from 5 to 25 we have the following information:

Version : The release version number, e.g. 5.

Release The release version as text, e.g. 'release-15.0'.

FChanged : The number of files changed since the previous version.

LInserts : The number of lines added since the previous version.

LDeletions : The number of lines deleted since the previous version.

releasedate : The release date for this version.

The items: FChanged, LInserts and LDeletions were obtained by extracting the tarballs for the source files for all the versions of Firefox and running a command such as:

```
diff -r moz-18.0/ moz-19.0/ | diffstat -s
```

Where `moz-18.0` and `moz-19.0` are the directory trees for Mozilla Firefox Release 18.0 and 19.0 respectively.

Note that the release dates for versions 23, 24 and 25 are after the cut-off date, 18th of July 2013, for bugs in this database.

Note too that where there is a minor change to a line, e.g. a single character is changed, then that will be recorded here as one line deleted and one line added.

Version	Release	FChanged	LInserts	LDeletions	releasedate
5	release-5.0	3618	72367	62832	2011-06-21
6	release-6.0	4276	90398	83114	2011-08-16
7	release-7.0	4493	92537	78987	2011-09-27
8	release-8.0	7341	74473	70222	2011-11-08
9	release-9.0	4377	109381	73509	2011-12-20
10	release-10.0	6073	148795	122633	2012-01-31
11	release-11.0	4805	108274	76724	2012-03-13
12	release-12.0	4457	123432	84129	2012-04-24
13	release-13.0	4688	150233	106764	2012-06-05
14	release-14.0.1	16835	355941	560277	2012-07-17
15	release-15.0	15728	257432	501298	2012-08-28
16	release-16.0	6349	132879	84268	2012-10-09
17	release-17.0	1051	28867	22179	2012-11-20
18	release-18.0	1315	29593	22844	2013-01-08
19	release-19.0	6133	129714	149368	2013-02-19
20	release-20.0	6393	250433	180606	2013-04-02
21	release-21.0	6840	146689	108744	2013-05-14
22	release-22.0	9569	220244	171637	2013-06-25
23	release-23.0	11149	166918	121491	2013-08-06
24	release-24.0	6924	153191	115052	2013-09-17
25	release-25.0	10933	248838	201130	2013-10-29

Table 2.2: Covariate Information on Firefox Releases 5 to 25

2.5 Inequality of effort

An analysis of the raw Bugzilla database for Firefox shows that 29,515 unique email addresses recorded 60,801 defects between 2008-Nov-20 when the first of the rapid release bugs was recorded and the 2013-July-18 when the snapshot of the database was taken. Note that of the 29,515 unique email addresses who recorded bugs, an astonishing 24,392 or 83% recorded only one bug. At the other extreme, one particular email address has recorded 551 unique defects.

Clearly with hundreds of million users and only 29,515 unique email addresses recording bugs, there must be bugs that have been seen many times and other bugs that have been seen and not recorded.

Putting it another way, has the reader experienced an issue with Firefox? If so, have they reported this to the Mozilla project?

As a simple illustration of the sorts of analyses that can be done with the Firefox-2013 dataset, we look at the number of people who just report one bug and compare it with the number of people who report more than one bug we can see that across all versions of the dataset, that as time progresses that fewer people are doing more of the work. Refer to Figures 2.4 to 2.8. A loess smoother[30] has been added to these plots. The extent of this can be measured using the Gini coefficient (Gini, 1912; Ceriani and Verme, 2012), which is normally used for measuring income inequality. In it's common usage, a Gini Coefficient of 1.0 means that all of the wealth of a country is concentrated in one person and a coefficient of 0.0 means that the wealth is perfectly evenly distributed. So in the normal context of the Gini coefficient, a time increasing Gini coefficient increasing means that that the wealth of that country is becoming concentrated in fewer people.

In this case, the higher the Gini coefficient, the more that a small number of people are reporting more bugs, and also that a large number of people are reporting only one bug.

In the Firefox-2013 dataset there are a total of 1,761 separate email addresses[31] who recorded 10,420 bugs. However, the top ten testers recorded 2,397 bugs, or 23% of the total, and 1,103 testers only recorded one bug. This corresponds to a Gini coefficient of 0.758, Figure 2.4. If the Gini coefficient was zero, then there would be a single vertical bar in this graph, i.e. everybody would be reporting the same number of bugs.

Curiously, the Gini coefficient for bug reporting email addresses seems to be increasing over time, see Figure 2.8. The implication of an increasing Gini coefficient is that there is a smaller and smaller group of testers who are doing more and more of the work and that the majority of the testers recording bugs only record a small number of bugs.

[30]R function '`stats::loess()`'

[31]Here we assume email addresses are a proxy for people

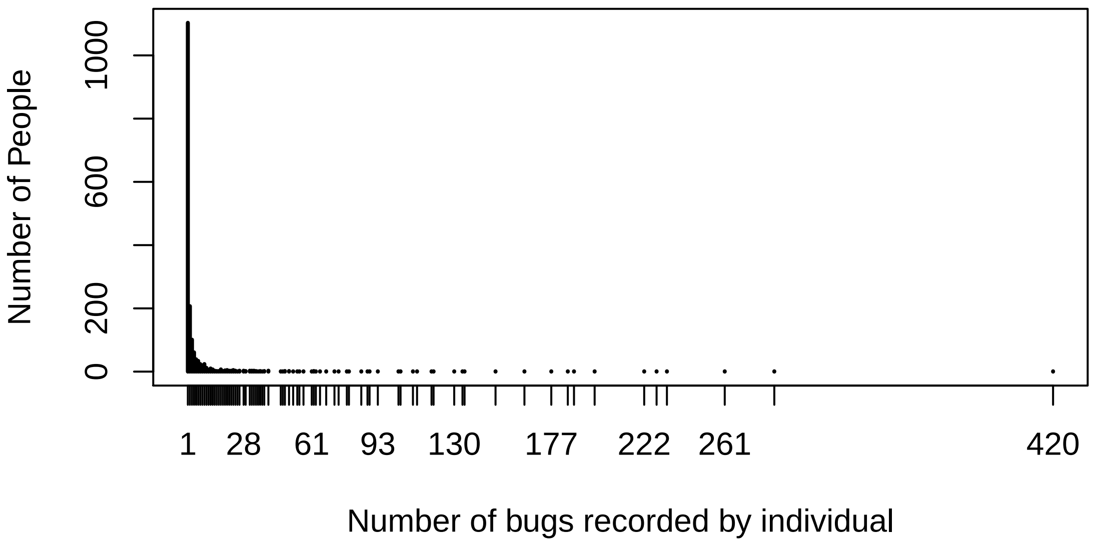


Fig. 2.4: A plot of the number of bugs reported per individual. Many people reported only one bug, one person reported 420 bugs.

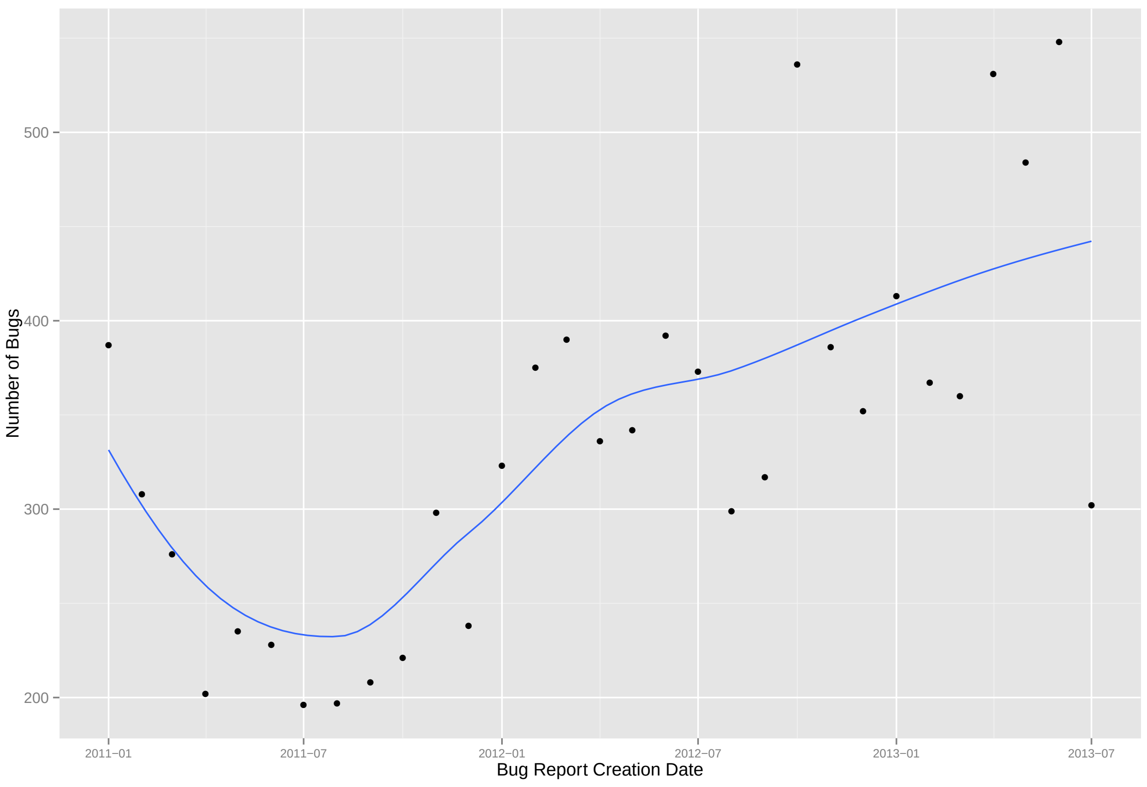


Fig. 2.5: Number of Bugs logged to Bugzilla by Month with a loess smoother.

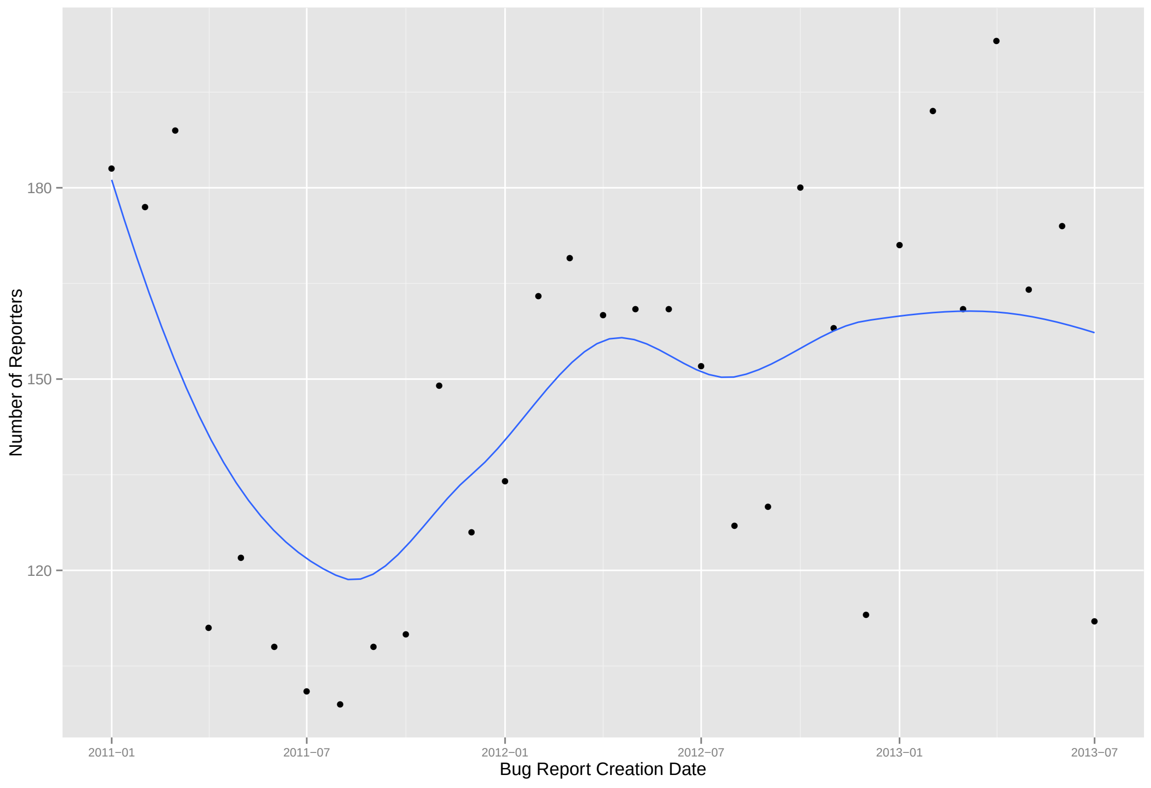


Fig. 2.6: Number of Reporters logging bugs to Bugzilla by Month with a loess smoother.

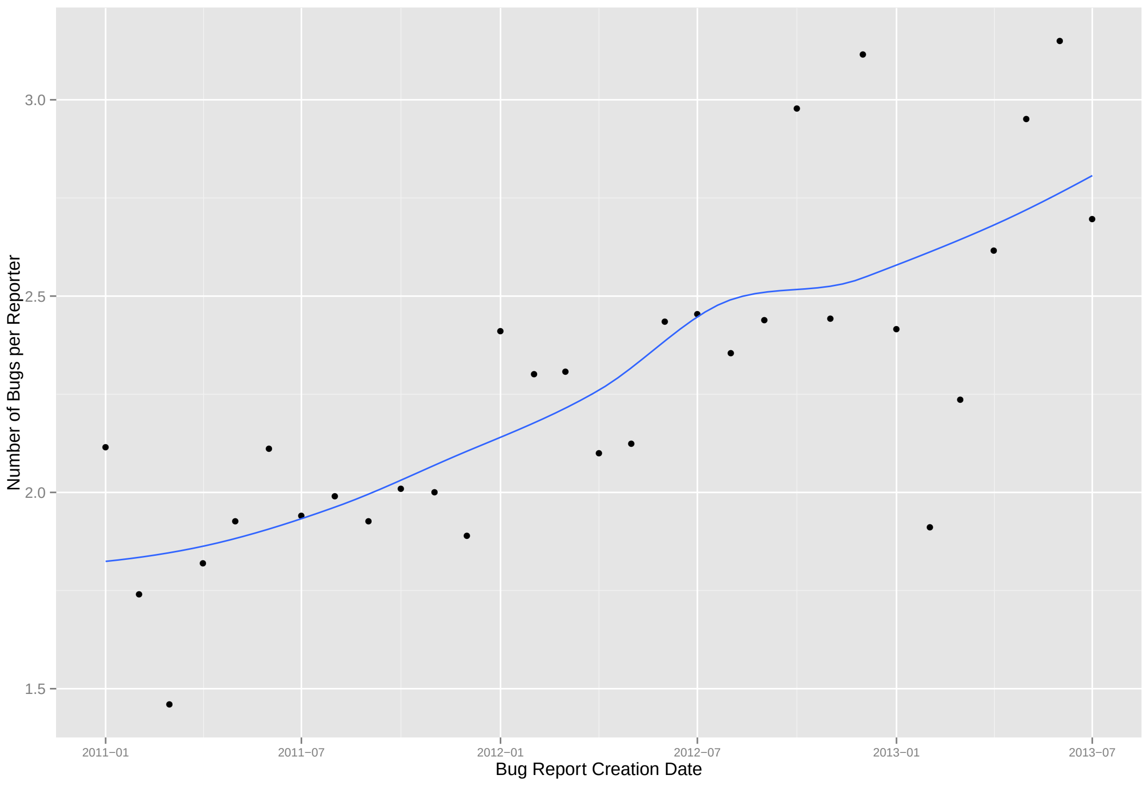


Fig. 2.7: Number of Bugs per Reporter logging bugs to Bugzilla by Month with a loess smoother.

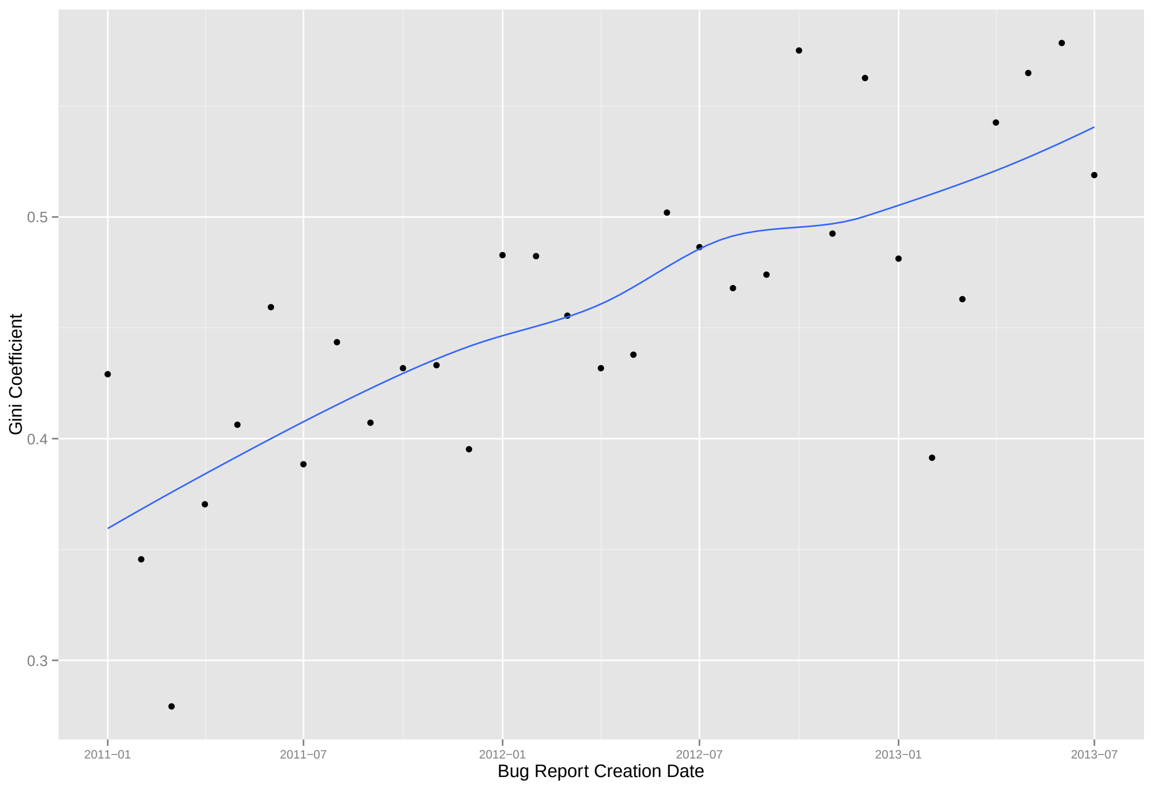


Fig. 2.8: Gini Coefficient of Effort by reporters logging bugs to Bugzilla by Month with a loess smoother.

Chapter 3

Statistical Theory

3.1 Introduction

This chapter is intended as a compendium of the methods used in later chapters in order to allow the latter to be more readable. The interested reader might consider referring back from those later chapters when they are looking for more detail. This chapter reviews statistical inference and the Bayesian approach, it outlines Monte Carlo theory and the Markov Chain Monte Carlo (MCMC) methodology. The theory of classification is briefly reviewed and the distinction between generative and declarative models reviewed. Finally some background on the modelling of software reliability models is covered.

3.2 Decision Making

When there is no uncertainty, decision making is easy. However, in real life there is always uncertainty about the future. All decisions must be made under some uncertainty since we cannot predict the future. Even if were to know the lottery numbers in advance of the draw, the world is an unpredictable place and the lottery may be cancelled! French (1986); Lindley (1985) are two good and very readable texts on decision making.

Following Gelman et al. (2013) very closely, in order to make a decision in a Bayesian context we must follow four steps:

1. Enumerate the entire decision space, i.e. all possible decisions, d and all possible

outcomes, x.

2. Determine the probability distribution of x for each possible decision occurring, i.e. $p(x|d)$, the conditional posterior distribution.

3. Define a utility function, $U(x)$ which maps the outcomes to some real life value, e.g. a monetary value. For example, if we make decision d_1, then the net value is 800, but if we make decision d_2, then the net value is 700.

4. Compute the expected utility, $\mathbb{E}(U(x)|d) = \sum_x U(x)p(x|d))$ and chose the decision which maximises the expected utility. Where the decisions are discrete, e.g. d_1 we will buy a red car, or d_2 we will buy a black motorcycle, then we can create a decision tree and we calculate the expected utility of each of the end nodes of the tree and pick the node with the highest utility. In a continuous setting, e.g. on what future date should we release this software given our current knowledge, then we might have a utility function of time and we optimize for the maximum value of utility to find the time when the software should be released. Note that decision trees can be of arbitrary depth and complexity.

While in this work we only consider one-step-ahead decisions, more complicated processes for software release decisions also exist, refer McDaid (1998); McDaid and Wilson (2001); Singpurwalla and Wilson (1999).

3.3 Bayesian Inference

Bayes' theorem dates back to somewhere between 1746 and 1749 years and was published posthumously by his friend Richard Price who recognized its importance, (McGrayne, 2011). In short it says that:

$$\text{Posterior probability} \quad \propto \quad \text{Likelihood} \times \text{Prior belief} \tag{3.1}$$

Laplace independently re-discovered it some fifty years later and used it extensively in a variety of settings. McGrayne (2011) goes on to describe the fascinating story of how Bayes' theorem has been used and suppressed through the centuries. For instance, Turing used prior knowledge of the contents of German encrypted messages to help to probabilistically determine the contents, e.g. "Beacons lit as ordered". However, it was not until Metropolis et al. (1953) and the practical use of *Markov Chain Monte Carlo* that the real value of it was realized in statistical physics, though at the time only places like Los Alamos had computers which were required to use it. Hastings (1970) generalized the algorithm to what is now known as Metropolis-Hastings and allowed asymmetric proposal distributions. However, the larger statistical community was quite unaware of this according to Geyer (2011); Robert and Casella (2011). Despite what are considered to be well known papers now, (Geman and Geman, 1984; Tanner and Wong, 1987), it was not until (Gelfand and Smith, 1990) that it was brought to the wider statistical community.

In modern usage we write Bayes' rule as either of the following:

$$\begin{aligned} p(y|x) &= \frac{p(x|y)p(y)}{p(x)} && (3.2) \\ &= \frac{p(x|y)p(y)}{\sum_y p(x|y)p(y)} && (3.3) \end{aligned}$$

Where the latter is used when we can enumerate all possible values of y. For continuous variables this becomes:

$$p(y|x) = \frac{p(x|y)p(y)}{\int p(x|y)p(y)dy} \tag{3.4}$$

This is often written as:

$$\underbrace{p(\theta|D)}_{\text{Posterior}} = \underbrace{p(D|\theta)}_{\text{Likelihood}} \quad \underbrace{p(\theta)}_{\text{Prior}} \; / \; \underbrace{p(D)}_{\text{Evidence}} \tag{3.5}$$

Where D is the data which is considered fixed and θ are the parameters of the model. Since the denominator is not a function of θ,

$$p(\theta|D) \propto p(D|\theta) \times p(\theta) \tag{3.6}$$

is all that is needed in many practical instances.

A fundamental tenet of Bayesian statistics is that everything that is unknown can be described by a probability distribution, i.e. a random variable, though it might be difficult to describe this probability distribution, refer Bernardo and Smith (2000). Furthermore, as de Finetti (1937) says in his opening paragraph as he outlines the contents of the paper, probability is subjective:

‘Il s’agit, d’une part, de la définition de la probabilité (que je considère comme une entité purement subjective) et de la signification de ses lois, et, d’autre part, des notions et de la théorie des événements et nombres alétoires « équivalents »; . . . ’

Which very roughly translated is:

‘On the one hand this paper is about the definition of probability (which I consider as something purely subjective) and the importance of these laws, and on the other hand about the theory of events and random numbers . . . ’

In theory it is thus easy to evaluate the posterior, unfortunately in many instances it can turn out to be difficult in practice. In the simplest scenario where we only have a few parameters, we might evaluate the likelihood on a grid and multiply by the prior, however, how does one know where to put the grid and how finely it should be spaced? In higher dimensions, it is simply not practical to evaluate on a grid or using quadrature (Acton, 1990; Evans and Swartz, 2000; Press, 2007), so Markov Chain Monte Carlo is one common technique that is used, see Section 3.4.

3.4 Markov Chain Monte Carlo Methods

Having described some of the background above in Section 3.3, Markov Chain Monte Carlo (MCMC) methods are a family of techniques used for statistical analysis in high dimensions. In particular when we have a likelihood that is of low dimensions, say five or less, then it is possible to evaluate it on a grid (possibly an adaptive grid) using standard numerical analysis techniques of numerical integration or quadrature. If we use a fixed grid of g points in each of d dimensions, then the number of function evaluations, N, rises exponentially with the dimension, i.e. $N = g^d$. However in higher dimensions, because of the so called *curse of dimensionality*[1], this is no longer possible because with each extra dimension, calculations grow exponentially with dimension.

In contrast, using Monte Carlo techniques, we are not limited by the number of dimensions but more by the complexity of the geometry of the sample space, since the variance is not a function of the dimension. Here also we specify the number of

[1]`http://en.wikipedia.org/wiki/Curse_of_dimensionality`

function evaluations, N.

$$p(\theta|D) \to \theta_1, \ldots, \theta_n \tag{3.7}$$

$$\mathbb{E}[f(\theta)] \approx \frac{1}{N}\sum_{i=1}^{N} f(\theta_i) \tag{3.8}$$

Following Bolstad (2010, Chapter 6) and Chib and Greenberg (1995), we attempt to find a Markov chain that has the posterior distribution we are looking, $\pi(\cdot)$, for given a long run. In continuous space, A, where A is any measurable subset, and $\pi(\cdot)$ is the distribution of interest, we attempt to find $P(\theta, A)$ such that

$$\int_A \pi(\theta')d\theta' = \int_A \pi(\theta)P(\theta, A)d\theta \tag{3.9}$$

where $P(\theta, A)$ is the transition kernel. Metropolis et al. (1953) and then extended by Hastings (1970) came up with a general way of doing this in practice. As Gilks et al. (1996, Page 6) explains, if we have a proposal function[2], $q(\theta', \theta)$, and a likelihood function, $g(x)$, where x is the current location and θ' is the proposed new location, then to ensure detailed balance or *reversibility*, we need to ensure that

$$g(\theta)q(\theta, \theta') = g(\theta')q(\theta', \theta) \tag{3.10}$$

The Metropolis-Hastings algorithm supplied the solution by only moving with probability:

$$\alpha(\theta, \theta') = \min\left[1, \frac{g(\theta')q(\theta', \theta)}{g(\theta)q(\theta, \theta')}\right] \tag{3.11}$$

In the following, we outline the Metropolis-Hastings algorithm for a single parameter θ, though it is trivial to extend this algorithm to a multi-dimensional $\vec{\theta}$:

1. Pick an initial value, $\theta^{(0)}$, which you think is reasonable for your parameter based on your prior knowledge.

2. Initialize the iteration counter $i = 0$, and then iterate the following many times, say N times:

   (a) Increment the iteration counter i

[2]Note that Chib and Greenberg (1995) describes $q(\theta', \theta)$ as the *candidate-generating density*, defined as $q(\theta', \theta) = \int q(\theta, \theta')d\theta' = 1$ and interpreted as when a process is at the point θ, the density generates a value θ' from $q(\theta', \theta)$.

(b) Generate $\theta' \sim q(\theta^{(i-1)}, \theta')$

(c) Calculate $\alpha(\theta^{(i-1)}, \theta') = \min \left[1, \frac{g(\theta')q(\theta',\theta^{(i-1)})}{g(\theta^{(i-1)})q(\theta^{(i-1)},\theta')}\right]$

(d) Draw a uniform random number $u = \mathcal{U}(0, 1)$

(e) if $u < \alpha(\theta^{(i-1)}, \theta')$ then accept the proposal and let $\theta^{(i)} = \theta'$, else reject the proposal and let $\theta^{(i)} = \theta^{(i-1)}$,

When we have carried out the N iterations we obtain an array $\theta^{(1:N)}$, or a chain. Typically we would carry out the above procedure four times simultaneously on a quad-core CPU from slightly different starting points and using different random number generator sequences to obtain four chains. At one stage there was a lack of agreement in the community as to whether it was better to have one very long chain or a number of shorter chains (Gilks et al., 1996, Section 1.4.4), however with cheaper computing power and multi-core processors, it seems to be general practice now to create multiple chains. At this point, it is normal practice to plot the chains $\theta^{(1:N)}$ on the Y-axis against the iteration count, i.e. $(1 : N)$ on the X-axis and review the chains for *mixing* and convergence, i.e. do all the chains converge to the same approximate value and does it bounce around this value for a while (Gelman et al., 2013). In practice, this almost never works first time, and we adjust the initial values and the parameters of the proposal function, and possibly run longer chains and chains of more than 100,000 iterations are not uncommon. In the case of *Random Walk Metropolis* we might use Gaussian proposal distribution in which case the mean will be the current position and the standard deviation is an adjustable parameter.

When we are happy with the convergence, then Gelman et al. (2013) recommends throwing away the first halves of the chains and merging the remaining half-chains. We can then calculate the mean and quantiles which should approximate our desired distribution.

The well known Gibbs sampler (Geman and Geman, 1984) was been shown to be a special case of Metropolis-Hastings by Gelman (1992).

We further note that this is a field with a wide and diverse literature and much active research is being carried out.

3.5 Hamiltonian Monte Carlo

Neal (2011) gives a clear description of the history of Hamiltonian Monte Carlo(HMC) and then gives a good description of the technique. In this paper he states that the first statistical use of HMC was his own in a tutorial paper in 1993 (Neal, 1993). Informally, HMC uses the idea that if we are making good progress in exploring a probability distribution by looking in one particular direction, then we should keep going in this direction. In contrast Random Walk Monte Carlo (RWMC) will be heading in a good direction and then will pick a new random direction, thus throwing away the information that it has gained about going in a good direction. HMC is done in a similar way as in Hamiltonian mechanics by adding the concept of momentum, potential energy and kinetic energy. Following Neal's description, we have our current position and our speed and as we start to climb, our speed will decrease, our kinetic energy will decrease and our potential energy will increase. To do this the gradient at the current position needs to be calculated and in Stan (Stan Development Team, 2014*a*), this is done using automatic differentiation. So clearly this will only work when we are using continuous and differentiable parameters and will not work for integer valued parameters. Gelman et al. (2013) gives a clear description of the algorithm, which we will skip here for space reasons.

3.6 Markov Chain Convergence

Following Connor and Goldschmidt (2012), we will outline the properties of Markov chains in the discrete case (time and space), but this argument can be extended to the continuous case. A Markov Chain $X = X_0, X_1, \ldots$ has state X_t at time t and X must have the Markov property, that its current state must only depend on its last state, i.e. that it has no memory, $P(X_{i+1}|X_i, \ldots, X_0) = P(X_{i+1}|X_i) \quad \forall\, i \geq 0$.

If a Markov Chain is *irreducible* and *positive-recurrent* then it has a unique equilibrium distribution. If in addition it is *aperiodic*, then the equilibrium distribution is also the limiting distribution.

These well known conditions are:

irreducible

A discrete Markov chain is irreducible if for all i, j, it has a positive chance of visiting j at some positive time, if it starts at i.

positive-recurrent

A discrete Markov chain that returns to a given state i in finite time T is said to be positive-recurrent.

aperiodic

If one cannot divide state-space into non-empty subsets such that the chain progresses through the subsets in a periodic way, then it is aperiodic.

In theory, if these conditions are held to be true, then a Markov Chain will converge to a unique stationary distribution (Meyn and Tweedie, 1993). In theory and in practice if there is detailed balance, then these conditions will hold, (Bolstad, 2010; Chib and Greenberg, 1995).

If $p_{i,j}$ is the probability that the chain will move from i to j, then a Markov chain satisfies detailed balance if there is a non-trivial solution of:

$$\pi_i p_{i,j} = \pi_j p_{j,i} \tag{3.12}$$

where $\pi_.$ is the distribution of interest.

In practice, there are no firm rules to tell us whether convergence will occur in a finite (and reasonable) number of iterations, nor whether we have actually achieved convergence. However, we do have some rules of thumb to guide us: (Gelman et al., 1996; Gelman and Shirley, 2011; Gelman et al., 2013)

- Looking at the trace plot of a given parameter, i.e. a plot of the value on the Y-axis against the iteration number on the X-axis, it should stabilize to a value after some initial instability. Thereafter it should not jump around too much but remain stable with what is known as mixing, i.e. that the parameter value of high density is reasonably covered.

- Run a few chains, commonly four, from different starting points in the parameter space and check that they appear to converge to the same value.

- Following Gelman et al. (2013), the chains are tested by splitting them in half and then testing the variance between the eight halves (if you had four chains).

It is important to note that no methodology is perfect and all tests are susceptible to deception. Particularly in high dimensions, it is not possible to know for certain that a chain or set of chains has converged. We guard against this by initializing the parameters for each chain at different values and when we see the same regions being explored by each of the chains, then this is considered in the community to be indicative of good convergence.

Kruschke (2011, Section 7.1) gives a nice tutorial example of how Metropolis-Hastings works.

When we have a Markov chain, there is almost inevitably some autocorrelation and Gelman et al. (2013, Section 11.5) describes how the effective sample size might be calculated.

3.7 Adaptive Monte Carlo

When doing Markov Chain Monte Carlo (MCMC) analysis, one problem that crops up again and again is *tuning*, the process of persuading the chains to converge in a reasonable time. This process involves, manually running the analysis a number of times and looking at density plots and trace plots for the parameters and seeing whether they make sense and whether the chains converge with good mixing and looking at the acceptance rate for the chains. For a given MCMC setup we do not know the 'best' rate of acceptance, but we have the guidance from Gelman et al. (1996) that a rate between 0.1 and 0.6 is probably optimal, though their research did not use our MCMC chains, it has been found to be reasonable in common practice. As Rosenthal (2011) clearly describes, the process of adjusting the initial values and proposal distributions is generally done in an *ad hoc* manner, they further point out that Metropolis et al. (1953) recognized this early. In high dimensions this is clearly difficult to do manually, so why not try and automate the process?

We could create an optimization 'wrapper' around our code to find better parameters and restart over and over again, in effect mimicking the manual tuning process. The most well known, early attempt at the automatic identification of 'optimal' proposal distributions is probably Haario et al. (2001), but a lot of other work has been done on this and it is still an active research area. Well known work includes Andrieu and Thoms (2008); Roberts and Rosenthal (2009) and indeed the first of which was in a special issue of *Statistics and Computing* on Adaptive Monte Carlo Methods, with guest editor Paul Fearnhead. There is even an R library which implements many of these methods [3].

It is important to point out that one key difficulty with adaptive MCMC is that it is easy to lose the important properties of MCMC and thus invalidate the work, e.g. if we change the proposal distribution after 100 iterations, then we strongly risk breaking reversibility and detailed balance. Roberts and Rosenthal (2007) gave some important results relating to adaptive MCMC whereby if there is diminishing adaption and containment, in particular the former, then Ergodicity should hold adaption and thus the validity of using summaries of the full chains for inference.

An alternative approach is to run one of the many adaption algorithms to find a better set of proposal parameters, and then to run from scratch using plain, non-adaptive MCMC.

3.8 Imputation of Missing Data

Data can be missing from a dataset, and it is important to understand that there are different ways of describing this 'missingness' (Carpenter and Kenward, 2013)[4].

Missing data can be Missing Completely At Random (MCAR) where there is no pattern what so ever to the missingness. Missing at Random (MAR) describes the case where the factors that cause the data to be missing are independent of our covariates. Finally, data can also be missing in a way which is dependent on our covariates - this is a serious issue and more difficult to manage - Missing Not At Random (MNAR).

Following (Carpenter and Kenward, 2013), if we have a dataset $\underline{Y}$, and we say that $\underline{Y} = \{\underline{Y_o}, \underline{Y_m}\}$, where $\underline{Y_o}$ is observed and $\underline{Y_m}$ is missing. We further say that R is an indicator variable, which is zero if an observation is missing and 1 if it is observed. Thus we denote the probability that a set of values are missing given the values taken by the observed and missing observations, as $P(R|\underline{Y_o}, \underline{Y_m})$. Then we can say for MCAR that $P(R|\underline{Y_o}, \underline{Y_m}) = Pr(R)$. While for MAR that $P(R|\underline{Y_o}, \underline{Y_m}) = Pr(R|\underline{Y_o})$. Unfortunately, we cannot make assumptions about missing data which is MNAR.

[3]`http://www.bayesian-inference.com/software`

[4]`www.missingdata.org.uk`

When we have missing data, one possibility is to just pretend we did not know it was there in the first place which is commonly done. However, this can lead to a loss of efficiency because of the reduced size of your database. Rubin (1977, 1987) was the first to come up with the idea of *multiple imputation* which seeks to replace the missing data with a distribution and when this is done a number of times, then the analyst can examine the effect on the outcome to get a better understanding of the full dataset and the missing data. Further important work in the area was also carried out by Tanner and Wong (1987); Tanner (1996).

In an MCMC context where we have missing data, it is generally not difficult to replace missing data with a distribution and in each iteration we sample from that distribution, and calculate our statistics on a nominally full dataset - except that in each successive iteration the dataset will be different because of random sampling. The advantage of this is that we will better understand the variability of the statistics we wish to calculate than if we had discarded the missing data. A difficulty arises in determining the distribution to use for the missing data. How this is done will depend on whether we think that the missing data are missing at random (MAR) or missing completely at random (MCAR). Special care is needed when we think that the data are missing not at random (MNAR). In the context of recent election polling results in the UK, it has been suggested that one of the reasons for the poor predictions was that a larger proportion of Conservative voters who were polled either refused to answer or answered that they would vote Labour than *vice versa*.

3.9 The Recursive Property of Bayes' Theorem

If we take a dataset D_1, a prior P_0 and put these into a model M, we generate the posterior P_1. When we get a new updated dataset, D_2, we know more about the dataset and we can use the posterior P_1 above as an informed prior. Using P_1 as the *prior* for the updated dataset D_2, we then generate a new posterior P_2.

Alternatively, we can put the two datasets D_1, D_2 together in an appropriate fashion and use the same uninformative prior P_0 and the result will be theoretically identical to the above posterior P_2. This has the nice advantage that we do not have to sample from the empirical distribution P_1 to be able to get to P_2 in the first scenario.

3.9.1 Proof of Recursive Property of Bayes' Theorem

By the direct application of Bayes' Law to our problem we say:

$$P(\theta|D_1, D_2) = \frac{P(D_1, D_2|\theta) \times P(\theta)}{P(D_1, D_2)} \tag{3.13}$$

Where the term $P(D_1, D_2|\theta)$ is known as the 'likelihood', $P(\theta)$ is known as the 'prior' and the denominator, $P(D_1, D_2)$ is a constant, since it doesn't involve the parameter θ.

In the case where the data D_1, D_2 are independent (technically this is not the case in our scenario), we can say the following:

$$\begin{align}
P(\theta|D_1, D_2) &= \frac{P(D_2|\theta)P(D_1|\theta)P(\theta)}{P(D_1, D_2)} \tag{3.14}\\
&= \frac{P(D_1)}{P(D_1)} \times \frac{P(D_2|\theta)P(D_1|\theta)P(\theta)}{P(D_1, D_2)} \tag{3.15}\\
&= \frac{P(D_1)}{P(D_1, D_2)} \times P(D_2|\theta)\left[\frac{P(D_1|\theta)P(\theta)}{P(D_1)}\right] \tag{3.16}\\
&= \frac{P(D_1)}{P(D_1, D_2)} \times P(D_2|\theta) \times P(\theta|D_1) \tag{3.17}\\
&= \text{Constant} \times \text{Likelihood} \times \text{Prior} \tag{3.18}
\end{align}$$

In the case where D_1, D_2 are not independent which is true in our case, the importance of this depends on how different the *Likelihood* term is from the independent case.

$$P(\theta|D_1, D_2) = \frac{P(D_1|\theta) \times P(D_2|\theta, D_1) \times P(\theta)}{P(D_1, D_2)} \tag{3.19}$$

In the case where D_1, D_2 are independent, the term $P(D_2|\theta, D_1)$ is, by definition equal to $P(D_2|\theta)$.

3.10 Non-Homogeneous Poisson Process

Closely following the definition in Ross (1996, Chapter 2.4), a Non-Homogeneous Poisson process is a counting process $\{N(t), t \geq 0\}$ with a rate (intensity) function $\lambda(t)$ for $t \geq 0$ if:

1. $N(0) = 0$.

2. $\{N(t), t \geq 0\}$ has independent increments.

3. $P\{N(t+h) - N(t) \geq 2\} = o(h)$

4. $P\{N(t+h) - N(t) = 1\} = \lambda(t)h + o(h)$

where by definition[5] a function f is said to be $o(h)$ if

$$\lim_{h \to 0} \frac{f(h)}{h} = 0 \tag{3.20}$$

However, going back to basics, a plain Poisson process looks like:

$$P[N(t) = k|\lambda] = \frac{(\lambda t)^k}{k!} e^{-\lambda t} \tag{3.21}$$

A Non-Homogeneous Poisson Process (NHPP) has a rate $\Lambda(t)$ which is a function of time, so it looks like:

$$P[N(t) = k|\Lambda(t)] = \frac{\Lambda(t)^k}{k!} e^{-\Lambda(t)} \tag{3.22}$$

In our case that is an estimate of the number of bugs that will be discovered by time t. In our case, we have the actual times of the N bugs up to time T, i.e. we have observed the values of the times:

$$t_1 < t_2 < \ldots < t_n \tag{3.23}$$

up to time T, for $T >= t_n$ and thus we can write the likelihood of this as:

$$\prod_i \Lambda(t_i) \times e^{-\Lambda(T)} \tag{3.24}$$

3.11 Hierarchical Models

Gelman and Hill (2007, Chapter 11) introduce *multilevel structures* in a text book situation and they describe these models as:

> '... extensions of regression in which data are structured in groups and coefficients can vary by group.'

[5]Ross (1996, Chapter 2, page 60)

In other words there are layers of groups of data which are related as in pupils, classroom, school, district.

Lindley (1969*a*) appears to be the first reference to Bayesian hierarchical regression and this is followed by a series of other papers, (Lindley, 1969*b*, 1970), published by the Educational Testing Service (ETS) in Princeton under Melvin Novak who was the principal investigator.

The ETS publications were followed up by Lindley and Smith (1972) though it appears to be similar to non-Bayesian work by Zellner and Theil (1962). Other papers followed by Aitkin et al. (1981) who apply the technique in an educational setting as does the slightly better known paper by Raudenbush and Bryk (1986), though this appears to be significantly pre-dated by Novick et al. (1972), who cite Lindley and Smith (1972) as the originator.

In our situation, we have multiple versions and we make the assumption that the parameters for each version come from a pool which has its own distribution, e.g. we will later discuss the model by (Goel and Okumoto, 1979) which has two parameters (a, b), and when used in this situation, we will be assuming that there is a distribution for each of (a, b) and that the versions are interchangeable and un-ordered. We might assume that the (a, b) are each log-normally distributed, with given means (μ_a, μ_b) and variances (σ_a^2, σ_b^2). For a particular version, then we will get a particular (a_k, b_k) for which, in a Bayesian setting, we will see a distribution on that pair of values. We might then be trying to estimate the distribution of the 'parent' (a, b).

The clear advantage of hierarchical modelling in a Bayesian setting is the ability to set the priors in a meaningful way and 'share strength'.

Exchangeability is the concept closely related to *Independent and Identically Distributed (IID)*, and informally says that given a set of data points, the order doesn't matter. Clearly exchangeability does not apply for time series or in many spatial models, but it is quite widely applicable. Exchangeability is generally attributed to de Finetti (1929, 1974), though strictly speaking the representation theorem only applies as the data set size goes to infinity. More formally, if we have data, $x_1, x_2, \ldots, x_n$, then $p(x_1, \ldots, x_n)$ is invariant to any permutation of the subscripts $\{1, \ldots, n\}$. Exchangeability is particularly important in the context of hierarchical models as it allows us to make inference based on the hierarchical parameters. For instance, if we look at a set of data based on different versions of software, and another version is given to us, then under the assumption of exchangeability, we can infer the properties of this new version.

As it turns out, it is a bad idea to use uninformative priors for the hyper-parameters, i.e. the hyper-priors, as discussed in Hobert and Casella (1996) and Bolstad (2010, Chapter 10). If we were to use an uninformative prior such as a Jeffreys prior for the hyper-parameters, then the joint posterior will be improper and though the output might appear to be reasonable, it can converge to the wrong value because the Markov chain will be null recurrent rather than positive recurrent.

3.12 Model Comparison

When we have two statistical models using the same dataset, it is useful to be able to objectively compare the effectiveness of the two models. There are a number of different methodologies used, in the field of machine learning, the effectiveness of prediction is prioritised, i.e. given a subset of the original data which was not used for generating the models, which model is in some sense 'better' at predicting the known values that we have, whether in regression or classification (Kuhn and Johnson, 2013), i.e. the out of sample predictive accuracy. In Bayesian statistics there is a preference for methodologies based on information criteria such as AIC (Akaike, 1974) and BIC (Schwarz, 1978), or so called *Bayes factors* which are credited to Jeffries (1939) but significant work also goes to Kass and Raftery (1995); DiCiccio et al. (1997); Han and Carlin (2001); Lavine and Schervish (1999). It is important to note that all of the above methodologies require that the exact same dataset is used. Using Bayes factors to compare two models, (M_1, M_2) for the same data set is done as follows:

$$\text{BF} = \frac{P(\text{Data}|M_1)}{P(\text{Data}|M_2)} \tag{3.25}$$

Kass and Raftery (1995) suggest that the if the value of BF is greater than 20, then there is strong evidence for M_1.

