A Nonparametric Approach to Software Reliability

Axel Gandy and Uwe Jensen*†

Department of Stochastics, University of Ulm, D-89069 Ulm, Germany

Summary

In this paper we present a new, nonparametric approach to software reliability. It is based on a multivariate counting process with additive intensity, incorporates covariates and includes several projects in one model. Furthermore, we present ways to obtain failure data from the development of open source software. We analyze a dataset from this source and consider several choices of covariates. We are able to observe a different impact of recently added and older source code onto the failure intensity.

KEY WORDS: software reliability, open source software, multivariate counting processes, Aalen model, additive risk model, survival analysis

1 Introduction

In 1972, Jelinski and Moranda [13] proposed a model which helped create the field of software reliability. Since then, lots of models have been proposed, most are based on counting processes, some rely on classical statistics, some are Bayesian (see [17], [18], [19]). Most models are parametric. During the last 30 years none of these models proved superior. One of the reasons could be the lack of suitable, big datasets to test the models. Usually, software companies do not publish failure data of their development process. An indication is that the biggest dataset publicly available today is more than 20 years old (see [16]), even though software development progresses rapidly. We describe a way that could help out of this predicament.

In recent years, a new way of developing software emerged: open source software. Some projects not only publish their source code, but they also publish failure data (mostly bug reports). Since large datasets can be obtained from this source, we were able to try a new, nonparametric approach to software reliability.

For this, we chose a model proposed by Odd Aalen ([4], [5], [6]). We consider n software projects and let $\boldsymbol{N}(t) = (N_1(t), \dots, N_n(t))$ be the process counting the number of failures up to time t. For each project i we furthermore observe k covariates $Y_{i1}(t), \dots, Y_{ik}(t)$. The main assumption of the model is that the intensity $\boldsymbol{\lambda}(t) = (\lambda_1(t), \dots, \lambda_n(t))$ of $\boldsymbol{N}(t)$ can be written as

$$\boldsymbol{\lambda}(t) = \boldsymbol{Y}(t)\boldsymbol{\alpha}(t), \tag{1}$$

where $\boldsymbol{\alpha}(t) = (\alpha_1(t), \dots, \alpha_k(t))$ is a vector of unknown deterministic baseline intensities.

*Correspondence to: U. Jensen, Department of Stochastics, University of Ulm, D-89069 Ulm, Germany.

†E-Mail: jensen@mathematik.uni-ulm.de

We use the above model to analyze a dataset from open source software. We explore different choices of covariates, which are based on the size of the source code of the projects and on past failures. When using recently added source code and older source code as covariates we find differences between the corresponding baseline intensities.

The paper is organized as follows. In section 2 we discuss problems in software reliability that lead to our approach. The statistical model is introduced in section 3. Estimators for this model and methods to assess goodness of fit are also presented. How to obtain up-to-date failure data of many projects that include covariates is discussed in section 4. What we describe was made possible by the rise of open source software in the last decade. Results of applying the statistical model to such datasets are the topic of section 5. In the last section alternative approaches and possibilities for future research are discussed.

2 Remarks on Software Reliability

A classical model of software development is the "waterfall" model. It structures development into sequential phases, i.e. a new phase does not begin before the previous phase has been completed. For our purposes it is sufficient to consider only 5 phases: analysis, design, coding, test and operation. In the analysis phase, the problem to be solved is analyzed and requirements for the software are being defined. In the design phase, the software's system architecture and a detailed design is developed. During coding, the actual software (the "code") is written. In the test phase, it is checked whether the requirements from the analysis and design phases are met by the software. Finally, during operation, the software is deployed.

Most models in software reliability focus on the test phase. The setup is usually as follows.

A time interval $\mathcal{T} = [0, \tau], 0 < \tau < \infty$ is fixed, during which the software is tested. Whenever the software exhibits a behavior not meeting the requirements (this is called a "failure"), the time is recorded. Call these times T_i. Assuming that no two failures occur at the same time, we can define a counting process N by

$$N(t) = \sum_i 1_{\{T_i \leq t\}}, t \in \mathcal{T},$$

where $1_{\{T_i \leq t\}} = 1$ if $T_i \leq t$ and $1_{\{T_i \leq t\}} = 0$ otherwise. $N(t)$ counts how many failures have occurred up to time t.

We denote the information available up to time $t \in \mathcal{T}$ by $\mathcal{F}_t$. Formally $(\mathcal{F}_t), t \in \mathcal{T}$ is an increasing family of σ-algebras. In most models, $\mathcal{F}_t = \sigma(N(s), s \leq t)$ is chosen, i.e. the information available at time t is the path of N up to time t.

Models differ by the way the intensity $\lambda(t)$ of $N(t)$ is modeled. Heuristically $\lambda(t)$ satisfies

$$E(N(t + dt) - N(t)|\mathcal{F}_t) = \lambda(t)dt,$$

i.e. $\lambda(t)$ is the rate at which failures occur. In the last equation, the symbol E denotes expectation. More formally, the intensity $\lambda(t)$ of $N(t)$ is a process such that $M(t) = N(t) - \int_0^t \lambda(s)ds$ is a martingale.

As a reminder, a process $M(t)$ is called a martingale if for all $0 \leq s \leq t \leq \tau$, $M(t)$ is $\mathcal{F}_t$-measurable for each t, $M_0 = 0$, $E|M(t)| < \infty$ and $E(M(t)|\mathcal{F}_s) = M(s)$. The last requirement for martingales can be interpreted as follows. The best guess for the expected future value of a martingale is its value today. An immediate consequence of this definition is that for all $t \in \mathcal{T}$, $EM(t) = 0$.

One of the earliest models in software reliability is the model by Jelinski and Moranda published 1972 in [13]. It uses the following intensity.

$$\lambda_t = \Phi(n - N(t-)),$$

where $N(t-) = \lim_{s \to t, s<t} N(s)$. An interpretation of the model is that n is the initial number of faults (bugs) present in the software and Φ is the rate at which bugs are discovered. Every bug initially present in the software contributes equally to the failure rate and upon discovery (i.e. it causes a failure) it is instantaneously removed (perfect debugging).