If we have two possible models, then the motivation for this is as follows:

$$P(M_1|\text{Data}) = \frac{P(\text{Data}|M_1)P(M_1)}{P(\text{Data}|M_1)P(M_1) + P(\text{Data}|M_2)P(M_2)} \tag{3.26}$$

$$P(M_2|\text{Data}) = \frac{P(\text{Data}|M_2)P(M_2)}{P(\text{Data}|M_1)P(M_1) + P(\text{Data}|M_2)P(M_2)} \tag{3.27}$$

Looking at the ratio between the priors and the posteriors:

$$\frac{P(M_1|\text{Data})}{P(M_2|\text{Data})} = \underbrace{\frac{P(\text{Data}|M_1)}{P(\text{Data}|M_2)}}_{\text{Bayes Factor}} \frac{P(M_1)}{P(M_2)} \tag{3.28}$$

$$\text{Posterior Ratio} = \text{Bayes Factor} \times \text{Prior Ratio} \tag{3.29}$$

Note that $P(M_1|\text{Data}) + P(M_2|\text{Data}) = 1$.

By the partition law we have:

$$P(\text{Data}|M_1) = \int_{\text{Parameters}} \underbrace{P(\text{Data}|\text{Parameters in } M_1)}_{\text{Likelihood of } M_1} \times \underbrace{P(\text{Parameters})}_{\text{Prior}} \tag{3.30}$$

$$= \mathbb{E}\left(P(\text{Data}|\text{Parameters in } M_1)\right) \tag{3.31}$$

$$\approx \frac{1}{K}\sum_k P(\text{Data}|\text{Parameters}^{(k)}\text{in} M_1) \tag{3.32}$$

$$\text{Parameters}^{(k)} \leftarrow P(\text{Parameters}) \text{ Prior} \tag{3.33}$$

Unfortunately if we calculate the Bayes factors in this way, then the variance of the estimate from MCMC is very high.

$$P(M_1|\text{Data}) = \int P(M_1|\text{Params})P(\text{Params}|\text{Data}) \tag{3.34}$$

Thus, we use the method of 'Harmonic Mean', (Kass and Raftery, 1995, Section 4.3). If we have k iterates through MCMC samples

$$P(\text{Data}|M_1) \approx \left[\frac{1}{K}\sum_k \frac{1}{P(\text{Data}|\text{Parameters})}\right]^{-1} \tag{3.35}$$

Where

$$\text{Parameters}^{(k)} \leftarrow P(\text{Parameters}|\text{Data}) \tag{3.36}$$

$$k = 1, \ldots, K \tag{3.37}$$

In practice we compute

$$\left[\frac{1}{M}\sum_{m=1}^{M}\frac{1}{\prod_{k=1}^{K} L(\underline{t}_k|a_k^{(m)}, b_k^{(m)})}\right]^{-1} \tag{3.38}$$

Where

$$\log(L) = -\, b_k^{(m)}\sum \underline{t}_k + n_k\left[\log(a_k^{(m)}) + \log(b_k^{(m)})\right] - a_k^{(m)}(1-\exp(-b_k^{(m)}T_k)) \tag{3.39}$$

as before and where $(a_k^{(m)}, b_k^{(m)})$ are samples of (a_k, b_k) from the respective MCMC iteration $m = 1, \ldots, M$.

3.13 Classification Background

In *Supervised learning*, which Murphy (2012) calls *predictive learning* we aim to learn how to map inputs, x to outputs, y, given a training set, i.e. given a new input x', how can we predict a new output y'? When y is categorical we call the process classification or pattern recognition and when y is real-valued we call the process regression. An example of categorical supervised learning is where we have photographs of men and women with a gender label associated with each one and we are then asked to be able to predict the gender of a new unlabelled photograph.

In contrast in *Un-Supervised learning*, which Murphy (2012) calls *descriptive learning*, we are looking for “interesting patterns” in the data. Barber (2012, Section 13.2) describes both supervised and unsupervised learning as mature fields. As Hastie et al. (2009, Chapter 14) point out, un-supervised learning is like learning without a teacher and the goal is to make some inference on the properties of the probability distribution of the data. Bishop (2006, p.3) points out that the goal may be to find clusters of similar examples within the data, or to estimate densities, or to project down from high dimensional space for visualization.

We use *Semi-Supervised Learning* when we have some labelled data, which might have been hand labelled and we wish to infer labels for a large quantity of unlabelled data. As described in Chapelle et al. (2006, Section 1.1.2), this is like unsupervised learning where the small quantity of labelled data gives some guidance. Other forms of semi-supervised learning allow for constraints to be placed on the output labels.

In our case we have labelled and unlabelled data but we also are prepared to make some assumptions about the unlabelled data. We assume that bugs for a particular version are more likely to be closer to the nominal release date for that version than those from a release which is long beforehand or long afterwards. We further assume that the density of bugs is 'bell-shaped' around a nominal date which is at or near the official release date - which is known for each of the releases. We will discuss this further in Chapter 5.

Transduction is a term introduced by Vapnik (1998) to distinguish between a learning process which can generalize from a set of points to be able to predict anywhere from a process which is only capable of inferring the values at specific points. Vapnik insists that one should not create a system which was too general as that is wasteful.

More formally, if we have a data set of n points, $\mathcal{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ with labels $\mathcal{Y} = (y_1, \ldots, y_n)$, then in un-supervised learning we try to infer y_i from $\mathbf{x}_i$ with no knowledge of y_i. In supervised learning we know the values of y_i for the training set and we try to infer some predictive knowledge of why $\mathbf{x}_j$ is labelled as y_j, in order to be able to predict for a new $\mathbf{x_k}$ with unknown label y_k. In contrast, during un-supervised learning we only have some labels in our training set. This can occur if, for instance, it requires a human to look at the data to say what class a given $\mathbf{x}$ is and it is comparatively expensive in time and/or money to do this.

Assessing the performance of classification models is a large and active field of research, we the interested reader to Kuhn and Johnson (2013); Hastie et al. (2009); Agresti (1990). According to Kuhn and Johnson (2013, p.254), a Confusion Matrix is '. . . a simple cross-tabulation of the observed and predicted classes for the data.' In the simple binary class case we have four possible cases, True Positives (TP), True Negatives (TN), False Positives (FP) otherwise known as Type-I errors and False Negatives (FN) otherwise known as Type-II errors. In the following, $\sum$TP and $\sum$TN correspond to the count of TP and TN cases respectively. Similarly $\sum$P and $\sum$N are the count of all Positive and Negative cases respectively.

$$\text{Accuracy} = \frac{\sum \text{TP} + \sum \text{TN}}{\sum P + \sum N} \tag{3.40}$$

$$\text{Kappa} = \frac{O - E}{1 - E} \tag{3.41}$$

Where O is the Observed accuracy, and E is the Expected accuracy.

Given a confusion matrix, there are many ways to summarize the performance of the classification model. We understand that it is still an open research question as to whether it is possible to summarize a confusion matrix into a single measure of performance. Frankly we consider it to be unlikely given the number of degrees of freedom in a square confusion matrix that has 22 levels. By this we mean that there are measures such as the imbalance in the true counts of each level, the symmetry and others which need to be taken account of in any performance measure. We have chosen to use the Kappa measure as implemented in the function `confusionMatrix()` in the R `caret` package. In general, the Kappa statistic attempts to highlight the real gain in the performance of the classifier compared to a classifier which was completely random, or as Agresti (1990, p.366) says "is the excess of the observer agreement over that expected purely by chance, i.e. if the ratings were statistically independent."

3.14 Software Reliability

In general, software reliability models assume that a single piece of software is developed by a homogeneous team who then pass it over to a single homogeneous test team who work on it and record all the bugs they find perfectly and as soon as they "occur". Any bugs found are assumed to be fixed instantaneously and perfectly, i.e. the fixing of the first bug does not introduce any new bugs. More recent work, e.g. Ruggeri and Soyer (2008), looks at imperfect debugging where new bugs are introduced with a small probability when an old bug is being fixed.

In the Open Source world, most people who record a bug do not work for the organization and only ever record one bug; a very small group of testers record a large number of bugs.

As can be seen in Figure 2.2, there are four distinct regimes of bug finding: an initial flurry of activity in October 2011, a steady stream of bugs until release date with clear curvature downwards, then the line straightens and the rate at which bugs are recorded increases until the middle of March when version 11 of Firefox was released, thereafter is the fourth phase where the rate of bug recording is significantly slower than previously - no doubt due to the fact that relatively few people are using it and most have the option of upgrading to a later version.

The development process for open source software is not as clear cut as elsewhere and testers can pull development code at any time and build it for themselves and record bugs. These testers (and indeed the developers) can be anywhere in the world and are not necessarily competent. Many bugs are incorrectly recorded in the bug database and some of these errors are detected and corrected by the triage team, and others by developers. Some of these incorrect bug reports are never corrected.

Another big difference between models described in the literature and modern Open Source practice is that the older literature expects there to be one computer running one instance of *the* software and there are no other instances of relevance. We all know the phones in our pockets are more powerful than these old 1970s era mainframe computers, and those same phones can also run Mozilla Firefox. According to Mozilla there are 500 million people around the world who use Firefox[6] on all sorts of devices from Desktop PCs to laptops to tablets to phones and a variety of other devices. Statistics are available on the use of older versions of Firefox[7].

3.14.1 Models for Software Reliability

The statistical modelling of software reliability has a long history since the seminal paper of Jelinski and Moranda (1972) to more recent times. Goel and Okumoto, which we will explore in more detail later, was one of the first Non-Homogeneous Poisson Process (NHPP) models and this work has since been extended by for instance Jeske and Pham (2001) who identified difficulties with the MLE estimates of the parameters. Yamada et al. (1983) take this analysis further. Other significant references in the domain include Littlewood and Verrall (1973); Littlewood and Mayne (1989); Musa (1975); Musa et al. (1987); Musa (2004). Thorough reviews have been written on the topic of software reliability including Singpurwalla and Wilson (1994, 1999) and Pham (2010). While Soyer (2011) is a recent survey article.

Following (Singpurwalla and Wilson, 1994), there are two basic kinds of model, Type-I: those that model the inter-failure time which as Soyer (2011) notes is often accomplished using a failure rate function of time; Type-II: those that model the failure count at a given time which are point processes such as a Poisson process with mean value $\Lambda(t)$.

Note that many of these models make the explicit assumption that any errors found

[6]http://blog.mozilla.org/press/ataglance/

[7]`http://www.w3schools.com/browsers/browsers_firefox.asp`

are instantaneously repaired; for the case where software has been released, then this does not make sense - once released on the Internet it cannot be recalled. To be fair though, the Internet did not exist when these models were created.

Some examples of models include:

- Jelinski and Moranda (1972) is a Type-II model and used a homogeneous Poisson process and is considered to be the first software failure rate model. Many other Type-II models are variants on this so the assumptions that it made are important and according to Pham (2010) these are:
  - There are a fixed but unknown number, N, of faults in the program.
  - Faults are independent and do not affect other faults.
  - The inter-fault time is independent.
  - Faults are removed instantaneously and reliably.
  - The failure rate $\lambda(t)$ is constant in a failure interval and proportional to the number of faults remaining. So

$$\lambda(t_i) = \phi[N - (i - 1)], \quad i = 1, \ldots, N \tag{3.42}$$

- Goel and Okumoto (1979) is another Type-II model and the first to use a non-homogeneous Poisson process. We will discuss this model in more depth in Chapter 4.

- Yamada et al. (1983) describes an S shaped reliability growth model based on a non-homogeneous Poisson process where the mean value function is

$$\Lambda(t) = K\left[1 - (1 + \lambda t)e^{-\lambda t}\right] \tag{3.43}$$

  where t is time, λ is the error detection rate and K is the total number of bugs. The assumptions made in the model are:
  - failures occur at random times.
  - The time between failures $(k-1)$ and k depend on the time to failure $(k-1)$.

  As is common in these older papers, the language used makes the failures sound like hardware failures.

One class of model that can be both a Type-I or a Type-II model is known as ‘imperfect debugging’ where there is an assumption that there is a small probability that a bug fix will fail or introduce another bug.

More recent work includes:

- Ray et al. (2006) which looks at the use of covariates to improve modelling.
- Ruggeri and Soyer (2008) considers a hidden Markov model which assumes failure times are exponentially distributed with parameters depending on an evolving latent variable. They also look at using a self-exciting point process for modelling bug fixing, which looks like a really good idea, since of the order of 20% of bug fixes create new bugs, (Yin et al., 2011).
- Ravishanker et al. (2008) looks at Markov switching with a multivariate model to try to model the more modern iterative software development practices found in industry.
- Pievatolo et al. (2010, 2012) uses a Hidden Markov Model (HMM) to look at how debugging does not reliably fix bugs.

The books Singpurwalla and Wilson (1999); Musa et al. (1987); Pham (2010) go into considerable detail on many aspects of software reliability modelling and compliment each other.

3.14.2 Modelling Multiple Version Reliability

There is very little published work on the modelling of the reliability of multiple versions. Musa et al. (1987, p.165) briefly mentions the issue of multiple versions, but there is no specific modelling done. Musa et al. (1987, Section 6.3) discusses *Evolving Programs*, i.e. programs that are changing in a number of ways and the issues around modelling their reliability. Pham (2010) touches on the subject, but does not explore it in any depth.

Singpurwalla and Soyer (1985) refers to multiple versions, but by this they are assuming instantaneous bug fixing upon first sight of a bug and that the period between each bug is a new version - quite a different scenario to what we discuss.

3.15 Modelling Defect Identification

It should be emphasized at this point that the data that we are attempting to model are the time stamps of the first recording of bugs. By this is meant that the time element is the moment that this defect was first recorded in the Bugzilla database. There is no attempt to model when or if these defects will be fixed nor is this an attempt at identifying when a bug was first seen.

It should also be emphasized that where a problem is seen more than once, that this will be marked as a duplicate when after being recorded in the database and will be ignored in our analysis.

Chapter 4

Goel-Okumoto

4.1 Introduction

This chapter reviews the seminal paper by Goel and Okumoto (1979) and looks at appropriate extensions to this model where projects have multiple versions of software released sequentially.

4.2 The Goel and Okumoto model

The paper by Goel and Okumoto (1979) was ground-breaking in modelling the failure phenomenon with a Non-Homogeneous Poisson Process (NHPP). This model has been heavily studied over the last 35 years and as of June 2015 there were 1490 citations on Google Scholar and 615 citations in the Web of Science core collection. The model makes the quite reasonable assumption that the rate bugs are found is proportional to the number remaining undiscovered in the system, i.e. when there are few bugs left, they are harder to find, so at a constant level of effort, the interval between finding bugs will increase as time goes on. If $\Lambda(t)$, known as the *mean value function* (MVF), gives the expected number of bugs detected in the system at time t, then Goel and Okumoto (1979) assumed that:

$$\frac{d}{dt}\Lambda(t) \;\propto\; a - \Lambda(t) \tag{4.1}$$

$$= b(a - \Lambda(t)), \qquad \Lambda(0) = 0 \tag{4.2}$$

Which has the solution:

$$\Lambda(t) = a(1 - e^{-bt}), \tag{4.3}$$

where a is the total number of bugs in the system and b can be interpreted as a measure of how hard the testers are working and finding bugs, or as Goel and Okumoto put it "the occurrence rate of an error".

Note that there is an implicit assumption that the rate of testing work is assumed constant and that the system does not change during this period.

More formally, if $\mathbb{E}\left\{N(t)\right\}$ is the expected number of bugs that have been discovered at time t, and $\mathbb{E}\left\{\bar{N}(t)\right\}$ is the expected number of bugs remaining undiscovered, then $\mathbb{E}\left\{N(t)\right\} + \mathbb{E}\left\{\bar{N}(t)\right\} = a$. The expected number of errors remaining in the system at time t is $\mathbb{E}\left\{\bar{N}(t)\right\} = ae^{-bt}$ by inspection.

The intensity function, which they also call the *error-detection rate*, $\lambda(t)$ of the model is:

$$\lambda(t) = \Lambda'(t) = abe^{-bt} \tag{4.4}$$

An illustration of the MVF can be seen in Figure 4.1 where the full NTDS dataset (Goel and Okumoto, 1979) is plotted with the cumulative bug count on the Y-axis against time in days on the X-axis. The parameters used in the plot were the MLE parameters published in Goel and Okumoto (1979) of $(a = 34, b = 0.00579)$ which is based on the first 26 points in the dataset. The dashed vertical line shows the nominal date when the production phase was finished at about 250 days and the software was released for testing after 26 had been discovered. A further 8 bugs were subsequently discovered in the system. This model appears to be a good fit for this dataset.

If $N(T)$ is the cumulative number of bugs detected to time T, and $N(T)$ is a non-homogeneous Poisson process with mean value function $\Lambda(T)$, then:

$$\begin{aligned} P\left[N(T) = n | a, b\right] &= \frac{[\Lambda(T)]^n}{n!} e^{-\Lambda(T)} && (4.5)\\ &= \frac{a^n}{n!}\left(1 - e^{-bT}\right)^n e^{-a\left(1-e^{-bT}\right)} && (4.6)\end{aligned}$$

This is appropriate if we know how many bugs occurred from $t = 0$ to $t = T$, however, if we also know what times these n bugs occurred, then it is appropriate to use this information and thus we say:

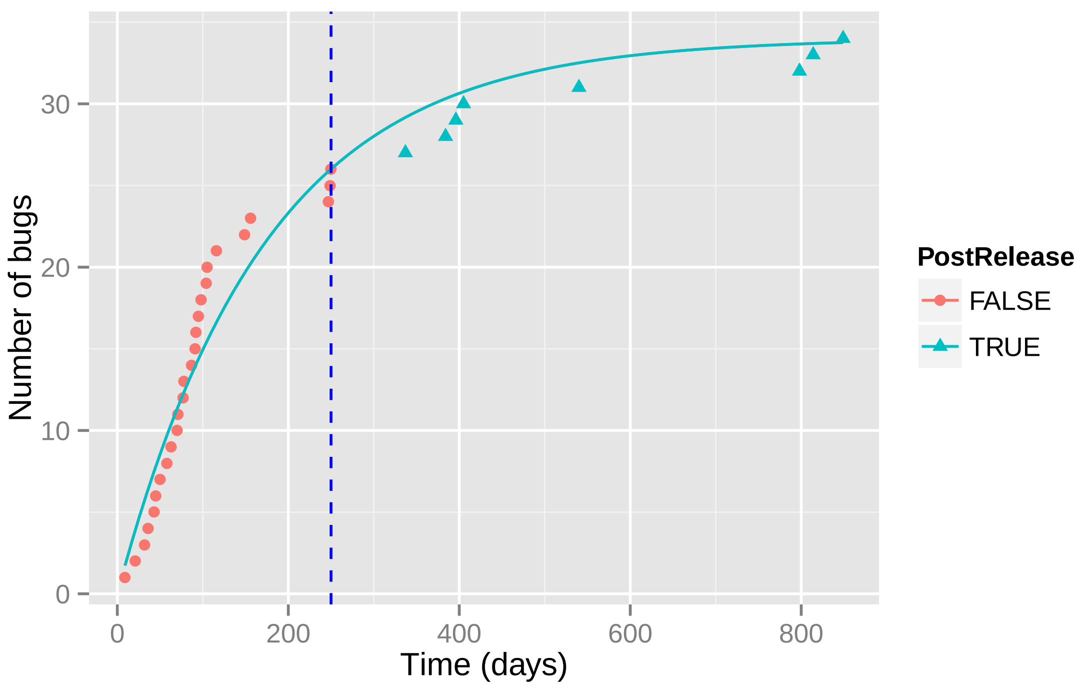


Fig. 4.1: A plot of the Mean Value Function of the Goel-Okumoto model for the NTDS dataset based on MLE parameters for the first 26 data points as published by Goel-Okumoto.

When we have already encountered n bugs to time T, and we know the times recorded for these bugs $t_1, \ldots, t_n$ then the log-likelihood as given by (Goel and Okumoto, 1979) in our notation is:

$$p(t_1, \ldots, t_n, T|a, b) = n \log(a) + n \log(b) - a \left(1 - e^{-bT}\right) - b \sum_{i=1}^{n} t_n \tag{4.7}$$

Note that $\sum_{i=1}^{n} t_i$ and n are sufficient statistics.

Interestingly, the Goel-Okumoto model allows positive probability to possible values of $a < n$ as it is a simple model. A more complicated model may well constrain $a \geq n$ but will likely be more wieldy to work with, however the Goel-Okumoto model does guarantee that the maximum likelihood estimate $\hat{a} \geq n$.

4.3 Approximations

The Goel-Okumoto model says that the predicted number of bugs, n by time t is:

$$n \approx a \left(1 - e^{-bt}\right) \tag{4.8}$$

Hence:

$$\frac{n}{a} \approx 1 - e^{-bt} \tag{4.9}$$

$$b \approx -\frac{1}{t} \log\left(1 - \frac{n}{a}\right) \tag{4.10}$$

By using the Taylor expansion

$$\log(1+x) = \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n} x^n \tag{4.11}$$

$$= x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \dots \qquad \text{for} \quad |x| \leq 1 \tag{4.12}$$

in Equation (4.10) this gives us:

$$b \approx -\frac{1}{t}\left[-\frac{n}{a} - \frac{1}{2}\left(\frac{n}{a}\right)^2 - \frac{1}{3}\left(\frac{n}{a}\right)^3 - \dots\right] \tag{4.13}$$

$$\approx \frac{n}{ta} + o\left(\frac{n^2}{a^2}\right) \tag{4.14}$$

Hence as a first approximation we have $b \propto \frac{1}{a}$. This is an expected result as we often see plots of the log-likelihood which have this $b \propto \frac{1}{a}$ structure, such as in example plots Figure 4.2 and Figure 4.3 where we see the characteristic $\frac{1}{x}$ structure when we plot on plain axes and a linear structure when we plot on log-log axes.

When we evaluate on a grid and use Gamma priors i.e. the prior on a is $\Gamma(1.1, \frac{1}{100})$, and the prior on b is $\Gamma(1.05, \frac{1}{1.01})$, we get a much more constrained posterior as shown in example plots Figure 4.4 and Figure 4.5.

Looking at the marginals, Figure 4.6, of the log-likelihood and the posterior for (a, b) we see that the influence of the data are very strong despite there only being 26 of the 34 data points used in this analysis. If we used the full 34 data points, the effect would be even stronger.

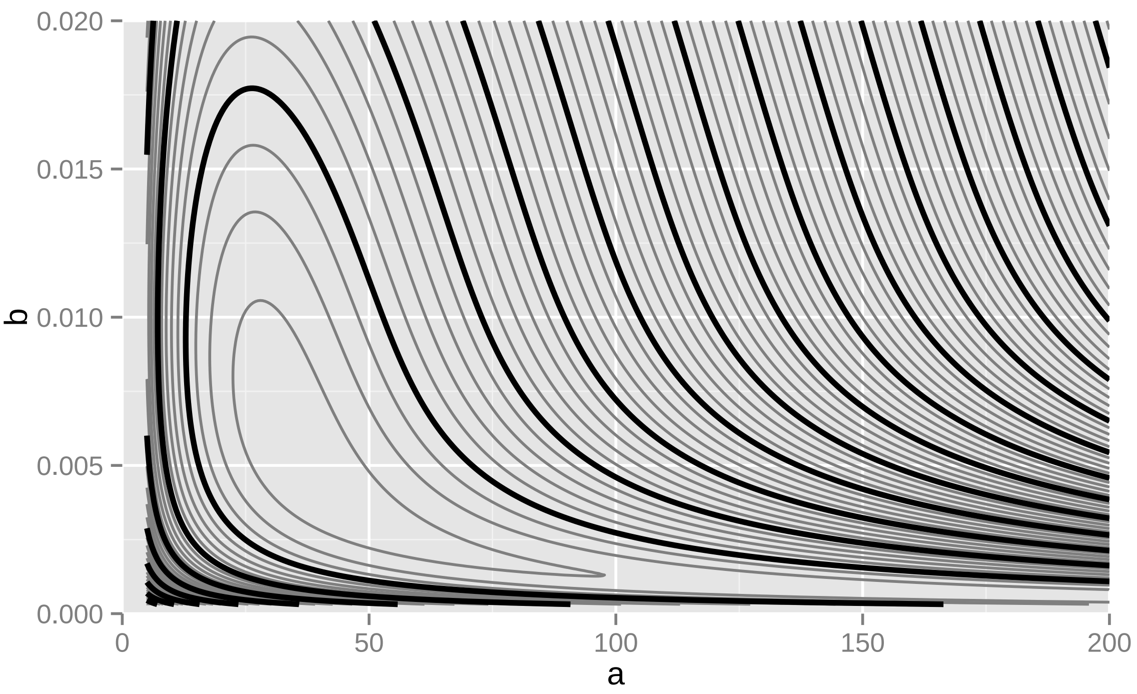


Fig. 4.2: A contour plot of the log-likelihood as in Equation (4.7) of the Goel-Okumoto model for the 26 point NTDS dataset, calculated on a rectangular grid.

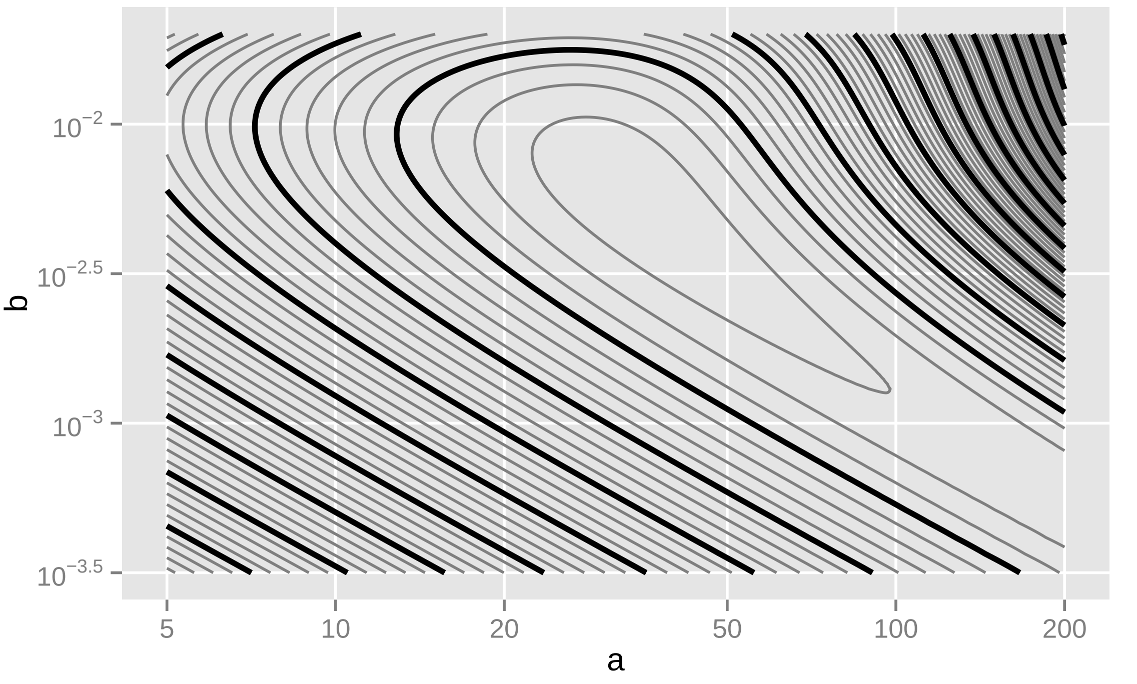


Fig. 4.3: A contour plot of the log-likelihood of the Goel-Okumoto model for the 26 point NTDS dataset on a log-log scale, calculated on a rectangular grid.

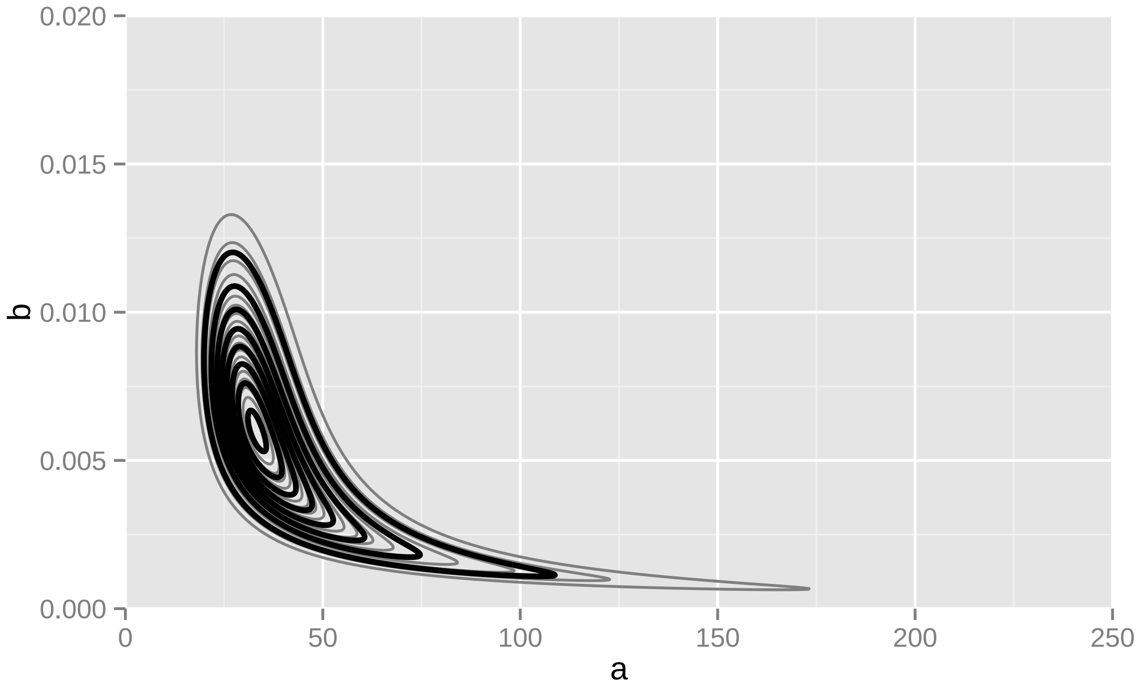


Fig. 4.4: A contour plot of the posterior of the Goel-Okumoto model for the NTDS dataset.

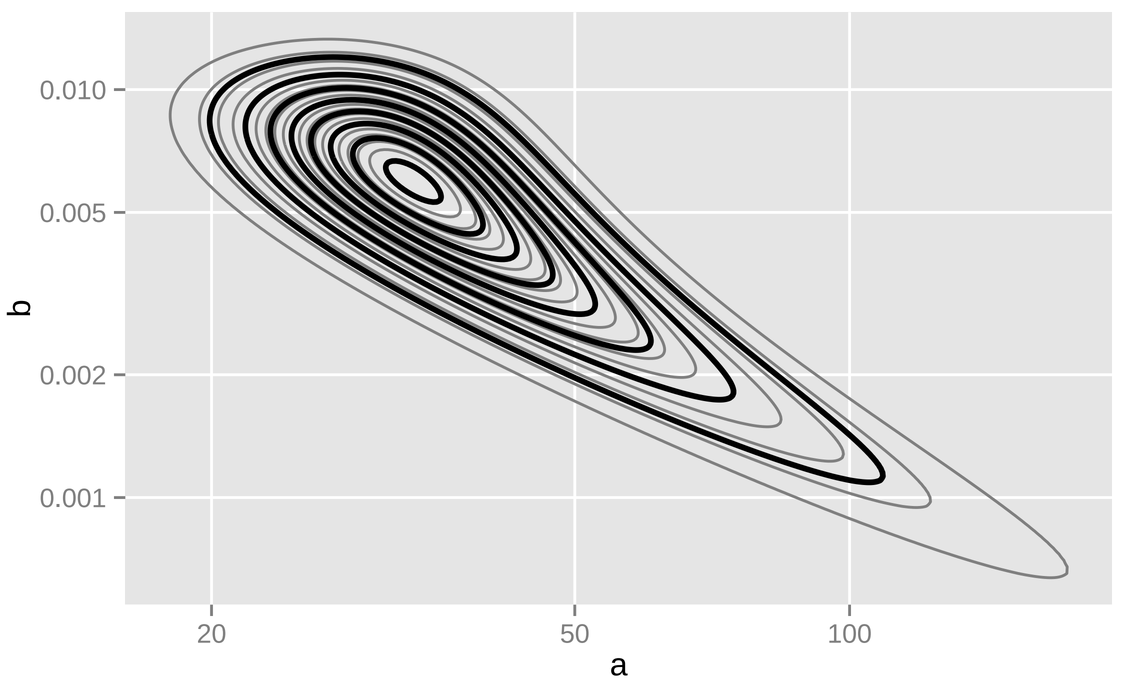


Fig. 4.5: A contour plot of the posterior of the Goel-Okumoto model for the NTDS dataset on a log-log scale.

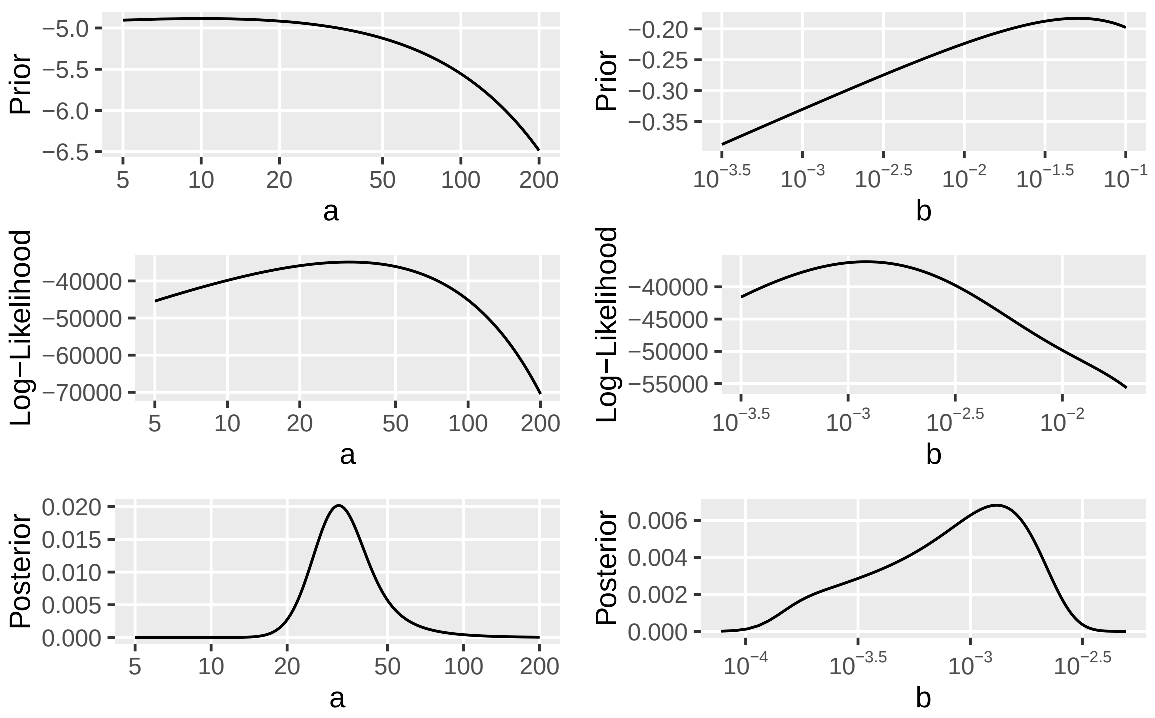


Fig. 4.6: A plot of the marginals of (a, b) of the Goel-Okumoto model for the NTDS dataset.

4.4 RStan

RStan (Stan Development Team, 2014a) is relatively new software for doing Bayesian modelling following in the footsteps of WinBUGS and JAGS. Stan and RStan are being developed by a group led by Andrew Gelman at Columbia University, New York. RStan is an R package which interfaces to the underlying Stan library (Stan Development Team, 2014b) from R and uses the No-U-Turn Sampler (NUTS) (Hoffman and Gelman, 2011, 2014; Betancourt, 2013). As Stan is based on Hamiltonian Monte-Carlo (HMC), it uses the gradient at each point to improve its trajectory around the sample space, so while the evaluation of each point on the trajectory might take longer, it should traverse the sample space more evenly and thus converge more quickly obtaining a better *effective* sample size. It is not clear yet at this stage that there is a big advantage for Stan over the older alternatives like WinBUGS and JAGS, but Stan is being actively developed and has a very helpful user mailing list[1]. Note in particular that because Stan uses Hamiltonian Monte-Carlo, and thus the gradient, it is not possible within Stan to use integer parameters as it is not possible to calculate

[1]`http://mc-stan.org/groups.html`

the gradient with non-continuous parameters.

The Stan model code for the Goel-Okumoto model is in Figure 4.7. In particular the last line of the model section defines the increment to the log-probability. Note also that there is quite a bit of R code which is used to prepare the data and make the call to Stan that is not visible in this figure.

When working with the NTDS dataset in RStan, it became clear that the best results, given that we know the answer from our previous work with a grid solution, comes when we only specify a vague prior for one of the parameters. Since it is more intuitive to specify a prior for the total number of bugs, a, that is what we have done.

When we look in more detail at the plot in Figure 4.10 and output statistics in Figure 4.8 from RStan for the NTDS model for 26 points, we see that the 50%ile estimate of the a parameter is 33.94, and the $(0.025, 0.975)$ quantile range is $(20.94, 74.90)$. When we look at the plot in Figure 4.11 and output in Figure 4.9 from RStan based on the full 34 points we see that the range for a has considerably tightened, and similarly for the b parameter. This is to be expected when we re-inspect Figure 4.1 as we can see that the initial data points, pre-release are in a comparatively straight line and when we add in the post-release points, the post-release points give more certainty to the location of the asymptote. Compare and contrast the relative certainty shown by the density plots from Figure 4.11 for 34 points as opposed to that for 26 points in Figure 4.10. Note in particular the vastly different scales on the two plots for both parameters.

Some further points to note are the column `n_eff` which is an estimate of the effective sample size of this run, and the column `Rhat` which is a measure of convergence fully described in Gelman et al. (2013) and originally published in Gelman and Rubin (1992) and extended in Brooks and Gelman (1998). Note too that Gelman has modified the definition of Gelman's $\hat{R}$, in Gelman et al. (2013) as compared with the first edition[2]. As convergence improves and as $n \to \infty$, then $\hat{R} \to 1$. Thinning, or only saving every m parameter values will improve both $\hat{R}$ and $n_{\text{effective}}$ for a given saved chain length n. The argument for *not* thinning, if the computer memory is available, is that thinning is throwing away information, i.e. why should we expect a better answer by throwing away data?

[2]This is mentioned in the footnote in Gelman et al. (2013) on page 285

```
// This is a Stan model for the Goel-Okumoto model with
// MVF = a(1 - exp(-b*t))
// where t is time
//   a corresponds to the expected number of bugs
//        in the system
//   b is a parameter relating to the rate at which
//        the bugs are discovered

data {
  // number of data points (bugs observed)
  int<lower=0> N;

  // the observed bug times - in days or decimal years
  real<lower=0> bugs[N];
}

// here we pre-"calculate"" some values once to save having
// to calculate them many times when sampling in the model
transformed data {
  // bugsn and sumt are minimal sufficient statistics
  real<lower=0> bugsn; // last value of time
  real<lower=0> sumt;  // the sum of all the time values
  // used as a rough estimator for the prior on a
  real<lower=0> arat;
  real<lower=0> amid; // "mid"-point of a prior
  // real<lower=0> bmid; // "mid"-point of a prior

  // minimal sufficient statistics (MSS)
  // Note that N is the third and final of the MSS
  bugsn <- bugs[N];
  sumt <- sum(bugs);

  // Very crude estimates for the mid-points for a and b
  // GROSS estimate of the prior on a that is 20% bigger
  // than the number of bugs seen thus far
  arat <- 1.2;
  amid <- log(N*arat); // i.e. a is about 1.1*N
  // based on approximation for b in terms of a
  //  bmid <- -log(log(1+1/arat)/bugsn); //
}
parameters {
  // estimated number of bugs in the system
  real<lower=1, upper=15000> a;

  // finding "rate" parameter
  real<lower=0, upper=1> b;
}
model {
 // priors on a and b
     // or for the log-gamma, a - shape=1.1, rate=1/100
     // or for the log-gamma, b - shape=1.05, rate=1/1.01
  a ~ lognormal(amid, 1);
  // flat prior on b for now so comment out...
  // b ~ lognormal(bmid, 1);

  // GO model itself
  increment_log_prob(-b*sumt + N*(log(a)+log(b)) -
                   a*(1-exp(-b*bugsn)));
}
```

Fig. 4.7: The full Goel-Okumoto model in Stan model code

```
Inference for Stan model: goel_okumoto.
4 chains, each with iter=10000; warmup=5000; thin=1;
post-warmup draws per chain=5000, total post-warmup draws=20000.

         mean se_mean      sd     2.5%      25%      50%      75%
a     37.2274  0.3350 15.9460  20.9362  28.5150  33.9447  41.1979
b      0.0063  0.0000  0.0026   0.0017   0.0044   0.0061   0.0079
lp__ -88.8778  0.0182  1.0788 -91.7627 -89.3275 -88.5439 -88.0986
        97.5% n_eff   Rhat
a     74.8985  2266 1.0027
b      0.0119  3886 1.0019
lp__ -87.8107  3523 1.0011
```

Fig. 4.8: Output from a run of the Stan model for the Goel-Okumoto model for the NTDS dataset using only 26 points.

```
          mean se_mean     sd      2.5%       25%       50%       75%
a      35.2945  0.0591 5.9893   24.5406   31.0240   35.0003   39.1276
b       0.0045  0.0000 0.0010    0.0027    0.0038    0.0045    0.0051
lp__ -135.1138  0.0136 1.0180 -137.8425 -135.5214 -134.7960 -134.3832
         97.5% n_eff   Rhat
a      47.9673 10269 1.0003
b       0.0065 10178 1.0002
lp__ -134.1147  5620 1.0007
```

Fig. 4.9: Output from a run of the Stan model for the Goel-Okumoto model for the NTDS dataset using only 34 points.

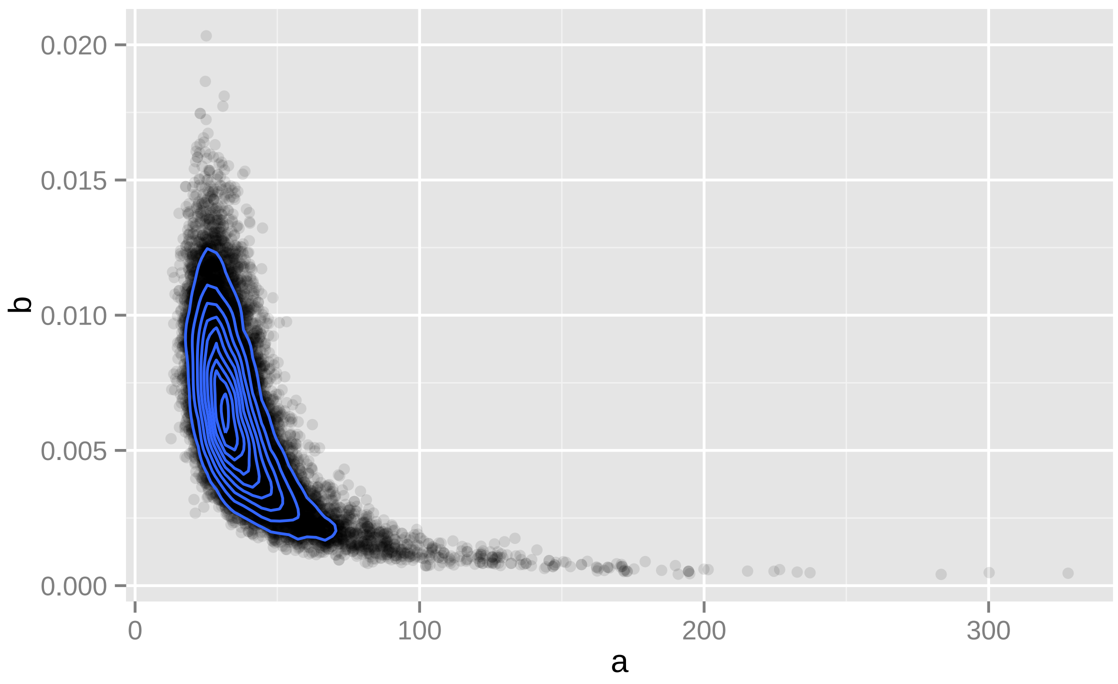


Fig. 4.10: A plot of the posterior of the Goel-Okumoto model from RStan for the NTDS dataset for $N = 26$.

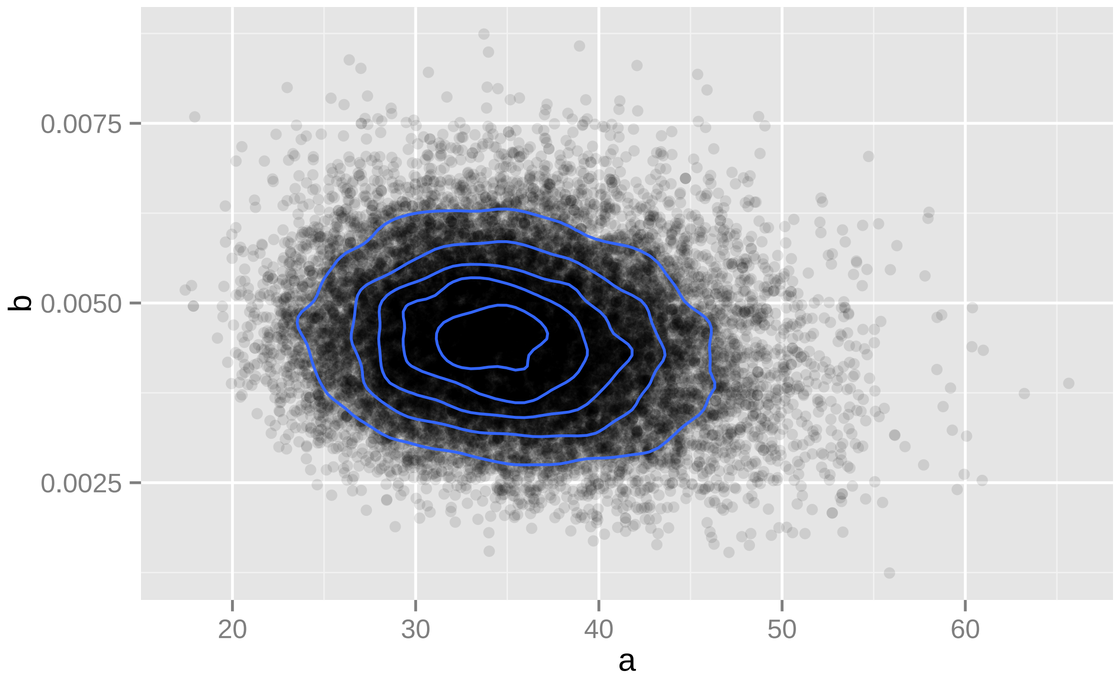


Fig. 4.11: A plot of the posterior of the Goel-Okumoto model from RStan for the NTDS dataset for $N = 34$. Note the very different scales on both the X and Y axes as compared with Figure 4.10.

4.5 Multiple Versions

When looking at multiple versions of software, we can infer the values of the Goel-Okumoto parameters (a_k, b_k) separately for each version of software, k. Thus the probability model for each release, k, separately is as follows:

$$p(t_1, t_2, \ldots, t_n, a_k, b_k) = p(t_1, t_2, \ldots, t_n | a_k, b_k) p(a_k, b_k)$$

If the times recorded for the bugs for each of releases $(1 \ldots K)$ are:

$$\underline{t}_k = (t_{k1}, t_{k2}, \ldots, t_{kn})$$

We can write

$$p(\underline{t}_1, \ldots, \underline{t}_K, \underline{a}, \underline{b}) = \prod_{k=1}^{K} p(\underline{t}_k | a_k, b_k) p(a_k) p(b_k)$$

where we assume that each release is an observation from a separate and independent Goel-Okumoto model. We call this model *Independent* and an example of this applied to the Firefox dataset can be seen in Figure 4.12. We note a profound difference in the pattern in both a and b from version 19 onwards, which is also reflected in our later models, in particular there is a sudden increase in the variance of the estimates of the parameters (a, b), which we cannot explain. We further note that the independent model handles each and every version identically, so this difference in the parameter estimates arises from the data.

As an aside, note that the pattern of increasing and decreasing a mirrors the decrease and increase in the parameter b as we would expect given our approximation of $a \propto \frac{1}{b}$.

However, if we have already calculated a distribution for one version k, then we can use this information to help us calculate the values of the parameters in the next version $(k + 1)$, i.e. use the previous posterior of (a_k, b_k) as the prior for the next version and similarly for subsequent versions. This respects the chronological order of the release versions. We call this model *Cumulative*. Here we assume that each release is an observation of the Goel-Okumoto model with the same parameters, which is a strong assumption. Refer to Figure 4.13 to see a plot for the Firefox dataset.

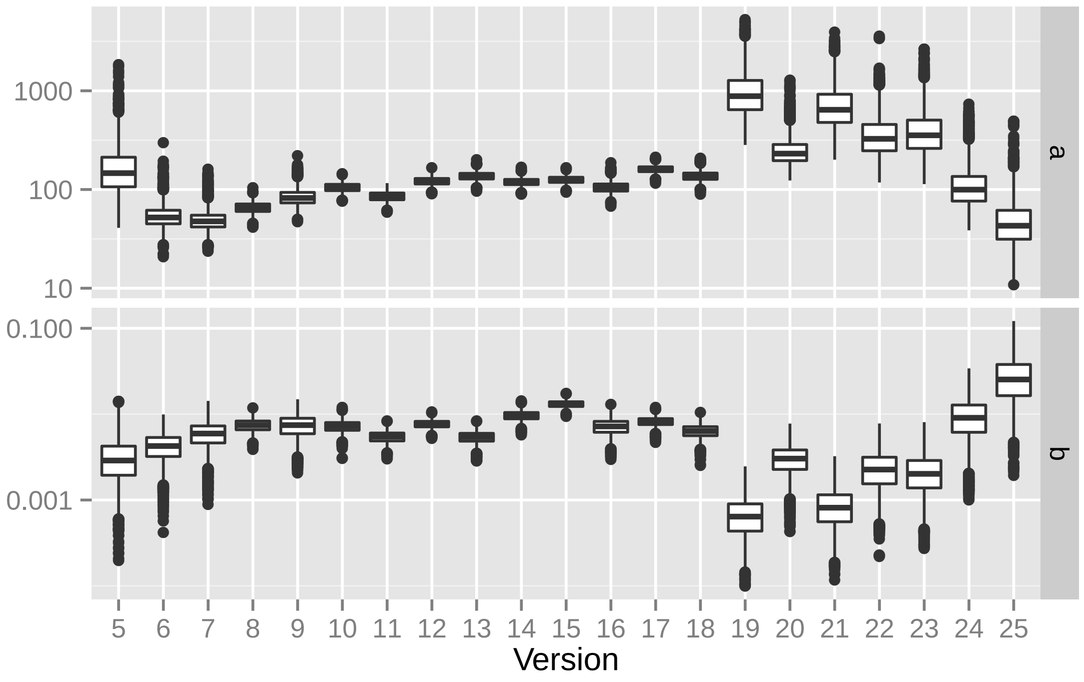


Fig. 4.12: A box plot of the Goel-Okumoto `a` and `b` parameters for each of the versions calculated using Stan using the *independent* model

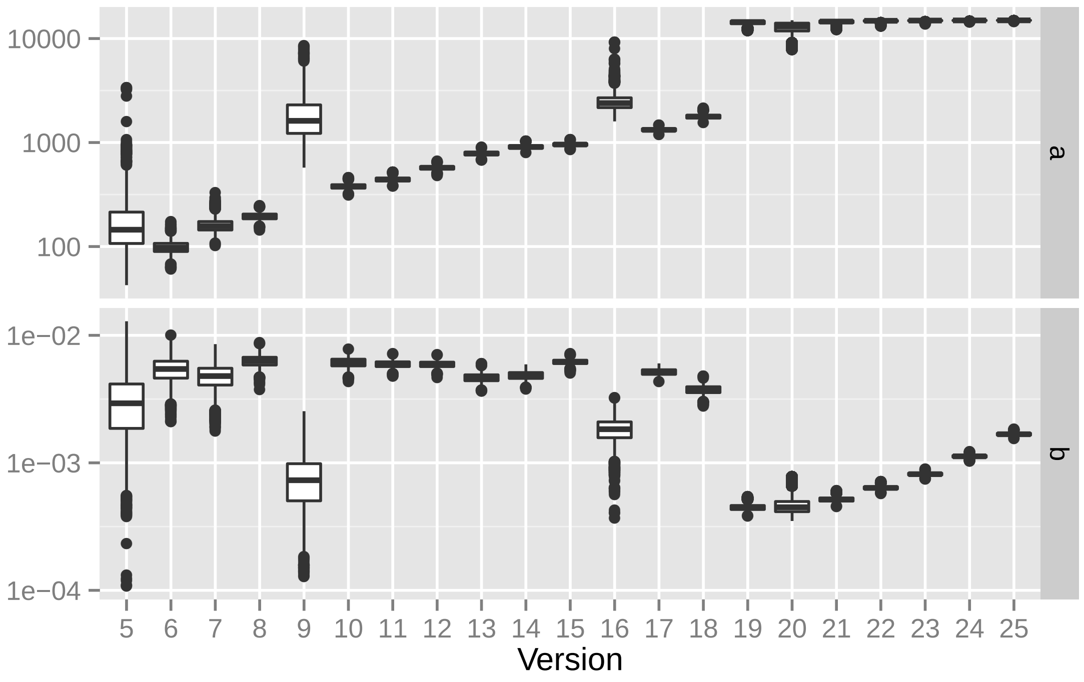


Fig. 4.13: A box plot of the Goel-Okumoto `a` and `b` parameters for each of the versions calculated using Stan using the *cumulative* model

We can say that:

$$p(\underline{t}_1, \underline{t}_2, \ldots, \underline{t}_k, a, b) = p(a, b) \prod_{i=1}^{k} p(\underline{t}_i | a, b)$$

$$p(a, b | \underline{t}_1, \underline{t}_2, \ldots, \underline{t}_k) \propto p(a, b) \prod_{i=1}^{k} p(\underline{t}_i | a, b) \qquad \text{‘batch’}$$

$$\propto \underbrace{p\left(a, b | \underline{t}_1, \ldots, \underline{t}_{k-1}\right)}_{\text{Prior}} \times \underbrace{p(\underline{t}_k | a, b)}_{\text{Likelihood}} \qquad \text{‘streaming’}$$

By ‘batch’, we mean that we calculate everything at once when we have all the data. In contrast in ‘streaming’ mode we only get the data version by version so after we receive each version, j, we calculate the new value of $p(a, b|\underline{t}_{1..j})$ by updating the value for version $(j - 1)$. Thus we might start the chain of versions by using a very uninformative prior for (a, b) for version $j = 1$, but as j increases, we have more information. Note that in our dataset the time ranges for bugs in each of the versions overlap considerably.

An alternative to the cumulative model is to borrow strength from the other versions by using a *Hierarchical* model as described in Section 3.11. This ignores the time ordering of the releases and implies that they are exchangeable which is a weaker assumption than the independence model. Here we are back to each release having its own (a, b) but now they are conditionally independent given the hyper-parameters $(A, B, \sigma_a^2, \sigma_b^2)$. In particular, the hyper-parameter (A, σ_a^2) are the unknown ‘true’ value of a and its variance, and similarly for (B, σ_b^2) and b. This gives us the advantages of a hierarchical approach, the borrowing of strength and the ability to predict the values of (a, b) for future releases allowing for some differences between releases.

$$p\left(\underline{t}_1, \ldots, \underline{t}_k, \underline{a}, \underline{b}, A, B, \sigma_a^2, \sigma_b^2\right) = \left(\prod_{k=1}^{K} p\left(\underline{t}_k | a_k, b_k\right) p\left(a_k | A, \sigma_a^2\right) p\left(b_k | B, \sigma_a^2\right)\right) \\ \times p\left(A, B, \sigma_a^2, \sigma_b^2\right)$$

where the final $p(\cdot)$ term is a prior on the hyper-parameters. Note that the first $p\left(\underline{t}_k | a_k, b_k\right)$ is NHPP, and the second pair of terms $p\left(x_k | X, \sigma_x^2\right)$ for $X = (A, B)$, are the log-normal hyper-priors.

This can also be spelt out as:

$$\underbrace{\left(\prod_{k=1}^{K} p(\underline{t}_k | a_k, b_k)\right)}_{\text{Likelihood}} \times \underbrace{\left(\prod_{k=1}^{K} p(a_k | A, \sigma_a^2)\right)}_{\text{Prior on a}} \times \underbrace{\left(\prod_{k=1}^{K} p(b_k | B, \sigma_b^2)\right)}_{\text{Prior on b}} \times \underbrace{p(A, B, \sigma_a^2, \sigma_b^2)}_{\text{Hyper-Prior on (a,b)}}$$

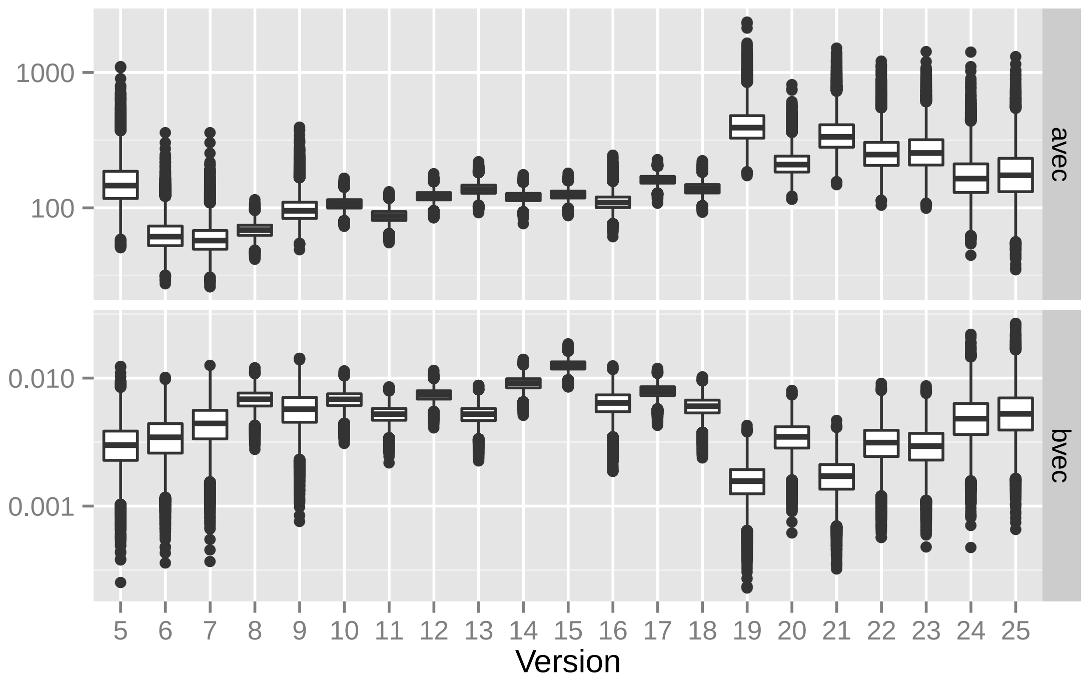


Fig. 4.14: A box plot of the Goel-Okumoto `a` and `b` parameters for each of the versions calculated using Stan using the *hierarchical* model. Compare with Figure 4.13.

This is a simple hierarchical model, and assumes that there is no particular order to the versions and that the parameters from each version are fully exchangeable with the other versions. The results are shown in Figures 4.14 to 4.16. The corresponding parameter values for this Stan run of the hierarchical model can be seen in Appendix A.

When we calculate the Bayes factors following the methodology in Section 3.12, we find that in comparing the cumulative and independent model, the Bayes Factor is 1.67×10^{1551}, which Kass and Raftery (1995) describe as *very strong* evidence in favour of the cumulative model.

When we compare the hierarchical model over the independent model, the Bayes factor is 9.04×10^{239}, or *very strong* evidence in favour of the hierarchical model.

Finally when we compare the cumulative model and the hierarchical model, the Bayes factor is 1.8×10^{1311} or *very strong* evidence for the cumulative model over the hierarchical model.

These results should be taken with some caution given the difficulties in estimating Bayes factors from MCMC results.

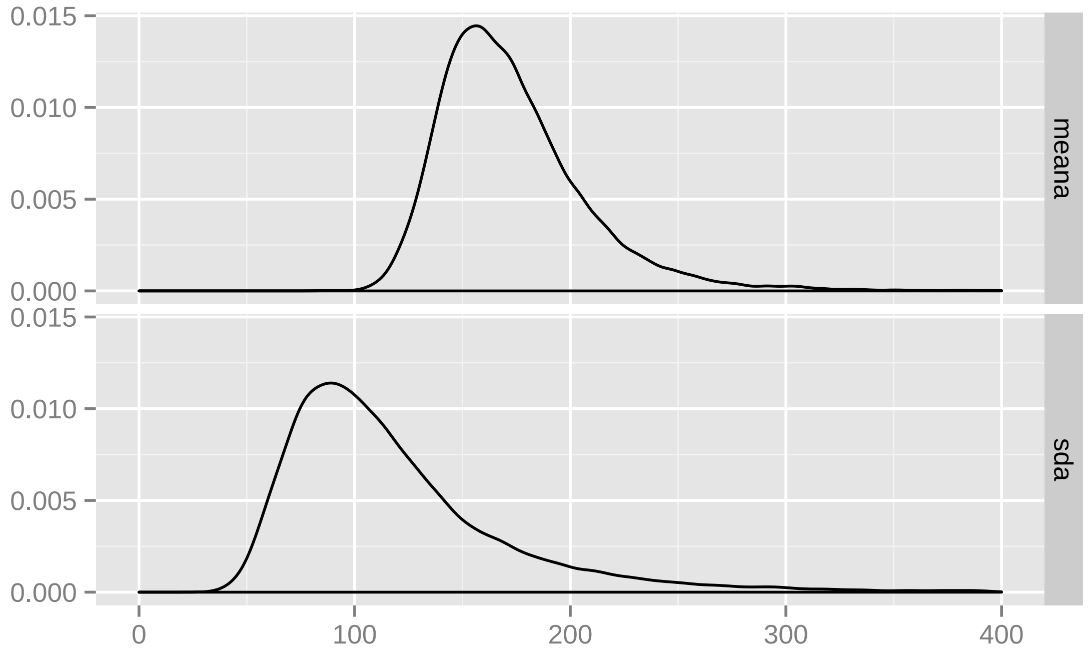


Fig. 4.15: Density plots of the Goel-Okumoto `meana` (mean of a) and `sda` (standard deviation of a) parameters from the Stan *hierarchical* model.

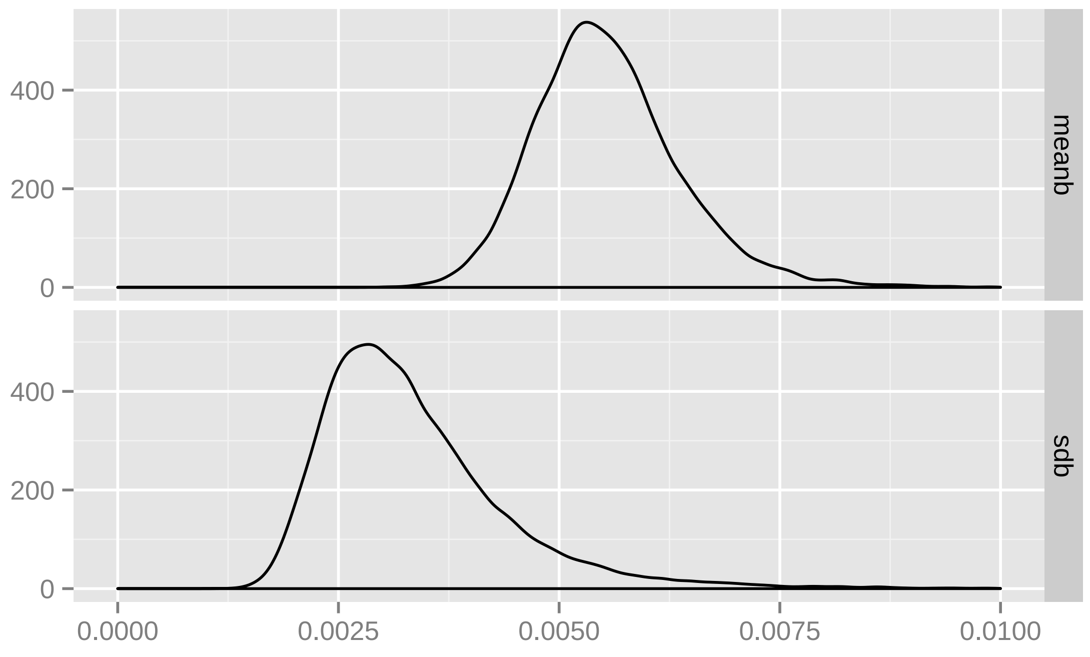


Fig. 4.16: Density plots of the Goel-Okumoto `meanb` (mean of b) and `sdb` (standard deviation of b) parameters from the Stan *hierarchical* model

4.5.1 Practical Issues

Obviously for the independent model, the bugs for each version of Firefox were independently extracted and the RStan model was run. However, for the box plots seen in Figures 4.12 and 4.13, it was necessary to run each model so that the $N_{\text{effective}}$ was approximately the same so that the box plots were comparable between versions. Note that the RStan implementation of $N_{\text{effective}}$ is used, refer Gelman et al. (2013, Section 11.5) for a text-book description.

The process of running the Stan code to obtain an effective sample size is done by running for one chain of 100 iterations and then using a plain linear estimator to try and scale this to an $N_{\text{effective}} = 500 \ \pm 10\%$. When there are more than two iterations, plain linear regression is used to estimate $N_{\text{effective}}$ against the number of iterations. This cycles until it comes within the $\pm 10\%$ bound at which time the details are stored and the function moves to the next version.

For the individual model the bug time data was extracted for a given release only as follows:

```
bugt <- ffi[ffi$rver == Nrel, 'age'] # NB NOTE THE ==
# now rescale so that the first bug is at time=1
# 1 is just a nominal offset
bugt <- bugt - min(bugt) + 1
```

For the cumulative model, the data set was extracted for all releases up to and including the given release N_{rel}, in contrast to the individual model which only looks at that particular release.

```
bugt <- ffi[ffi$rver <= Nrel, 'age'] # NB NOTE THE <=
# now rescale so that the first bug is at time=1
# 1 is just a nominal offset
bugt <- bugt - min(bugt) + 1
```

As shown in Section 3.9, this is equivalent to calculating the posterior for release $N_{\text{rel}} - 1$ and then using this as the prior for N_{rel}.

4.6 Hand written sampler

In order to meld with the later work we have written a Metropolis-Hastings sampler by hand in R. In particular, as Stan is based on Hamiltonian Monte Carlo, it cannot use integer parameters which we will need in later work. An acceptance rate of about

	a	b
Prior	Log-Normal$(\exp(30), 0.7)$	Uniform$(10^{-4}, 0.05)$
Proposal	Normal(a, 0.4)	Normal(b, 0.4)

Table 4.1: A summary of the details of the hand sampler

0.39 is easily achieved, and is within the range highlighted by Gelman et al. (1996) and Gelman et al. (2013, Section 12.2) as being optimal, i.e. 0.44 for single parameter reducing to 0.23 in dimensions > 5. By optimal we mean that there is a trade-off between exploring the probability space quickly and proposals being rejected. If a high proportion of samples are rejected, then the proposal is likely too large and it is trying to move too far. However, if all samples are accepted, then the proposal is likely too small and is likely moving too little at each step so it will take too long to explore the sample space - so there is a happy medium. Note that this is a general rule and while quite widely applicable is not perfect.

The sampler is based on a simple Metropolis-Hastings step using the above likelihood and a summary can be seen in Table 4.1. The prior on the a parameter is log-normal on a support of $(0, \infty)$, and uses the density, as written in R as:

```
dlnorm(x, meanlog=log(30), sdlog=0.7)
```

which we felt to be "reasonable". As we had little intuition for the prior on the b parameter and we found previously that it was so highly correlated with the a parameter, we have given it a uniform density on the support of $(10^{-4}, 0.05)$ also as per the models above.

The proposal step is normal based on the log of the previous successful step and a standard deviation of 0.4 for both parameters worked adequately. Since this step is a relatively fast operation in comparison to the rest of the model and thus a lower priority for optimization, we have not spent much time optimizing it for speed as recommended by Knuth (1974). This sampler has been found to be very similar in operation to both the RStan and grid based samplers and this gave us some confidence that we were on the right track. As we obtained similar results to previous runs, we have not gone into further details with results plots and output tables.

In addition some work was put into running the chains in parallel and merging the output into one data structure for post-processing and analysis.

4.7 Discussion

As mentioned in Section 4.2 there is an implicit assumption in the model that both the environment and the rate of testing work stay constant. In reality, this is not the case and in our scenario, particularly before the official release date, the environment, i.e. the software, is being released on a nightly basis. Furthermore, some testers will update to a new nightly release on a daily basis whereas others might only test some of the nightly releases.

The number of people who are testing is also in constant flux with some working full-time on testing an upcoming release. In other cases, somebody else might see a bug in the live release, then download the latest nightly release and check that the bug is also there and file the bug. As seen in Figure 2.4 most people only log a single bug! Significantly, the number of people who will look at a particular release will be measured in the thousands before the release date, and within days of the release will be measured in the tens and then hundreds of millions.

While the models described above are for the most part a reasonable fit with the Firefox data, it is clear when looking at the plot for the cumulative data in particular that the a parameter is not modelled well from version 19 onwards as the values are collapsed up against the support maximum for a of 15,000, but that said, the Bayes factors strongly prefer the cumulative model.

In particular, this can be interpreted as very strong evidence for single Goel-Okumoto pair (a, b) underlying all release versions.

Chapter 5

Semi-Supervised Classification

5.1 Introduction

This chapter reviews semi-supervised classification and applies it to the sorts of dataset found in bug databases. Notably in a bug database there are normally bugs associated with a particular release or version, and then other bugs which are not assigned to a particular release. This chapter outlines one possible solution to this problem by imputing a version label for unlabelled bugs. All the labelled bugs can then be analysed using a model such as that of Goel and Okumoto (1979) as discussed in Chapter 4 and further analysed in Chapter 6.

5.2 Data

We will continue to use the Firefox dataset described in Section 2.4 and seen in Figure 2.3. As can (just about) be seen in Figures 5.1 and 5.2, the density of each of the Rapid-Releases are for the most part approximately ‘bell-shaped’, though some are bimodal. Note that the density of version 25 is particularly intense, starting shortly before the cut-off date of 2013-07-18 and it was not due to be released for some time and was actually released on the 2013-10-29.

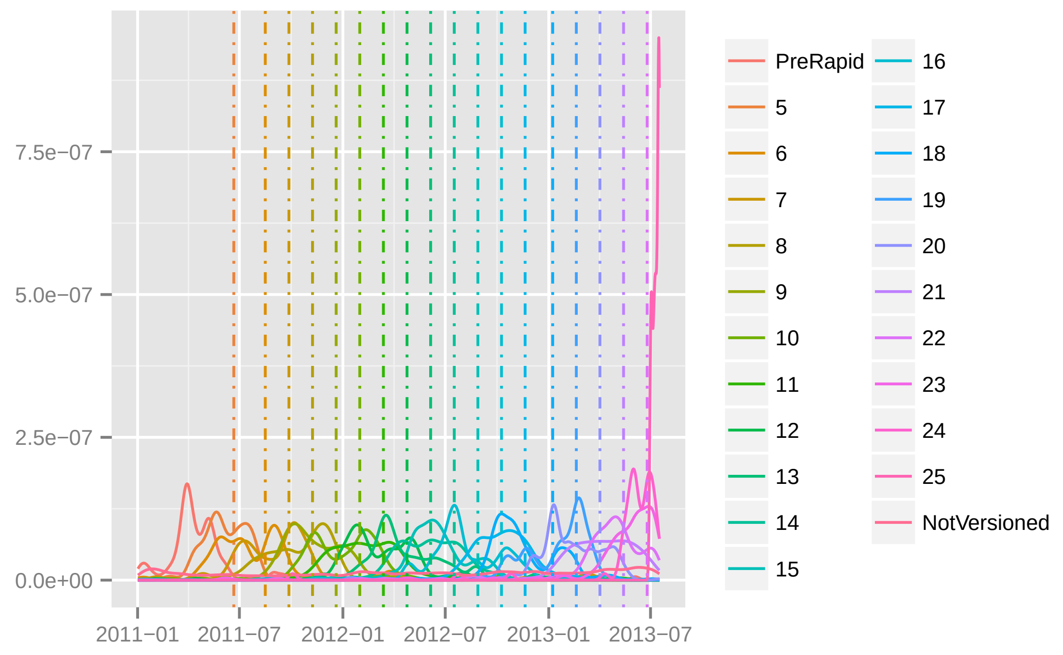


Fig. 5.1: A plot of the densities of the different versions of Firefox with related release date for each version plotted in a vertical dashed line of the same colour.

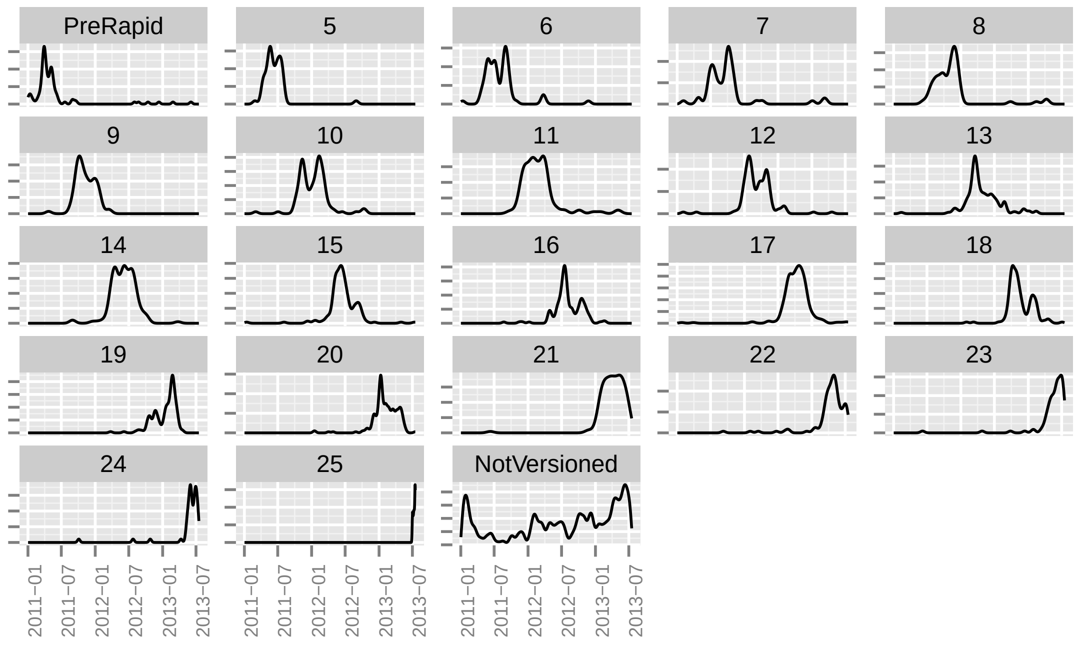


Fig. 5.2: A facet plot of the densities of the different versions of Firefox. Note that for clarity, the y-scale is ‘free’, i.e. not identical, in each facet.

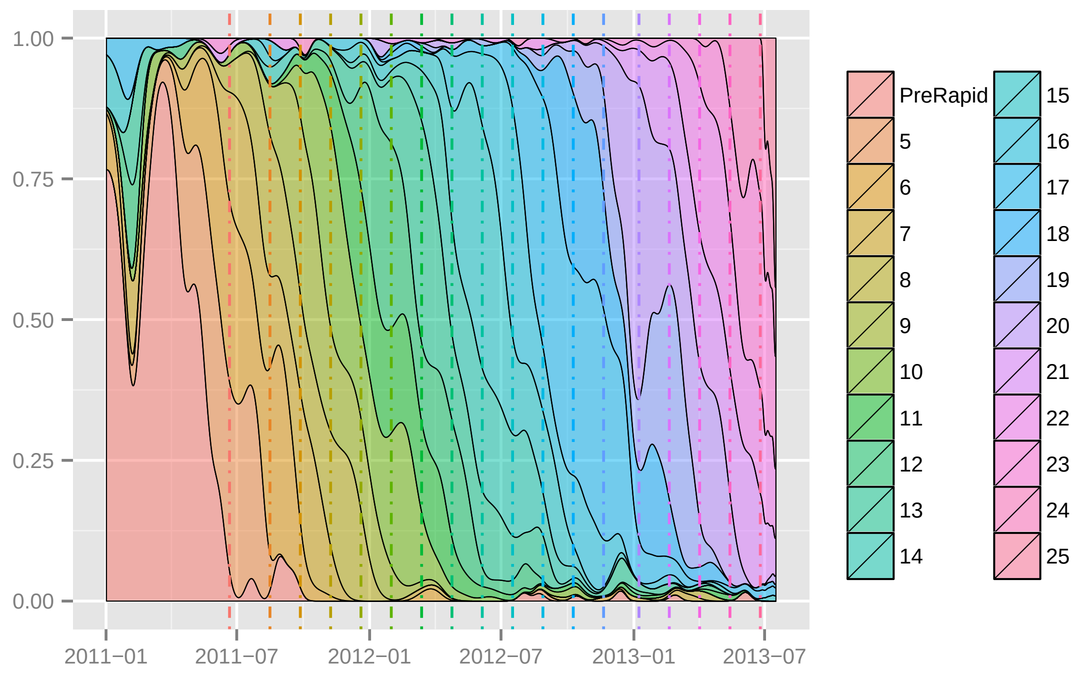


Fig. 5.3: An area plot of the densities of the different versions of Firefox. At a given date, the probability that a bug is from a particular release will be strongly approximated by the proportion of colour associated with a particular version in a vertical transect at that date. The release dates of each version (5-22) are shown as dashed vertical lines in the colour of the release. It is not expected that the reader can read this in detail, merely that they see the overall pattern.

5.3 Informal Model Outline

We expect bugs which are recorded 'near' the time of an official release date are more likely to be from that release than from another release from which they are not as 'near'. Detailed examination of Figures 5.1 and 5.2 shows that this is not an unreasonable assumption.

Looking at Figure 5.4, we show a schematic of the version model where the density of the time stamp of the identification of each bug is modelled as a Gaussian kernel which is offset from the official release date (T_k) for that particular version (k), by a time offset (α).

As a consequence of exchangeability we assume that the labelled data are representative of the unlabelled data. When we look at Figure 5.3 and pick a particular date, say 2013-07-01, we imagine a vertical transect at that date and look at the proportions of colour/versions at that date and we say that these proportions will be representative of probabilities of the labels that we will apply at that date.

5.3.1 Assumptions:

Our model makes the following assumptions:

- The version label is Missing At Random (MAR). By this we mean that we assume that the label is missing randomly in a way which does not affect the model itself, which is slightly less strict than Missing Completely At Random (MCAR).
- Bugs are exchangeable in the sense of de Finetti.
- Bug distribution is bell-shaped around a date that is 'near' the release date. This is partly supported by the density plots as seen in Figures 5.1 and 5.2, though we will return to this assumption later.
- The version parameters, (d_j, α_j) are independent.

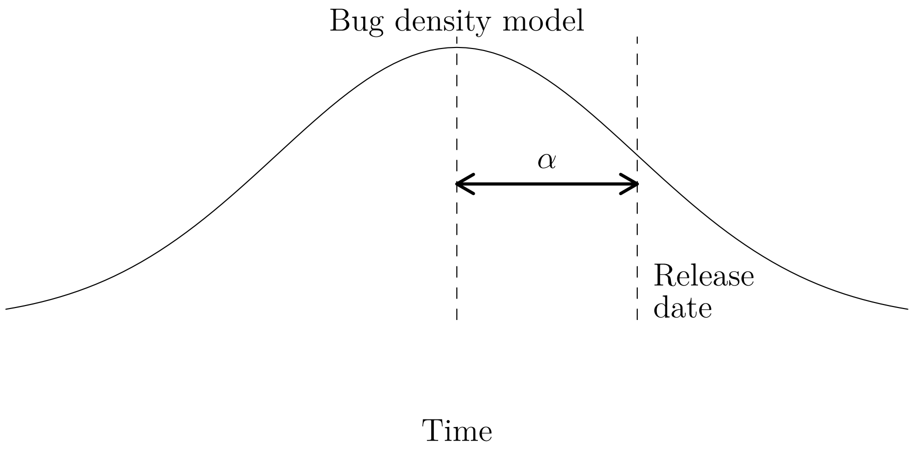


Fig. 5.4: A schematic of the version model, showing the Gaussian kernel density and the known release date.

5.4 Formal description of the model

The probability that a given bug t_i is from a particular release j is:

$$P(R = j | d_{1:k}, \alpha_{1:k}, t_i, T_j) = \frac{e^{-d_j(t_i - T_j - \alpha_j)^2}}{\sum_{k=1}^{K} e^{-d_k(t_i - T_k - \alpha_k)^2}} \tag{5.1}$$

This uses the *softmax* function of Bridle (1989)[1]. The value T_j is the official release date of release j and can be considered as a co-variate information about the release - rather than about the main dataset. The parameters $d_{1:k}$ are not-unlike the standard deviation of a Gaussian distribution - one for each release $(1 : k)$. The parameters $\alpha_{1:K}$ are the offsets between the official release date of a given version and the 'centre' of the 'bell-curve', refer Figure 5.4.

5.5 Algorithm

This section outlines the actual implementation of the model.

The priors for the model are:

$$\alpha_k \sim \mathcal{N}(-0.1, 1) \tag{5.2}$$

$$d_k \sim \mathrm{Exp}(\mathrm{rate} = 1) \tag{5.3}$$

[1]softmax$(j, x_{1\ldots n}) = \frac{\exp(x_j)}{\sum_{i=1}^{n}\exp(x_i)}$

The likelihood is:

$$\prod_{i=1}^{N}\prod_{k=1}^{K}\frac{e^{-d_j(t_i-T_j-\alpha_j)^2}}{\sum_{k=1}^{K}e^{-d_k(t_i-T_k-\alpha_k)^2}} \tag{5.4}$$

The posterior is thus:

$$\prod_{i=1}^{N}\prod_{k=1}^{K}\frac{e^{-d_j(t_i-T_j-\alpha_j)^2}}{\sum_{k=1}^{K}e^{-d_k(t_i-T_k-\alpha_k)^2}}\prod_{k=1}^{K}\frac{1}{\sqrt{2\pi}}e^{-\frac{1}{2}(\alpha_k+0.1)^2}\prod_{k=1}^{K}e^{-d_k} \tag{5.5}$$

First, we initialize ($\alpha_k = -0.1, d_k = 12$), jittered by 10 percent[2]. Then we initialize all missing version labels with a linear estimate from 1 to K as a first very rough guess at the labels. Then, for each MCMC iteration i:

1. Iterate the version counter, k from $1 : K$

   (a) For version k we make the following proposals:

   - $\alpha_k^* \sim \mathcal{N}(\alpha_k, \sigma_\alpha^2)$ where $\sigma_\alpha = 0.0125$.
   - $d_k^* \sim \mathcal{U}\left(d_k c_d, \frac{d_k}{c_d}\right)$ where $c_d \in [0, 1]$, currently 0.9.

   (b) Based on the likelihood in Equation (5.1), jointly accept or reject the (α_k^*, d_k^*) via a Metropolis-Hastings step, with priors:

   - $\alpha \sim \mathcal{N}(\mu_\alpha, 1)$, where μ_α is the initial value of α, in our case $\mu_\alpha = -0.1$ as mentioned above.
   - $d \sim \mathit{Exp}(\mathit{rate} = 1)$, i.e. the exponential distribution.

   We calculate the acceptance probability as follows:

```
g_num <- loglikelihood(alpha.star, d.star) + log(
   prior.alpha(alpha.star[[k]])*prior.d(d.star[[k]]))
g_den <- loglikelihood(alpha.star, d.star) + log(
   prior.alpha(alpha[i, k])*prior.d(d[i, k]))
accept.prob <- min(1, exp(g.num - g.den)*d[i,k]/d.
   star[[k]])
```

   Which is equivalent to the following:

$$\begin{aligned}\text{Accept Prob} =& \frac{\text{Likelihood}(\text{data}|\alpha_{1:K}^*, d_{1:K}^*)}{\text{Likelihood}(\text{data}|\alpha_{1:K}, d_{1:K})} \times \frac{\mathcal{N}(\alpha_{1:K}^* + \mu_\alpha, \sigma_\alpha)}{\mathcal{N}(\alpha_{1:K} + \mu_\alpha, \sigma_\alpha)} \\ & \times \frac{\mathit{Exp}[d_k^*, \lambda = 1]}{\mathit{Exp}[d_k, \lambda = 1]} \times \frac{d_k^*}{d_k}\end{aligned} \tag{5.6}$$

2. Impute release version labels for all unlabelled bugs using the new set of $(\alpha_{1:K}, d_{1:K})$.

[2]We are using time in units of years with an origin at the start of the time series to reduce the possibility of numerical overflow with the operator $\sum_i \exp(x_i)$ as we had issues when using long time series measured in units of days.

(a) Given the set of $i = 1 \ldots N$ time stamps, t_i, calculate over the $k = 1 \ldots K$ versions to get the P_{ver} matrix:

$$P_{\text{ver}}(i,k) = \frac{\exp\left(-d_k\left(t_i - T_k - \alpha_k\right)^2\right)}{\sum_{j=1}^{K}\exp\left(-d_j\left(t_i - T_j - \alpha_j\right)^2\right)} \tag{5.7}$$

(b) For each bug, i, pull weighted sample, i.e. take index weighted by P_{ver}, or in R:

```
sample(K, size=1, prob=pver[i,])
```

Note that we initially tried to do a Metropolis-Hastings accept/reject step with all $(\alpha_{1\ldots K}, d_{1\ldots K})$ simultaneously, but the acceptance rate was virtually zero, so we changed to doing the accept/reject on a single pair of (α_k, d_k) and we found a much improved acceptance rate which we were able to tweak by hand to get into the acceptable range, being of the order of 0.2.

5.6 Acceptance Ratios

There is a curious *exponential decay* shape to the plot of acceptance rate for the version model of Chapter 5 as seen in Figure 5.5. By adjusting the proposal rate, this curve can be moved up and down and can be made more linear, however, it will have the same basic shape with the highest acceptance rate for the earliest release (in fact the Pre-Rapid releases) and the lowest acceptance rate for the latest release, 25.

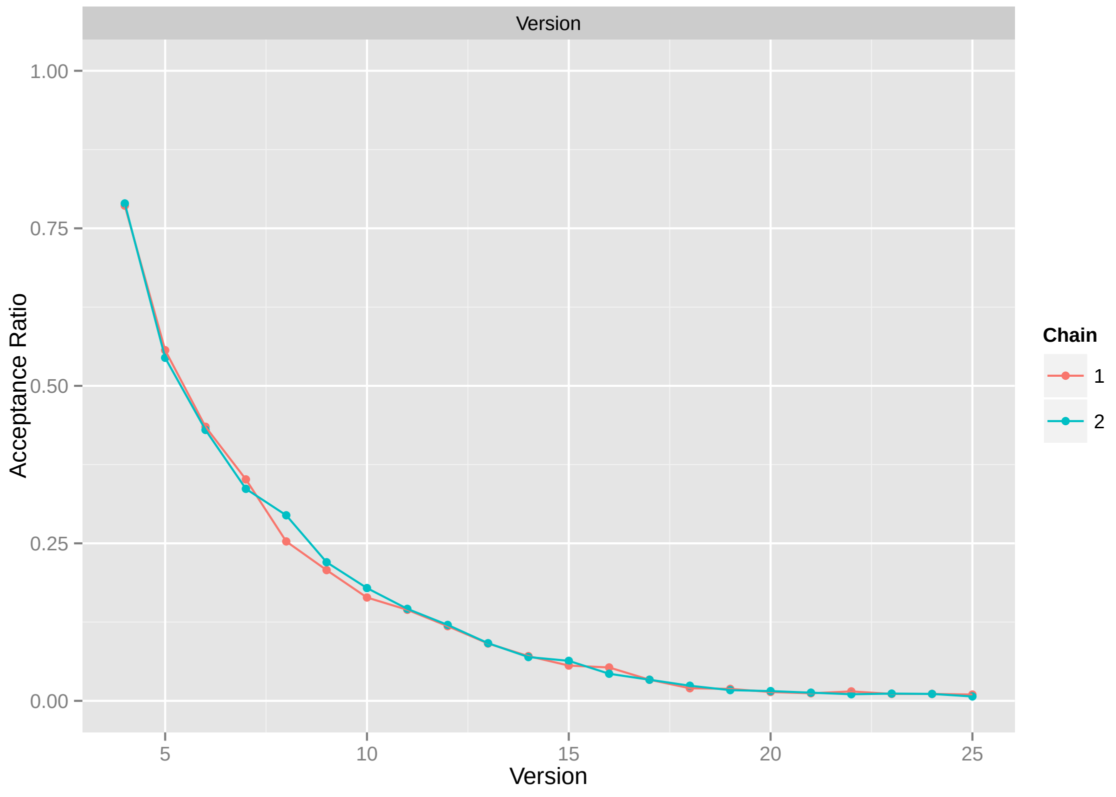


Fig. 5.5: A plot of the acceptance ratios by version for the Chapter 5 version model. This is based on an MCMC run with two chains and 2,000 iterations, however the same pattern is visible in chains with 100,000 iterations.