Another well-known model is due to Goel and Okumoto [11]. Here

$$\lambda(t) = abe^{-bt},$$

where a and b are some positive constants.

We will not attempt to list all models. According to [18], more than 50 models have been proposed. Overviews can be found in [17], [18] and [19]. [19] focuses on a Bayesian perspective.

The general idea behind models in software reliability is that the removal of bugs leads to "reliability growth", i.e. the failure intensity is decreasing (at least in some respect). This is a major difference to classical reliability theory, where hardware is subject to wear and tear, which increases the failure intensity over time. In contrast to that software does not degrade.

A common feature of the models is their parametric approach and that they only consider one single project. Exceptions to the parametric approach are the use of the semi-parametric Cox regression model (see e.g. [18]) and a nonparametric order statistics model (see [8]).

Since software development can be influenced by many factors (like size, number of testers available, programming language used, ...) it might be helpful to incorporate covariates into models. But most models, proposed up to now do not incorporate covariates. An exception is given in [20], which allows the addition of code, whenever a failure occurs. As we will see, our approach allows a much more flexible use of covariates.

The development of software is a very complicated process which differs hugely between projects. Trying to describe the failure process resulting from this by a simple parametric form (for all software development projects) might not be possible.

These considerations led the authors to evaluate the possible use of nonparametric models incorporating covariates in software reliability. Furthermore, we wanted to consider multiple projects within one statistical model.

Many software projects today incorporate preexisting software and do not start from scratch. This can be in the guise of a new version or in the use of components developed earlier or by a third party. Models found in the literature are not designed for this. Musa [17] referred to this as "evolving software" and suggested to use a transformation of the timescale to cope with it. Our approach can deal with this by using covariates.

3 An Additive Model

3.1 The Model

The model to be described in this section will be the main tool for our application to software reliability. It was introduced by Odd Aalen ([4], [5], [6]).

We fix a time interval $\mathcal{T} = [0, \tau], 0 < \tau < \infty$ during which we observe an n-variate counting process $\boldsymbol{N} = (N_1, \ldots, N_n)^T$ adapted to a given filtration $(\mathcal{F}_t), t \in \mathcal{T}$.

For us, n will be the number of software projects observed and $N_i(t)$ the number of failures that occurred in the i-th project up to time t.

We assume that $\boldsymbol{N}$ admits an intensity $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_n)^T$, meaning that for each i, λ_i is the intensity of N_i. Hence

$$\boldsymbol{M}(t) := (M_i)(t) := \boldsymbol{N}(t) - \int_0^t \boldsymbol{\lambda}(s) ds \tag{2}$$

is a martingale, meaning that each component M_i of $\boldsymbol{M}$ is a martingale. The key assumption in the model is that $\boldsymbol{\lambda}(t)$ can be written as

$$\boldsymbol{\lambda}(t, \omega) = \boldsymbol{Y}(t, \omega)\boldsymbol{\alpha}(t), \quad t \in \mathcal{T}, \tag{3}$$

where for some $k \leq n$, $\boldsymbol{\alpha}$ is a k-variate deterministic process and $\boldsymbol{Y}$ is an $n \times k$-matrix of predictable locally bounded processes. k will represent the number of covariates, $\boldsymbol{Y}(t)$ contains all these covariates and $\boldsymbol{\alpha}(t)$ represents the unknown time-dependent influence of the covariates on the failure rate. In our examples, the covariates will be based on the size of the projects considered and on past failures.

For convenience, we will assume that $\boldsymbol{Y}(t)$ always has full rank. How to proceed if this is not the case can be found in [7], [14], [10].

In the next section we describe ways to estimate $\int_0^t \boldsymbol{\alpha}(s) ds$ and $\boldsymbol{\alpha}(t)$.

Remark. In the literature, different names are used for this model. In [12] and [14], it is called "Aalen's additive risk model". In [7], the term "matrix multiplicative intensity model" is mentioned and a section called "nonparametric additive hazard models" discusses only this model. When Aalen introduced this model, he called it "a matrix version of the multiplicative intensity model" [4].

3.2 Nelson-Aalen Estimator

An estimator for $\boldsymbol{B}(t) := \int_0^t \boldsymbol{\alpha}(s) ds$ can be motivated as follows. (2) and (3) can be written in differential form as

$$d\boldsymbol{N}(t) - d\boldsymbol{M}(t) = \boldsymbol{Y}(t)\boldsymbol{\alpha}(t) dt.$$

If $\boldsymbol{Y}^-(t) = (Y_{ij}^-(t))$ satisfies $\boldsymbol{Y}^-(t)\boldsymbol{Y}(t) = \boldsymbol{I}$, where $\boldsymbol{I}$ is the $k \times k$ identity matrix then

$$\boldsymbol{\alpha}(t) dt = \boldsymbol{Y}^-(t) d\boldsymbol{N}(t) - \boldsymbol{Y}^-(t) d\boldsymbol{M}(t).$$

Such a $\boldsymbol{Y}^-(t)$ will be called "generalized inverse" of $\boldsymbol{Y}(t)$. Since $\boldsymbol{Y}(t)$ has full rank by assumption, a possible choice for $\boldsymbol{Y}^-(t)$ is

$$\boldsymbol{Y}^-(t) = (\boldsymbol{Y}(t)^T \boldsymbol{Y}(t))^{-1} \boldsymbol{Y}(t)^T. \tag{4}$$

This particular choice of $\boldsymbol{Y}^-(t)$ can be motivated by a formal least squares argument (see [4]), which is why it is called "least squares generalized inverse". Another choice for $\boldsymbol{Y}^-(t)$ (the "weighted least squares generalized inverse") can be found for example in [14], [7] and [10]. In this paper we will only use the "least squares generalized inverse".

Since $\boldsymbol{M}$ is a martingale, a natural estimator for $\boldsymbol{B}(t)$ is

$$\hat{\boldsymbol{B}}(t) := \int_0^t \boldsymbol{Y}^-(s) d\boldsymbol{N}(s) = \sum_{0 \leq s \leq t} \boldsymbol{Y}^-(s) \Delta \boldsymbol{N}(s), \tag{5}$$

where $\Delta \boldsymbol{N}(s) = \boldsymbol{N}(s) - \boldsymbol{N}(s-)$ denotes the jumps of $\boldsymbol{N}$ at time t. $\hat{\boldsymbol{B}}(t)$ is called Nelson-Aalen estimator (see [7]). Note that we are estimating a continuous function $\boldsymbol{B}$ by the right-continuous step-function $\hat{\boldsymbol{B}}$.

Under boundedness conditions on $\boldsymbol{Y}^-(t)$ and $\boldsymbol{\alpha}(t)$, $\hat{\boldsymbol{B}}(t) - \boldsymbol{B}(t)$ is a martingale and thus $E\hat{\boldsymbol{B}}(t) = \boldsymbol{B}(t)$. Furthermore, under certain conditions, $\sqrt{n}(\hat{\boldsymbol{B}} - \boldsymbol{B})$ converges to a Gaussian process as $n \to \infty$ (see [14], [10]). This implies that $\sqrt{n}(\hat{\boldsymbol{B}}(t) - \boldsymbol{B}(t))$ is asymptotically normally distributed for each t.

In order to use the result about asymptotic normality of $\sqrt{n}(\hat{\boldsymbol{B}}(t) - \boldsymbol{B}(t))$ to compute asymptotic confidence intervals we need an estimator for the covariance $\mathrm{Cov}(\hat{\boldsymbol{B}}(t) - \boldsymbol{B}(t))$. We will use $\hat{\boldsymbol{\Sigma}}(t)$ which is defined as follows.

$$\hat{\boldsymbol{\Sigma}}(t) := \int_0^t \boldsymbol{Y}^-(s)\mathrm{diag}(d\boldsymbol{N}(s))\boldsymbol{Y}^-(s)^T, \quad t \in \mathcal{T} \tag{6}$$

Hereby, $\mathrm{diag}(d\boldsymbol{N}(s))$ is a diagonal matrix with entries $dN_i(s), i \in \{1, \ldots, n\}$ and the above is to be interpreted as matrix product. More formally,

$$\hat{\boldsymbol{\Sigma}}_{ij}(t) = \sum_{\nu=1}^{n} \int_0^t Y^-_{i\nu}(s)Y^-_{j\nu}(s)dN_\nu(s).$$

Under certain regularity conditions on $\boldsymbol{Y}^-$ (e.g. assume it is bounded), $\hat{\boldsymbol{\Sigma}}(t)$ is an unbiased and consistent estimator for $\mathrm{Cov}(\hat{\boldsymbol{B}}(t) - \boldsymbol{B}(t))$ (see [12], [10]).

3.3 Smoothing the Estimator

Up to now, we developed an estimator for $\boldsymbol{B}(t) = \int_0^t \boldsymbol{\alpha}(s)ds$. What we are really interested in is estimating $\boldsymbol{\alpha}$ itself.

The estimator for $\boldsymbol{\alpha}$ which we will consider involves so called "kernels". A measurable, bounded function $K : \mathbb{R} \to \mathbb{R}_+$ is a *kernel*, if it vanishes outside $[-1, 1]$ and $\int_{-1}^1 K(t)dt = 1$. Three well-known kernels are the following. $K_U(t) = \frac{1}{2}1_{[-1,1]}(t)$, called the *uniform kernel*, $K_E(t) = \frac{3}{4}(1 - t^2)1_{[-1,1]}(t)$, called the *Epanechnikov kernel* and $K_B(t) = \frac{15}{16}(1 - t^2)^2 1_{[-1,1]}(t)$, called the *bigweight kernel.*

Let $b > 0$ and K a kernel. We will consider the following estimator for $\boldsymbol{\alpha}$.

$$\hat{\boldsymbol{\alpha}}(t) = \frac{1}{b}\int_{\mathcal{T}} K\left(\frac{t-s}{b}\right) d\hat{\boldsymbol{B}}(s), \quad t \in [b, \tau - b]. \tag{7}$$

Note that since K vanishes outside $[-1, 1]$, the integration is really only over $[t - b, t + b] \cap \mathcal{T}$. The parameter "b" is called bandwidth. Another way to write $\hat{\boldsymbol{\alpha}}$(t) is given by

$$\hat{\boldsymbol{\alpha}}(t) = \sum_{s \in \mathcal{T}} \frac{1}{b} K\left(\frac{t-s}{b}\right) \boldsymbol{Y}^-(s)\Delta\boldsymbol{N}(s).$$

For $t < b$ and $t > \tau - b$, adjustments on the estimator should be made to estimate $\boldsymbol{\alpha}(t)$. We will not deal with this here and refer to [7] for further discussion of this. We will call the problem arising here "boundary effect".

3.4 Martingale residuals

To assess goodness of fit one might want to look at the residual process $\boldsymbol{M}(t)$ given by (2), which is not observable. The heuristic calculation

$$d\boldsymbol{M}(t) = d\boldsymbol{N}(t) - \boldsymbol{Y}(t)\boldsymbol{\alpha}(t)dt \approx d\boldsymbol{N}(t) - \boldsymbol{Y}(t)d\hat{\boldsymbol{B}}(t) = d\boldsymbol{N}(t) - \boldsymbol{Y}(t)\boldsymbol{Y}^-(t)d\boldsymbol{N}(t)$$

gives rise to the estimator $\hat{\boldsymbol{M}}(t)$ which is given by

$$\hat{\boldsymbol{M}}(t) = \boldsymbol{N}(t) - \int_0^t \boldsymbol{Y}(s)\boldsymbol{Y}^-(s)d\boldsymbol{N}(s) = \sum_{0 \le s \le t} (\boldsymbol{I} - \boldsymbol{Y}(s)\boldsymbol{Y}^-(s))\Delta\boldsymbol{N}(s)$$

where $\boldsymbol{I}$ denotes the n-dimensional identity matrix.