This pattern is all the more curious if we examine the trace plots of (α, d) for the later versions where we see that the traces still have not converged after 100,000 iterations.

To overcome the very low acceptance ratios for the most recent releases, a simple scheme to use a scaling of the proposals so they were smaller in a way which attempts to mirror the 'exponential decay' type curve of the acceptance ratios. Our solution is to apply a multiplier to the proposal variances. Here we create a global fixed parameter β which we use to set the 'decay rate' of the multiplier:

$$\text{Multiplier} = \frac{e^{-\beta k}}{e^{-\beta}} \tag{5.8}$$

where k varies from $1 : K$ and in our case $K = 22$, i.e. we have 22 versions (PreRapid, $5, \ldots, 25$). Through experimentation, we have found that a value of $\beta = 0.25$ gives an improved acceptance ratio for the more recent versions, i.e. for the higher values of k, than $\beta = 0$ which is a flat proposal rate across all k versions. Typical acceptance ratios might start at about 0.5 and rapidly decay within a few versions to an asymptote of about 0.1, but these values can be tweaked by adjusting β and the base proposal values for α, d.

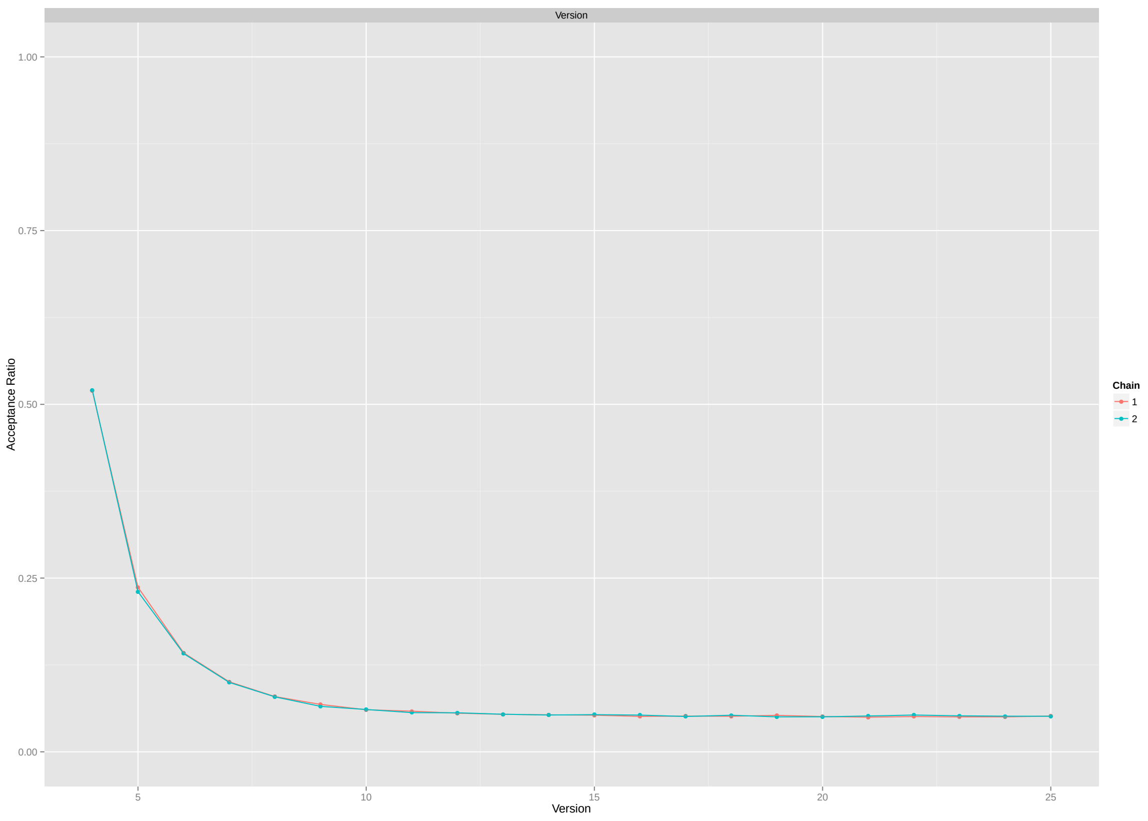


Fig. 5.6: A plot of the acceptance ratios by version for the Chapter 5 version model. This is based on an MCMC run with two chains and 40,000 iterations, with a decaying proposal as k increases and $\beta = 0.25$.

There is a trade-off between a ‘good’ acceptance ratio and good convergence of the traces, i.e. that a given chain for each of the parameters appears to have come to be stable about a particular value, and also that other chains come to be stable about the same value. In some cases, particularly at higher values of k, i.e. more recent versions, there can be a tendency to have chains which appear to ‘converge’, but each chain has ‘converged’ to a different value - which is a problem. However, in our case the data for the later versions is far from complete and in particular the data for the last version, 25, has only just started to be recorded, so we feel that the lack of proper MCMC convergence can be justified for the later versions - without destroying the argument for the model as a whole.

5.7 Validation with synthetic dataset

A synthetic dataset was created over the support $(0, 1)$, with three labelled Cauchy distributions with scale 0.2. The points from the set A were centred at 0.0, B were centred were at 0.55 and C were centred at 1.0. There were $(50, 91, 52)$ points in each of the sets A,B and C. We then labelled these points as 'unknown' with probability 0.25.

The distribution of the points in the dataset can be seen in Figure 5.7 where we show a rug plot of the three dataset with their corresponding original Cauchy distribution densities in black (A), red (B) and blue (C).

In Figure 5.8, we show a density plot of the three synthetic sets, weighted by the number of points in each set. The density curves we calculated using the "SJ" methodology, Sheather and Jones (1991) with a grid size of 512. In Figure 5.9 we show the corresponding weighted area density plot of the three synthetic sets.

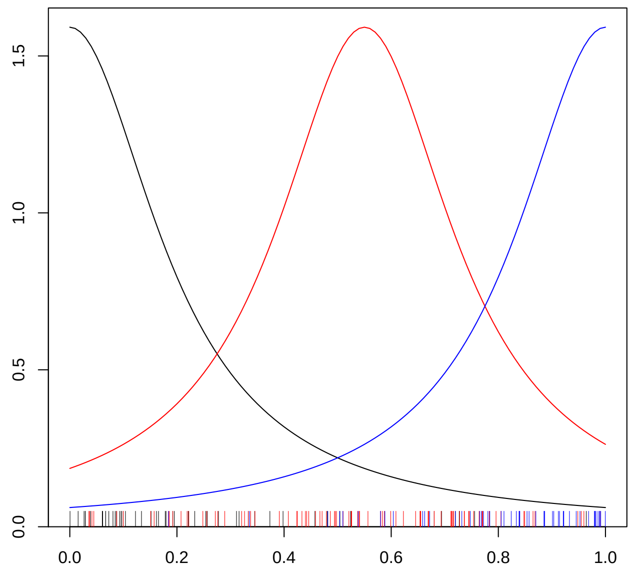


Fig. 5.7: A rug plot of distribution of the points in the synthetic dataset with their corresponding original Cauchy distribution densities in black (A), red (B) and blue (C).

We set the nominal 'true' values of α, to be $c(0.05, 0.5, 0.95)$ respectively for the sets (A,B,C) and we ran our MCMC code with four chains, each of length 200,000 iterations and the results are shown in Figures 5.10 to 5.12. The trace plots show that

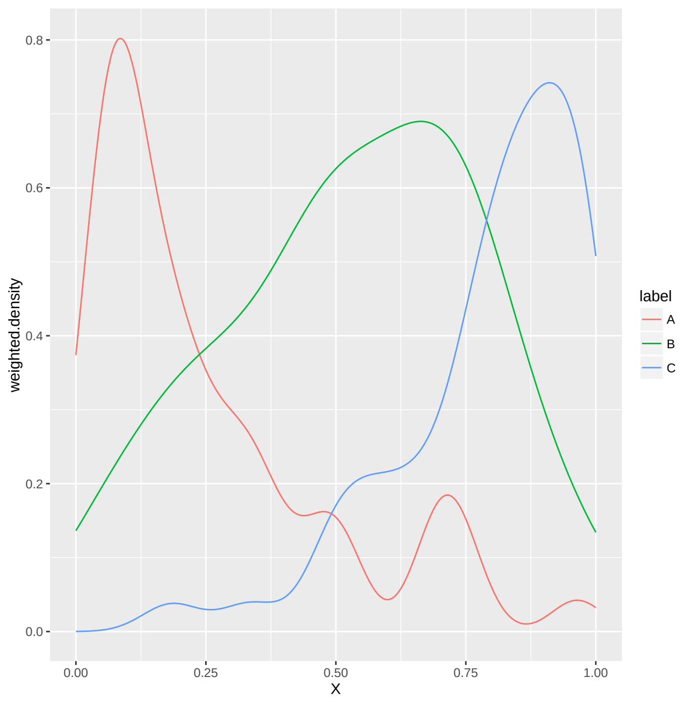


Fig. 5.8: A density plot of the three synthetic sets, weighted by the number of points in each set.

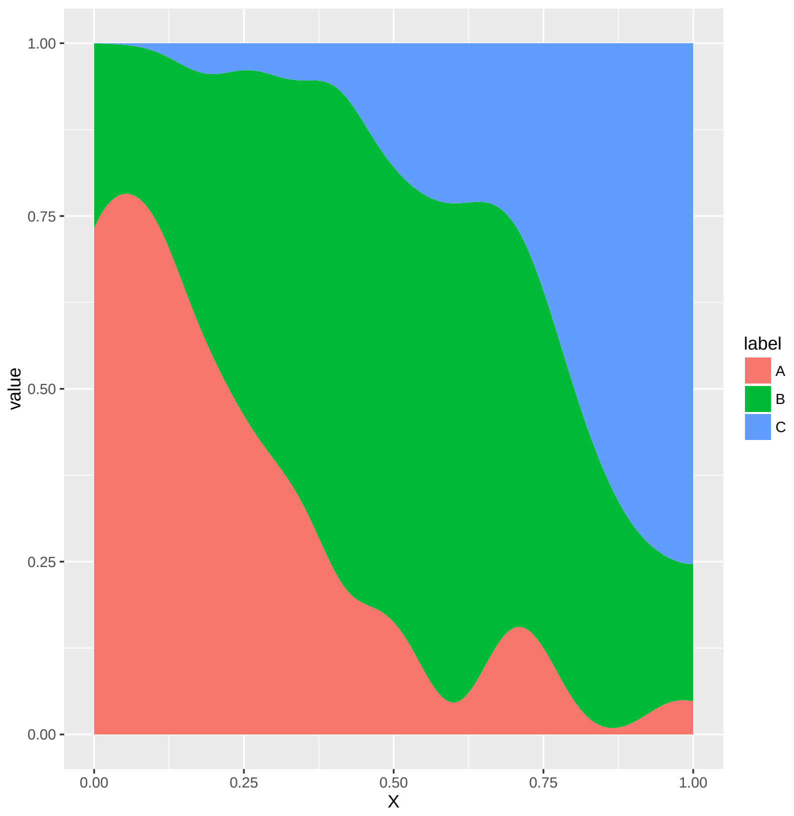


Fig. 5.9: An area density plot of the three synthetic sets, weighted by the number of points in each set.