$\hat{\boldsymbol{M}}(t)$ can be shown to be a martingale with $\hat{\boldsymbol{M}}(0) = 0$. Thus $\hat{\boldsymbol{M}}(t)$ should fluctuate around 0. Plotting $\hat{\boldsymbol{M}}(t)$ against t gives an impression of the goodness of fit of the model (see [6]).

Note that in the case $n = 1, k = 1$, we always have $\hat{M}(t) = 0$.

4 Datasets

The most widely used reference for software development datasets was published by Musa [16]. It describes 16 different software projects developed in the mid 1970s. It is, intentionally, a very heterogeneous dataset, so comparisons between projects in this dataset are difficult. The dataset is not really new, in a field as rapidly developing as software engineering, the mid 1970s can be considered “antique”. To the authors knowledge, no datasets comparable in size were published since, and smaller datasets published did not include useful covariates. This could be due to the proprietary nature of software development, almost no company likes to publish how many failures its software produced.

For our approach datasets found in the literature were not sufficient so we chose a different path. In recent years, open source software has received much attention. Its main feature is that the source code and not only the compiled program is available. Prominent examples are the Linux operating system, the web-server Apache and the desktop environments GNOME and KDE. Many developers are volunteers, distributed around the globe (companies support some projects, though). Since the participants of theses projects cannot meet physically, every aspect of development uses the Internet. Development does not adhere to the “waterfall” model described earlier. It is constantly going on and everybody can access the newest version. In the language of Musa [17] this is called “evolving software”.

To be able to control who is allowed to change the code, sophisticated tools are being employed. One of the most popular is called CVS, which stands for “Concurrent Version System”. For our purpose it is important that CVS allows to retrieve projects as they were at any given date and that we can observe changes made in a certain period. This way we can get the size of projects during our observation period. Quantities derived from this will be used as covariates. For more information on CVS we refer to [9] and the CVS home page [2].

Many projects also use bug (defect) tracking systems that allow everybody to submit bug reports and enable developers to process them. A sophisticated and popular example for such a system is called Bugzilla. It allows classification of bugs by various criteria such as severity, status and resolution. Furthermore, it contains a powerful query tool to search for bug reports in a given time interval satisfying certain criteria. We will use this query tool to obtain the failure data needed. For more on Bugzilla, we refer to its home page [1].

We want to elaborate some more about the specific dataset we will analyze. It is based on severals programs which are part of the GNOME desktop environment [3]. The advantage is that all programs considered are stored in one CVS database and use the same Bugzilla bug tracking system. We wrote scripts and programs in Perl and C++ to obtain and process the data.

We exclude some bug reports from our study in order to enhance the quality of the datasets. We only use the most severe reports (“blocker” and “critical”) and do not include “unconfirmed” reports. Furthermore bug reports marked as “invalid”, as a “duplicate”, as not being a bug (“notabug”) or as not pertaining to GNOME (“notgnome”) are excluded as well. For example, not allowing duplicate reports for a bug is reasonable, since we do not want to count the same failure twice and since people making bug reports are encouraged not to report bugs that had already been reported (but they do not always do so).

Concerning the size of projects, we considered two possibilities. The first is to count the number of lines contained in the entire project directory (for the i-th project, at time t this number divided by 1000 will be denoted by $P_i(t)$). This includes many files that do not contain source code such as change logs, manuals, documentation or to-do lists. The second possibility is to distinguish between source code files and other files. Since the projects we consider are using the C-programming language, we took files ending with '.c', '.h' and makefiles as approximation for the source code files. We denote the number of lines (divided by 1000) contained in theses files in project i at time t by $S_i(t)$. To get the number of lines in a certain file at a certain time, we started with the number of lines it contained at the beginning of the observation period and added the lines inserted since then. Deleted lines were not counted. The reasoning behind this is as follows. If subtracting deleted lines, then changing one line does not change our covariates, since CVS reports in this case that one line was added and one removed. We want to avoid this. For fixed t, $(P_i(t))$ and $(S_i(t))$ are highly correlated (> 0.9). Changes in $(P_i(t))$ and $(S_i(t))$ (i.e. for some t and ν, $(P_i(t) - P_i(t-\nu))$ and $(S_i(t) - S_i(t-\nu))$) are less correlated. From now on we only work with $S_i(t)$. The advantage of using $S_i(t)$ is that in our model $\alpha(t)$ can be interpreted as failures per thousand lines of code per year.

Our method to obtain the failure data is similar do [15]. In that paper the entire size of the project directory is used. Concerning software reliability, only the number of failures per line is measured and no other software reliability model is considered.

For the present application, we take 73 projects which are part of the GNOME desktop environment. For these projects, data from CVS and Bugzilla could be easily matched.

Our observation period is March 1st, 2001 up to October 1st, 2002. As unit for our measurements we have chosen years.

5 Results

5.1 Total Size as Covariate

We consider the size of the source code (in thousand lines of code) as the only covariate (i.e. $k = 1$), i.e.

$$Y_{i1}(t) = S_i(t).$$

In Figure 1, the least squares estimator $\hat{B}(t)$ can be seen. We included an asymptotic pointwise confidence interval to the level 95%.

The smoothed estimator $\hat{\alpha}(t)$ can be seen in Figure 2, where the Epanechnikov kernel was used together with a bandwidth of $b = 60$ days. The vertical lines indicate the first and last 60 days, during which boundary effects appear.

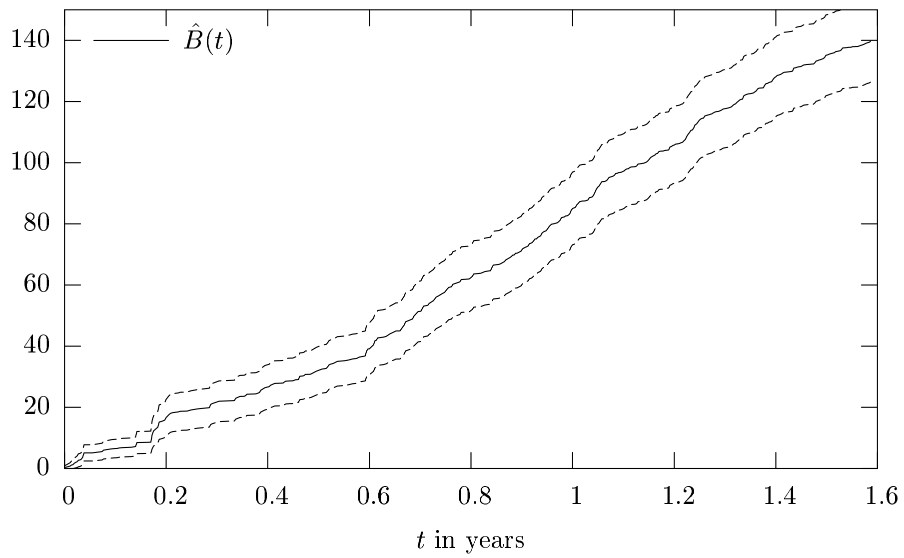

Figure 1: $k = 1$, $Y_{i1}(t) = S_1(t)$

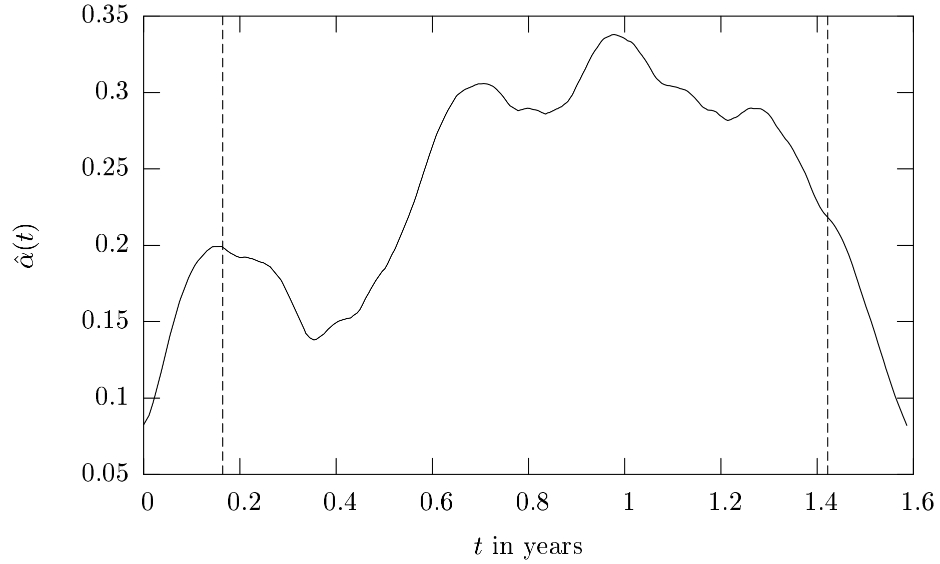

Figure 2: $k = 1$, $Y_{i1}(t) = S_i(t)$, Epanechnikov kernel, $b = 60$ days

5.2 Old and new Code

Now we consider two covariates ($k = 2$), which represent "old code" and "new code". More precisely, with $\nu := 30$ days,

$$\begin{aligned} Y_{i1}(t) &:= S_i(t - \nu), \\ Y_{i2}(t) &:= S_i(t) - S_i(t - \nu). \end{aligned}$$

In order to have the necessary covariates available, our plots start $\nu = 30$ days later, i.e. $t = 0$ is March 31st, 2001.

Figure 3 contains the smoothed estimators $\hat{\alpha}_1(t)$ and $\hat{\alpha}_2(t)$.

For $b < t < 1$ year, $\hat{\alpha}_2(t) > \hat{\alpha}_1(t)$, meaning that during that time "old code" contains less bugs than "new code" After that the relation changes. This could be because during that time a new release of GNOME was prepared (which was released at the end of June 2002, which corresponds to $t = 1.2$ years). Before the release, development of new features was restricted, the main focus was to get the different projects together into one reliable, stable package. This may explain why the newly added code during that period was less responsible for the failures.

The variation of $\hat{\alpha}_2(t)$ is bigger than the variation of $\hat{\alpha}_1(t)$. This can be explained by the greater variation in the covariates (the amount of source code added in the last ν days varies stronger than the amount of source code before ν days).

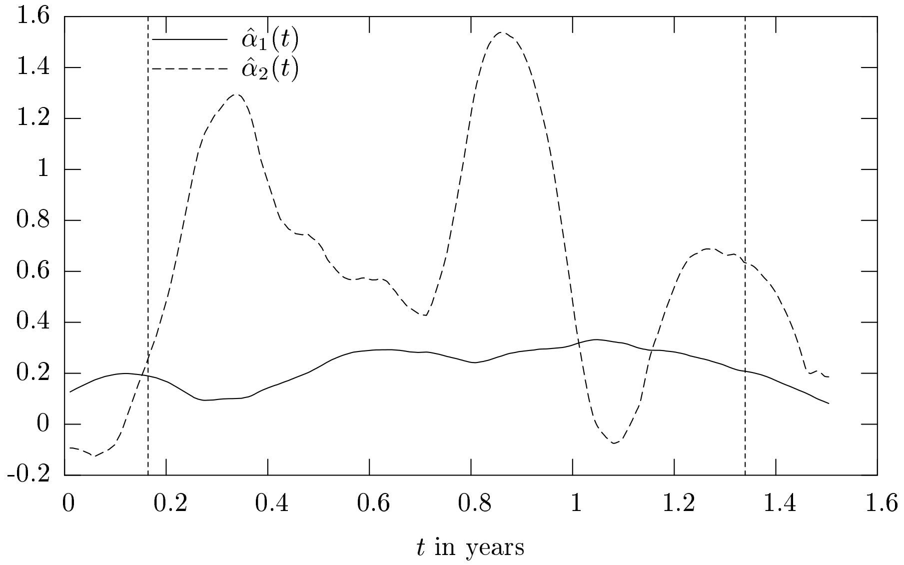

Figure 3: $k = 2$, $\nu = 30$ days, $Y_{i1}(t) = S_i(t - \nu)$,$Y_{i2}(t) = S_i(t) - S_i(t - \nu)$, Epanechnikov kernel, $b = 60$ days

To assess the goodness of fit, Figure 4 shows the martingale residuals of the alphabetically first 30 projects.