the chains are not mixing particularly well.

```
Inference for the input samples (4 chains: each
with iter=200000; warmup=1e+05):

              mean    se_mean        sd        2.5%        25%
alpha.1 -0.5106671 0.06740459 0.3924952 -1.54378652 -0.6779093
alpha.2  0.2105992 0.25464919 1.0723841 -1.84682250 -0.1912538
alpha.3  0.5303048 0.11747280 0.3936637  0.08566444  0.2691472
d.1      2.2514311 0.15743345 1.1798654  0.60049799  1.3416202
d.2      2.4435176 0.59868077 2.2656859  0.16946966  0.5230970
d.3      2.7456790 0.22816172 1.4635900  0.67072027  1.5946887

                 50%        75%       97.5% n_eff     Rhat
alpha.1 -0.4086104120 -0.2370744 -0.03082931    34 1.146626
alpha.2 -0.0002651797  0.8568064  2.57493313    18 1.354177
alpha.3  0.4231749520  0.6688123  1.73586912    11 1.257576
d.1      2.0538821680  2.9462958  5.01700139    56 1.081420
d.2      1.4565390434  4.1432472  7.54789172    14 1.098672
d.3      2.5405387268  3.6468223  6.08915422    41 1.095703

For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

Fig. 5.10: The results of the MCMC simulation for the three class synthetic dataset. Note that the values of the alpha that we are expecting to see are $(-0.05, 0.05, 0.05)$, and clearly these are not close.

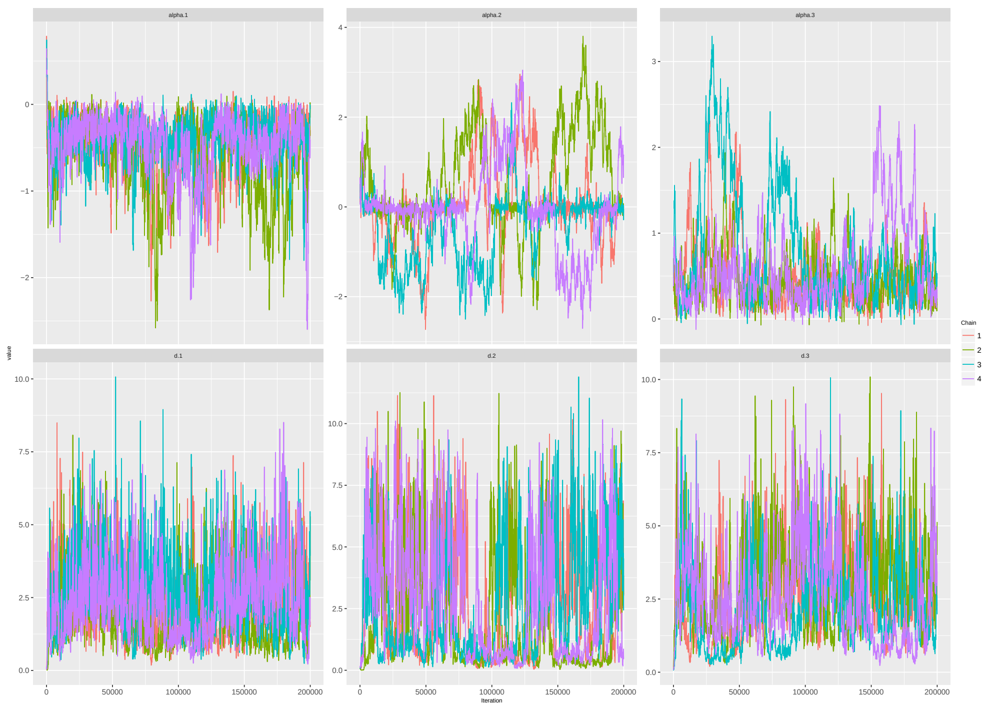


Fig. 5.11: Trace plots of the parameters $(\alpha_{i=1:3}, d_{i=1:3}$ for an MCMC simulation with 4 chains each of length 200,000 iterations. The top line of plots shows the parameter values of $(\alpha_{i=1:3}$ for each of the three sets and the bottom line of plots shows the corresponding values for $d = i = 1:3$.

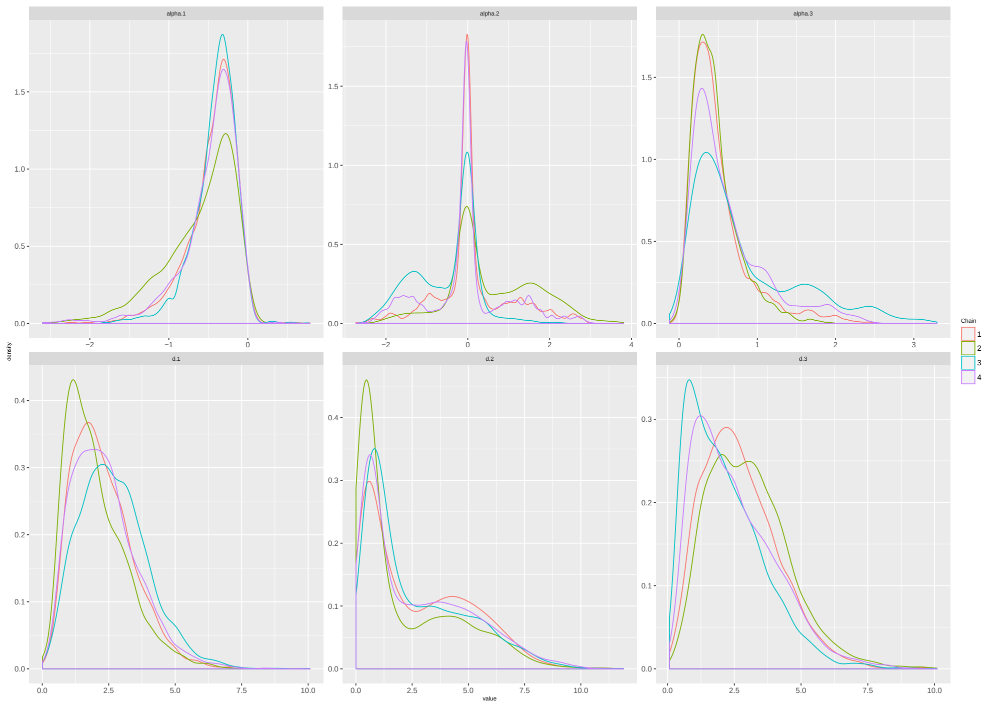


Fig. 5.12: Density plots of the parameters $(\alpha_{i=1:3}, d = i = 1:3$ for an MCMC simulation with 4 chains each of length 200,000 iterations. The top line of plots shows the parameter values of $(\alpha_{i=1:3}$ for each of the three sets and the bottom line of plots shows the corresponding values for $d = i = 1:3$.

5.8 Validation with Firefox-2013 dataset

To validate the model, we took our Firefox-2013 dataset and removed any bugs that have no version label, leaving us with 1,959 bugs out of the full dataset of 10,420 or 18.8% of the original dataset size. The next step was to temporarily pretend that we did not know some of the labels, and then impute these missing labels by sampling using our model. Finally we compared the predicted against the actual version labels using a confusion matrix and compared the accuracy of our predictions.

However, if we have N classes, and a uniform distribution of the classes, then there is a $1/N$ chance of predicting the correct class at random and the Kappa statistic of Cohen (1960) and described by Kuhn and Johnson (2013, Section 11.2) is intended to remove this random chance element.

$$\text{Kappa} = \frac{O - E}{1 - E} \tag{5.9}$$

Where O is the observed accuracy and E is the expected accuracy.

The actual implementation used was the function `confusionMatrix()` from the R package `caret` (Kuhn et al., 2014).

To validate our model we selected proportion Q of the full dataset, a subset which was selected at random, and temporarily pretended that we did not know the labels for this subset. We then imputed the labels and validated against the known label for this subset.

We selected nine random subsets with values of Q from 10% to 90% in increments of 10%. We then ran our label prediction model for a number of chains of length 500,000 iterations for each of the nine subsets. The predicted class for a bug is the one that was sampled most frequently in the MCMC. At each MCMC iteration we saved the full set of predictions and at the end of the run these were saved to a file, thus allowing us to do more detailed analysis afterwards. Typical average acceptance rates were of the order of 0.28 which is reasonably close to the Gelman suggested value of 0.24 for high dimensional MCMC.

Figure 5.13 shows a box plot of the *Kappa* score for each of the nine subsets of data and it is interesting to note that there is only a small decreasing trend as we rise from 10% of the dataset unknown to 90%. As we had the full chain information we also re-did the analysis for chains which were much shorter, i.e. for chains of lengths 1000,

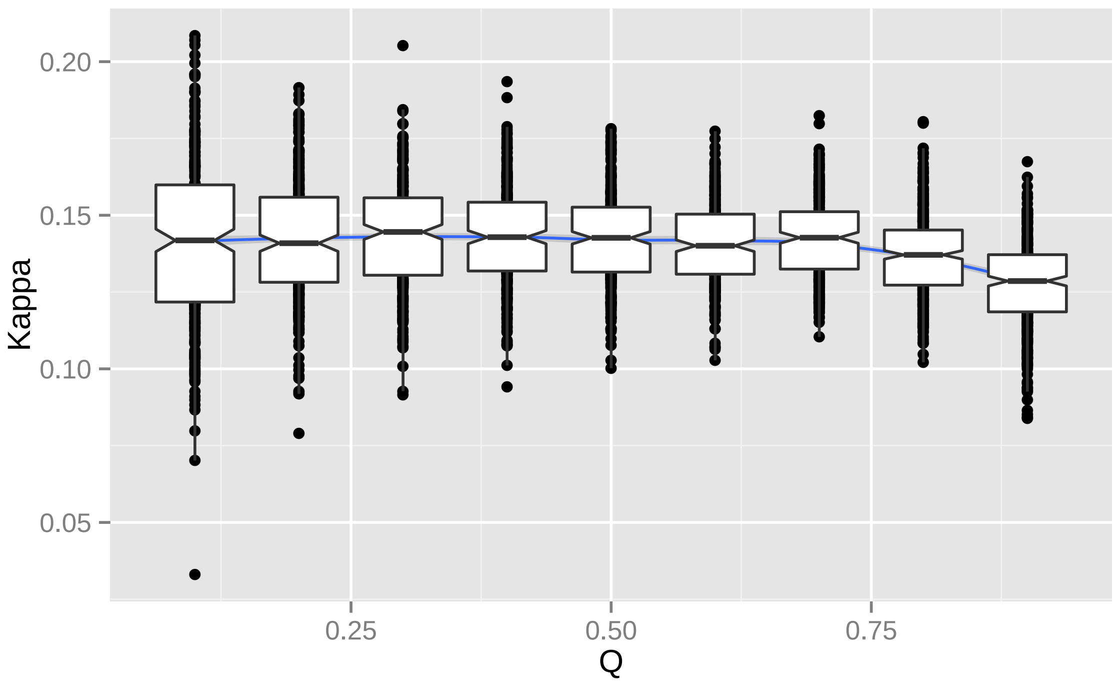


Fig. 5.13: Box plot of the values of Kappa for the nine increasing values of Q (proportion of missing data).

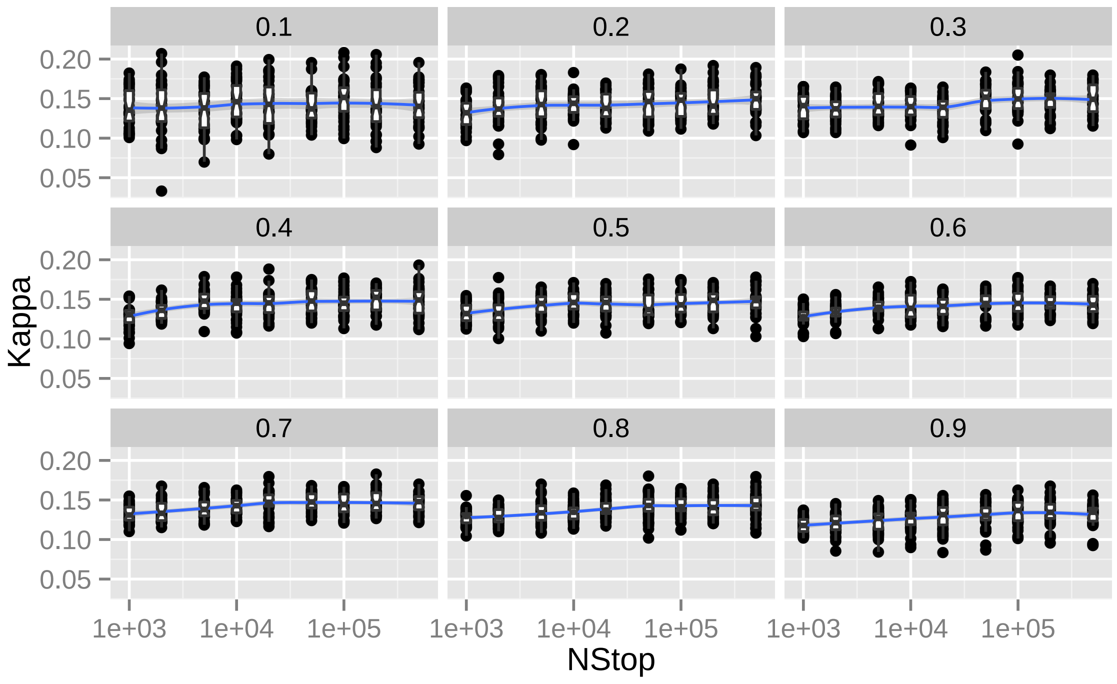


Fig. 5.14: Box plot of the values of Kappa against NStop faceted against Q (proportion of missing data).

2000, 5000, 10000, 20000, 50000, 100000, 200000 and 500000. The faceted box plot in Figure 5.14 shows the results of this re-analysis and despite poor values for Gelman's $\hat{R}$ (Gelman and Rubin, 1992), and thus low values of the estimated effective N, the results are reasonable for even relatively low numbers of iterations, i.e. prediction accuracy improves slowly with the number of MCMC iterations. As we might expect there is a slight increasing trend as we increase the number of iterations. Note that we used Gelman's recommendation of using only the second half of the chain in each case(Gelman et al., 2013). In total 294 chains were run with length 500,000 iterations, at the different values of Q.

We note that looking at the shape of the typical confusion matrix produced, they show a strong band down the diagonal, which we might expect given our model assumption that a label is more likely to be associated with a version whose release date is nearby, and the matrix labels are ordered.

Figures 5.15 and 5.16 show a set of trace plots for four chains and the corresponding density plots for an MCMC run of 500,000 iterations for $Q = 0.10$. Traces for the α, d parameters from the last few versions, 22 to 25, are notably less well behaved than those for the earlier versions, but this is to be expected as there is much less information. By this we mean that in the early days of a version, like 25 which was released long after the cut-off date for the dataset, the rate that bugs are being found is still climbing rapidly and has not yet levelled off as we see for the much older versions. We see this levelling off most clearly in Figure 2.3 for the 'PreRapid' bugs at the far left of the plot, and the steep rate of climb for releases 23 to 25 at the right hand side of the plot.

We have also looked at the change in labels between two MCMC iterations and a trace plot for one sample chain is shown in Figure 5.17. In each MCMC iteration the labels will randomly change under the likelihood model described earlier. How far will each bug change label between any two iterations - this is what we try to describe in Figure 5.17. We have N MCMC iterations and J bugs that are unlabelled and we have a matrix P which has N rows and J columns, and each element of the matrix holds the sampled version label for that bug for that iteration. For a given MCMC iteration $i = 2 \ldots N$, the Change between this MCMC iteration and the last is:

$$\text{Change}_i = \frac{\sum_{j=1}^{J} |P_{i,j} - P_{i-1,j}|}{J} \tag{5.10}$$

The intent of this metric is to give a sense of the "average" distance that a label will

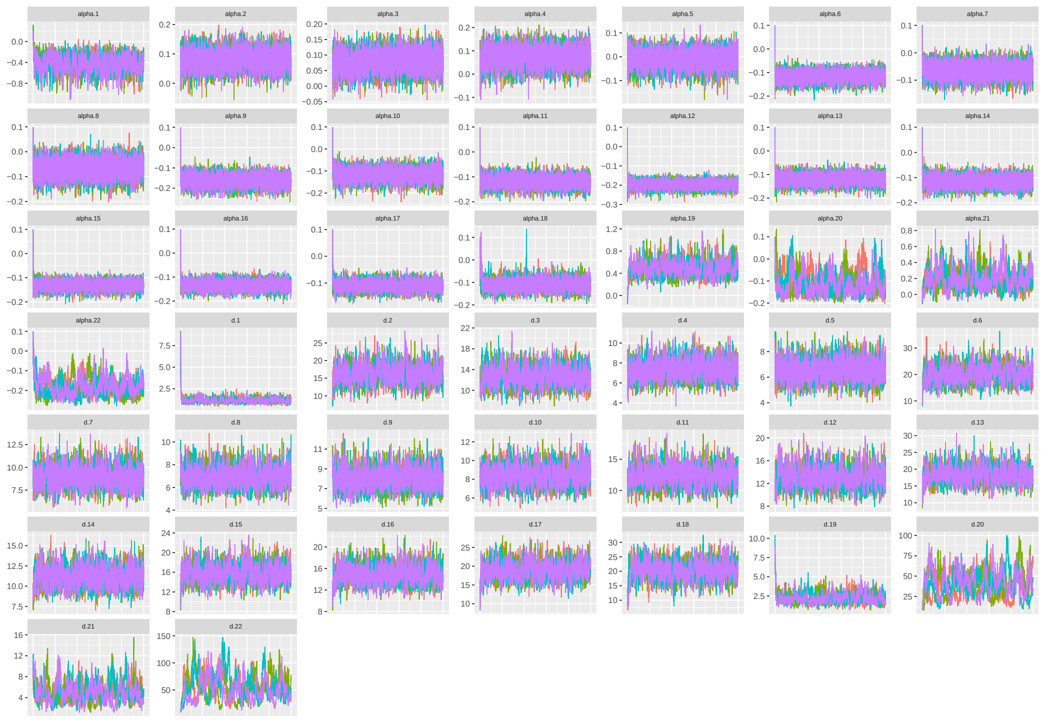


Fig. 5.15: Trace plots of all α and d for four chains for $Q = 0.1$ and 500,000 MCMC iterations. Note that from the top left the first 22 plots are the traces of $\alpha_{5:25}$ followed by the corresponding $d_{5:25}$.

change each iteration.

In Figure 5.17, we have over plotted the change value in red using Friedman's SuperSmoother, (Friedman, 1984*a*,*b*) implemented in the R function `stats::supsmu()` which is a very strong smoother and it is clear that there is no particular trend, and the smoother values start at 2.15 and after half a million iterations, the last value is 2.154295.

In Figure 5.18, we show a plot of the empirical probability that a given bug is from a particular version. Note that the bell curves strongly overlap, particularly in the region between bug numbers 2000 to 4000. Note also that initially, at the start of 2011, the Rapid-Releases do not exist so the probability that a bug is from the PreRapid releases is almost 1.0. The slightly jagged shape of the curves is due to the empirical nature of the curves combined with plotting on bug number on the x-axis rather than time.

We have experimented with different priors for (α, d) and we have observed small shifts in the plot and it is possible to have wider or narrower 'bells', but essentially the results are similar.

In Figure B.1, we can see a table of the results for one run of 100,000 iterations of

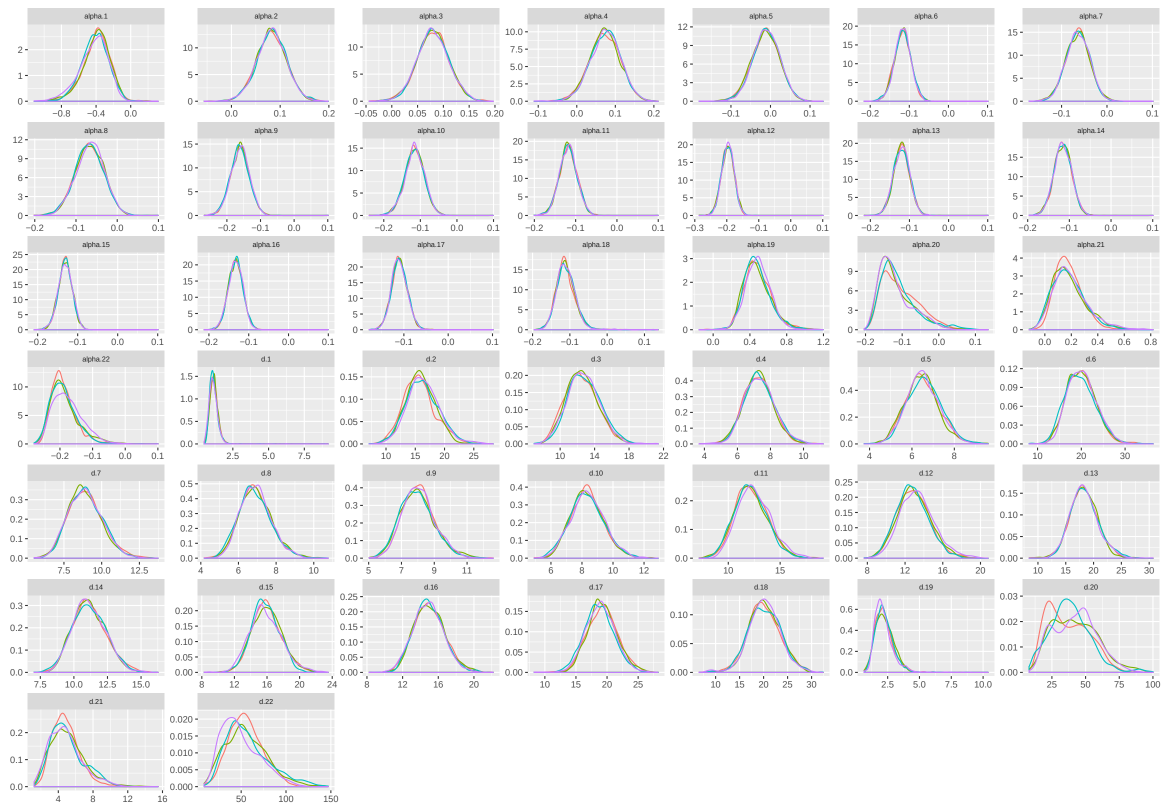


Fig. 5.16: Density plots of all α and d for four chains for $Q = 0.1$ and 500,000 MCMC iterations and corresponding to the traces in Figure 5.15. Note that from the top left the first 22 plots are the densities of $\alpha_{5:25}$ followed by the corresponding $d_{5:25}$.

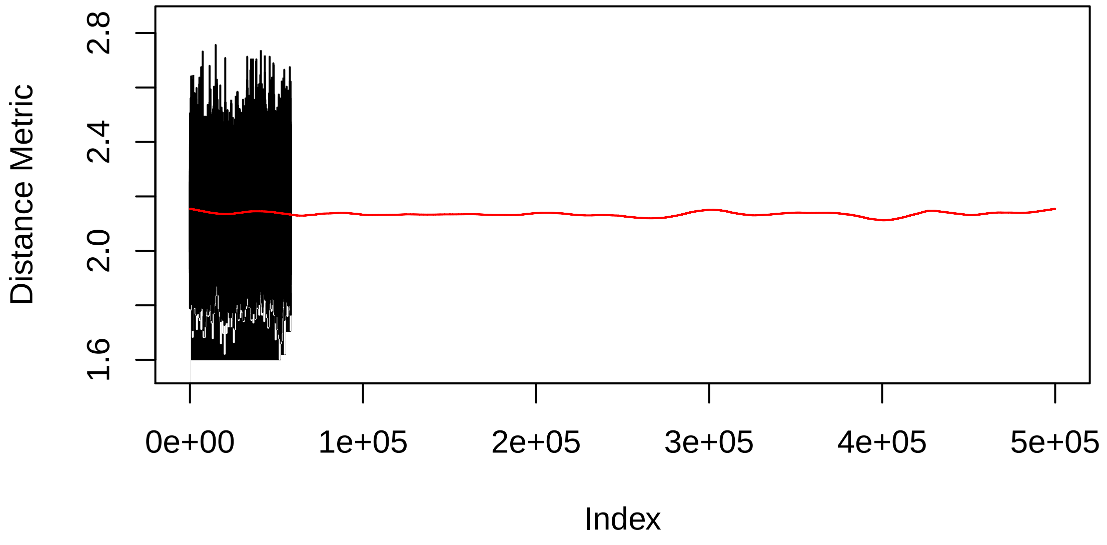


Fig. 5.17: Plot of the change in label for one chain for $Q = 0.1$ where the MCMC iteration is on plotted on the X axis. Described further in the text.

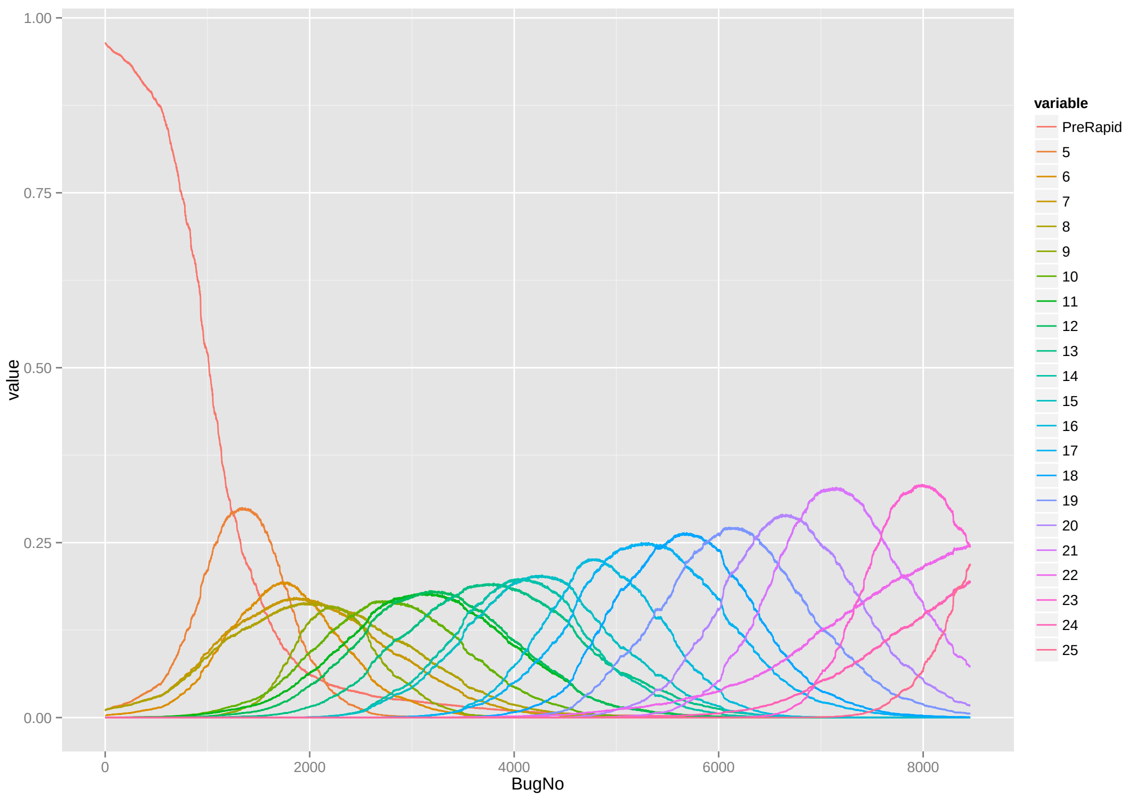


Fig. 5.18: A typical plot of the empirical probability that a given unlabelled bug is from a particular version.

the version model with 4 chains. Note in particular that the values for the effective N are highest for the versions near the low teens, i.e. in the middle, but that they do not necessarily coincide for α and d. This might imply that the Metropolis update for the d parameter could be better tuned, which could of course be done adaptively. We also note that the maximum effective N is 1111 which is 0.56% of the 200,000 draws examined which is in stark contrast with the results for the Goel-Okumoto model in Figure A.1 which reaches 100% in some instances for only 20,000 draws examined.

5.9 Discussion

Our results described in Section 5.8 show that the ability to predict the version labels is far from perfect, but it is genuine as can be seen in Figure 5.18, and shows promise. This is due to the substantial overlap in bug reporting with release times. So the question that arises is whether the model captures this uncertainty well. There is uncertainty with how a given bug is labelled and we think that our model captures this uncertainty.

One issue with the use of the *Kappa* statistic is that it is a single number summarising a 22×22 confusion matrix. Clearly there is an issue with the loss of degrees of freedom and it is clearly impossible to summarise this succinctly with a single number.

A further issue with the use of the *Kappa* statistic is that there are a number of different versions of *Kappa* for ordered classes which is done by appropriate weighting. How these weighting might be used to better understand the results has not been explored due to time constraints.

Unbalanced classes is a further issue which we have mostly ignored. The numbers of bugs in each version class are moderately imbalanced before relabelling (refer Table 2.1). If we choose the most smallest and largest classes, with 23 bugs counted towards release 25 and a count of 135 bugs towards release 17, a ratio of $\frac{135}{23} = 5.9$. For one particular sample, after relabelling the ratio counts are 18 and 294 for the same releases, 25 and 17 respectively, a ratio of $\frac{294}{18} = 16.3$.

Clearly there is a considerable amount of further work that could be done with the measure of change, in the labels, as seen in Figure 5.17. One possibility would be to look at things vertically instead of horizontally, does the "amount" of change for a given bug change as the MCMC chain progresses?

As there is a considerable amount of data, and comparatively few parameters, a very significant increase in speed could be achieved by re-writing the likelihood function for a GPU. The big issue with using a GPU is the copying of large quantities of data to the GPU from the computers main memory and vice versa. The speed of one of the 'processors' in a GPU is an order of magnitude slower than one core in a modern CPU, however even this four year old laptop has 96 GPU cores, and more recent commodity GPU cards on desktop hardware have 2048 cores[3]. With a sufficiently large power supply, a number of these cards can be fitted into one chassis to work in parallel. Each GPU generation is running faster and with more cores. As the data itself could be cached in the GPU memory and only the $K = 44$ parameters (α_i, d_i) are passed to the GPU at each iteration this should mean that using a GPU would significantly increase the speed of the calculations. In fact, the entire Metropolis algorithm could be run on the GPU.

It has been suggested in Semi-Supervised Learning that sometimes it is better to

[3]Nvidia GTX 980 `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980/specifications`

feed in unknown labels slowly and then use the results of the imputation to better label the next incremental feed of unknown labels. Unfortunately due to time constraints, this has not yet been explored.

We have worked with different priors, by changing the size of the variance and the mean for α and the rate for d, and it appears to make little difference except to make the 'bells' in Figure 5.18 more or less uniform in height and width.

Chapter 6

Combined Model

6.1 Introduction

In Chapter 4 we discussed the Goel-Okumoto model as a stand alone model given a full data set. In Chapter 5 we discussed a model for imputing the missing version labels using a Gaussian kernel. In this chapter we present and discuss a model which combines both of the above models in a coherent way.

Following Bishop (2006, Section 1.5) and Bishop (2007), a *generative* approach explicitly or implicitly models the distribution of inputs as well as outputs, because by sampling from them it is possible to generate synthetic data points in the input space. By contrast, a *discriminative* approach directly models the posterior probability.

Our version model from Chapter 5 can be viewed as a discriminative model, $p(r_i|t_i)$, the distribution of version given discovery time, refer to the DAG in Figure 6.1. Note the arrow directions ($t_i \to r_i$). By contrast our Goel-Okumoto model from Chapter 4 is a generative model, $p(t_i|r_i)$, refer Figure 6.2. Note the arrow directions ($r_i \to t_i$), where r_i is a class and t_i is a feature in Bishop's terminology.

When we try and combine these two models together we end up with a 'DAG' which has an arrow both directions between the parameters version label, r_i, to time, t_i, which clearly will not work. We have resolved this by jointly modelling the t_i and the r_i as can be seen in Figure 6.3.

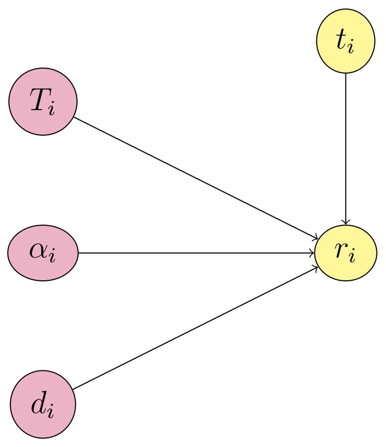


Fig. 6.1: A DAG for the version model.

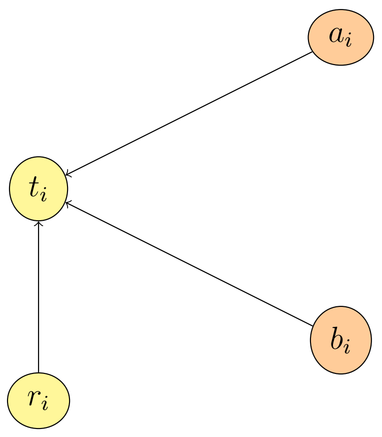


Fig. 6.2: A DAG for the Goel-Okumoto model.

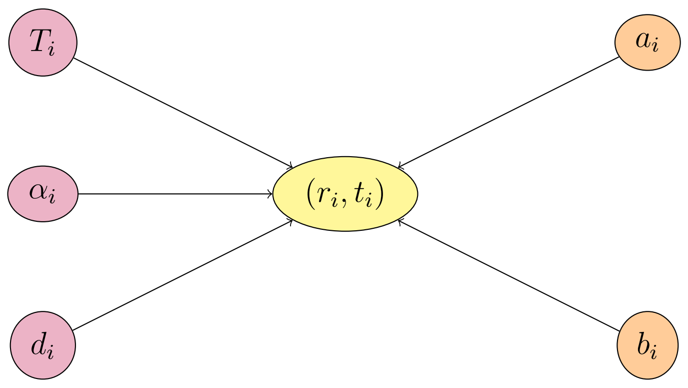


Fig. 6.3: A DAG for the Combined model.

6.2 Model details

The combined model relies on the version model in Chapter 5, namely that we use the version model to impute version labels, and then we use the Goel-Okumoto likelihood function using time stamps for the known and imputed labels to obtain distributions for a_k, b_k for each version k. The likelihood of labels are imputed, as in Chapter 5, using:

$$L(d, \alpha | t_{1:N}, T_{1:K}, (r_{1:N})) = \prod_{i=1}^{N} \frac{\exp\left(-d_k \left(t_i - T_k - \alpha_k\right)^2\right)}{\sum_{j=1}^{K} \exp\left(-d_j \left(t_i - T_j - \alpha_j\right)^2\right)} \tag{6.1}$$

We then sample from the imputed labels. However, because of the dependence as seen in Figure 6.3, we need to allow for the Goel-Okumoto model in our sampling procedure. The details of this are in Appendix C.

The full conditional for r_i, where $r_i = 1, \ldots, K$ is:

$$p(r_i = k | \underline{t}, \underline{a}, \underline{b}) \propto p(\underline{t} | \underline{a}, \underline{b}, \underline{r}) p(\underline{a}) p(\underline{b}) p(\underline{r}) \tag{6.2}$$

$$\propto b_{r_i} a_{r_i} e^{-b_{r_i} t_i} \times p(r_i) \tag{6.3}$$

where r_i is the release version for bug i. This allows us to sample a set of bugs with label $r_i = k$ from the bugs with no labels, and then combine them with the bugs with known label $r_i = k$ to get a vector of time stamps when bugs were recorded for version k.

This can then be put into the model for Chapter 4 in a straight forward manner - with the minor difficulty that the time stamps need to be adjusted to be ‘small’ and positive so that when we calculate the exponential of a sum we do not overflow the numerical capacity of a real number represented on the computer in limited precision as an IEEE double (Goldberg, 1991; IEEE Task P754, 2008). To do this we used the minimum value of the time stamp for each version for the known data, i.e. the minimum t_k for the known data and just pass this in - then the probability for any t_i which is less than this T'_k is just zero, i.e.

$$T'_k = min\{t_i | r_i = k\} \tag{6.4}$$

$$p(r_i = k | \underset{\sim}{t}, \underset{\sim}{a}, \underset{\sim}{b}, \underset{\sim}{\alpha}, \underset{\sim}{d}) \propto \begin{cases} b_k a_k e^{-b_k(t_i - T'_k)} e^{-d_k(t_i - T_k - \alpha_k)^2} & t_i > T'_k \\ 0 & t_i <= T'_k \end{cases} \tag{6.5}$$

$$k = 1 \ldots K \tag{6.6}$$

Thus we have the combined model:

$$p(\underset{\sim}{\Theta}, \underset{\sim}{\Psi}, \underset{\sim}{r}_{n+1:N} | \underset{\sim}{t}, \underset{\sim}{r}_{1:n}) \propto p(\underset{\sim}{\Theta}) p(\underset{\sim}{\Psi}) \left(\prod_{i=1}^{N} p(\underset{\sim}{t}_i | \underset{\sim}{\Theta}) \right) \left(\prod_{i=1}^{N} p(\underset{\sim}{r}_i | \underset{\sim}{\Psi}) \right) \tag{6.7}$$

Where $p(\underset{\sim}{t}_X | \underset{\sim}{\Theta})$ is the Goel-Okumoto likelihood for $\underset{\sim}{t}_X$ and $p(r_i | \underset{\sim}{\Psi})$ is the version model for $\underset{\sim}{r}_X$.

$$\underset{\sim}{t}_X = \{t_i | r_{i=X}\} \tag{6.8}$$

$$\underset{\sim}{\Theta} = \{(a_x, b_x) | x = 5, \ldots, 25\} \tag{6.9}$$

$$\underset{\sim}{\Psi} = \{d_x, T_x, \alpha_x) | x = 5, \ldots, 25\} \tag{6.10}$$

In merging the two models, we started with the code implementing the Chapter 5 model, then we:

- We added the extra parameters for the Goel-Okumoto model from Chapter 4, i.e. (a, b), along with their initialization.
- The way in which the sample from P_{ver} was chosen had to be modified because of the joint dependency between t and r as described in Section 6.1. Obviously the Goel-Okumoto parameters, (a, b) are also passed into this procedure.
- The Goel-Okumoto model code was inserted after suitable extraction and correct labelling of the bugs from the P_{ver} matrix, as described in Section 5.5. This code was parameterized so that we can repeat it a number of times within each MCMC iteration to get better convergence of the (a, b) parameters.
- When the time stamps for the original and the imputed bugs for a particular version have been extracted, there will inevitably be a small number of bugs, most likely from the original dataset which are assigned to a particular version

a long time before the release date. This gave the growth curve an elongated flattened 'S' shape where the bottom half of the 'S' left a very long and flat tail to the left. We manually looked at the history for a sample of these bugs and found that they were incorrectly assigned, or had been originally assigned to a previous version but for operational reasons due to resource constraints had been re-assigned to the current version. Since we found no such 'early' bug which was correctly assigned, we decided to discard the handful of bugs which were so assigned if they were more than three release periods - i.e. 3×42 days before the official release date for a particular version.

- To improve convergence we changed the initialization for the parameters $\vec{\alpha}, \vec{d}, \vec{a}, \vec{b}$ so that they are done individually instead of en masse with the same value for each vector - based on experience running the chains - in effect this means that a chain running for say 15,000 iterations has a much longer effective length since it is starting from values similar to the closing values of a previous chain. That said, the initial values are still jittered by 10% for each chain.

Figures 6.4 and 6.6 show the trace plots for two different MCMC runs with 100,000 iterations and four chains each. The corresponding density plots can be seen in Figures 6.5 and 6.7. It is interesting to note that the densities for the parameters (α, d) for imputing the versions seem less stable than the Goel-Okumoto parameters (a, b); this can be at least partly explained by the fact that (α, d) pairs are only updated every $K = 22$ iterations, whereas for the Goel-Okumoto parameters (a, b), all K parameters are updated every iteration. As we have been hand turning the proposal parameters, and it is a slow process, we have not managed to get the acceptance rates for the version model above 0.006 - which is clearly very low and this must also be a reason for the apparent 'noisiness' and poor quality of the density plots. Details of the distributions of the posterior parameters along with corresponding $N_{\text{effective}}$ and $\hat{R}$ can be seen in Figure C.1. Note that the effective N for the version model parameters is lower than for the Goel-Okumoto parameters as we would expect given that the former are updated only every $K = 22$ iterations.

It is also interesting to note that in Figures 6.4 and 6.5 we can see that the parameters `alpha.18, d.18` show one chain, of the four, which does not seem to converge to the same mode as the other three chains. When we look at the corresponding

Goel-Okumoto parameters, `a.18, b.18`, we note that there is no noticeable difference between the four chains. This shows that our overall goal of modelling the Goel-Okumoto parameters (a, b) for the imputed data is robust to significant perturbations in the version model.

In our version model, we created a matrix, `pver` which we used to store the imputed versions for each bug at each iteration. Each column in the matrix corresponds to a particular bug and each row corresponds to an MCMC iteration. We used the `pver` matrix to examine the effectiveness of increasing the number of MCMC iterations on the predictive power on accuracy, or more precisely *Kappa*. In the combined model, we calculate the `pver` matrix slightly differently because we need to take account of the dependence between the two models, as seen in Figure 6.3 and discussed above.

Figure 6.8 shows a plot of the imputed bug labels, i.e. from the `pver` matrix, for the bugs without labels. We can see that approximately the first 1,000 bugs are from the first version, i.e. *PreRapid*. Then as time passes, we come across bugs in the known set and we allow these versions to be used with the imputed set. We can see that typically, there is a band approximately 7 versions wide for a given bug.

In Figure 6.9 we show a plot of our normalized distance metric, Equation (5.10), for the first 1,000 iterations of the model on one chain (removing the first two zeros for clarity). By normalized, we mean that we have divided the metric by K to give the mean absolute distance that each label has moved per iteration.

When we compare the results for the parameters (α, d) for the version model in Chapter 5 as seen in Figure B.1, with the results for the combined model which can be seen in Figure C.1, the values of the mean show a remarkable similarity. On the other hand when we look at the values of the predicted Goel-Okumoto parameters (a, b) and compare the values with those of the Chapter 4 model as seen in Figure A.1, the results are significantly different. For the combined model the mean values for a are significantly higher as there are many more labelled bugs to start with, e.g. in the Figure A.1, the mean value of `avec[10]` is 120.6 whereas in Figure C.1, the corresponding value of `a.10` is 1605.7. There are a total of 1,959 labelled bugs used in Chapter 4 with 99 labelled as being due to version 10. By comparison, there are 10,420 bugs used in the Combined model in this chapter and in one sample iteration chosen there were a total of **1,120** bugs labelled as version 10 in the P_{ver} matrix which

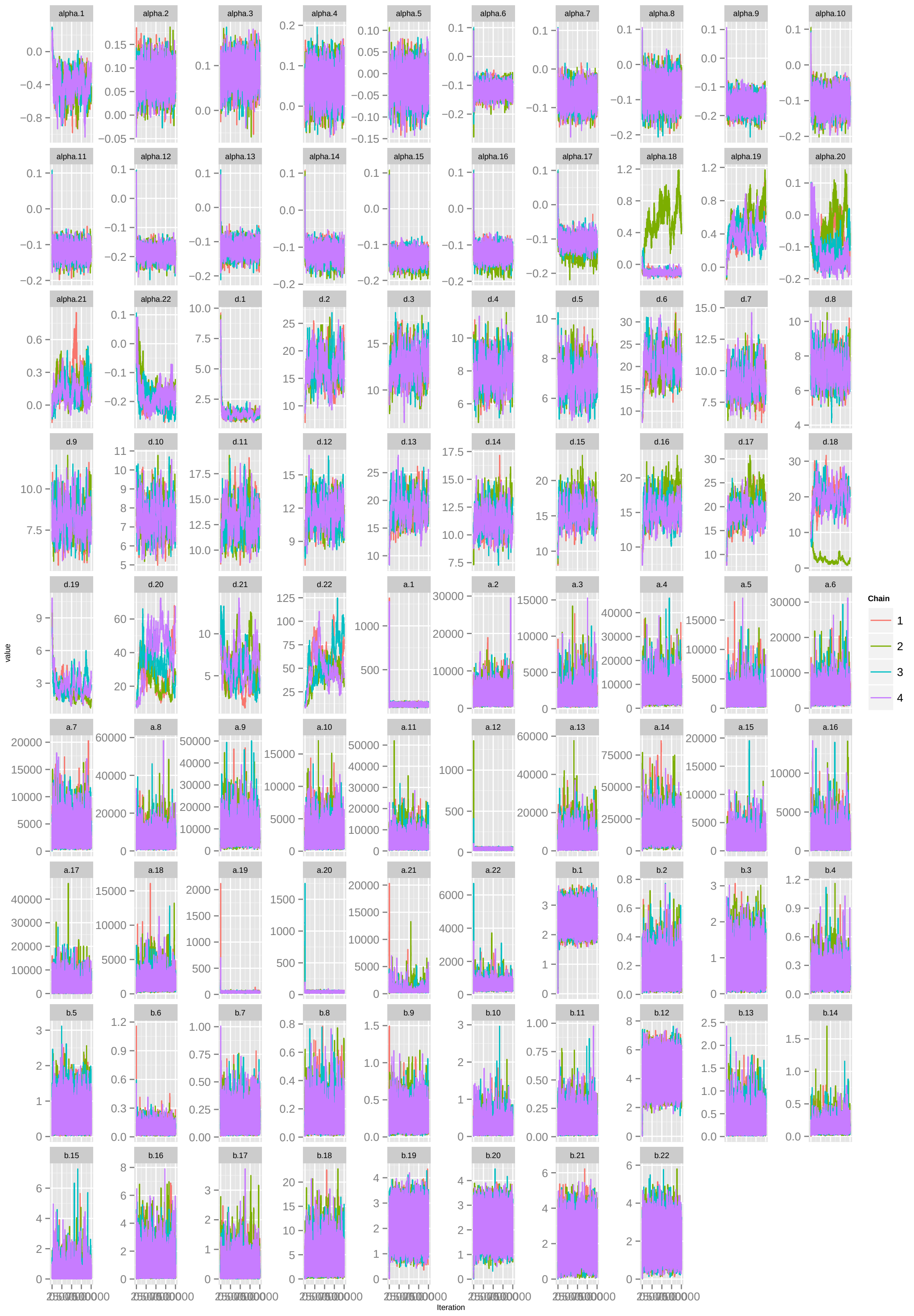


Fig. 6.4: A trace plot of four chains for 100,000 iterations of the Combined model. Note in particular the excursion for d.18 and alpha.18 and the lack of corresponding excursion for a.18 and b.18.

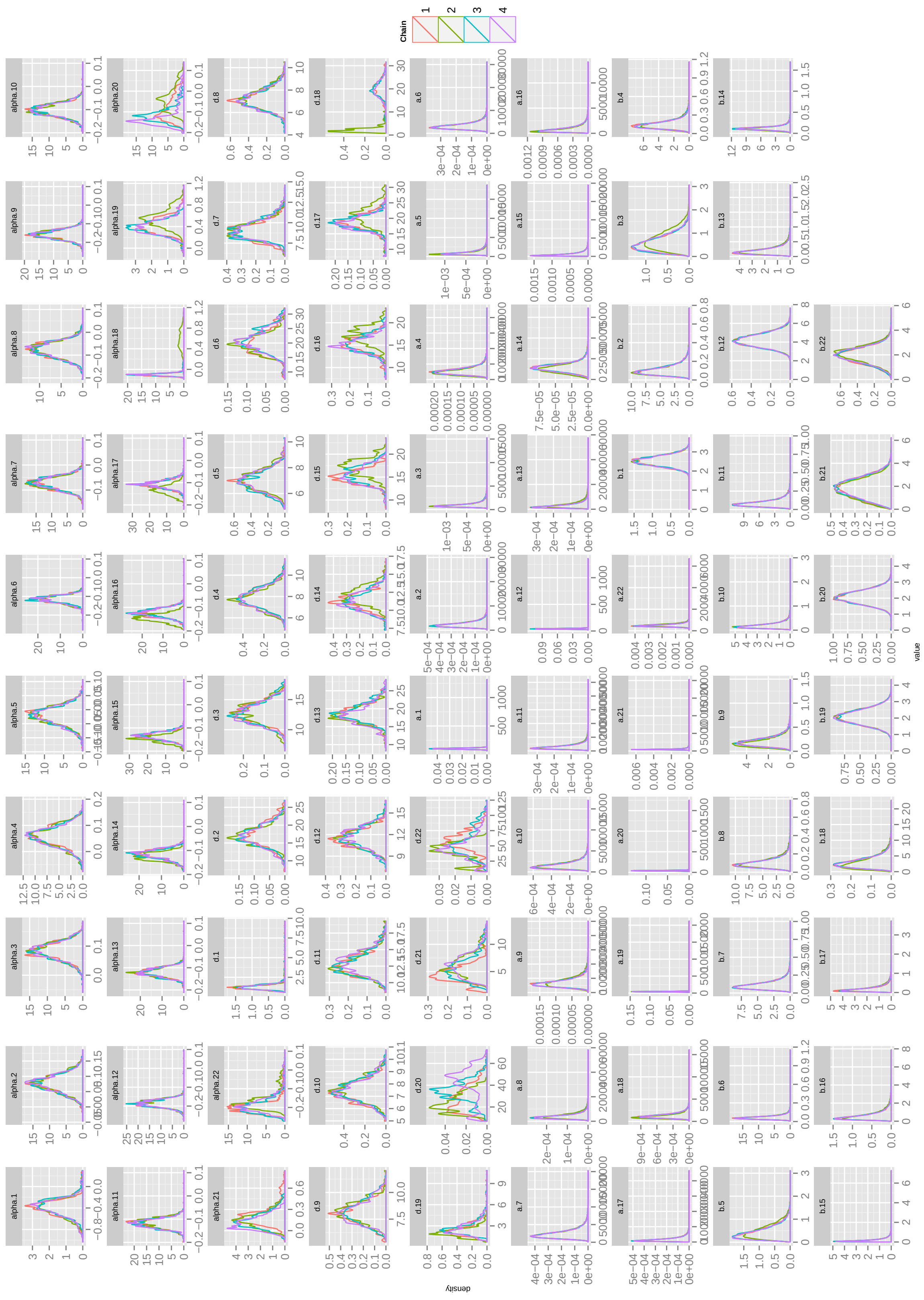


Fig. 6.5: A density plot of four chains for 100,000 iterations of the Combined model. Note in particular the excursion one chain for d.18 and alpha.18, i.e. the $(d, \alpha$ for Firefox version 21, and most interestingly the corresponding Goel-Okumoto model parameters, a.18 and b.18, seem quite robust to this.

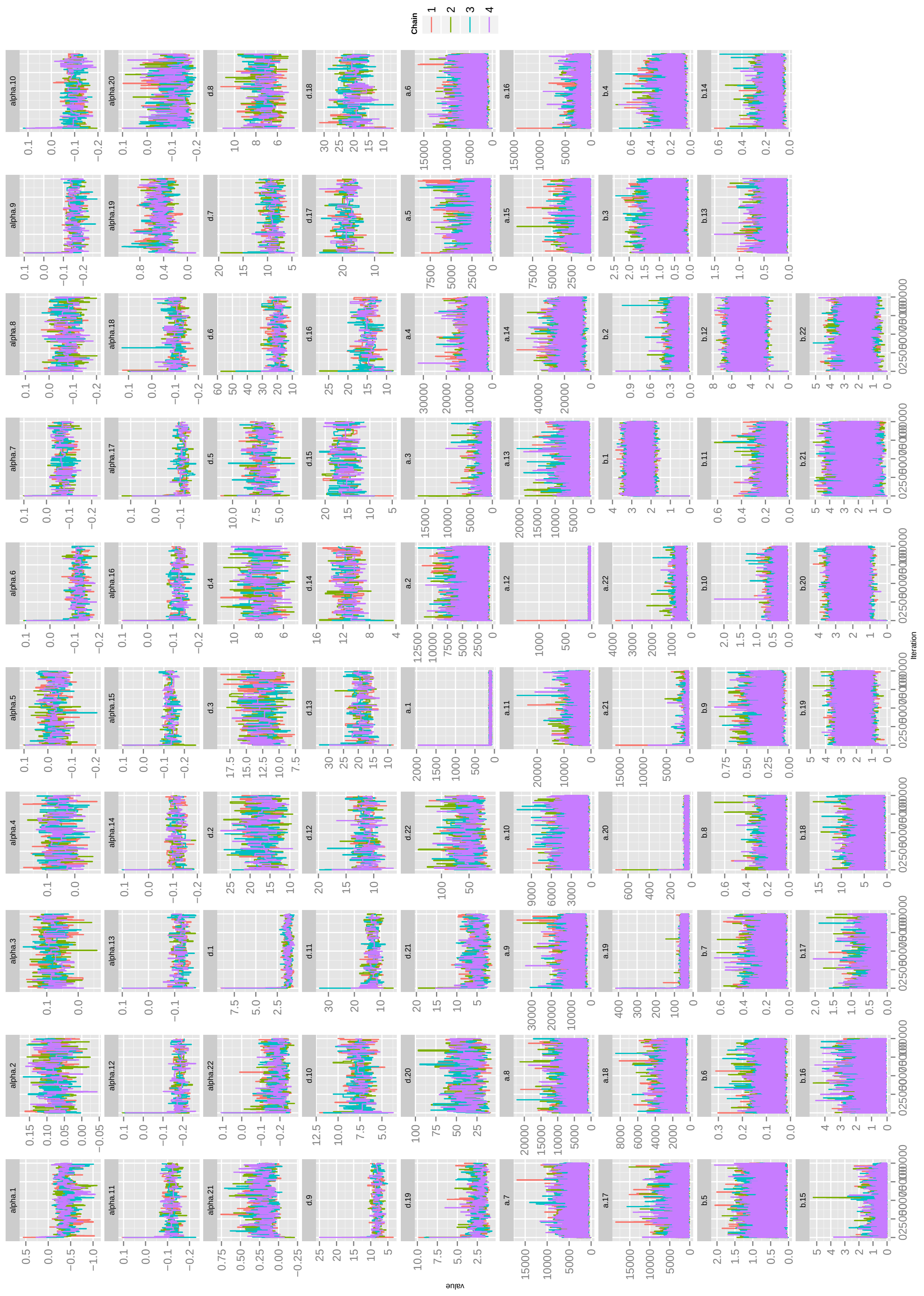


Fig. 6.6: Based on another run, a trace plot of four chains for 100,000 iterations of the Combined model.

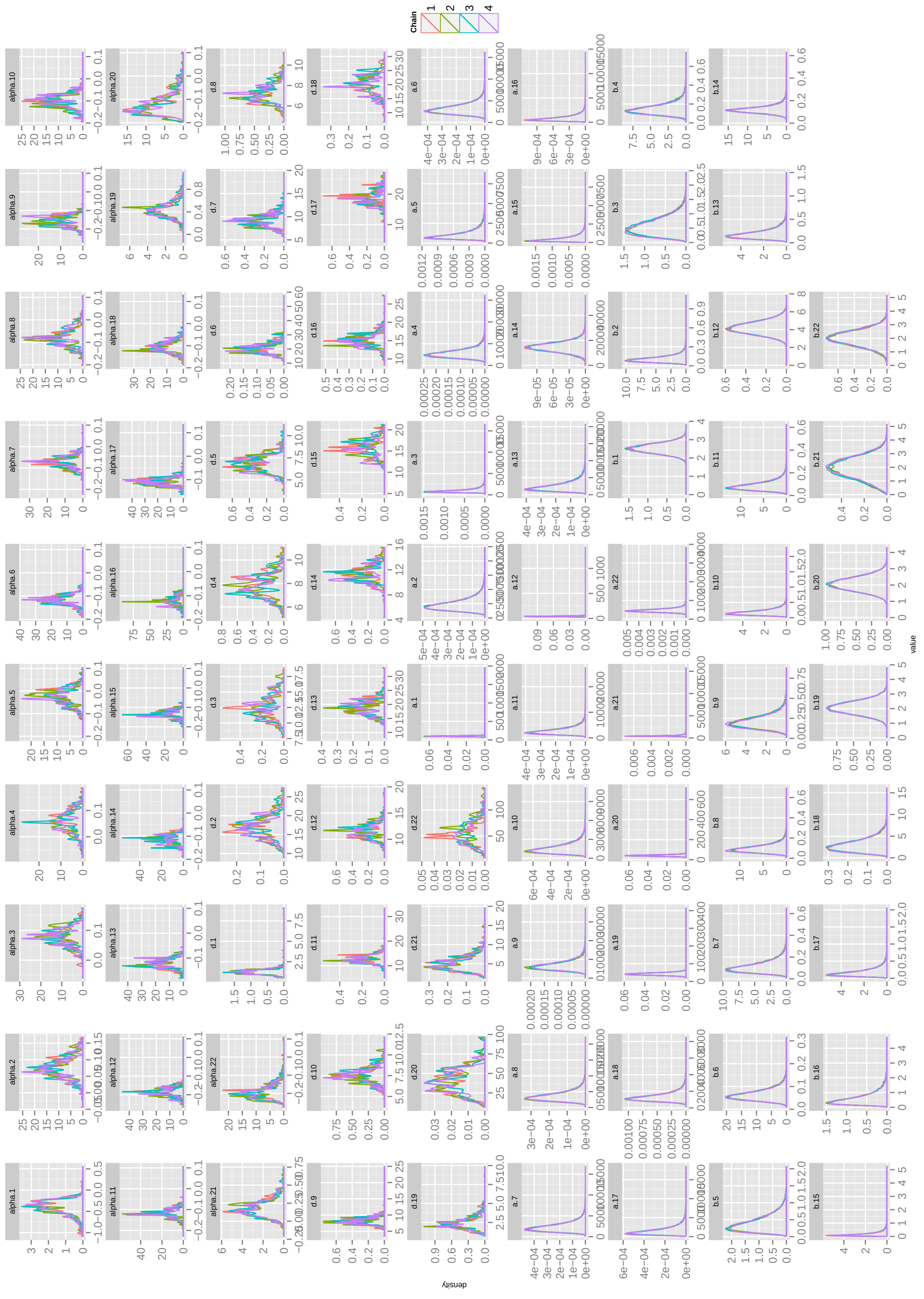


Fig. 6.7: Based on another run, a density plot of four chains for 100,000 iterations of the Combined model.

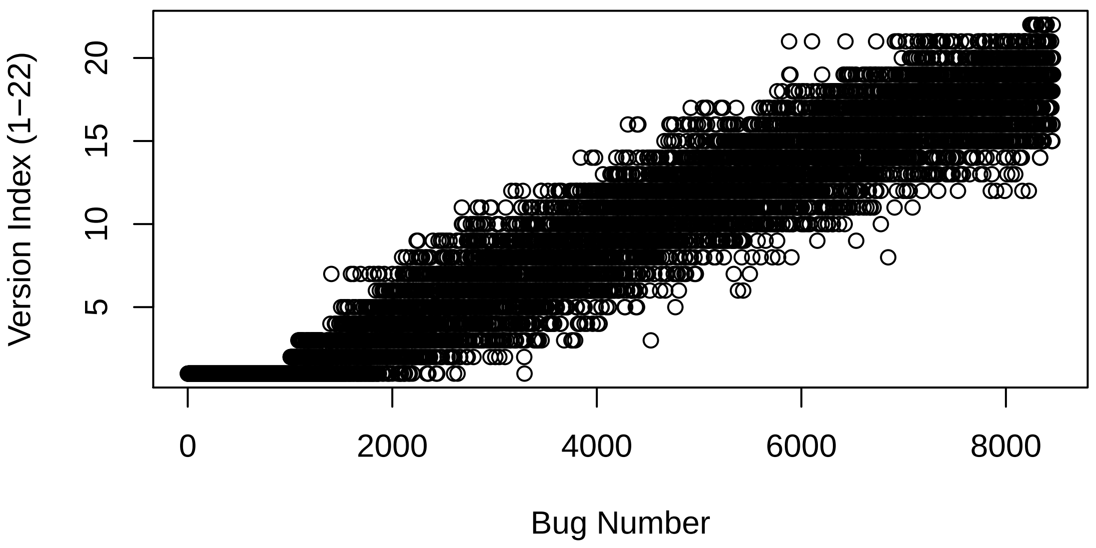


Fig. 6.8: A plot of one sample of bug labels returned by the combined model before being processed by the Goel-Okumoto model. This was taken early in an early MCMC iteration before warm-up had completed and is illustrative. In particular we can see that version index of 1 (PreRapid) applies to the early bugs only and version 25 has only been applied to the final bugs.

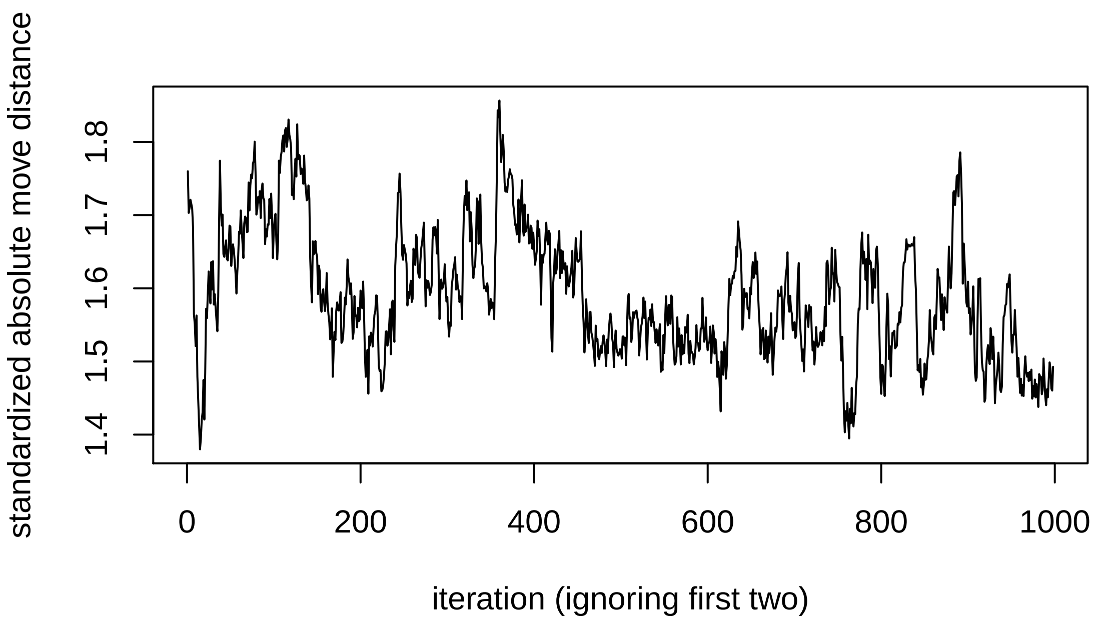


Fig. 6.9: A plot of the normalized absolute difference between two iterations in the version section part of the combined model.

were originally labelled as such and imputed labels.

6.3 Hierarchical Model

In order to allow us to make inference on the Goel-Okumoto parameters (a, b) which in turn sit on the version model, we need to have exchangeability of the parameters which we do by re-writing the model so that the Goel-Okumoto part of the model is hierarchical.

In particular, we will use a hierarchical model for the (a, b) parameters and assume a gamma distribution. The hyper-parameters will thus be $(A_{\text{rate}}, A_{\text{shape}}, B_{\text{rate}}, B_{\text{shape}})$ which we derive as follows, firstly we will look at A_{rate}, noting that the derivation for $(A_{\text{rate}}, A_{\text{shape}})$ can be applied to $(B_{\text{rate}}, B_{\text{shape}})$ by simply swapping B for A.

$$\begin{aligned} p(A_{\text{rate}}|\ldots) &\propto \left[\prod_k p(\underline{t}_k|a_k, b_k) p(a_k|A_{\text{rate}}, A_{\text{shape}}) p(b_k|B_{\text{rate}}, B_{\text{shape}})\right] \\ &\quad \times p(A_{\text{rate}}) p(A_{\text{shape}}) p(B_{\text{rate}}) p(B_{\text{shape}}) && (6.11) \\ &\propto \prod_k p(a_k|A_{\text{rate}}, A_{\text{shape}}) \times p(A_{\text{rate}}) && (6.12) \\ &= \left(\frac{A_{\text{rate}}^{A_{\text{shape}}}}{\Gamma(A_{\text{shape}})}\right)^K \prod_k a_k^{(A_{\text{shape}}-1)} e^{-a_k A_{\text{rate}}} \times p(A_{\text{rate}}) && (6.13) \\ &= A_{\text{rate}}^{(KA_{\text{shape}})} e^{-A_{\text{rate}} \sum_k a_k} \times p(A_{\text{rate}}) && (6.14) \end{aligned}$$

We can see that the first term, i.e. $A_{\text{rate}}^{(KA_{\text{shape}})} e^{-A_{\text{rate}} \sum_k a_k}$ is proportional to the density of a standard gamma distribution - i.e. where we let $A_{\text{rate}} = \beta$ and $a_k = x$ in the rate parameterization of the gamma distribution density:

$$\frac{1}{\Gamma(\alpha)} \beta^\alpha x^{\alpha-1} e^{-x\beta} \qquad (6.15)$$

and drop the terms $\frac{1}{\Gamma(\alpha)} \times x^{\alpha-1}$ which do not depend on β.

Similarly for A_{shape}:

$$\begin{aligned}
p(A_{\text{shape}}|\ldots) \propto & \left[\prod_k p(\underline{t}_k|a_k, b_k)p(a_k|A_{\text{rate}}, A_{\text{shape}})p(b_k|B_{\text{rate}}, B_{\text{shape}})\right] \\
& \times p(A_{\text{rate}})p(A_{\text{shape}})p(B_{\text{rate}})p(B_{\text{shape}}) && (6.16) \\
& \propto \prod_k p(a_k|A_{\text{rate}}, A_{\text{shape}}) \times p(A_{\text{shape}}) && (6.17) \\
& = \left(\frac{A_{\text{rate}}^{A_{\text{shape}}}}{\Gamma\left(A_{\text{shape}}\right)}\right)^K \prod_k a_k^{(A_{\text{shape}}-1)} e^{-a_k A_{\text{rate}}} \times p(A_{\text{shape}}) && (6.18) \\
& = \left(\frac{A_{\text{rate}}^{A_{\text{shape}}}}{\Gamma\left(A_{\text{shape}}\right)}\right)^K \prod_k a_k^{(A_{\text{shape}}-1)} \times p(A_{\text{shape}}) && (6.19)
\end{aligned}$$

Note that Equation (6.13) and Equation (6.18) are the same aside from the prior, i.e. $\times p(A_{\text{rate}})$ versus $\times p(A_{\text{shape}})$. It's really only in the following step that things cancel out to get Equation (6.19) - and that first term $\left(\frac{A_{\text{rate}}^{A_{\text{shape}}}}{\Gamma\left(A_{\text{shape}}\right)}\right)^K \prod_k a_k^{(A_{\text{shape}}-1)}$ does not look anything like a gamma distribution - thus we have to do a Metropolis-Hastings step.

So for Metropolis-Hastings, we use $\left(\frac{A_{\text{rate}}^{A_{\text{shape}}}}{\Gamma\left(A_{\text{shape}}\right)}\right)^K \prod_k a_k^{(A_{\text{shape}}-1)}$ as the likelihood in the Metropolis step.

However on implementation it became obvious that as the values of a_k are of the order of 50, summing these to the power of A_{shape} quickly causes numerical overflow, so moving onto the log scale we have the following. For clarity here, we will use the substitution S = Shape and R = Rate, and drop all mention of A, in particular because the same function will be used similarly for both A and B.

$$\begin{aligned}
\text{log-likelihood} = & \log\left(\left(\frac{R^S}{\Gamma(S)}\right)^K \prod_k a_k^{(S-1)}\right) && (6.20) \\
= & K \log\left(\frac{R^S}{\Gamma(S)}\right) + \log \prod_k a_k^{(S-1)} && (6.21) \\
= & KS \log R - K \log \Gamma(S) + \sum_k \log(a_k^{(S-1)}) && (6.22) \\
= & KS \log R - K \log \Gamma(S) + \sum_k (S-1) \log a_k && (6.23) \\
= & K \left[S \log R - \log \Gamma(S)\right] + (S-1) \sum_k \log a_k && (6.24)
\end{aligned}$$

6.4 Current implementation - a description

For the Goel-Okumoto hierarchical model, the following is an outline description of how it is implemented.

Parameter	Distribution		
α	Gaussian	mean=0	sd=1
d	Exponential	rate=1	
A_{shape}	log-normal	meanlog=log(1.1)	sdlog=1
A_{rate}	gamma	shape=1	rate=100
B_{shape}	log-normal	meanlog=log(1.1)	sdlog=1
B_{rate}	gamma	shape=1	rate=0.01

Fig. 6.10: Hierarchical Priors for the combined hierarchical model

As compared with the non-hierarchical model, we changed the model to iterate across all K releases of the version model every single MCMC iteration, instead of previously only visiting each release every K^{th} iteration.

6.4.1 Hierarchical priors

The priors and proposals and steps for the version model which is embedded in the combined hierarchical model are identical to what we used before. The priors for the Goel-Okumoto model are shown in Figure 6.10.

A `dgamma(x, shape=Ashape[[i]], rate=Arate[[i]], log=TRUE)`

B `dgamma(x, shape=Bshape[[i]], rate=Brate[[i]], log=TRUE)`

6.4.2 Proposals

The proposals for both (a, b) are effectively `exp(rnorm(1, log(x), 0.5))`.

6.4.3 Step Functions

The 'Step' functions, so called because they allow us to make an MCMC step and are as follows:

A shape A Metropolis step:

`runif(1, oldshape*0.5, oldshape/0.5)` and log-likelihood function:

```
abshapeloglikelihood <- function(shape, rate, abk) {
  K <- length(abk)
  K*(shape*log(rate) - lgamma(shape) + (shape - 1)*sum(log(abk)))
}
```

A rate A Gibbs step: Gamma($KA_{\text{shape}}, \sum a$)

B shape exactly as per A shape, with b instead of a.

B rate A Gibbs step: Gamma($KB_{\text{shape}}, \sum b$)

6.4.4 Results

Detailed results plots can be seen in Appendix C.0.1.

6.5 Discussion

Looking at the parameters seen in the box plots Figures C.2 to C.5, we notice some patterns. Firstly there is more certainty with the parameters (a, b) for the earlier versions, however, this is reversed for the parameters (α, d) for the version model where there is more certainty for the more recent versions.

The more recent Goel-Okumoto a parameter indicates both more uncertainty in its value and that it is higher, i.e. there are more bugs - which could be due to more professional testers being employed by Mozilla, or that the code has more bugs. Over time, the Goel-Okumoto b parameter is relatively steady, but might be decreasing which implies that bugs are being found more slowly but there is large uncertainty here, so this cannot be said to be clear.

There is a dramatic decrease in the α in the early Rapid-Releases where initially the Gaussian kernel is centred *after* the release date, i.e. suggesting that many bugs were found post release. Then until `alpha.12` - i.e. Release-15.0, there is a steady movement to centre the Gaussian before the release date - which makes sense - it is better to find the bugs before the release! After Release-15, and depending on which MCMC run we use, α seems to stabilise about 0.1 years, ie. 42 days or one release cycle *before* the release date. This matches our intuition that in the period between 2×42 days and 1×42 days before release, the software is in 'Alpha' or as Mozilla calls it 'aurora'[1], then 1×42 days before release, it goes into 'Beta', i.e. getting ready for release, so they are trying to fix the outstanding bugs and checking there are no more bugs - or as few as possible and that bugs claimed to be fixed, really are fixed.

The version model d parameter is reducing over time with some exceptions, i.e. the width of the Gaussian bell is reducing over time, implying that there is more concentration of effort into a particular release. As Mozilla becomes a more professional

[1]`https://wiki.mozilla.org/RapidRelease/Calendar`

organization, this is to be expected. As we saw in Section 2.5, more and more effort is being put into finding bugs by a small number of people.

Overall, we are quite happy with the performance of the combined model, particularly in the hierarchical form, as it seems quite robust to wild excursions in the (α, d) parameters without visibly affecting the Goel-Okumoto parameters (a, b).

When we take the last set (for instance) of parameters from a chain and plot the corresponding original and imputed data points and then take the sampled hierarchical parameters for $(A_{\text{shape}}, A_{\text{rate}}, B_{\text{shape}}, B_{\text{rate}})$, and then use these parameters to sample the Goel-Okumoto parameters (a, b) and plot the mean value function (MVF)

$$\text{MVF}(t|a, b) = a(1 - e^{-bt}) \tag{6.25}$$

We get plots like in Figure C.14. As can be seen, there is quite a range in the sampled MVF traces, however they are generally in the same ball park as the data for all versions, despite the fact that the MVF traces are not version specific. We find this a powerful supporting argument for our model.

Chapter 7

Case Study

This chapter discusses a case study which uses the combined model.

7.1 Utility

Extending the work in McDaid (1998); McDaid and Wilson (2001) and in earlier chapters, we will look at the idea of making a decision, one-step-ahead, as to when to release given the information we have. We will avoid any more complicated decision theoretic approaches.

In Chapter 6 we fitted a hierarchical model for the K known versions of Firefox. We can thus ask a number of questions then about version $K+1$, such as what is the optimal release time?

In order to do this define a utility for releasing every T days and then find the value of T that maximises the expected utility.

We define specific cost factors C_1^*, C_3^*, C_3^* in 'real' units, such that these are respectively, the cost of finding a bug before release, the cost of finding a bug after release and the cost per day of continuing to test. Note that $N(T)$ is the number of bugs found by time T, and $\bar{N}(T)$ is the number of bugs remaining in the system at time T. Thus we can write a utility function as:

$$U^* \left[T, N(T), \bar{N}(T)\right] = -\, C_1^* N(T) - C_2^* \bar{N}(T) - C_3^* T \qquad (7.1)$$

Alternatively, we can write a function by dividing across by C_1^*, to get a function relative to C_1^*, i.e. C_2 is the relative cost of finding a bug after release compared with the cost of finding a bug before release, C_1. We know from studies that $\frac{C_2^*}{C_1^*}$ is of the order of 10 - in the case where the bug is found just after release, compared with finding the bug in the final phase of testing.

$$U\left[T, N(T), \bar{N}(T)\right] = -\,N(T) - C_2\bar{N}(T) - C_3T \tag{7.2}$$

However, we are really interested in the Expected Utility:

$$\begin{aligned}\text{Expected Utility} &= -\,\mathbb{E}\left[N(T)|\text{data}\right] - C_2\mathbb{E}\left[\bar{N}(T)|\text{data}\right] - C_3T &(7.3)\\ &= -\,\mathbb{E}\left[a_{K+1}(1-e^{-b_{K+1}T})|\text{data}\right] - C_2\mathbb{E}\left[a_{K+1}e^{-b_{K+1}T}|\text{data}\right] - C_3T &(7.4)\end{aligned}$$

Which we can approximate as follows, assuming the we have M, MCMC iterations and in our notation, $x^{(m)}$ refers to the value of x at the m^{th} iteration:

$$\approx -\frac{1}{M}\sum_{m=1}^{M} a_{K+1}^{(m)}(1-e^{-b_{K+1}^{(m)}T}) - C_2\frac{1}{M}\sum_{m=1}^{M} a_{K+1}^{(m)}e^{-b_{K+1}^{(m)}T} - C_3T \tag{7.5}$$

Where the term C_3T corresponds to the cost per day of prolonging testing.

We simulate $a_{K+1}^{(m)}$ from $p(a|\text{hyper-parameters}^{(m)})$. We also simulate $b_{K+1}^{(m)}$ from $p(b|\text{hyper-parameters}^{(m)})$. Where hyper-parameters$^{(m)}$ is the m^{th} set of hyper-parameters in MCMC samples.

We evaluate the above for different T to maximise utility, e.g. by simply iterating from a small to a large T and plot to find the optimal release time T^*. In our case we might look at selecting T from 1 to say 365 to look at the expected value over a year.

Note that as C_2 increases, so does T^*, i.e. as the relative cost of a bug being found post-release increases compared with finding it before the release, the testing period will be increased to reduce the number of bugs found post-release.

Alternatively, we could fix $T^* = 42$ days and examine the relationship between C_2 and C_3 - which are of course relative to C_1, as mentioned above.

7.2 Results

We have an MCMC simulation with four chains which ran for 15,000 iterations. At every point in the chain we simulated the hyper-parameters $A_{\text{shape}}, A_{\text{rate}}, B_{\text{shape}}, B_{\text{rate}}$. From these hyper-parameters, we sample from the Gamma distribution with these shape and rate parameters to get pairs (a, b) from which we can calculate the utility as described above and shown in Figure 7.1. Note that the values of T^* where the

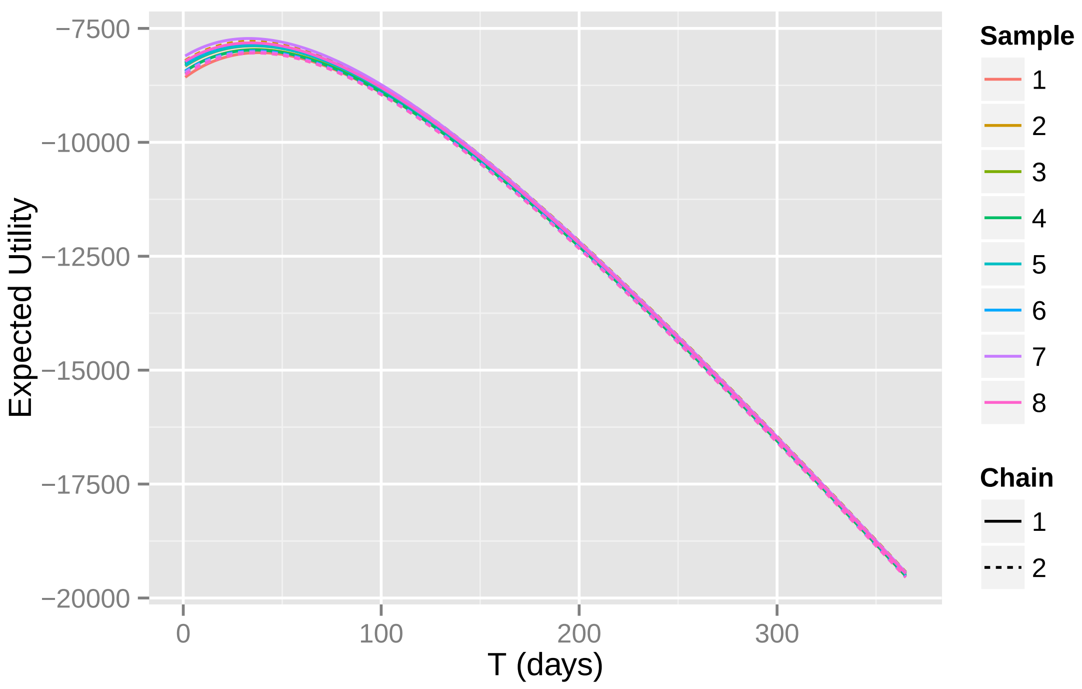


Fig. 7.1: A plot of the utility for four samples from two chains.

utility is maximised is similar for the four chains. In this case we have defined the cost parameter vector, $C_{1:4} = (1, 10, 18750, 0)$.

We also looked at the standard deviation of T^* from the generated samples and it is of the order of 4.5 days given our chains and samples.

If we fix $C_2 = 10$ and then examine the relationship between T^* and C_3, the cost per day of extra testing relative to the cost of finding a bug before release, we see the following:

We note in particular that as C_3 increases, T^* decreases until about 24,600 when T^* goes below 1 day, which realistically is not reasonably practical - i.e. on an ongoing basis it is very difficult to regularly release software at a rate faster than once per day. In emergency situations, for instance when a severe security flaw is detected in a release, it is reasonable that software gets re-released only hours after the previous release, but on an ongoing basis, it is not practical.

Knowing that Firefox is regularly released every 42 days, we then fix $C_3 = 17550$ and look at the relationship between T^* and C_2, as it is not that far from the T^* we found above which is of the order of 35 days. In more traditional, non-open source software development, C_2 has been found to be an order of magnitude greater than C_1 which is why we started by using $C_2 = 10$, but how sensitive is this? In Figure 7.3

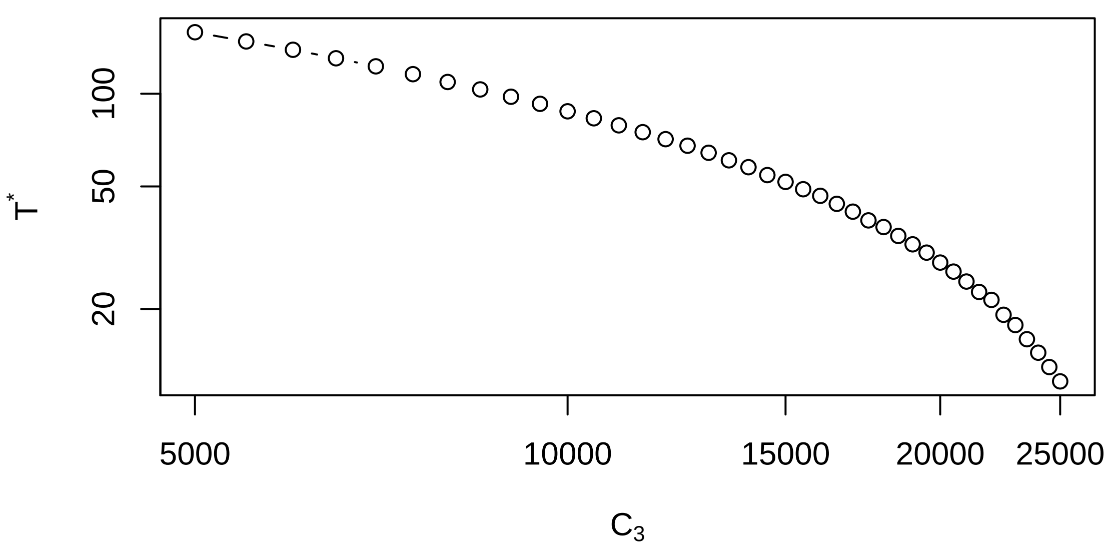


Fig. 7.2: A plot of the relationship between C_3 and T^*.

we look at the relationship between C_2 and T^* and we see that T^* increases as C_2 increases which we would expect, i.e. as the relative cost of a bug being found after release goes up, it makes sense to spend more time looking for bugs before release since that is cheaper. Conversely if the relative cost drops down toward 7, then T^* drops dramatically down towards one day.

It would also be interesting to fix $T^* = 42$, the fixed release schedule of Firefox, in order to look at the relationship between C_2 and C_3. We have carried such an analysis with 40 replicates at each value of C_2 and we have added a smoother, refer Figure 7.4. Noting in advance that there is obvious heteroscedasticity, the approximate straight line through these points on the plot is $C_3 = -1106 + 1721C_2$. We further note that for $C_2 < 5$ the value of $C_3 \approx 7000$, we have not ascertained why there is a change point as yet.

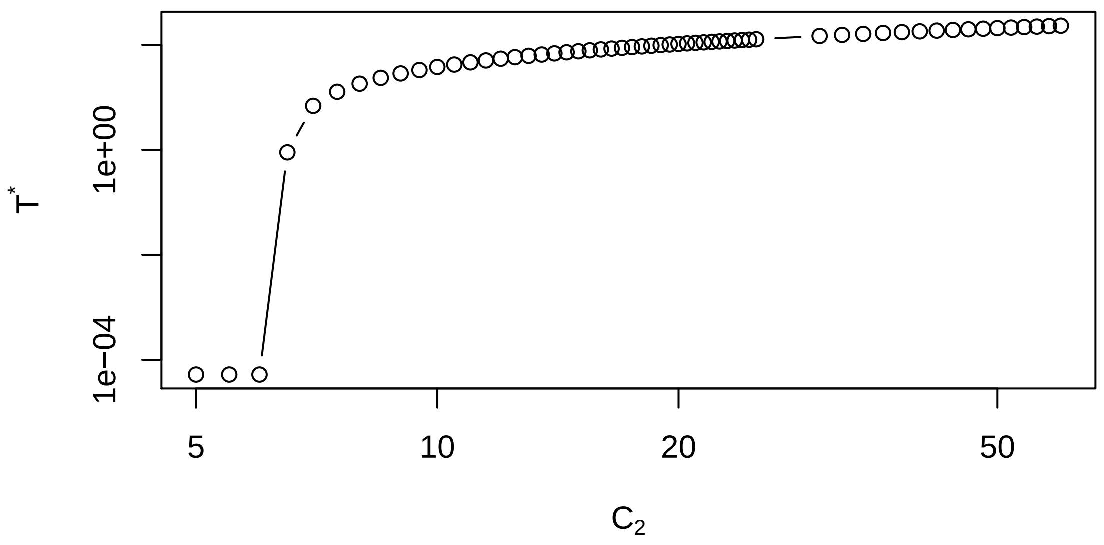


Fig. 7.3: A plot of the relationship between C_2 and T^*.

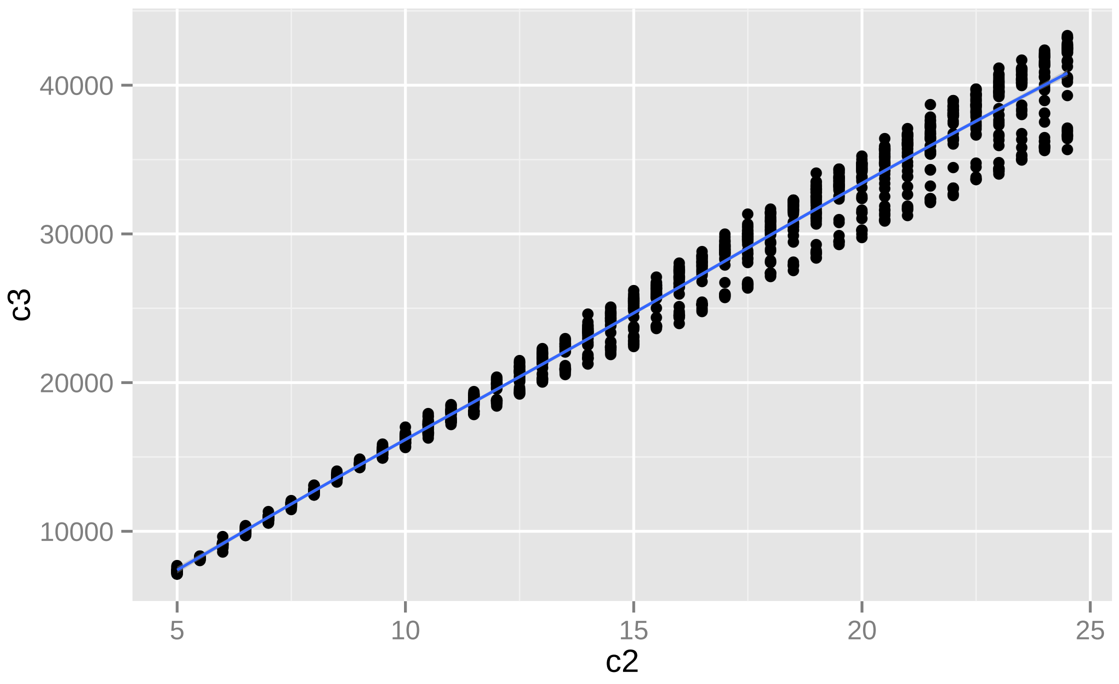


Fig. 7.4: A plot of the relationship between C_2 and C_3 for a fixed $T^* = 42$.

7.3 Discussion

The standard figure of a bug being an order of magnitude more expensive if found after release, is consistent with what we have found. The C_3 coefficient is of the order of 16,100 in time units of a year, or of the order of 44 in time units of a day. By this we mean that delaying the release by one day is equivalent to finding 44 bugs!

7.4 Further Work

Further work that could be done on the case study.

The following is a list of possible scenarios that might be developed.

- Look at different priorities of bugs, assigning different utility values to each priority.
- Look at different severities of bugs, assigning different utility values to each severity.
- Look at bugs from different subsystems - possibly at multiple priorities and severities.
- We could also add a further term, $C_4\frac{1}{T}$ which we might describe as a marketing parameter to keep the time distance between releases low. We feel that it would be interesting to further explore the use of C_4.

Chapter 8

Discussion

8.1 Discussion

This work has taken an old model (Goel and Okumoto, 1979) and shown that it can be extended to model the rate at which bugs are found in modern open source software using a hierarchical model.

Historically, references such as Jelinski and Moranda (1972); McDaid and Wilson (2001) look at the reliability of software and use language like that relating to hardware reliability. These older papers make assumptions about the instantaneous repair of bugs, possibly in a faulty manner. Given that there are 157 open bugs in Firefox at the time of writing, some of which have been open for more than four years, this assumption might need to be reviewed. What we have examined is the rate at which bugs are recorded in a database which says nothing about the repair rate of said bugs or the reliability. Before the official release of Firefox, there are so called *nightly* releases which are automatic builds from the source code control system and then released to the testing community. After the official release, the code used does not change except in the rare circumstances of a security patch release - in other words there are no bug fixes post release. Furthermore, many bugs are repaired by developers which were never recorded in a bug database. By this we are saying that we are not really looking at the reliability of Firefox, merely the rate at which bugs are recorded, by a tiny fraction (thousands) of the total user base which is of the order of 500 million. The assumptions and the language used by the software reliability community might need to change.

An interesting line of further inquiry would be to examine the Mercurial version control logs[1] for information on the fixing bugs during the same period. In a discussion about the source code of R with Prof. Brian Ripley, he suggested that this would be a more reliable way of looking at bugs in R. An ambitious project would be to simultaneously examine the version control system for bug fixes, look at the bug database for the opening and possible closing of cases, and to create a full reliability model for such complicated software as Firefox with more than 10 million lines of code and which is in a state of constant flux.

8.1.1 Data

We define the *age* of a bug as the release date minus the creation date - which clearly only works for bugs with a known version label. Then looking at the raw data for age in more detail, Figures 8.1 to 8.3, we can see that the intensity of bugs logged can be divided into two distinct regimes, firstly before the release, and secondly from the date of release. Before the release there is a peak about two releases ($2 \times 42 = 84$ days), before the release date - this corresponds with when Firefox is marked as *aurora*, or being in alpha state. At 42 days before release, the code goes from aurora into beta[2] and the next batch of code is promoted to aurora. A lot of internal testing and fixing by staff (paid or otherwise) in the Mozilla organization occurs when the code goes into aurora. After the first peak, the bug creation rate then falls away until the release date when there is a sharp peak followed by an exponential type decay in the bug creation rate. Since there are no code changes after the release (except urgent security patches), the rapid exponential type decay rate supports the Goel-Okumoto hypothesis.

In further work, it would be most interesting to explore the insights in this histogram further, possibly looking at a two peak mixture model for the bug creation rate and exploring the implications of this.

[1]Mercurial is the version control system used by the Mozilla project to manage source code, `https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Source_Code/Mercurial`. Version control or Revision control systems are used to manage changes in files in a system and include functions for reverting to older versions and merging (often automatically) changes from more than one person.

[2]`https://wiki.mozilla.org/RapidRelease`

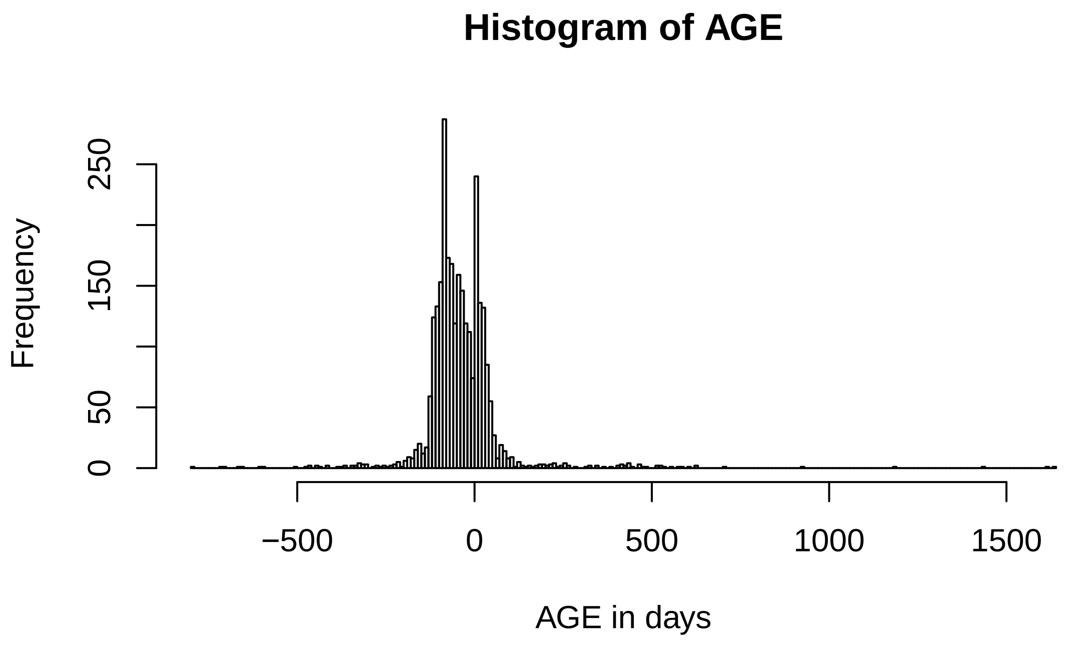


Fig. 8.1: A histogram of the age (defined in the text) of bugs

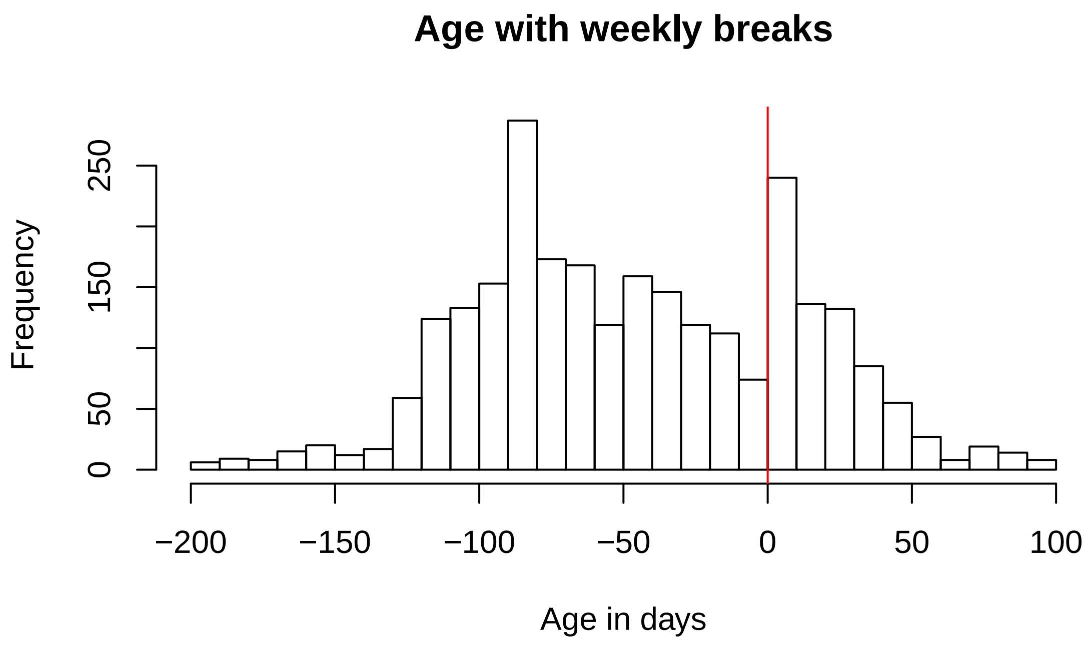


Fig. 8.2: A histogram of the age (defined in the text) of bugs, looking at -200 to +100 days, with daily breaks

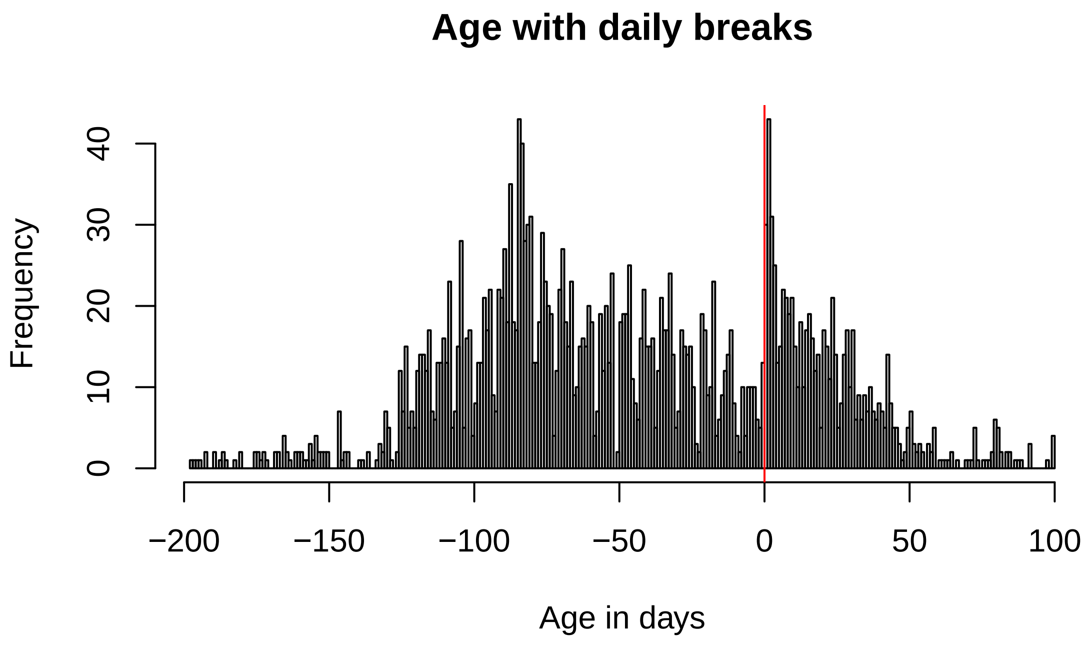


Fig. 8.3: A histogram of the age (defined in the text) of bugs, looking at -200 to +100 days, with weekly breaks

8.1.2 Covariates

In Section 2.4.2 we described three covariates for each version, namely the number of files changed between versions (FChanged), the number of lines added in this version (LInserts) and the number of lines deleted in this version (LDeletions). It is worth recalling that where there is a minor change to a line, e.g. a single character is changed, then that will be recorded as a file change, as one line deleted and one line added. When we plot these three covariates against version number as seen in Figure 8.4, a clear pattern is visible and indeed as seen in the correlation matrix in Figure 8.5, the three covariates are, in our opinion, highly correlated.

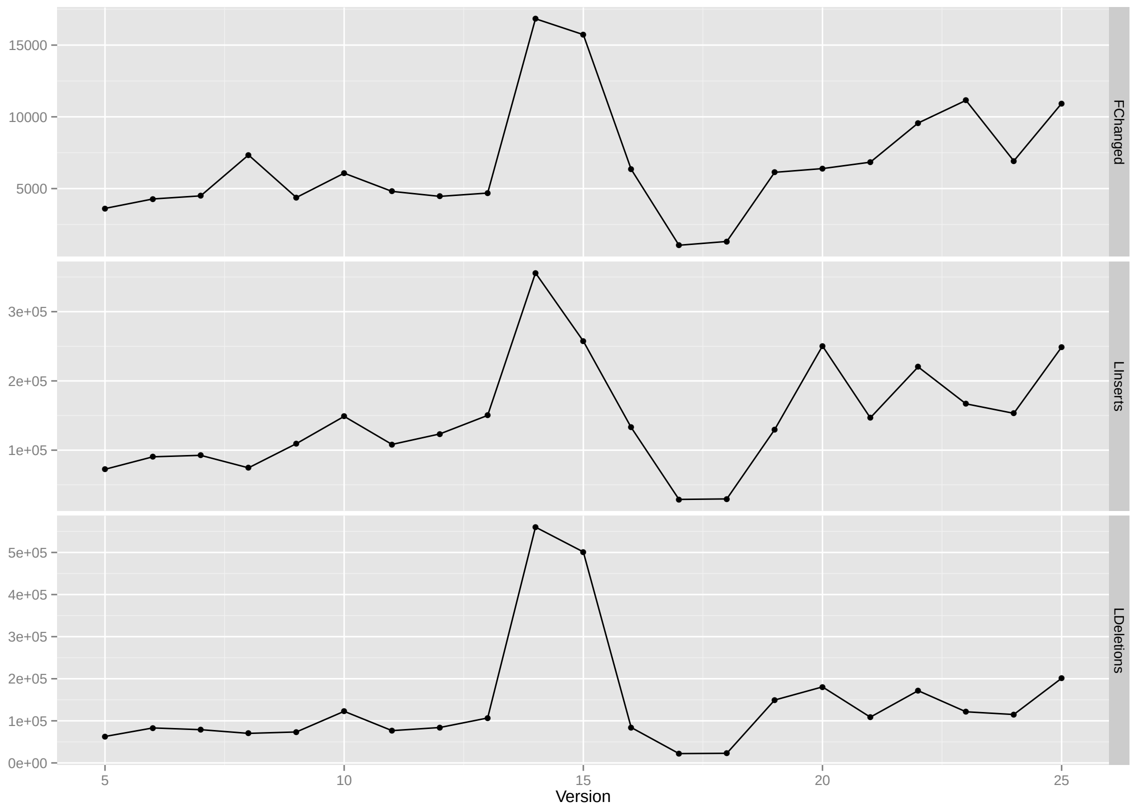


Fig. 8.4: The three covariates plotted as a function of version covariate information: number of files changed, number of lines of code inserted and number of lines deleted.

In Figure 8.6, the posterior distributions of the a_k and b_k are plotted against covariate information for the release. Notably, we see little relationship between the parameters and covariates.

We note the wide variation in the parameters as seen in Figure 8.6, i.e. the large vertical spread seen in the figure for the parameter values for some versions, noting also the $\log_{10}$ scale on the Y-axis. We do a linear regression of the mean values Goel-

```
> cor(ffcov)
           FChanged LInserts LDeletions
FChanged      1.000    0.876      0.897
LInserts      0.876    1.000      0.861
LDeletions    0.897    0.861      1.000
```

Fig. 8.5: The correlation matrix for the version covariate information: number of files changed (FChanged), number of lines of code inserted (LInserts) and number of lines deleted (LDeletions).

Okumoto parameters (a, b) from the combined model on the three covariates as seen in Figure 8.7. We note that there is a significant relationship between the parameter a and the covariates. However, we note that no significant relationship appears to exist between the parameter b and the covariates.

We conclude that, there is some evidence that the number of bugs and their rate of discovery are a function of these covariates but the relationship is not straightforwardly linear. This may be due to the nature of open source testing, where the bug creation and discovery processes are rather isolated from each other. It would be interesting to extend this initial work and look at it in more depth.

It would also be interesting to look at the bug covariates, e.g. the information on bug severity, as opposed to the version covariates we spoke of above.

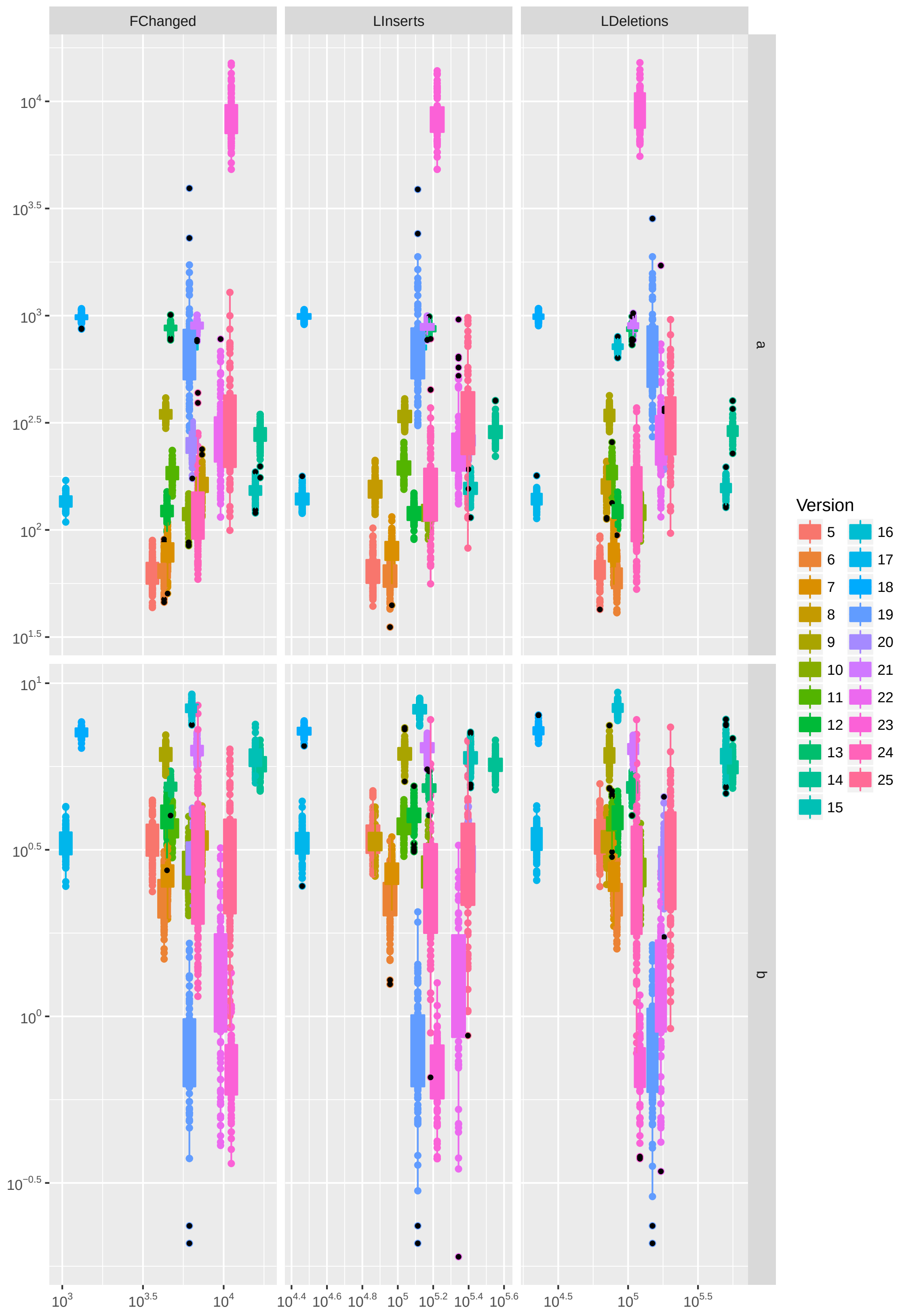


Fig. 8.6: The posterior distribution of the $\log_{10}(a_k)$ (top) and $\log_{10}(b_k)$ (bottom) plotted as a function of version covariate information: number of files changed, number of lines of code inserted and number of lines deleted.

```
> summary(lm(a~FChanged+LInserts+LDeletions, data=MPw))

Call:
lm(formula = a ~ FChanged + LInserts + LDeletions, data = MPw)

Residuals:
    Min      1Q  Median      3Q     Max
-2581.7  -461.0  -215.7   583.2  4855.4

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) -8.321e+02  8.945e+02  -0.930  0.36523
FChanged     7.040e-01  2.247e-01   3.133  0.00606 **
LInserts    -5.865e-03  9.797e-03  -0.599  0.55729
LDeletions  -1.647e-02  6.314e-03  -2.609  0.01833 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1601 on 17 degrees of freedom
Multiple R-squared:  0.3886,    Adjusted R-squared:  0.2807
F-statistic: 3.601 on 3 and 17 DF,  p-value: 0.03521

> summary(lm(b~FChanged+LInserts+LDeletions, data=MPw))

Call:
lm(formula = b ~ FChanged + LInserts + LDeletions, data = MPw)

Residuals:
    Min      1Q  Median      3Q     Max
-3.4821 -1.4286 -0.0966  0.5906  5.0410

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  5.002e+00  1.168e+00   4.282 0.000504 ***
FChanged    -2.804e-04  2.935e-04  -0.955 0.352728
LInserts    -7.594e-06  1.280e-05  -0.593 0.560714
LDeletions   1.349e-05  8.247e-06   1.636 0.120150
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.091 on 17 degrees of freedom
Multiple R-squared:  0.1361,    Adjusted R-squared:  -0.0163
F-statistic: 0.8931 on 3 and 17 DF,  p-value: 0.4648
```

Fig. 8.7: The output of linear regression for (a_k) (top) and (b_k) (bottom) regressed as a function of version covariate information: number of files changed (FChanged), number of lines of code inserted (LInserts) and number of lines deleted (LDeletions).

8.1.3 Version Model

There are weaknesses in the validation of the version model including:

- It is open for debate whether a *Kappa* of the order of 0.1 is sufficient to say that the model is good enough for use in the combined model.
- The model was validated by running 20 chains and assessing these chains against the true values and examining box plots of these 20 values at different values of Q and of chain length. This could have been done using leave out K, instead of the low volume ($N = 20$) bootstrapping, however, in our case it was not plain bootstrapping as the sample was without replacement.

It would be interesting to further explore the P_{ver} matrix, e.g. when we look at a longitudinal plot of the number of bugs assigned to each version by MCMC iteration as in Figure 8.8.

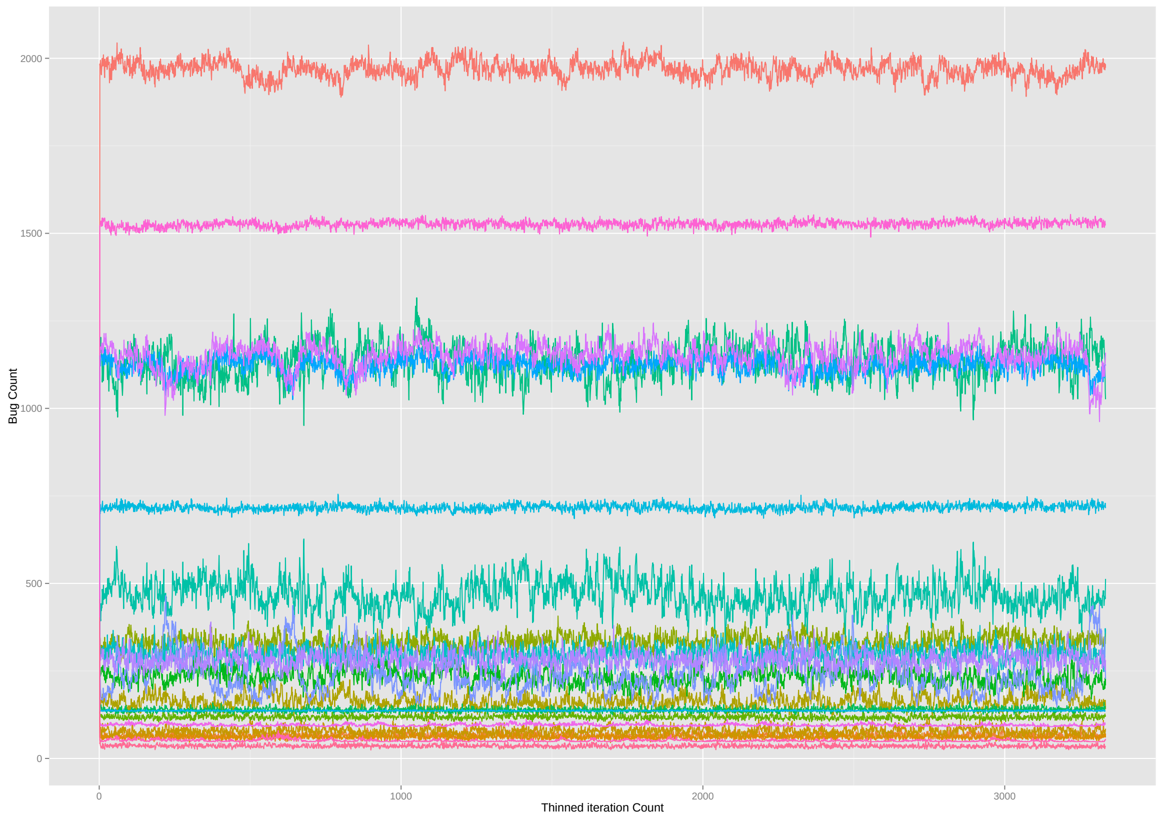


Fig. 8.8: A thinned sample plot of the counts in each of the versions PreRapid, $5, \ldots, 25$ against iteration count.

8.1.4 Model Speed

The models in Chapters 4 to 6 have all been written in R with the exception of the RStan model and the log-likelihood function for the model in Chapter 4 - in general the models can take a comparatively long time, e.g. one run four chains of 100,000 iterations of the Combined model took nearly 23 hours with most of that time being taken up by the evaluation of the two models. Aside from statistical considerations, such as using adaptive techniques as reviewed in Rosenthal (2011), these models could be rewritten to run considerably faster.

- The chains could be run in parallel independent processes if there was enough memory.

- Information from very long chains could be saved only intermittently, say every 10 iterations generating big memory savings and allowing longer chains to be used.

- As the log-likelihood functions are not vectorizable in R, considerable speed advantage can be gained just by re-writing these functions in C++, while leaving the rest of the Metropolis-Hastings algorithm in R. This has been done for the Goel-Okumoto model and initiated for the version model.

- As the dataset has thousands of points and the log-likelihood function is comparatively complex, considerable advantage can be had by using the graphics processor (GPU) which is ideally suited to calculating the same function many times over with different data points. Initial work has been done on calculating the log-likelihood function on the GPU and the function runs of the order of 30 times faster on the GPU than when written in R. Clearly if the entire version model was ported to the GPU there would be a very considerable gain in speed.

8.2 Further Work

Some further work ideas include:

- It would be useful to profile the MCMC code to see where the bulk of the time is being spent and then work on speeding up this core element as this is the weak link in the chain, as under the theory of constraints (Goldratt and Cox, 1984), trying to increase the speed of some other part of the code will be less useful.

- Speed up calculations in the version model by writing the core elements in C++ inside Rcpp, or going even further and moving the entire version model code onto a GPU, this work has been initiated.

- For the combined model, examine why $N_{\text{effective}}$ is an order of magnitude smaller for the version model than for the Goel-Okumoto model.

- Clearly, the model could be improved if different proposal standard deviations could be used across each of the versions as we have done in the hierarchical model, and we feel sure that adaptive MCMC would tune the parameters to much better acceptance rates and better convergence.

- Extend the GO model to (a, b, b') where the later two parameters refer to the before and after the release date for a particular version.

- Looking at covariates would be a useful extension of our work, as Ray et al. (2006), amongst others, have done.

- Autoregressive models have been examined by Singpurwalla and Soyer (1985), and it would be interesting to look at how they might be extended to software with multiple versions to describe the evolution of the parameters from one release to the next.

- If we look at a plot of the data for a particular release, i.e. Time on the X-axis and the cumulative number of bugs assigned to this version on the Y-axis, it appears that there might be changes in slope every 42 days (i.e. every release cycle) - this is worth looking at in more detail using cut-point analysis on the imputed data.

- When people find a bug, if they are curious, then this will lead them to find other similar bugs which could be modelled by a self-exciting process. It would be interesting to further explore whether this exists in the dataset. Singpurwalla and Wilson (1999) briefly mentions using self-exciting processes for software reliability models.

- It has been suggested that we should look at other fields in the raw Bugzilla database to identify if the version attached to a given bug should really be earlier than what is actually marked. We have done some initial work on this and of the order of 20% of bugs might be moved to an earlier version. This work might overcome some of the issues that we have with the small number of bugs which are recorded more than three release cycles before the release date, which we discard in the hierarchical model.

- This work was carried out based on the Firefox-2013 dataset where the last release is version 22 and the data goes as far as a few bugs in version 25. The cut-off date for the dataset is the 18th of July 2013. Two years later, we are using Firefox release 38 and it would be interesting to develop a new and more recent dataset and to look at the effectiveness of our work on a newer dataset.

- It would also be possible to go into more detail in this work by looking at the other data associated with each bug, e.g. the severity, and doing inference on say the number of bugs rated SEVERE for an upcoming release. To help decision making, a utility cost could be applied to each level of severity, e.g. 1 to TRIVIAL, to 1000 to SEVERE and ∞ to BLOCKER to calculate the expected cost of an upcoming release.

- For our purposes, we did not need to create a hierarchical version model in the combined model, but we think that it would be an interesting exercise.

  The following is an initial description of our implemented hierarchical version model.

$$p(\alpha_k) = \frac{1}{N2\pi \times 9^2} e^{\frac{1}{18}\alpha_k^2} \tag{8.1}$$

  - In hierarchical model:

$$p(\alpha_k|\mu_\alpha, \sigma_\alpha^2) = \frac{1}{N2\pi\sigma_\alpha^2} e^{-\frac{1}{2\sigma_\alpha^2}(\alpha_k - \mu_k)^2} \tag{8.2}$$

    and $(\mu_\alpha, \sigma_\alpha^2)$ are unknown and must be sampled.

  - So in Metropolis move for α_k, replace $p(\alpha_k)$ with $p(\alpha_k|\mu_\alpha, \sigma_\alpha^2)$.

  - We also have to sample $(\alpha_k|\mu_\alpha, \sigma_\alpha^2)$. These can be done by Gibbs sampling moves assuming $N(m, s^2)$ prior for μ_α and Inverse-Gamma(f, g) for σ_α^2.

- $*$ $p(\mu_\alpha|\ldots)$ is normal with mean

$$\frac{\frac{m}{s^2}+\frac{\Sigma\alpha_k}{\sigma_\alpha^2}}{\frac{1}{s^2}+\frac{1}{\sigma_\alpha^2}} \tag{8.3}$$

and variance:

$$\frac{1}{\frac{1}{s^2}+\frac{1}{\sigma_\alpha^2}} \tag{8.4}$$

- $*$ $p(\sigma_\alpha^2|\ldots)$ is Inverse-Gamma with scale= $f+\frac{1}{2}\Sigma_k(\alpha_k-\mu_k)^2$ and shape=$g+\frac{1}{2}K$.

- It would be interesting to look at change points in the parameters of the NHPP used in a release for the bugs recorded.

- Considering alternative intensity functions would also be interesting - there are many possible suggestions out there in the literature, some of which are mentioned in Section 3.14.1.

- In a fully subjective Bayesian approach one would elicit and employ the opinions of experts as much as possibble. In our case that information would be used to inform the initial values, the priors and hyper-priors in the models. That said, employing the opinions of experts in a coherent manner is an entire domain of research in itself.

- The calculation of the log-likelihood at each of the 10,000 data points can be done in parallel and is a prime candidate for being delegated to a GPU. Initial work has started on this and the results are very promising.

- Overall, we have shown that given a high proportion of imputed data (80%), our model appears to work in a robust manner.

- While the acceptance ratio in the Goel-Okumoto part of the model is quite acceptable, typically being of the order of 0.2, the acceptance ratio in the version model is not so clear cut having a strange 'exponential decay' type shape which needs more investigation.

Chapter 9

Conclusions

A brief summary of the important results.

- Firefox-2013, a multi-version bug dataset with covariates was published.
- We have demonstrated how the Goel-Okumoto model can be used in version models.
- A successful version model was created which imputes missing version labels.
- The above two models were combined together to predict the parameters for multi-version bug databases with missing data.
- The Firefox-2013 dataset was used as a case study to examine the implicit cost ratios for finding bugs before and after release.
- A working implementation of the above models.

Appendix A

RStan results table - Chapter 4

Figure A.1 shows the results of a sample run of RStan for the Goel-Okumoto model on the Firefox dataset in Chapter 4.