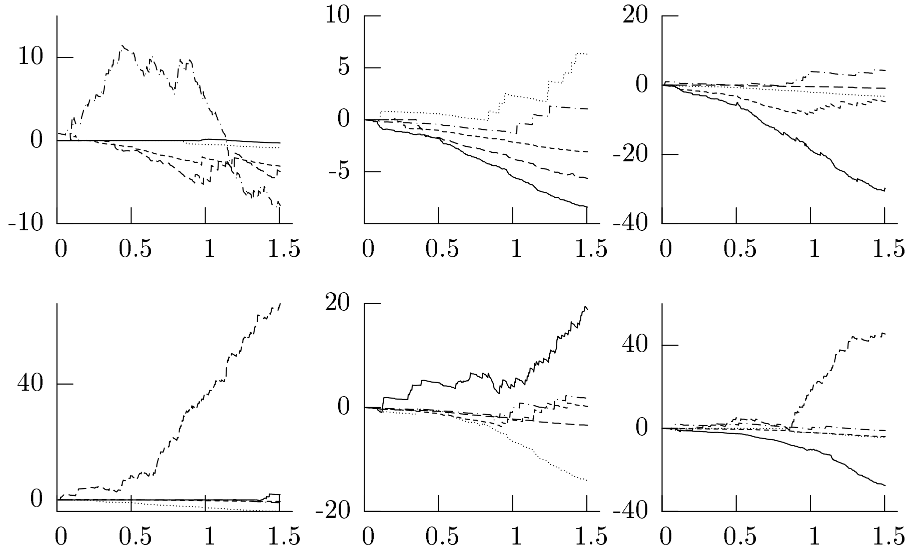

Figure 4: Some residuals $\hat{M}_i(t)$ for $k = 2$, $\nu = 30$ days, $Y_{i1}(t) = S_i(t-\nu)$,$Y_{i2}(t) = S_i(t) - S_i(t-\nu)$

5.3 Three Covariates

To improve the fit of the model we included yet another covariate, the number of recent failures. That is we choose $k = 3$, $\nu = 30$ days,

$$\begin{aligned} Y_{i1}(t) &= S_i(t-\nu) \\ Y_{i2}(t) &= S_i(t) - S_i(t-\nu) \\ Y_{i3}(t) &= N_i(t-) - N_i(t-\nu). \end{aligned}$$

Once again, $t = 0$ corresponds to March 31st, 2001.

In Figures 5 and 6 the smoothed estimators $\hat{\alpha}_1(t)$, $\hat{\alpha}_2(t)$ and $\hat{\alpha}_3(t)$ are displayed. Once again the Epanechnikov kernel was used together with a bandwidth of $b = 60$ days. When comparing Figures 5 and 3, we can observe that the same relation between $\alpha_1(t)$ and $\alpha_2(t)$ seems to hold in both cases.

The martingale residuals for the alphabetically first 30 projects of this model can be seen in Figure 7. Comparing this to Figure 4 (which is possible since the graphs in the two figures pertain to the same projects), the 3 covariates seem to indicate a better fit of the model (note the different scales).

In the model presented the intensity is additively separated into parts which can be attributed to the different covariates. From the plots thus far it cannot be determined how big these parts are. To get an impression of this we sum, over all projects, an estimate of these parts, i.e.

$$\hat{\rho}_j(t) := \hat{\alpha}_j(t) \sum_{i=1}^{n} Y_{ij}(t), \quad j = 1, 2, 3.$$

A plot of this can be seen in Figure 8. The third covariate (recent failures) seems to have a dominating effect on the total intensity.

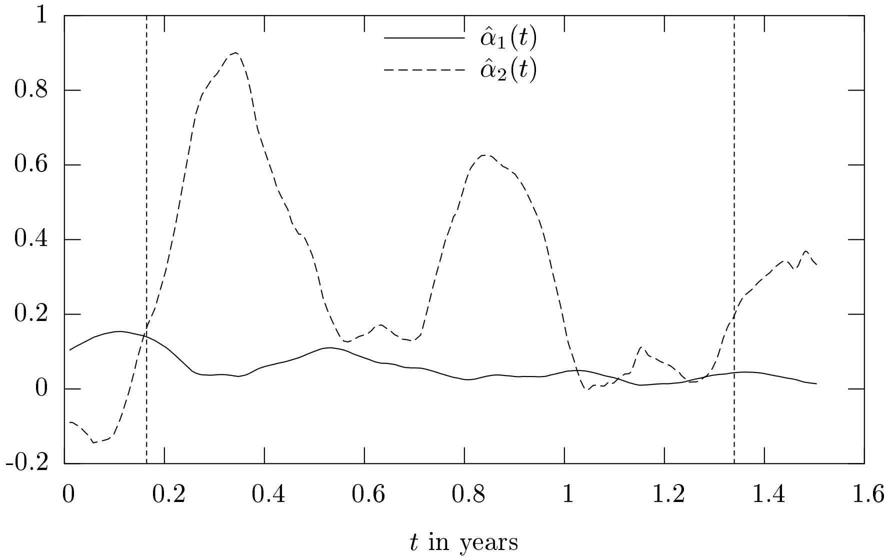

Figure 5: $k = 3$, $b = 60$ days, Epanechnikov kernel

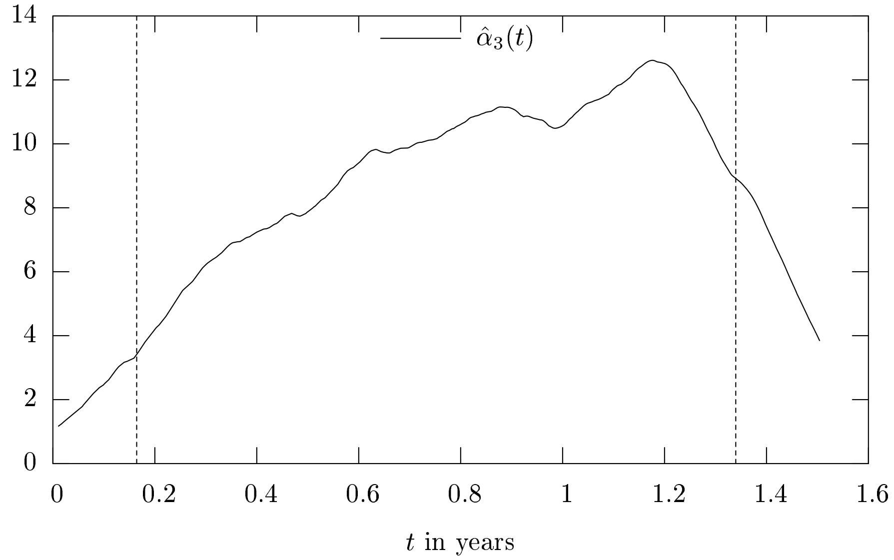

Figure 6: $k = 3$, $b = 60$ days, Epanechnikov kernel

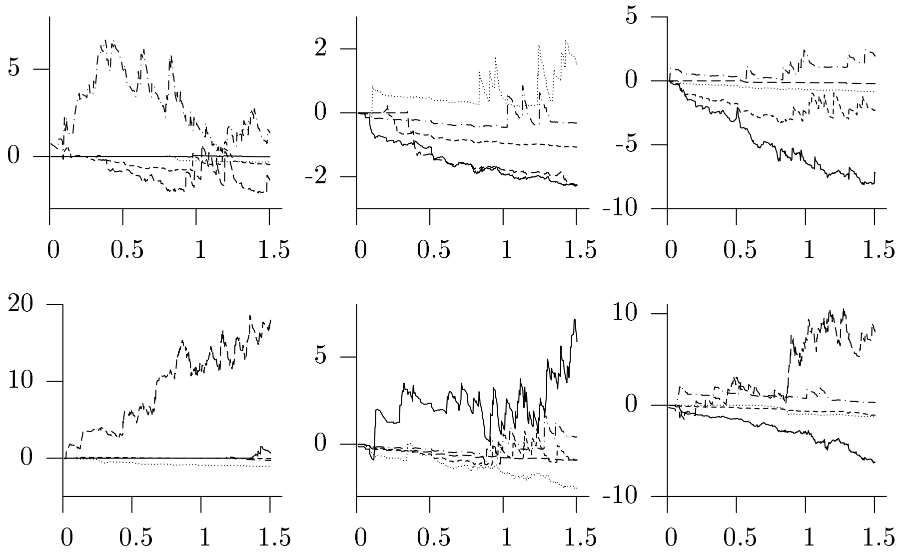

Figure 7: Some residuals $\hat{M}_i(t)$ for $k = 3$

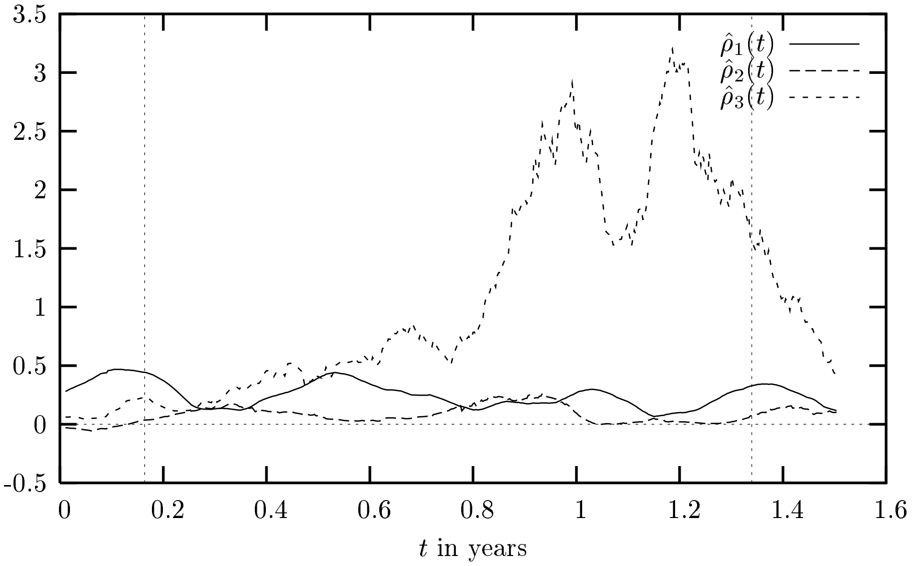

Figure 8: $k = 3$, $b = 60$, estimated additive parts of the total intensity

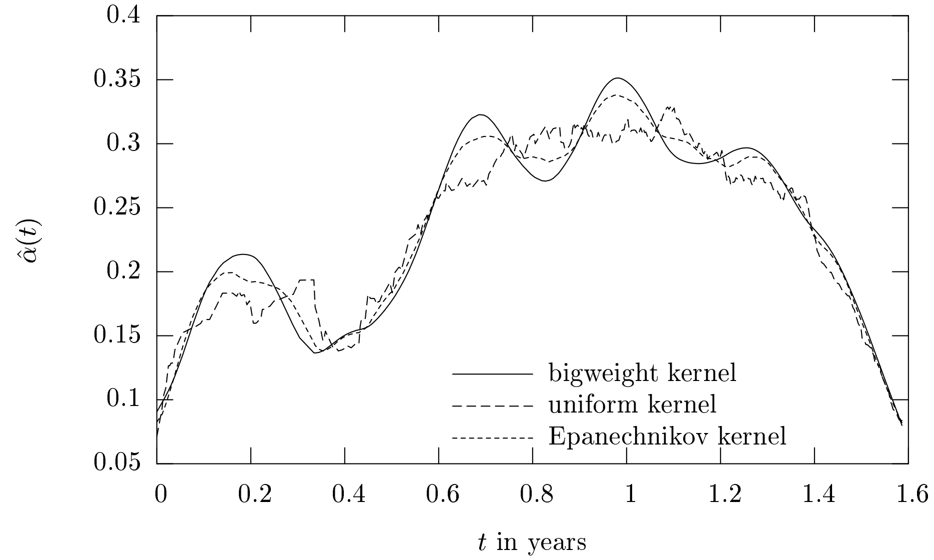

Figure 9: $k = 1$, $Y_{i1}(t) = S_1(t)$, $b = 60$ days

5.4 Effects of Bandwidth and Kernel

We return to our first choice of covariates, where we used as single covariate the size of the source code of the respective projects.