```
Inference for Stan model: ffh3.
4 chains, each with iter=10000; warmup=5000; thin=1;
post-warmup draws per chain=5000, total post-warmup draws=20000.

               mean se_mean        sd        2.5%         25%
avec[1]    159.68193 0.65766  63.79842    80.64330   117.30284
avec[2]     65.60814 0.21619  20.37350    40.00761    52.48957
avec[3]     60.79557 0.18384  17.36526    38.54920    49.53621
avec[4]     68.84632 0.06107   8.63714    53.12871    62.76686
avec[5]     99.64788 0.25953  24.47122    67.27905    83.49101
avec[6]    107.68871 0.08073  11.41676    86.92788    99.68443
avec[7]     87.48835 0.06864   9.70662    69.48722    80.80418
avec[8]    122.26919 0.07916  11.19525   101.48469   114.55922
avec[9]    138.32456 0.11071  14.49661   112.76338   128.27510
avec[10]   120.64820 0.07893  11.16279    99.90266   112.87870
avec[11]   125.86642 0.07800  11.03102   104.98854   118.27402
avec[12]   111.46957 0.15997  16.31860    85.23478   100.40570
avec[13]   161.72936 0.09544  13.49712   136.75076   152.33636
avec[14]   139.19065 0.12440  15.20481   112.61342   128.80322
avec[15]   420.90057 1.66474 138.39446   246.46961   327.56447
avec[16]   218.43788 0.52972  50.77972   149.81851   183.94140
avec[17]   359.66242 1.38123 118.26902   208.62969   280.63937
avec[18]   267.05681 1.01819  91.95994   151.33469   205.72791
avec[19]   275.35463 1.05100 100.43543   148.34959   207.30998
avec[20]   179.82893 0.73894  74.20362    85.30137   129.81680
avec[21]   192.69723 0.85208  89.83073    78.25725   131.60734
bvec[1]      0.00316 0.00001   0.00122     0.00127     0.00228
bvec[2]      0.00357 0.00001   0.00133     0.00132     0.00260
bvec[3]      0.00455 0.00001   0.00164     0.00176     0.00335
bvec[4]      0.00687 0.00001   0.00118     0.00465     0.00605
bvec[5]      0.00586 0.00002   0.00187     0.00264     0.00452
bvec[6]      0.00683 0.00001   0.00107     0.00478     0.00609
bvec[7]      0.00525 0.00001   0.00081     0.00371     0.00469
bvec[8]      0.00741 0.00001   0.00083     0.00582     0.00686
bvec[9]      0.00523 0.00001   0.00084     0.00362     0.00466
bvec[10]     0.00914 0.00001   0.00112     0.00700     0.00838
bvec[11]     0.01259 0.00001   0.00121     0.01030     0.01176
bvec[12]     0.00645 0.00001   0.00144     0.00375     0.00546
bvec[13]     0.00793 0.00001   0.00094     0.00613     0.00729
bvec[14]     0.00604 0.00001   0.00103     0.00407     0.00534
bvec[15]     0.00161 0.00001   0.00050     0.00075     0.00125
bvec[16]     0.00353 0.00001   0.00098     0.00178     0.00284
bvec[17]     0.00176 0.00001   0.00056     0.00080     0.00136
bvec[18]     0.00324 0.00001   0.00111     0.00141     0.00244
bvec[19]     0.00306 0.00001   0.00107     0.00131     0.00229
bvec[20]     0.00518 0.00002   0.00218     0.00203     0.00363
bvec[21]     0.00574 0.00002   0.00258     0.00219     0.00393
lreala       4.94471 0.00139   0.15075     4.66149     4.84430
lrealb      -5.35990 0.00152   0.15064    -5.67509    -5.45635
lrealasd     0.60057 0.00142   0.13436     0.38966     0.50505
lrealbsd     0.54849 0.00127   0.10562     0.37323     0.47311
meana      172.91786 0.39751  35.96692   124.80464   149.14025
mediana    142.05269 0.20553  21.93784   105.79364   127.01403
sda        120.47619 0.71590  64.41327    56.45722    82.55225
meanb        0.00556 0.00001   0.00084     0.00412     0.00498
medianb      0.00475 0.00001   0.00071     0.00343     0.00427
sdb          0.00335 0.00001   0.00106     0.00197     0.00263
lp__     -3455.82192 0.06718   5.19084 -3466.89670 -3459.16249
```

Fig. A.1: The details of the output of the Goel-Okumoto model in Stan

```
                50%          75%        97.5% n_eff    Rhat
avec[1]    146.18869    186.17966    319.11845  9411 1.00013
avec[2]     61.32779     73.42375    117.45153  8881 1.00046
avec[3]     57.35890     67.93651    103.76205  8923 1.00035
avec[4]     68.41388     74.46835     86.63972 20000 0.99993
avec[5]     95.17746    110.16974    158.90494  8891 1.00022
avec[6]    107.07977    114.92755    131.71347 20000 1.00004
avec[7]     86.97756     93.79626    107.52137 20000 0.99991
avec[8]    121.88586    129.59208    145.15844 20000 1.00003
avec[9]    137.44612    147.31625    169.80777 17145 1.00033
avec[10]   120.25200    127.94254    143.54955 20000 0.99992
avec[11]   125.61677    133.11121    148.44365 20000 0.99994
avec[12]   109.52230    120.23895    149.01234 10406 0.99988
avec[13]   161.17931    170.63574    189.16094 20000 0.99993
avec[14]   138.02128    148.49493    172.42518 14940 1.00044
avec[15]   391.43267    479.81282    763.51218  6911 1.00062
avec[16]   208.77062    241.29933    344.98943  9189 0.99990
avec[17]   334.70652    411.02502    657.03472  7332 1.00052
avec[18]   247.57732    304.67081    497.45641  8157 1.00037
avec[19]   254.12129    318.48798    528.98494  9132 1.00064
avec[20]   164.55777    211.31268    363.02779 10084 1.00025
avec[21]   173.96034    232.17983    418.92204 11115 1.00048
bvec[1]      0.00299      0.00385      0.00598 12028 1.00004
bvec[2]      0.00345      0.00441      0.00646 13763 1.00006
bvec[3]      0.00442      0.00561      0.00808 13419 1.00005
bvec[4]      0.00683      0.00764      0.00928 20000 1.00000
bvec[5]      0.00571      0.00707      0.00985 12498 1.00038
bvec[6]      0.00682      0.00754      0.00900 20000 1.00010
bvec[7]      0.00522      0.00579      0.00688 20000 1.00024
bvec[8]      0.00740      0.00796      0.00908 20000 0.99998
bvec[9]      0.00522      0.00579      0.00689 20000 1.00019
bvec[10]     0.00913      0.00988      0.01139 20000 0.99996
bvec[11]     0.01257      0.01338      0.01503 20000 0.99997
bvec[12]     0.00640      0.00739      0.00937 14582 0.99991
bvec[13]     0.00793      0.00856      0.00980 20000 0.99991
bvec[14]     0.00602      0.00673      0.00813 20000 1.00036
bvec[15]     0.00157      0.00193      0.00270  8393 1.00030
bvec[16]     0.00348      0.00416      0.00559 11806 0.99999
bvec[17]     0.00171      0.00211      0.00299  8982 1.00044
bvec[18]     0.00313      0.00391      0.00575 10654 1.00022
bvec[19]     0.00294      0.00370      0.00548 11116 1.00067
bvec[20]     0.00482      0.00633      0.01048 12196 1.00019
bvec[21]     0.00526      0.00700      0.01201 12694 1.00030
lreala       4.94035      5.03897      5.26017 11780 1.00027
lrealb      -5.35254     -5.25718     -5.08105  9765 1.00053
lrealasd     0.58300      0.67531      0.91497  8902 1.00025
lrealbsd     0.53729      0.61150      0.79020  6939 1.00030
meana      166.11722    188.37872    263.18890  8187 1.00040
mediana    139.81966    154.31098    192.51470 11393 1.00025
sda        105.49281    138.61223    277.17859  8096 1.00034
meanb        0.00548      0.00604      0.00741 20000 1.00024
medianb      0.00474      0.00521      0.00621  9998 1.00052
sdb          0.00313      0.00381      0.00599 10105 0.99996
lp__     -3455.48817 -3452.14235 -3446.67491  5971 1.00047

Samples were drawn using NUTS(diag_e) at Mon Dec  8 08:26:05 2014.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

Appendix B

RStan results table - Chapter 5

Figure B.1 shows the results of a sample run of RStan for the Version model on the Firefox dataset in Chapter 5 with four chains each of 100,000 iterations, and following Gelman's conservative procedure, the first half of each chain is dropped. This table was generated using `rstan::monitor()`.

```
Inference for the input samples (4 chains: each with iter=100000; warmup=50000):

         mean se_mean   sd 2.5%  25%  50%  75% 97.5% n_eff Rhat
alpha.1  -0.4     0.0  0.2 -0.8 -0.5 -0.4 -0.3  -0.2    61  1.1
alpha.2   0.1     0.0  0.0  0.0  0.1  0.1  0.1   0.1   568  1.0
alpha.3   0.1     0.0  0.0  0.0  0.1  0.1  0.1   0.1   454  1.0
alpha.4   0.1     0.0  0.0  0.0  0.0  0.1  0.1   0.1   478  1.0
alpha.5   0.0     0.0  0.0 -0.1  0.0  0.0  0.0   0.0   540  1.0
alpha.6  -0.1     0.0  0.0 -0.2 -0.1 -0.1 -0.1  -0.1   800  1.0
alpha.7  -0.1     0.0  0.0 -0.1 -0.1 -0.1 -0.1   0.0   656  1.0
alpha.8  -0.1     0.0  0.0 -0.1 -0.1 -0.1 -0.1   0.0   757  1.0
alpha.9  -0.2     0.0  0.0 -0.2 -0.2 -0.2 -0.1  -0.1   610  1.0
alpha.10 -0.1     0.0  0.0 -0.2 -0.1 -0.1 -0.1  -0.1   631  1.0
alpha.11 -0.1     0.0  0.0 -0.2 -0.1 -0.1 -0.1  -0.1   921  1.0
alpha.12 -0.2     0.0  0.0 -0.2 -0.2 -0.2 -0.2  -0.1  1111  1.0
alpha.13 -0.1     0.0  0.0 -0.2 -0.1 -0.1 -0.1  -0.1   974  1.0
alpha.14 -0.1     0.0  0.0 -0.2 -0.1 -0.1 -0.1  -0.1   873  1.0
alpha.15 -0.1     0.0  0.0 -0.2 -0.1 -0.1 -0.1  -0.1   699  1.0
alpha.16 -0.1     0.0  0.0 -0.2 -0.1 -0.1 -0.1  -0.1   611  1.0
alpha.17 -0.1     0.0  0.0 -0.1 -0.1 -0.1 -0.1  -0.1   131  1.0
alpha.18 -0.1     0.0  0.0 -0.1 -0.1 -0.1 -0.1   0.0    26  1.0
alpha.19  0.5     0.1  0.2  0.2  0.4  0.5  0.6   0.8     7  1.2
alpha.20 -0.1     0.0  0.1 -0.2 -0.1 -0.1 -0.1   0.0    10  1.3
alpha.21  0.2     0.0  0.1  0.0  0.1  0.1  0.2   0.5    10  1.2
alpha.22 -0.2     0.0  0.0 -0.2 -0.2 -0.2 -0.2  -0.1    22  1.2
d.1       1.2     0.0  0.3  0.7  1.0  1.2  1.4   1.8    76  1.1
d.2      17.4     0.4  2.8 12.2 15.3 17.3 19.4  23.0    40  1.1
d.3      12.6     0.1  1.7  9.6 11.5 12.5 13.6  16.1   122  1.0
d.4       7.7     0.1  0.9  6.1  7.1  7.6  8.3   9.6   186  1.0
d.5       6.8     0.1  0.8  5.3  6.3  6.8  7.3   8.6   147  1.0
d.6      19.9     0.3  3.1 14.1 17.7 19.6 21.7  26.7    88  1.0
d.7       9.1     0.1  1.2  6.7  8.2  9.1  9.9  11.7    97  1.0
d.8       7.1     0.1  0.9  5.6  6.5  7.1  7.7   8.9   220  1.0
d.9       8.0     0.1  1.1  6.1  7.2  8.0  8.8  10.2    87  1.0
d.10      7.4     0.1  0.9  5.9  6.9  7.4  8.0   9.2    72  1.0
d.11     12.8     0.1  1.5 10.2 11.7 12.6 13.7  16.3   122  1.0
d.12     11.3     0.1  1.3  8.8 10.4 11.3 12.2  13.8   103  1.0
d.13     18.7     0.2  2.4 14.3 17.1 18.8 20.4  23.4   108  1.0
d.14     11.4     0.1  1.2  9.3 10.6 11.4 12.1  14.1   143  1.0
d.15     15.9     0.2  1.7 13.0 14.7 15.8 17.0  19.5   118  1.0
d.16     15.2     0.2  1.8 11.9 13.9 15.3 16.5  18.8    92  1.0
d.17     19.1     0.2  2.1 15.3 17.5 19.0 20.6  23.0   120  1.0
d.18     21.0     1.0  3.4 14.4 18.8 21.1 23.3  27.9    12  1.1
d.19      2.2     0.3  0.8  1.1  1.6  2.1  2.7   4.0     8  1.2
d.20     35.0     3.9 14.2 13.7 24.0 34.1 42.9  70.5    13  1.3
d.21      5.5     0.7  2.1  2.1  3.9  5.5  6.9  10.0     9  1.2
d.22     53.7     5.3 22.1 21.7 33.8 52.7 68.0  98.9    17  1.3
```

Fig. B.1: The details of the output of the Version model.

Appendix C

Combined Model details - Chapter 6

In the following, T'_k is the minimum known value of t_i for version k.

For a single version:

$$p(t_1, \ldots, t_n | a, b, T, T') = \left[\prod_{i=1}^{n} \lambda(t_i - T') \right] e^{-\Lambda(T)} \tag{C.1}$$

$$= \left[\prod_{i=1}^{n} abe^{-b(t_i - T')} \right] e^{-a(1-e^{-bT})} \tag{C.2}$$

However for K versions we have:

$$\prod_{k=1}^{K} \left[\prod_{i=1}^{n_k} b_k a_k e^{-b_k(t_{ik} - T'_k)} \right] e^{-a_k(1-e^{-b_k T_k})} \tag{C.3}$$

And equivalently, writing it for n bugs:

$$p() = \left[\prod_{i=1}^{n} b_{r_i} a_{r_i} e^{-b_{r_i}(t_i - T'_k)} \right] \prod_{k=1}^{K} e^{-a_k(1-e^{-b_k T_k})} \tag{C.4}$$

$$\text{where:} \tag{C.5}$$

$$n_k = \sum_{i=1}^{n} \mathbb{I}(r_i = k) \tag{C.6}$$

Where n_k is the total number of bugs with version k.

We note that if we use the following prior in Equation (C.11), then things will cancel so that we get back the same as in Chapter 5.

$$p(r_i = k) \propto \frac{\exp\left(-d_{r_i}(t_i - T_k - \alpha_k)^2\right)}{b_k a_k e^{-b_k(t_i - T'_k)}} \tag{C.7}$$

then if r_i is not known, we have the full conditional for r_i as:

$$p(r_i = k | \underline{t}, \underline{a}, \underline{b}) \propto p(\underline{t} | \underline{a}, \underline{b}, \underline{r}) p(\underline{a}) p(\underline{b}) p(\underline{r}) \tag{C.8}$$

$$\propto b_{r_i} a_{r_i} e^{-b_{r_i}(t_i - T'_k)} \times p(r_i) \tag{C.9}$$

$$r_i = 1, \ldots, K \tag{C.10}$$

It is up to us to use what we choose as a prior for $p(r_i)$ which we could choose to be Gaussian as we previously used.

NB. T'_k is the offset towards zero for the Goel-Okumoto model, in our model for Chapter 6 it is set so that the first time stamp of the *known* dataset is 0, so it is a fixed function of the data, i.e. $T'_k = \min(t_i|k)$.

$$p(r_i = k|\underline{a}, \underline{b}, \underline{t}) = \frac{b_k a_k e^{-b_k(t_i - T'_k)} p(r_i = k)}{\sum_{j=1}^{K} b_j a_j e^{-b_j(t_i - T'_j)} p(r_i = j)} \tag{C.11}$$

$$= \frac{b_k a_k e^{-b_k(t_i - T'_k)} e^{-d_k(t_i - T_k - \alpha_k)^2}}{\sum_{j=1}^{K} b_j a_j e^{-b_j(t_i - T'_j)} e^{-d_j(t_i - T_j - \alpha_j)^2}} \tag{C.12}$$

$$= \frac{b_k a_k e^{-b_k(t_i - T'_k) - d_k(t_i - T_k - \alpha_k)^2}}{\sum_{j=1}^{K} b_j a_j e^{-b_j(t_i - T'_j)} e^{-d_j(t_i - T_j - \alpha_j)^2}} \tag{C.13}$$

Which allows us to sample the version release given everything else.

Or in log form:

$$\log(b_k) + log(a_k) - b_k(t_i - T'_K) - d_k(t_i - T_k - \alpha_k)^2 \tag{C.14}$$

Note that should we have the ratio $\frac{p(r_i=k)}{p(r_i=j)}$ in the Metropolis-Hastings calculations, the summation term on the bottom of each of the two $p()$ terms will cancel.

$$\frac{p(r_i = k)}{p(r_i = j)} = \frac{\frac{e^{-d_k(t_i - T_k - \alpha_k)^2}}{\sum_m e^{-d_m(t_i - T_m - \alpha_m)^2}}}{\frac{e^{-d_j(t_i - T_j - \alpha_j)^2}}{\sum_m e^{-d_m(t_i - T_m - \alpha_m)^2}}} \tag{C.15}$$

$$= \frac{e^{-d_k(t_i - T_k - \alpha_k)^2}}{e^{-d_j(t_i - T_j - \alpha_j)^2}} \tag{C.16}$$

So the full conditional that we need to use in our code is

$$p(r_i = k|\underline{t}, \underline{a}, \underline{b}, \underline{\alpha}, \underline{d}) \propto b_k a_k e^{-b_k(t_i - T'_k)} e^{-d_k(t_i - T_k - \alpha_k)^2} \tag{C.17}$$

$$k = 1 \dots K \tag{C.18}$$

where $(t_i - T'_k)$ is a normalized t_i such that the t in the Goel-Okumoto model is always positive, we do this by setting the minimum value of t for a particular version to be 0 for consistency. Note too that we have used time in decimal years to ensure that the summed values stay sufficiently small to avoid numeric overflow.

Figure C.1 shows the results of a sample run of the combined model on the Firefox dataset in Chapter 6. This output was generated using `Rstan::monitor()`. Note that

```
            mean se_mean     sd   2.5%     25%     50%     75%   97.5% n_eff Rhat
alpha.1     -0.4     0.0    0.2   -0.8    -0.5    -0.4    -0.3    -0.1    50 1.0
alpha.2      0.1     0.0    0.0    0.0     0.1     0.1     0.1     0.1   101 1.0
alpha.3      0.1     0.0    0.0    0.0     0.1     0.1     0.1     0.1   131 1.0
alpha.4      0.1     0.0    0.0    0.0     0.0     0.1     0.1     0.1    69 1.0
alpha.5      0.0     0.0    0.0   -0.1     0.0     0.0     0.0     0.0    80 1.0
alpha.6     -0.1     0.0    0.0   -0.2    -0.1    -0.1    -0.1    -0.1   176 1.0
alpha.7     -0.1     0.0    0.0   -0.1    -0.1    -0.1    -0.1     0.0    87 1.0
alpha.8     -0.1     0.0    0.0   -0.1    -0.1    -0.1    -0.1     0.0    89 1.1
alpha.9     -0.2     0.0    0.0   -0.2    -0.2    -0.2    -0.1    -0.1    74 1.0
alpha.10    -0.1     0.0    0.0   -0.2    -0.1    -0.1    -0.1    -0.1   154 1.0
alpha.11    -0.1     0.0    0.0   -0.2    -0.1    -0.1    -0.1    -0.1    60 1.0
alpha.12    -0.2     0.0    0.0   -0.2    -0.2    -0.2    -0.2    -0.1    39 1.1
alpha.13    -0.1     0.0    0.0   -0.2    -0.1    -0.1    -0.1    -0.1   123 1.0
alpha.14    -0.1     0.0    0.0   -0.2    -0.1    -0.1    -0.1    -0.1    73 1.1
alpha.15    -0.1     0.0    0.0   -0.2    -0.1    -0.1    -0.1    -0.1    39 1.1
alpha.16    -0.1     0.0    0.0   -0.2    -0.1    -0.1    -0.1    -0.1   141 1.0
alpha.17    -0.1     0.0    0.0   -0.1    -0.1    -0.1    -0.1    -0.1    35 1.1
alpha.18    -0.1     0.0    0.0   -0.2    -0.1    -0.1    -0.1    -0.1    61 1.1
alpha.19     0.4     0.0    0.1    0.2     0.4     0.4     0.5     0.7    82 1.1
alpha.20    -0.1     0.0    0.0   -0.2    -0.2    -0.1    -0.1     0.0   133 1.0
alpha.21     0.2     0.0    0.1    0.0     0.1     0.2     0.2     0.4   108 1.1
alpha.22    -0.2     0.0    0.0   -0.2    -0.2    -0.2    -0.2    -0.1   202 1.0
d.1          1.2     0.0    0.3    0.7     1.0     1.2     1.4     1.8    67 1.0
d.2         17.4     0.2    3.0   12.2    15.2    17.3    19.4    24.3   173 1.0
d.3         12.6     0.2    1.9    9.4    11.1    12.4    13.8    16.4    86 1.1
d.4          7.9     0.1    1.0    6.1     7.1     7.8     8.6     9.7   117 1.0
d.5          6.9     0.1    0.8    5.5     6.3     6.9     7.5     8.5    69 1.0
d.6         20.4     0.4    3.5   14.4    17.9    19.9    22.1    30.3    66 1.1
d.7          9.2     0.1    1.2    6.9     8.3     9.1    10.0    11.6   107 1.0
d.8          7.2     0.1    0.9    5.7     6.6     7.1     7.8     9.1   193 1.0
d.9          8.2     0.1    0.9    6.1     7.6     8.2     8.9    10.1   104 1.0
d.10         7.5     0.1    0.9    5.9     7.0     7.5     8.0     9.4   164 1.0
d.11        12.7     0.2    1.5   10.1    11.7    12.7    13.8    15.6    56 1.1
d.12        11.6     0.1    1.2    9.5    10.8    11.7    12.3    14.1   111 1.0
d.13        18.7     0.5    2.2   14.7    17.0    18.6    20.2    22.9    19 1.1
d.14        11.3     0.1    1.3    9.1    10.3    11.3    12.2    13.7    97 1.0
d.15        15.6     0.2    1.9   12.0    14.2    15.5    17.2    18.9    81 1.0
d.16        15.0     0.1    1.5   12.3    14.1    15.1    15.7    18.5    98 1.0
d.17        18.7     0.2    1.8   15.3    17.3    18.6    19.5    22.5   113 1.0
d.18        20.0     0.3    2.9   14.2    17.9    20.1    22.1    25.2   105 1.1
d.19         2.3     0.1    0.7    1.2     1.8     2.3     2.6     4.2    75 1.1
d.20        40.9     1.1   14.8   15.6    31.1    39.4    51.5    70.6   171 1.0
d.21         4.9     0.1    1.6    2.3     3.7     4.5     6.0     8.6   122 1.0
d.22        57.3     1.7   18.6   24.4    43.8    56.7    68.1    95.0   118 1.1
```

Fig. C.1: Details of the output of one run of the Combined Model from Chapter 6

as we might expect, the effective N is much higher for the Goel-Okumoto parameters, (a, b), than for the version model parameters (α, d), since the latter are only updated in pairs every $K = 22$ iterations, while the former are all updated every iteration.

```
          mean se_mean     sd   2.5%     25%     50%     75%   97.5% n_eff Rhat
a.1      101.0     0.1   10.1   82.0    94.1   100.6   107.6   121.9 19681  1.0
a.2     2406.0    21.7 1106.3  981.7  1640.4  2168.6  2907.9  5195.9  2605  1.0
a.3     1021.2    14.4  565.3  474.6   681.6   865.4  1167.1  2487.8  1543  1.0
a.4     5629.1    49.3 2011.7 2729.9  4242.6  5299.0  6613.4 10600.6  1664  1.0
a.5     1129.7    25.7  726.6  429.3   673.3   917.7  1341.0  3100.0   799  1.0
a.6     3191.5    20.9 1212.6 1501.6  2342.4  2973.0  3787.7  6141.9  3367  1.0
a.7     2361.6    18.4 1117.6  891.0  1576.9  2135.4  2889.8  5153.9  3680  1.0
a.8     3657.0    29.5 1586.1 1550.7  2571.6  3344.6  4368.5  7634.7  2898  1.0
a.9     6796.8    82.5 2544.0 3338.6  5082.0  6315.9  7928.8 13079.8   950  1.0
a.10    1605.7    18.6  842.5  561.0  1028.4  1417.4  1968.0  3782.9  2054  1.0
a.11    2793.1    32.7 1484.1 1006.2  1803.1  2464.2  3390.8  6545.0  2057  1.0
a.12      41.2     0.0    6.4   29.6    36.7    40.9    45.3    54.7 25578  1.0
a.13    2540.3    32.6 1533.4  769.5  1501.6  2164.4  3154.5  6493.1  2217  1.0
a.14   14832.9    74.2 4052.5 7547.2 12159.4 14669.3 17183.4 23518.9  2981  1.0
a.15     555.8    11.3  500.4   87.6   230.4   403.1   707.1  1922.1  1973  1.0
a.16     852.7    12.6  607.5  195.2   439.3   693.8  1086.2  2434.5  2307  1.0
a.17    1598.4    28.4 1150.9  325.2   812.0  1295.5  2040.5  4630.8  1644  1.0
a.18    1347.9     9.4  545.7  662.2   981.4  1228.5  1573.4  2761.5  3404  1.0
a.19      42.1     0.0    7.0   29.8    37.3    41.6    46.4    56.9 23293  1.0
a.20      41.4     0.0    6.6   29.6    36.8    41.1    45.7    55.4 29337  1.0
a.21     488.4     9.8  221.1  299.8   381.0   436.1   520.7  1030.6   511  1.0
a.22     469.0     2.8   96.9  320.7   404.7   455.7   518.3   690.7  1208  1.0
b.1        2.5     0.0    0.3    2.0     2.3     2.5     2.7     3.1 19604  1.0
b.2        0.1     0.0    0.1    0.0     0.1     0.1     0.1     0.2  2715  1.0
b.3        0.6     0.0    0.3    0.1     0.3     0.5     0.7     1.2   973  1.0
b.4        0.1     0.0    0.1    0.1     0.1     0.1     0.2     0.3  1308  1.0
b.5        0.4     0.0    0.2    0.1     0.2     0.3     0.5     0.9  1104  1.0
b.6        0.1     0.0    0.0    0.0     0.0     0.1     0.1     0.1  3584  1.0
b.7        0.1     0.0    0.1    0.0     0.1     0.1     0.1     0.2  3151  1.0
b.8        0.1     0.0    0.0    0.0     0.1     0.1     0.1     0.2  3582  1.0
b.9        0.2     0.0    0.1    0.1     0.2     0.2     0.3     0.4  1172  1.0
b.10       0.2     0.0    0.1    0.1     0.1     0.2     0.2     0.4  3197  1.0
b.11       0.1     0.0    0.0    0.0     0.1     0.1     0.1     0.2  3177  1.0
b.12       4.2     0.0    0.7    2.9     3.7     4.2     4.7     5.7 26352  1.0
b.13       0.2     0.0    0.1    0.1     0.1     0.2     0.2     0.4  3233  1.0
b.14       0.1     0.0    0.0    0.1     0.1     0.1     0.1     0.2  3866  1.0
b.15       0.2     0.0    0.2    0.0     0.1     0.1     0.2     0.7  1619  1.0
b.16       0.6     0.0    0.4    0.1     0.3     0.5     0.8     1.8  2629  1.0
b.17       0.2     0.0    0.1    0.0     0.1     0.1     0.2     0.5  2615  1.0
b.18       3.0     0.0    1.5    0.9     1.9     2.7     3.8     6.5  3431  1.0
b.19       2.1     0.0    0.5    1.2     1.8     2.1     2.4     3.0 18067  1.0
b.20       2.1     0.0    0.4    1.3     1.8     2.1     2.3     2.9 20872  1.0
b.21       1.9     0.0    0.8    0.5     1.4     1.9     2.5     3.4   433  1.0
b.22       2.0     0.0    0.6    0.9     1.6     2.0     2.4     3.1   465  1.0
```

C.0.1 Hierarchical Combined Model Results

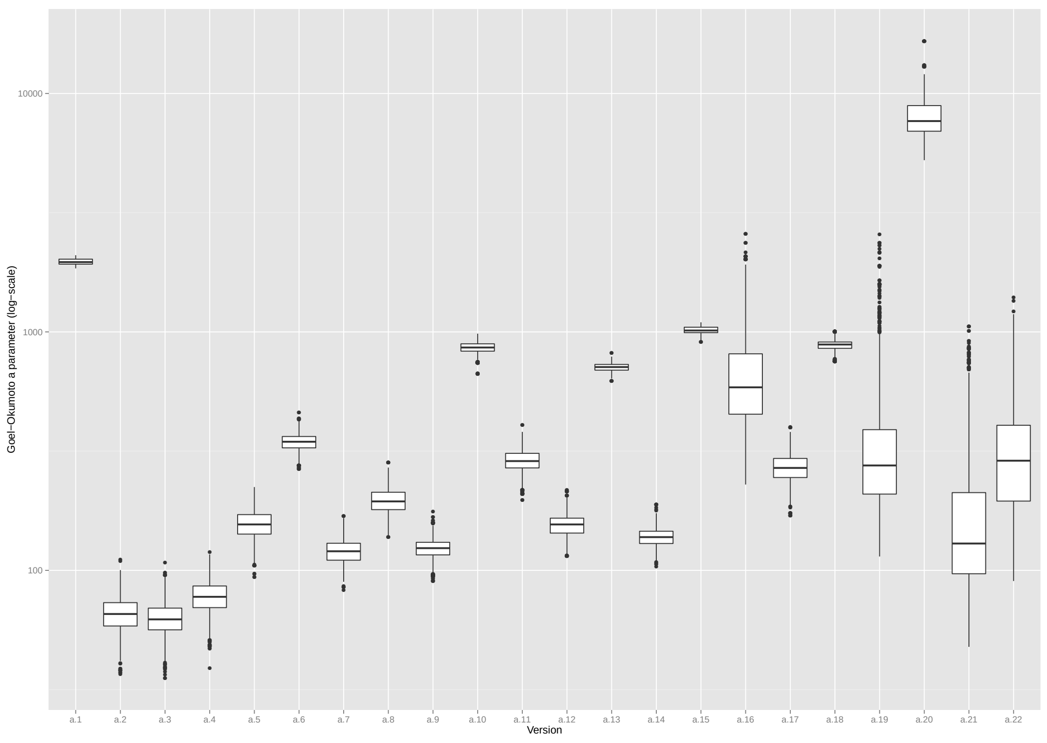


Fig. C.2: Box plot of the a parameters for the hierarchical model from Section 6.3.

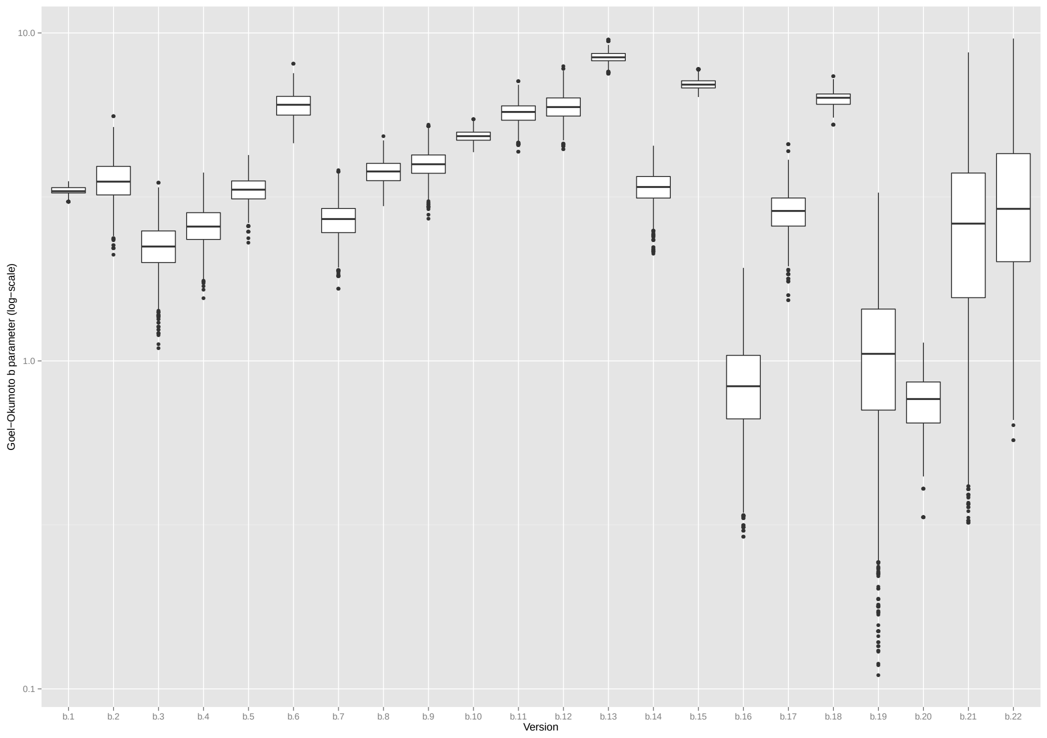


Fig. C.3: Box plot of the b parameters for the hierarchical model from Section 6.3.

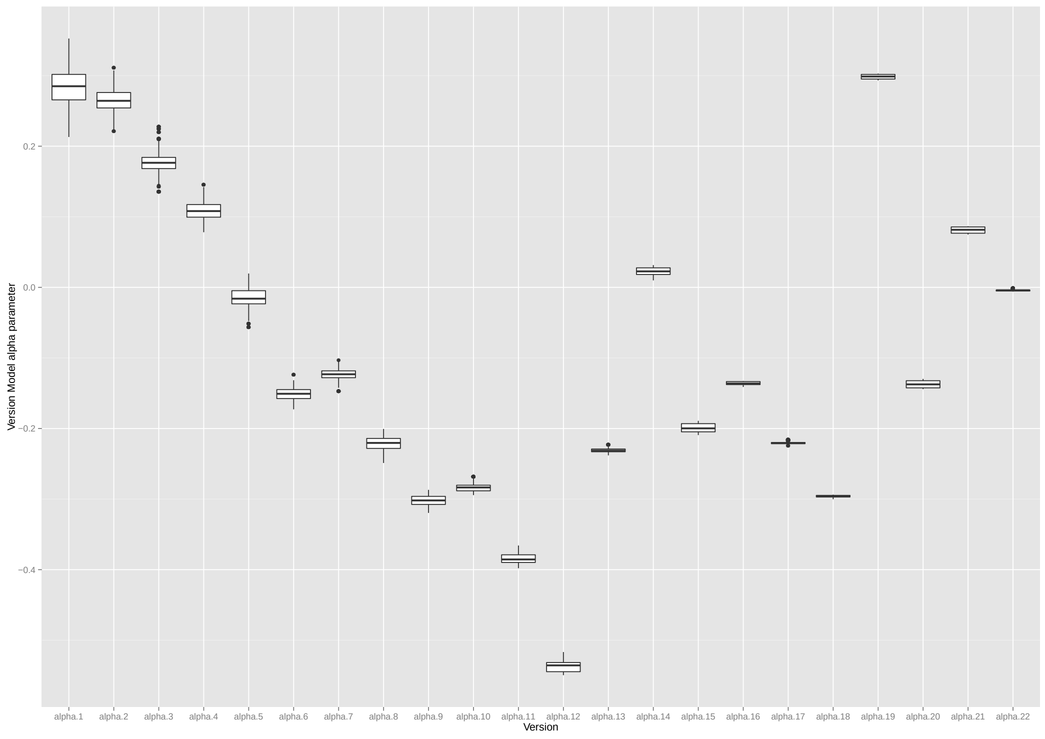


Fig. C.4: Box plot of the α parameters for the hierarchical model from Section 6.3.

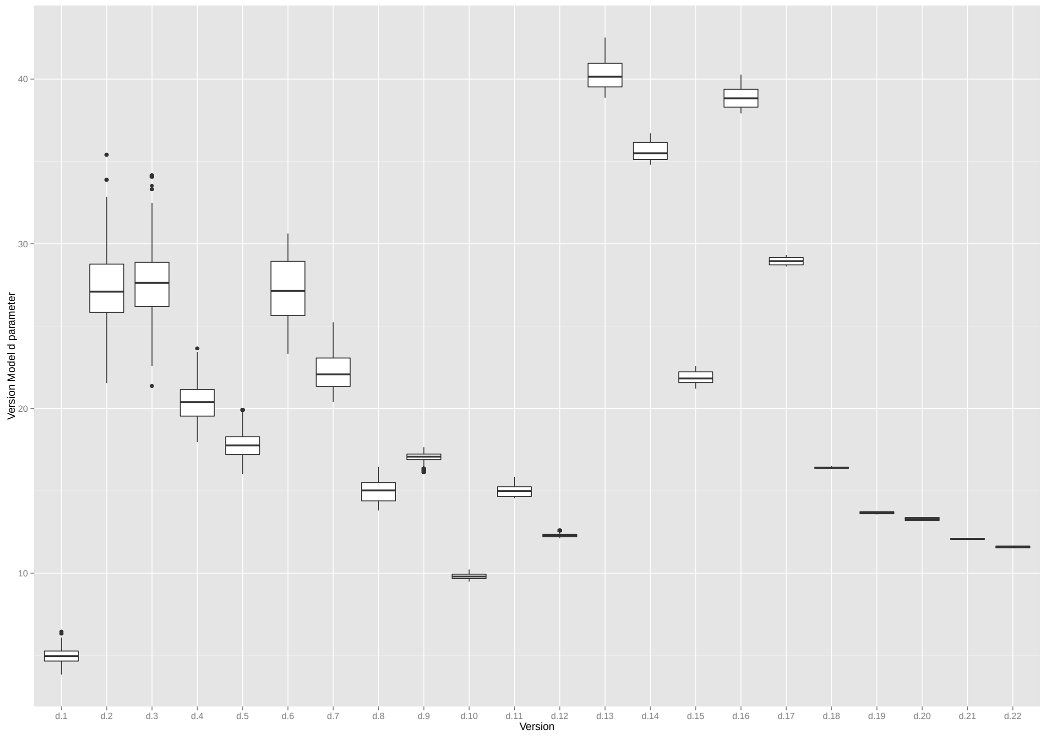


Fig. C.5: Box plot of the d parameters for the hierarchical model from Section 6.3.

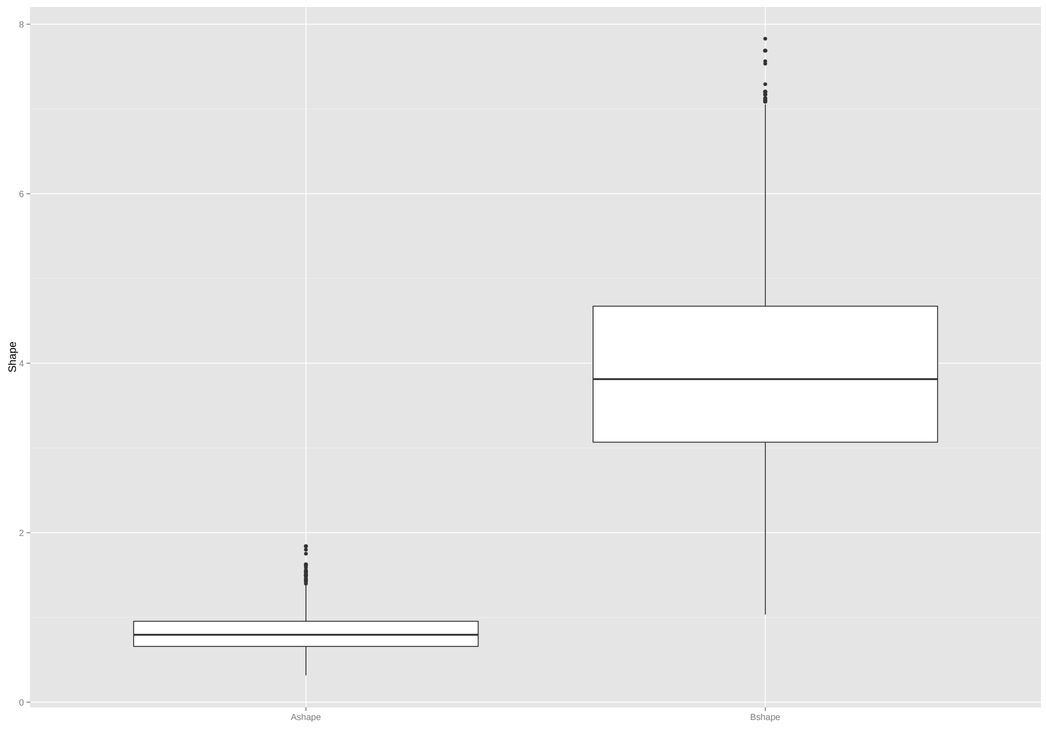


Fig. C.6: Box plot of the A and B Shape parameters for the hierarchical model from Section 6.3.

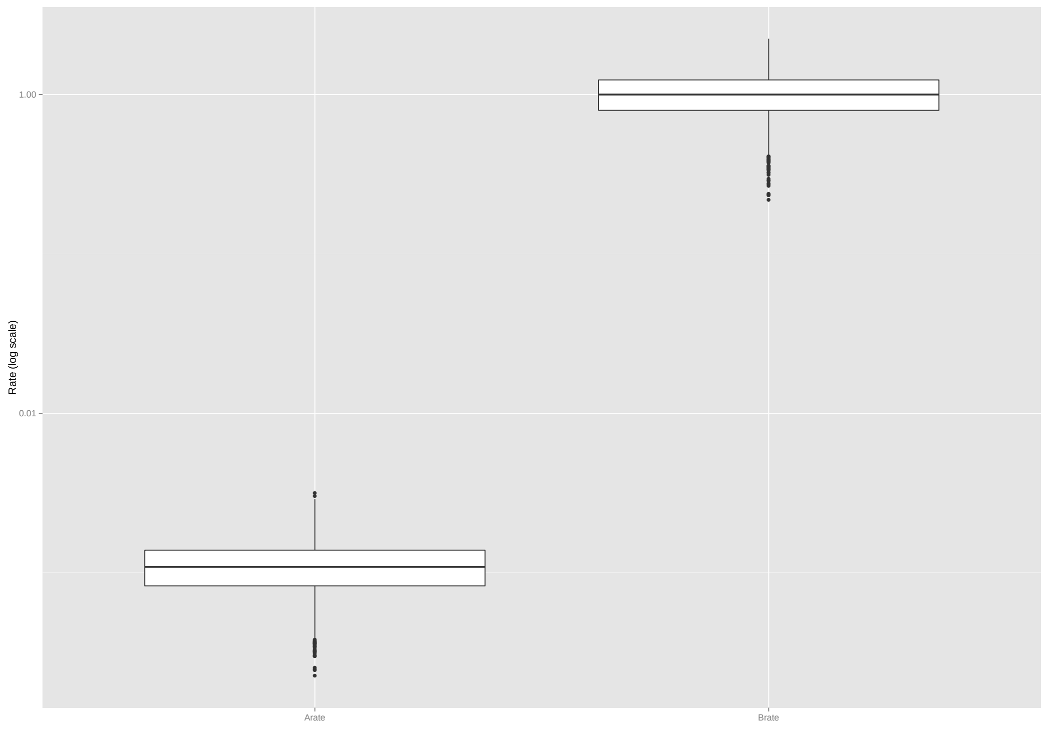


Fig. C.7: Box plot of the A and B Rate parameters for the hierarchical model from Section 6.3.

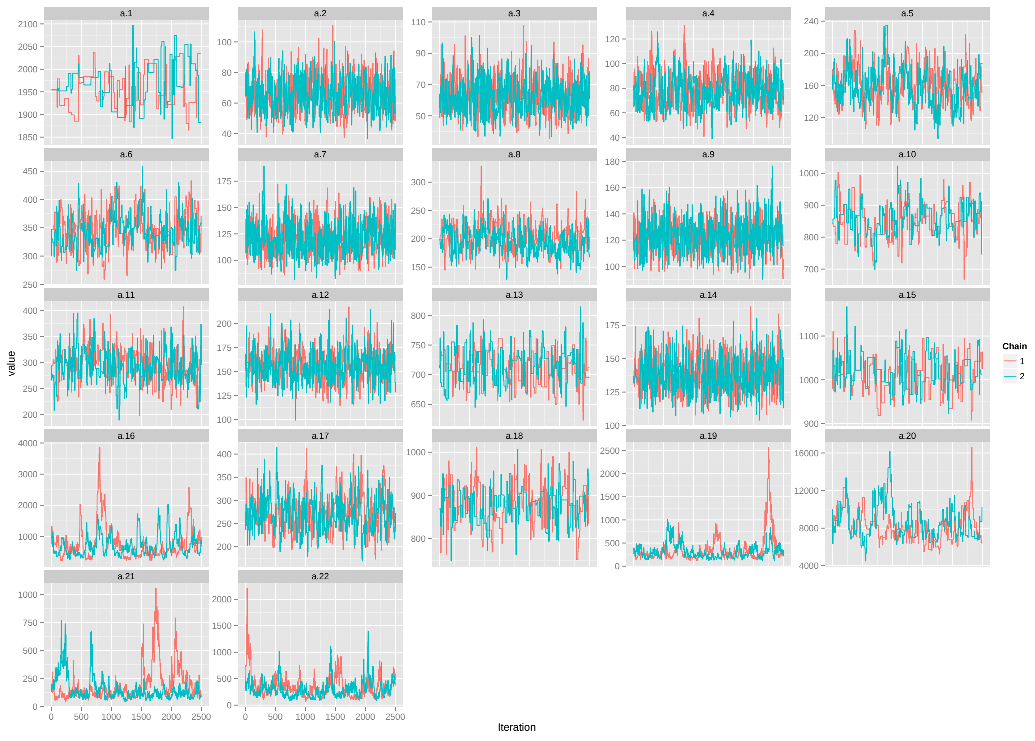


Fig. C.8: Trace plots of the a parameters for the hierarchical model from Section 6.3.

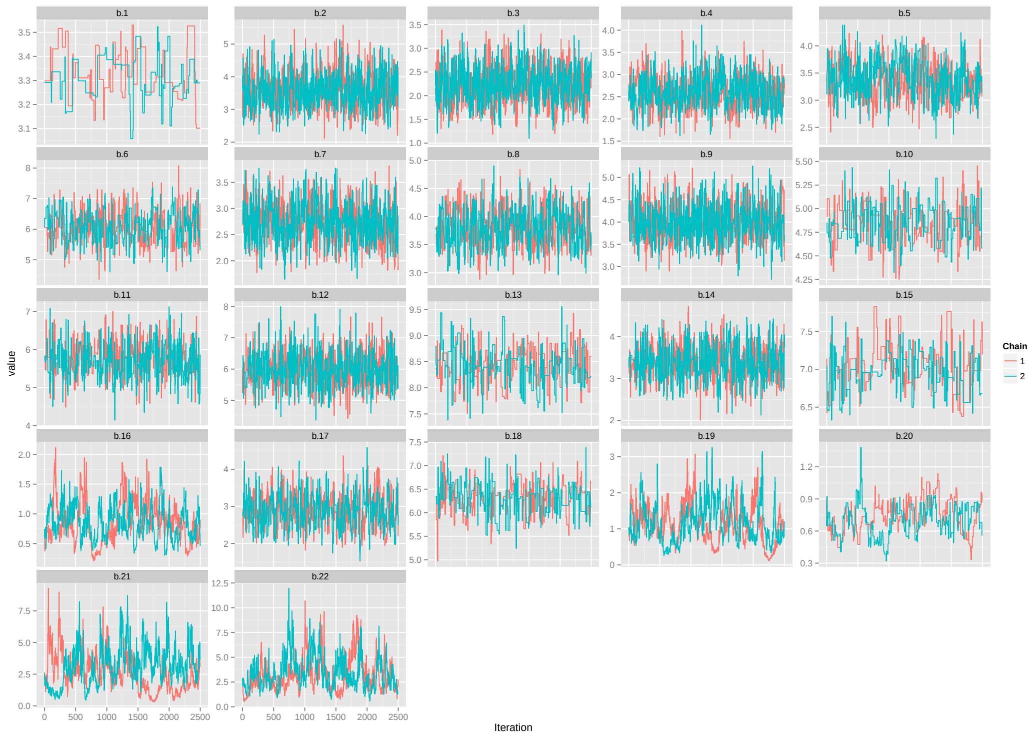


Fig. C.9: Trace plots of the b parameters for the hierarchical model from Section 6.3.

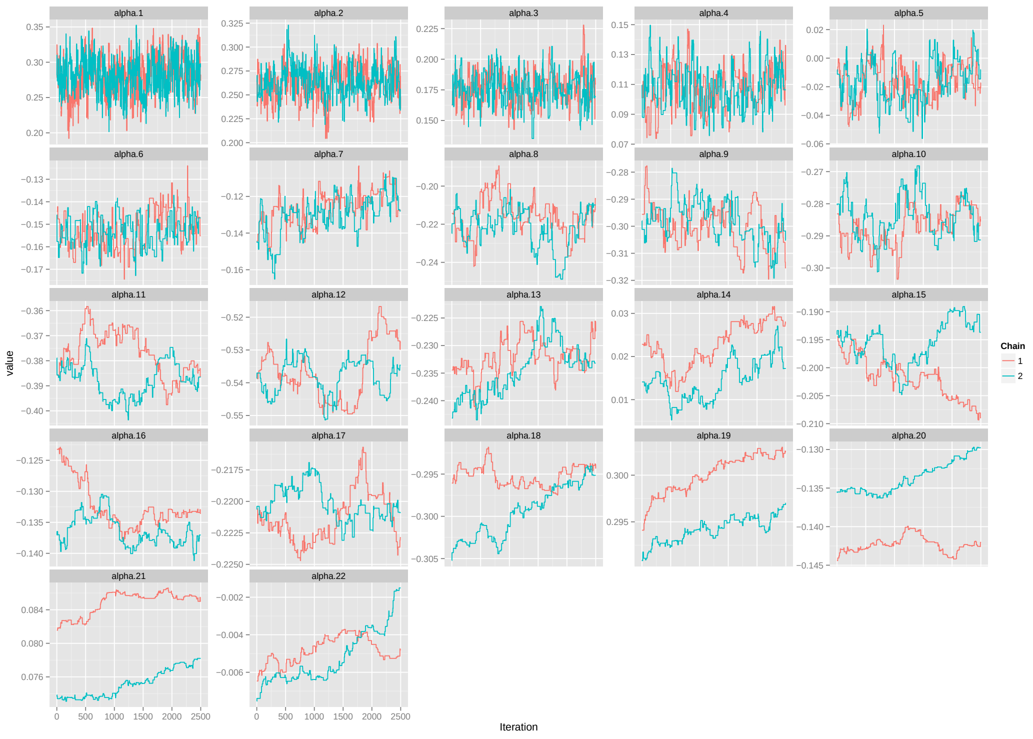


Fig. C.10: Trace plots of the α parameters for the hierarchical model from Section 6.3.

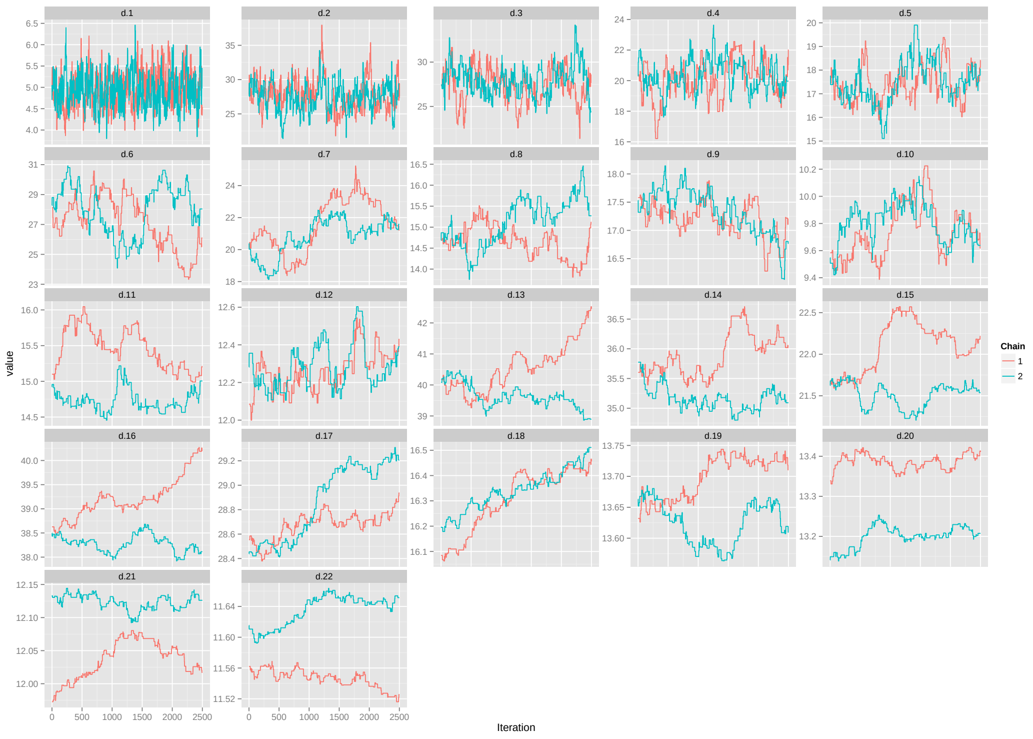


Fig. C.11: Trace plots of the d parameters for the hierarchical model from Section 6.3.

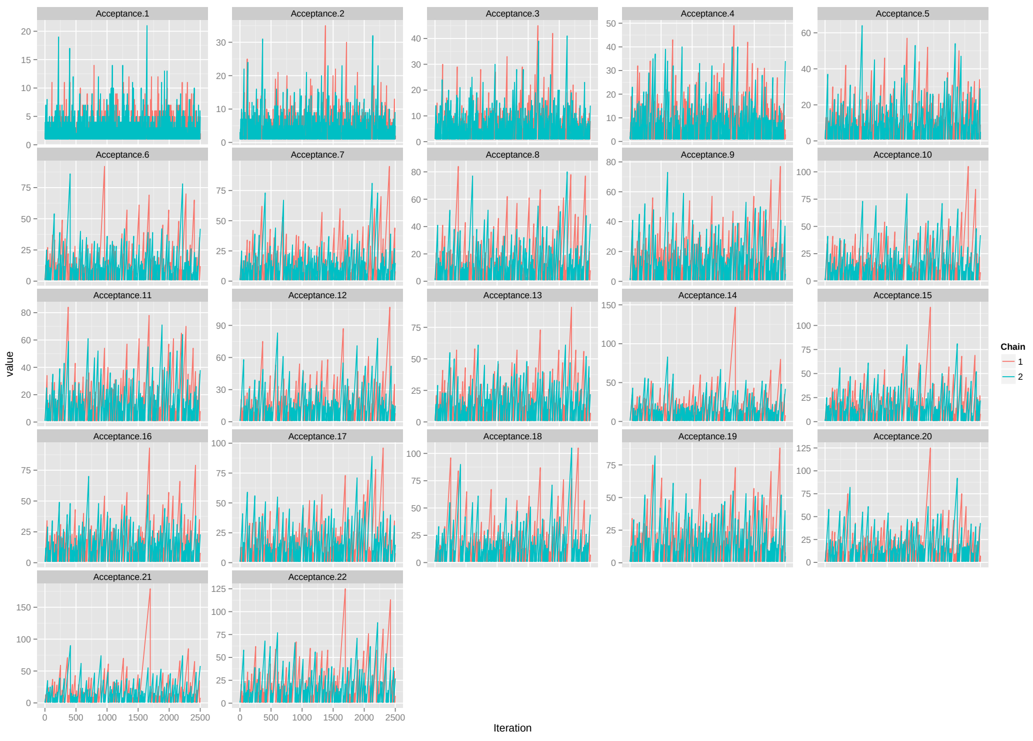


Fig. C.12: Trace plots of the Acceptance for the version model portion of the hierarchical model from Section 6.3. Note that this plots the number of rejects on the Y-axis +1.

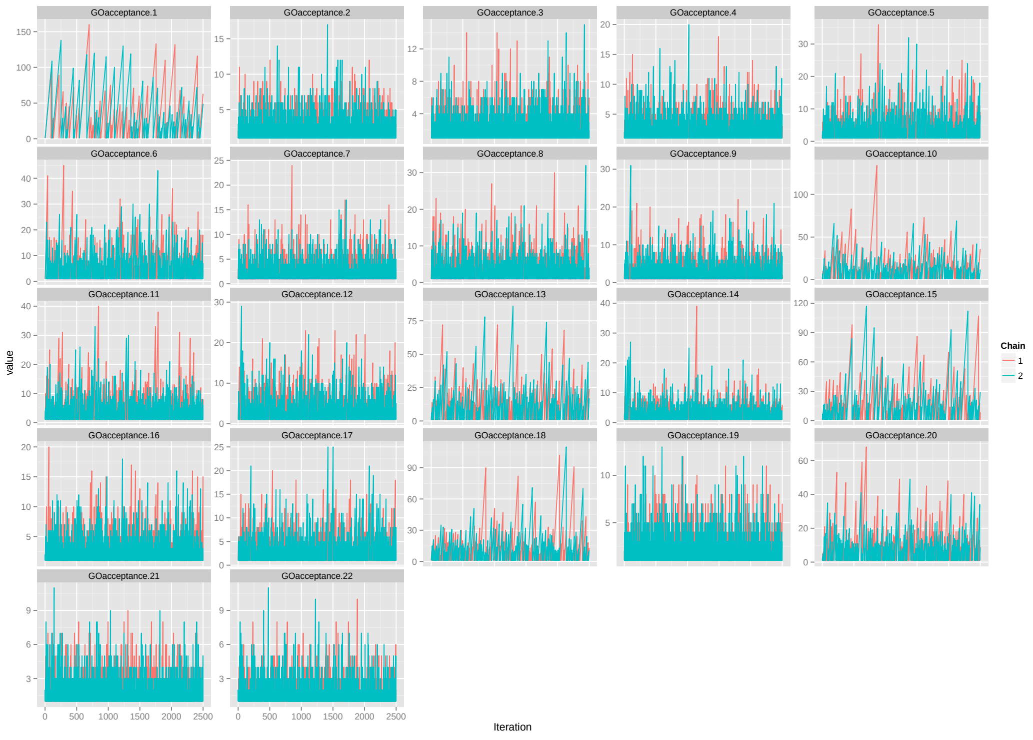


Fig. C.13: Trace plots of the Acceptance for the Goel-Okumoto model portion of the hierarchical model from Section 6.3. Note that this plots the number of rejects on the Y-axis +1.

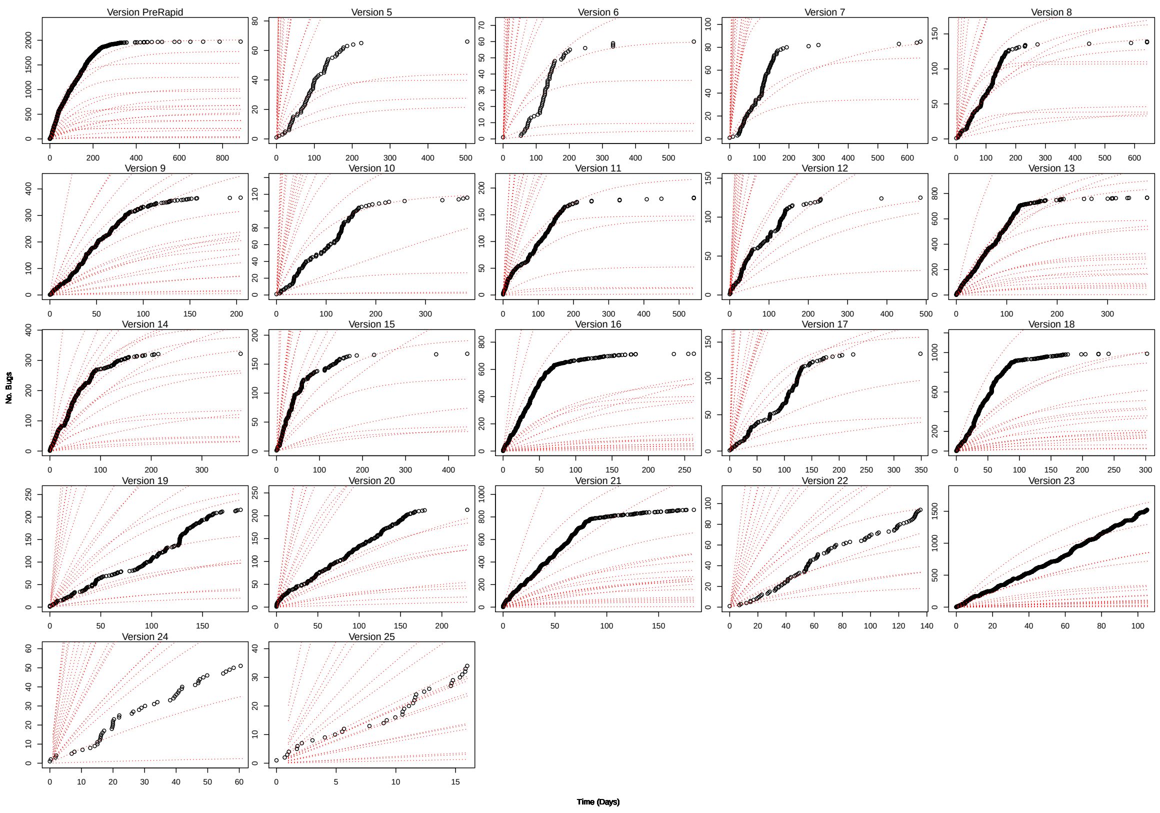


Fig. C.14: Sample plots of imputed and original points along with the corresponding Goel-Okumoto Mean Value Functions sampled from the last iteration of a 10k MCMC run.

```
Inference for the input samples (2 chains: each with iter=3334; warmup=1667):

           mean se_mean     sd   2.5%    25%    50%    75%    97.5% n_eff Rhat
alpha.1     0.3     0.0    0.0    0.2    0.3    0.3    0.3      0.3   134  1.0
alpha.2     0.3     0.0    0.0    0.2    0.3    0.3    0.3      0.3   237  1.0
alpha.3     0.2     0.0    0.0    0.1    0.2    0.2    0.2      0.2   172  1.0
alpha.4     0.1     0.0    0.0    0.1    0.1    0.1    0.1      0.1   112  1.0
alpha.5     0.0     0.0    0.0    0.0    0.0    0.0    0.0      0.0    98  1.0
alpha.6    -0.2     0.0    0.0   -0.2   -0.2   -0.2   -0.1     -0.1   135  1.0
alpha.7    -0.1     0.0    0.0   -0.2   -0.1   -0.1   -0.1     -0.1    52  1.0
alpha.8    -0.2     0.0    0.0   -0.2   -0.2   -0.2   -0.2     -0.2    33  1.1
alpha.9    -0.3     0.0    0.0   -0.3   -0.3   -0.3   -0.3     -0.3     3  1.2
alpha.10   -0.3     0.0    0.0   -0.3   -0.3   -0.3   -0.3     -0.3    39  1.0
alpha.11   -0.4     0.0    0.0   -0.4   -0.4   -0.4   -0.4     -0.4     2  1.5
alpha.12   -0.5     0.0    0.0   -0.6   -0.5   -0.5   -0.5     -0.5    11  1.1
alpha.13   -0.2     0.0    0.0   -0.2   -0.2   -0.2   -0.2     -0.2     8  1.2
alpha.14    0.0     0.0    0.0    0.0    0.0    0.0    0.0      0.0     9  1.5
alpha.15   -0.2     0.0    0.0   -0.2   -0.2   -0.2   -0.2     -0.2    13  1.1
alpha.16   -0.1     0.0    0.0   -0.1   -0.1   -0.1   -0.1     -0.1     3  1.8
alpha.17   -0.2     0.0    0.0   -0.3   -0.3   -0.2   -0.2     -0.2     1  3.4
alpha.18   -0.3     0.0    0.0   -0.3   -0.3   -0.3   -0.3     -0.3     6  1.9
alpha.19    0.3     0.0    0.0    0.3    0.3    0.3    0.3      0.3     1  2.4
alpha.20   -0.1     0.0    0.0   -0.1   -0.1   -0.1   -0.1     -0.1     1  3.9
alpha.21    0.1     0.0    0.0    0.1    0.1    0.1    0.1      0.1     1  2.9
alpha.22    0.0     0.0    0.0    0.0    0.0    0.0    0.0      0.0     5  2.1
d.1         5.0     0.0    0.4    4.2    4.7    4.9    5.2      5.8   409  1.0
d.2        27.3     0.2    2.0   23.7   25.9   27.1   28.6     31.7   180  1.0
d.3        27.8     0.1    1.7   24.6   26.5   27.7   29.0     31.1   146  1.0
d.4        20.3     0.2    1.1   18.1   19.5   20.2   21.0     22.3    48  1.0
d.5        17.3     0.1    0.9   15.5   16.7   17.3   17.9     19.1    47  1.1
d.6        27.3     0.8    1.6   23.9   26.3   27.2   28.1     30.7     4  1.2
d.7        21.5     0.3    1.1   19.6   20.8   21.7   22.2     23.9    12  1.1
d.8        15.1     0.2    0.7   13.8   14.6   15.0   15.7     16.6    12  1.1
d.9        16.5     0.6    0.8   15.0   15.8   16.6   17.1     18.4     2  1.7
d.10        9.5     0.1    0.3    9.0    9.3    9.5    9.7     10.0     7  1.3
d.11       14.1     0.4    0.5   13.2   13.7   14.1   14.6     15.0     1  2.0
d.12       12.0     0.1    0.3   11.4   11.8   12.0   12.2     12.4     8  1.4
d.13       40.6     1.2    1.4   37.3   39.7   40.6   41.8     42.5     1  2.5
d.14       37.2     1.7    1.7   34.9   35.4   37.1   39.1     39.9     1  3.5
d.15       24.3     0.4    0.5   23.6   23.9   24.4   24.7     25.0     1  2.6
d.16       39.3     1.0    1.0   37.9   38.4   39.3   40.4     40.7     1  5.9
d.17       29.9     0.1    0.1   29.7   29.8   29.9   30.0     30.3     3  1.3
d.18       17.7     0.3    0.4   17.1   17.4   17.5   17.9     18.3     1  3.6
d.19       13.7     0.0    0.1   13.6   13.7   13.7   13.7     13.8     3  1.2
d.20       13.4     0.0    0.0   13.3   13.3   13.4   13.4     13.4     7  1.6
d.21       12.2     0.1    0.1   12.0   12.1   12.2   12.3     12.4     1  8.2
d.22       11.9     0.1    0.1   11.8   11.8   11.9   12.0     12.0     1  6.0
```

This figure shows some sample results from the `rstan::monitor()` for a thinned run of the hierarchical combined model.

```
a.1       1965.5     4.7   48.9 1870.7 1942.6 1964.3 1996.2  2058.5  110  1.0
a.2         65.9     0.4   10.7   47.3   58.7   64.9   72.8    88.5  690  1.0
a.3         63.0     0.3   10.0   45.3   55.9   62.2   69.5    84.5  816  1.0
a.4         76.9     3.5   13.4   54.0   67.5   75.9   85.7   105.2   15  1.0
a.5        160.7     2.5   23.2  118.2  144.9  160.0  175.6   207.2   85  1.1
a.6        348.1     1.7   28.0  295.1  328.6  347.4  367.9   403.3  273  1.0
a.7        120.8     0.5   13.9   96.6  110.8  119.8  130.1   149.4  790  1.0
a.8        192.1     1.4   22.6  150.9  175.7  191.3  206.3   241.2  256  1.0
a.9        122.3     0.4   11.6  100.9  114.2  121.7  130.1   146.3  976  1.0
a.10       868.3     4.4   50.7  765.3  836.4  869.0  901.6   974.3  134  1.0
a.11       283.3     2.7   31.0  226.9  262.8  281.7  303.5   346.2  132  1.0
a.12       156.2     0.5   15.6  126.3  146.3  155.5  165.8   187.1 1068  1.0
a.13       715.9     1.6   29.6  661.1  693.5  714.9  735.6   770.5  348  1.0
a.14       138.7     0.3   12.5  116.7  130.0  138.1  146.9   165.3 1422  1.0
a.15       980.6     2.6   37.7  911.8  954.1  979.4 1006.3  1060.4  206  1.0
a.16       824.6    65.8  569.7  301.5  480.2  646.4  945.3  2594.7   75  1.0
a.17       251.2     1.3   32.6  196.1  229.5  248.8  268.3   323.9  584  1.0
a.18       895.1     4.2   45.1  796.9  867.1  896.8  923.7   980.9  114  1.0
a.19       324.0    27.8  215.2  139.0  209.1  268.6  358.0   865.0   60  1.0
a.20      8721.1   216.7 2088.9 5747.0 7299.9 8243.8 9556.0 13967.2   93  1.0
a.21       202.1    35.9  202.9   65.8  100.6  139.6  208.5   818.0   32  1.1
a.22       334.3    24.2  203.0  112.2  192.4  270.8  416.7   904.5   70  1.0
b.1          3.3     0.0    0.1    3.1    3.3    3.3    3.4     3.5   62  1.1
b.2          3.5     0.0    0.5    2.6    3.2    3.5    3.9     4.6  741  1.0
b.3          2.3     0.0    0.4    1.6    2.0    2.3    2.5     3.0  927  1.0
b.4          2.6     0.0    0.4    1.9    2.4    2.6    2.8     3.3  709  1.0
b.5          3.4     0.0    0.4    2.7    3.1    3.4    3.6     4.1  262  1.0
b.6          6.0     0.0    0.5    5.1    5.7    6.0    6.4     7.0  549  1.0
b.7          2.7     0.0    0.4    2.0    2.5    2.7    3.0     3.6 1028  1.0
b.8          3.8     0.0    0.3    3.1    3.5    3.8    4.0     4.5  746  1.0
b.9          4.0     0.0    0.4    3.2    3.7    4.0    4.3     4.7 1219  1.0
b.10         4.9     0.0    0.2    4.5    4.7    4.9    5.1     5.3  395  1.0
b.11         5.7     0.0    0.4    4.9    5.4    5.7    5.9     6.6  941  1.0
b.12         6.0     0.0    0.5    5.0    5.7    6.0    6.3     7.1 1488  1.0
b.13         8.4     0.0    0.3    7.7    8.2    8.4    8.7     9.1  235  1.0
b.14         3.4     0.0    0.4    2.6    3.1    3.4    3.6     4.2  891  1.0
b.15         7.2     0.0    0.3    6.7    7.0    7.2    7.4     7.7  188  1.0
b.16         0.8     0.0    0.3    0.3    0.6    0.8    1.0     1.5   91  1.0
b.17         3.0     0.0    0.4    2.1    2.7    3.0    3.2     3.9  682  1.0
b.18         6.4     0.0    0.3    5.7    6.2    6.4    6.6     7.1  421  1.0
b.19         1.2     0.1    0.6    0.3    0.8    1.1    1.5     2.6   78  1.0
b.20         0.7     0.0    0.2    0.4    0.6    0.7    0.8     1.1  101  1.0
b.21         2.6     0.2    1.5    0.4    1.6    2.4    3.5     6.1   45  1.1
b.22         3.2     0.2    1.6    0.9    1.9    2.9    4.2     7.0   91  1.0
Ashape       0.8     0.0    0.2    0.5    0.7    0.8    0.9     1.2  236  1.0
Arate        0.0     0.0    0.0    0.0    0.0    0.0    0.0     0.0  214  1.0
Bshape       3.6     0.1    1.1    1.9    2.9    3.5    4.2     6.0  119  1.0
Brate        1.0     0.0    0.3    0.5    0.7    0.9    1.1     1.6  170  1.0
```

Bibliography

Acton, F. S. (1990), *Numerical methods that usually work*, Mathematical Association of America Washington, DC.

Agresti, A. (1990), *Categorical Data Analysis*, first edn, John Wiley & Sons.

Aitkin, M., Anderson, D. and Hinde, J. (1981), 'Statistical modelling of data on teaching styles', *Journal of the Royal Statistical Society. Series A (General)* **144**(4), 419–461.
URL: *http://www.jstor.org/stable/2981826*

Akaike, H. (1974), 'A new look at the statistical model identification', *Automatic Control, IEEE Transactions on* **19**(6), 716–723.
URL: *http://dx.doi.org/10.1109/TAC.1974.1100705*

Andrieu, C. and Thoms, J. (2008), 'A tutorial on adaptive MCMC', *Statistics and Computing* **18**(4), 343–373.

Barber, D. (2012), *Bayesian Reasoning and Machine Learning*, Cambridge University Press.

Beck, K. (1999), *Extreme programming explained: embrace change*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Bernardo, J. M. and Smith, A. F. (2000), *Bayesian theory*, Vol. 405 of *Wiley series in probability and mathematical statistics*, John Wiley & Sons, Chichester, England.

Betancourt, M. J. (2013), 'Generalizing the no-U-turn sampler to riemannian manifolds', *arXiv* **1304**(1920).
URL: *http://arxiv.org/abs/1304.1920*

Bishop, C. M. (2006), *Pattern Recognition and Machine Learning*, Springer, New York.

Bishop, C. M. (2007), Generative or discriminative? getting the best of both worlds, *in* J. M. Bernardo, M. Bayarri, J. O. Berger, A. P. Dawid, D. Heckerman, A. Smith and M.West, eds, 'Bayesian Statistics 8', Oxford University Press, pp. 3–24.

Blackduck (2014), 'Open hub, mozilla firefox page', `https://www.openhub.net/p/firefox`, retrieved 2014-10-13.

Boehm, B. (1976), 'Software engineering', *IEEE Transactions on Computers* **25**(12), 1226–1241.

Boehm, B. (1984), 'Software engineering economics', *Software Engineering, IEEE Transactions on* **SE-10**(1), 4–21.

Bolstad, W. M. (2010), *Understanding computational Bayesian statistics*, John Wiley & Sons.

Bridle, J. S. (1989), Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition, *in* 'Neurocomputing: Algoriths, Architectures and Applications', Springer-Verlag, pp. 227–236.

Brooks, S. P. and Gelman, A. (1998), 'General methods for monitoring convergence of iterative simulations', *Journal of Computational and Graphical Statistics* **7**, 434–455.

Carpenter, J. R. and Kenward, M. G. (2013), *Multiple imputation and its application*, John Wiley & Sons, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, United Kingdom.

Ceriani, L. and Verme, P. (2012), 'The origins of the Gini index: extracts from Variabilità e mutabilità (1912) by Corrado Gini', *The Journal of Economic Inequality* **10**(3), 421–443.

Chapelle, O., Schölkopf, B. and Zien, A. (2006), *Semi–Supervised Learning*, The MIT Press, Cambridge, Massachusetts.

Chib, S. and Greenberg, E. (1995), 'Understanding the Metropolis-Hastings algorithm', *The American Statistician* **49**(4), 327–335.

Clicky (2013), 'Usage share of web browsers', `http://clicky.com/marketshare/global/web-browsers/`, retrieved 2013-09-30.

Cohen, J. (1960), ‘A coefficient of agreement for nominal data’, *Educational and Psychological Measurement* **15**(2), 37–46.

Connor, S. and Goldschmidt, C. (2012), ‘Applied stochastic processes’, Private communication.

Corbet, J., Kroah-Hartman, G. and McPherson, A. (2015), ‘Linux kernel development: How fast is it going, who is doing it, what are they doing and who is sponsoring the work’.
URL: *http://www.wwwlinuxfoundation.org*

D’Ambros, M., Lanza, M. and Robbes, R. (2010), An extensive comparison of bug prediction approaches, *in* ‘Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories)’, IEEE CS Press, pp. 31 – 41.

de Finetti, B. (1929), Funzione caratteristica di un fenomeno aleatorio, *in* Zanichelli, ed., ‘Atti del Congresso Internazionale dei Matematici, Bologna, dal 3 al 10 di settembre di 1928’, Vol. 6: Comunicazioni, sezione IV (A)-V-VII., Congresso Internazionale dei Matematici, Bologna, Italy, pp. 179–190.

de Finetti, B. (1937), La prévision: ses lois logiques, ses sources subjectives, *in* ‘Annales de l’institut Henri Poincaré’, Vol. 7, Presses universitaires de France, pp. 1–68.

de Finetti, B. (1974), *Theory of Probability*, Wiley, New York, USA.

DiCiccio, T. J., Kass, R. E., Raftery, A. E. and Wasserman, L. (1997), ‘Computing Bayes factors by combining simulation and asymptotic approximations’, *Journal of the American Statistical Association* **92**(439), 903–915.

Evans, M. and Swartz, T. (2000), *Approximating integrals via Monte Carlo and deterministic methods*, Oxford University Press.

Feller, J. and Fitzgerald, B. (2002), *Understanding Open Source software development*, Addison-Wesley.

Fitzgerald, B. (2006), ‘The transformation of open source software’, *MIS Quarterly* **30**(3), 587–598.

Fowler, M. (1999), *Refactoring: improving the design of existing code*, Addison Wesley.

French, S. (1986), *Decision theory: an introduction to the mathematics of rationality.*, Ellis Horwood series in mathematics and its applications, Ellis Horwood, Chichester, England.

Friedman, J. H. (1984*a*), SMART user's guide, Technical Report Technical Report No. 1., Laboratory for Computational Statistics, Stanford University.

Friedman, J. H. (1984*b*), A variable span scatterplot smoother, Technical Report Technical Report No. 5., Laboratory for Computational Statistics, Stanford University.

Gelfand, A. E. and Smith, A. F. (1990), 'Sampling-based approaches to calculating marginal densities', *Journal of the American Statistical Association* **85**(410), 398–409.

Gelman, A. (1992), 'Iterative and non-terative simulation algorithms', *Computing Science and Statistics (Interface proceedings)* **24**, 433–438.

Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A. and Rubin, D. B. (2013), *Bayesian Data Analysis*, Chapman & Hall/CRC Texts in Statistical Science, third edn, CRC Press, Taylor & Francis.

Gelman, A. and Hill, J. (2007), *Data Analysis Using Regression and Multilevel / Hierarchical Models*, Cambridge University Press.

Gelman, A., Roberts, G. and Gilks, W. (1996), 'Efficient Metropolis jumping rules', *Bayesian statistics* **5**, 599–608.

Gelman, A. and Rubin, D. B. (1992), 'Inference from iterative simulation using multiple sequences', *Statistical Science* **7**(4), 457–511.

Gelman, A. and Shirley, K. (2011), Inference from simulations and monitoring convergence, *in* S. Brooks, A. Gelman, G. L. Jones and X.-L. Meng, eds, 'Handbook of Markov Chain Monte Carlo', Taylor & Francis Group, Boca Raton, FL, pp. 163–174.

Geman, S. and Geman, D. (1984), 'Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images', *Pattern Analysis and Machine Intelligence, IEEE Transactions on* **PAMI-6**(6), 721–741.

Geyer, C. (2011), Introduction to Markov chain Monte Carlo, *in* S. Brooks, A. Gelman, G. L. Jones and X.-L. Meng, eds, 'Handbook of Markov Chain Monte Carlo', Taylor & Francis Group, Boca Raton, FL, pp. 3–48.

Gilks, W. R., Richardson, S. and Spiegelhalter, D. J. (1996), Introducing Markov chain Monte Carlo, *in* 'Markov Chain Monte Carlo in practice', Chapman & Hall/CRC.

Gini, C. (1912), *Variabilità e Mutuabilità. Contributo allo Studio delle Distribuzioni e delle Relazioni Statistiche*, C. Cuppini, Bologna, Italy.

Goel, A. L. and Okumoto, K. (1979), 'Time-dependent error-detection rate model for software reliability and other performance measures', *IEEE Trasactions on Reliability* **R-28**(3), 206–211.

Goldberg, D. (1991), 'What every computer scientist should know about floating-point arithmetic', *ACM Comput. Surv.* **23**(1), 5–48.
URL: *http://doi.acm.org.elib.tcd.ie/10.1145/103162.103163*

Goldratt, E. M. and Cox, J. (1984), *The Goal: A process of ongoing improvement*, first edn, North River Press.

Haario, H., Saksman, E. and Tamminen, J. (2001), 'An adaptive Metropolis algorithm', *Bernoulli* **7**(2), 223–242.
URL: *http://projecteuclid.org/euclid.bj/1080222083*

Han, C. and Carlin, B. P. (2001), 'Markov chain Monte Carlo methods for computing Bayes factors: A comparative review', *Journal of the American Statistical Association* **96**(455), pp. 1122–1132.
URL: *http://www.jstor.org/stable/2670258*

Hastie, T., Tibshirani, R. and Friedman, J. (2009), *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, second edn, Springer-Verlag, New York, USA.

Hastings, W. K. (1970), 'Monte Carlo sampling methods using Markov chains and their applications', *Biometrika* **57**(1), 97–109.

Hobert, J. P. and Casella, G. (1996), 'The effect of improper priors on Gibbs sampling in hierarchical linear mixed models', *Journal of the American Statistical Association* **91**(436), 1461 – 1473.

Hoffman, M. D. and Gelman, A. (2011), 'The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo', *arXiv* **1111**(4246).
URL: *http://arxiv.org/abs/1111.4246*

Hoffman, M. D. and Gelman, A. (2014), 'The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo', *Journal of Machine Learning Research* **15**(Apr), 1593–1623.

IEEE Task P754 (2008), *IEEE 754-2008, Standard for Floating-Point Arithmetic*, IEEE, New York, NY, USA.

International Electrotechnical Commission (2014), Medical device software - part 3: process reference model of medical device software life cycle processes, Technical Report IEC/TR 80002-3, International Organization for Standardization, Geneva, Switzerland.

International Telecommunications Union (2013), 'Internet usage', `http://en.wikipedia.org/wiki/Global_Internet_usage`, retrieved 2013-09-30.

Jeffries, H. (1939), *Theory of probability*, Oxford: Clarendon Press.

Jelinski, Z. and Moranda, P. B. (1972), 'Software reliability research', *Statistical Computer Performance Evaluation* pp. 465–484.

Jeske, D. R. and Pham, H. (2001), 'On the maximum likelihood estimates for the Goel-Okumoto software reliability model', *The American Statistician* **55**(3), 219–222.

Kan, S. H. (1995), *Metrics and models in software quality engineering*, Addison-Wesley.

Kass, R. E. and Raftery, A. E. (1995), 'Bayes factors', *Journal of the American Statistical Association* **90**(430), 773–795.

Knuth, D. E. (1974), 'Structured programming with go to statements', *ACM Computing Surveys (CSUR)* **6**(4), 261–301.

Kruschke, J. K. (2011), *Doing Bayesian Data Analysis: A Tutorial with R and BUGS*, Academic Press, Burlington, MA.

Kuhn, M. and Johnson, K. (2013), *Applied Predictive Modelling*, first edn, Springer.

Kuhn, M., Wing, J., Weston, S., Williams, A., Keefer, C., Engelhardt, A., Cooper, T., Mayer, Z. and the R Core Team (2014), *caret: Classification and Regression Training.* R package version 6.0-35.
URL: *http://CRAN.R-project.org/package=caret*

Lamkanfi, A., Perez, J. and Demeyer, S. (2013), The eclipse and mozilla defect tracking dataset: a genuine dataset for mining bug information, *in* 'MSR '13: Proceedings of the 10th Working Conference on Mining Software Repositories, May 18-19, 2013. San Francisco, California, USA', IEEE Press, pp. 203–206.

Lavine, M. and Schervish, M. J. (1999), 'Bayes factors: What they are and what they are not', *The American Statistician* **53**(2), pp. 119–122.
URL: *http://www.jstor.org/stable/2685729*

Lindley, D. V. (1969*a*), A Bayesian solution for some educational prediction problems, Technical Report RB-69-57, Educational Testing Service, Princeton, New Jersey.

Lindley, D. V. (1969*b*), A Bayesian solution for some educational prediction problems ii, Technical Report RB-69-91, Educational Testing Service, Princeton, New Jersey.

Lindley, D. V. (1970), A Bayesian solution for some educational prediction problems iii, Technical Report RB-70-33, Educational Testing Service, Princeton, New Jersey.

Lindley, D. V. (1985), *Making Decisions*, second edn, Wiley, King's Lynn, Great Britain.

Lindley, D. V. and Smith, A. F. M. (1972), 'Bayes estimates for the linear model', *Journal of the Royal Statistical Society. Series B (Methodological)* **34**(1), 1–41.

Littlewood, B. and Mayne, A. J. (1989), 'Predicting software reliability', *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences* **327**(1596), 513–527.

Littlewood, B. and Verrall, J. L. (1973), 'A Bayesian reliability growth model for computer software', *Journal of the Royal Statistical Society. Series C (Applied Statistics)* **22**(3), pp. 332–346.
URL: *http://www.jstor.org/stable/2346781*

Lutz, M. J., Naveda, J. F. and Vallino, J. R. (2014), 'Undergraduate software engineering', *Communications of the ACM* **57**(8), 52–58.
URL: *http://dx.doi.org/10.1145/2632361*

MacKay, D. J. C. (2003), *Information Theory, Inference, and Learning Algorithms*, Cambridge University Press.

McDaid, K. (1998), Deciding How Long to Test Software, PhD thesis, University of Dublin, Trinity College.

McDaid, K. and Wilson, S. P. (2001), 'Deciding how long to test software', *The Statistician* **50**(Part 2), 117–134.

McGrayne, S. B. (2011), *The theory that would not die*, Yale University Press.

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H. and Teller, E. (1953), 'Equation of state calculations by fast computing machines', *Journal of Chemical Physics* **21**, 1087–1092.

Meyn, S. and Tweedie, R. (1993), *Markov chains and stochastic stability*, Springer-Verlag, London.
URL: *http://probability.ca/MT*

Mozilla Corporation (2013*a*), 'Mozilla corporation', `http://www.mozilla.org/en-US/foundation/moco/`, retrieved 2014-10-13.

Mozilla Corporation (2013*b*), 'Mozilla corporation faq', `http://www.mozilla.org/en-US/foundation/annualreport/2011/faq/`.

Mozilla Foundation (2013), 'Mozilla foundation', `http://www.mozilla.org/en-US/foundation/about/`, retrieved 2014-10-13.

Mozillians (2011), 'Enterprise/firefox/extendedsupport:proposal', `https://wiki.mozilla.org/Enterprise/Firefox/ExtendedSupport:Proposal`, retrieved 2014-10-13.

Murphy, K. P. (2012), *Machine learning: a probabilistic perspective*, The MIT Press, Cambridge, Massachusetts, USA.

Musa, J. D. (1975), 'A theory of software reliability and its application', *Software Engineering, IEEE Transactions on* **SE-1**(3), 312–327.

Musa, J. D. (2004), *Software reliability engineering: more reliable software, faster and cheaper*, Tata McGraw-Hill Education.

Musa, J. D., Iannino, A. and Okumoto, K. (1987), *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill, New York, USA.

Neal, R. M. (1993), Probabilistic inference using Markov chain Monte Carlo methods, Technical Report CRG-TR-93-1, Department of Computer Science, University of Toronto.

Neal, R. M. (2011), MCMC using Hamiltonian dynamics, *in* 'Handbook of Markov Chain Monte Carlo', Chapman & Hall/CRC.

Novick, M. R., Jackson, P. H., Thayer, D. T. and Cole, N. S. (1972), 'Estimating multiple regressions in m groups: A cross-validation study', *British Journal of Mathematical and Statistical Psychology* **25**(1), 33–50.

O'Regan, G. (2002), *A Practical Approach to Software Quality*, Springer-Verlag, New York, USA.

Pham, H. (2010), *System Software Reliability*, Springer, London, England.

Pievatolo, A., Ruggeri, F. and Soyer, R. (2010), 'A Bayesian hidden Markov model for imperfect debugging', **Technical Report TR-2010-10**.

Pievatolo, A., Ruggeri, F. and Soyer, R. (2012), 'A Bayesian hidden Markov model for imperfect debugging', *Reliability Engineering & System Safety* **103**, 11–21.

Press, W. H. (2007), *Numerical recipes 3rd edition: The art of scientific computing*, Cambridge University Press.

R Core Team (2015), *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria.
URL: *http://www.R-project.org/*

Raudenbush, S. and Bryk, A. S. (1986), 'A hierarchical model for studying school effects', *Sociology of education* **59**(1), 1–17.

Ravishanker, N., Liu, Z. and Ray, B. K. (2008), 'NHPP models with Markov switching for software reliability', *Computational Statistics & Data Analysis* **52**(8), 3988–3999.

Ray, B. K., Liu, Z. and Ravishanker, N. (2006), 'Dynamic reliability models for software using time-dependent covariates', *Technometrics* **48**(1), 1–10.

Raymond, E. S. (1999), *The Cathedral and the Bazaar*, first edn, O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Robert, C. and Casella, G. (2011), A short history of MCMC: Subjective recollections from incomplete data, *in* S. Brooks, A. Gelman, G. Jones and X.-L. Meng, eds, 'Handbook of Markov Chain Monte Carlo', Chapman & Hall /CRC Press, Boca Raton, FL, pp. 49–66.

Roberts, G. O. and Rosenthal, J. S. (2007), 'Coupling and ergodicity of adaptive Markov chain Monte Carlo algorithms', *J. Appl. Probab.* **44**(2), 458–475.
URL: *http://dx.doi.org/10.1239/jap/1183667414*

Roberts, G. O. and Rosenthal, J. S. (2009), 'Examples of adaptive MCMC', *Journal of Computational and Graphical Statistics* **18**(2), 349–367.

Rosenthal, J. S. (2011), Optimal proposal distributions and adaptive MCMC, *in* 'Handbook of Markov Chain Monte Carlo', Chapman & Hall/CRC.

Ross, S. M. (1996), *Stochastic processes*, second edn, John Wiley & Sons, New York.

Rubin, D. B. (1977), 'The design of a general and flexible system for handling non-response in sample surveys'. Manuscript prepared for the U.S. Social Security Administration, republished in The American Statistician, Vol. 58, No. 4 (Nov., 2004), pp. 298-302.

Rubin, D. B. (1987), *Multiple Imputation for Nonresponse in Surveys*, John Wiley and Sons, Hoboken, New Jersey.

Ruggeri, F. and Soyer, R. (2008), Advances in Bayesian software reliability modelling, *in* T. Bedford, J. Quigley, L. Walls, B. Alkali, A. Daneshkhah and G. Hardman, eds, ‘Advances in Mathematical Modeling for Reliability’, IOS Press, Amsterdam.

Schwaber, K. (1997), Scrum development process, *in* J. V. Sutherland, D. Patel, C. Casanave, J. Miller and G. Hollowell, eds, ‘Business Object Design and Implementation: OOPSLA ’95 Workshop Proceedings 16 October 1995, Austin, Texas’, Springer, London. Also available on page 56ff of Sutherland 2012 at `http://jeffsutherland.com/ScrumPapers.pdf`, retrieved 2014-10-14.

Schwarz, G. (1978), ‘Estimating the dimension of a model’, *Ann. Statist.* **6**(2), 461–464.
URL: *http://dx.doi.org/10.1214/aos/1176344136*

Sheather, S. J. and Jones, M. C. (1991), ‘A reliable data-based bandwidth selection method for kernel density estimation’, *Journal of the Royal Statistical Society, Series B* **53**(3), 683–690.

Singpurwalla, N. D. and Soyer, R. (1985), ‘Assessing (software) reliability growth using a random coefficient autoregressive process and its ramifications’, *IEEE Transactions on Software Engineering* .

Singpurwalla, N. D. and Wilson, S. P. (1994), ‘Software reliability modeling’, *International Statistical Review* **62**(3), 289–317.

Singpurwalla, N. D. and Wilson, S. P. (1999), *Statistical Methods in Software Engineering: Reliability and Risk*, first edn, Springer-Verlag, New York, USA.

Soyer, R. (2011), ‘Software reliability’, *WIREs Comp Stat* **3**, 269–281.

Stan Development Team (2014*a*), ‘Rstan: the r interface to stan, version 2.4’.
URL: *http://mc-stan.org/rstan.html*

Stan Development Team (2014*b*), *Stan Modeling Language Users Guide and Reference Manual, Version 2.5.0*.
URL: *http://mc-stan.org/*

Stats, S. G. (2013), 'Browser share', `http://gs.statcounter.com/#browser-ww-monthly-201306-201306-bar`, retrieved 2013-09-30.

Sutherland, J. (2012), 'The scrum papers: nuts, bolts and origins of and agile framework', `http://jeffsutherland.com/ScrumPapers.pdf`, retrieved 2014-10-14.

Tanner, M. A. (1996), *Tools for Statistical Inference, Methods for the Exploration of Posterior Distributions and Likelihood Functions*, third edn, Springer.

Tanner, M. and Wong, W. (1987), 'The calculation of posterior distributions by Data Augmentation (with discussion).', *Journal of the American Statistical Association* **82**(398), 528–540.
URL: *http://www.jstor.org/stable/2289457*

The Bugzilla Team (2014), 'The bugzilla guide - 4.4.6+ release', `http://www.bugzilla.org/docs/4.4/en/html/`, retrieved 2014-10-21.

van Vliet, H. (2000), *Software Engineering: Principles and Practice*, second edn, Springer-Verlag, New York, USA.

Vapnik, V. N. (1998), *Statistical Learning Theory*, Wiley, New York.

w3counter (2013), 'Browser share', `http://www.w3counter.com/globalstats.php?year=2013&month=06`, retrieved 2013-09-30.

Wikimedia (2013), 'Browser share', `http://stats.wikimedia.org/archive/squid_reports/2013-06/SquidReportClients.htm`, retrieved 2013-09-30.

Yamada, S., Ohba, M. and Osaki, S. (1983), 'S-shaped reliability growth modeling for software error detection', *IEEE Transactions on Reliability* **R-32**(5), 475–484.

Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S. and Bairavasundaram, L. (2011), How do fixes become bugs?, *in* 'Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering', ESEC/FSE '11,

ACM, New York, NY, USA, pp. 26–36.
URL: *http://doi.acm.org/10.1145/2025113.2025121*

Zeileis, A. (2014), *ineq: Measuring Inequality, Concentration, and Poverty.* R package version 0.2-13.
URL: *http://CRAN.R-project.org/package=ineq*

Zellner, A. and Theil, H. (1962), 'Three-stage least squares: Simultaneous estimation of simultaneous equations', *Econometrica* **30**(1), 54–78.
URL: *http://www.jstor.org/stable/1911287*