What happens if instead of the Epanechnikov kernel, we employ different kernels? The effects of using the the bigweight kernel or the uniform kernel on $\hat{\alpha}(t)$ can be seen in Figure 9.

As bandwidth we always used $b = 60$ days. In Figure 10 it can be seen that - as to be expected - a higher bandwidth yields smoother graphs of $\hat{\alpha}(t)$.

6 Outlook

In this section we want to mention some different ways we could have used (and which we may explore in the future).

The first thing, we want to mention, is our handling of lines of code deleted during development. We chose to ignore them. Instead, we could subtract them from our covariates. This does not strongly affect the results.

Using the size of the entire project directory $P_i(t)$ instead of $S_i(t)$ does not lead to very different results. Other software metrics besides size could be used as covariates. Examples are Halstead's software metric or McCabe's cyclomatic complexity metric (for a short review see e.g. [18]).

Other open source projects use the same tools (CVS, Bugzilla) as GNOME does. So it is possible to obtain data from these projects and make comparisons.

In our opinion, there is no reason why nonparametric methods should not be used in traditional software development as well. The only requirement is the availability of sufficiently large datasets which is, at least in the publicly available literature, not the case thus far. But inside one (big) company, data should be available and nonparametric methods could be applied. Our approach could for example be useful to compare different programming paradigms.

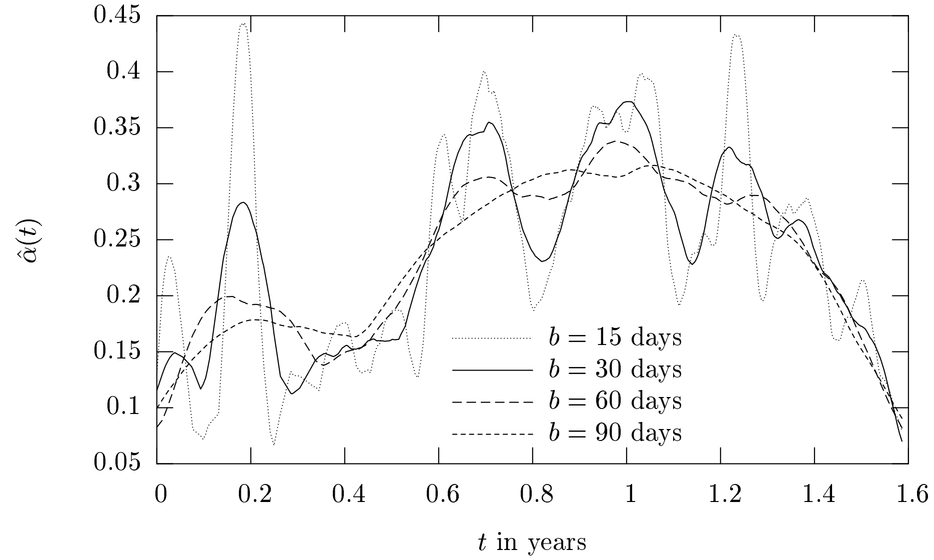

Figure 10: $k = 1$, $Y_{i1}(t) = S_1(t)$, Epanechnikov kernel

The last point is that other nonparametric models incorporating covariates could, of course, be used as well. We chose these models as flexible, relatively easily usable examples.

Acknowledgements

Financial support of this research by the Deutsche Forschungsgemeinschaft through the interdisciplinary research unit (Forschergruppe) 460 is gratefully acknowledged.

References

[1] Bugzilla project. http://www.bugzilla.org [27 Januray 2003].

[2] CVS home. http://www.cvshome.org [27 Januray 2003].

[3] GNOME project. http://www.gnome.org [27 Januray 2003].

[4] Odd Aalen. A model for nonparametric regression analysis of counting processes. In *Mathematical Statistics and Probability Theory - Proceedings, Sixth International Conference, Wisla (Poland)*, volume 2 of *Lecture Notes in Statistics*, pages 1–25. Springer-Verlag, New York, 1980.

[5] Odd O. Aalen. A linear regression model for the analysis of life times. *Statistic in Medicine*, 8:907–925, 1989.

[6] Odd O. Aalen. Further results on the non-parametric linear regression model in survival analysis. *Statistics in Medicine*, 12:1569–1588, 1993.

[7] Per Kragh Andersen, Ørnulf Borgan, Richard D. Gill, and Niels Keiding. *Statistical Models Based on Counting Processes*. Springer-Verlag, New York, 1993.

[8] May Barghout, Bev Littlewood, and Abdallah A. Abdel-Ghaly. A non-parametric order statistics software reliability model. *Software Testing, Verification&Reliability*, 8(3):113–132, 1998.

[9] Per Cederqvist. *Version Management With CVS*. Available at http://www.cvshome.org [27 Januray 2003].

[10] Axel Gandy. A nonparametric additive risk model with applications in software reliability. Diplomarbeit, Universität Ulm, 2002.

[11] Goel and Okumoto. Time-dependent error-detection rate model for software reliability and other performance measures. *IEEE Transactions on Reliability*, R-28(3):206–211, 1979.

[12] Fred W. Huffer and Ian W. McKeague. Weighted least squares estimation of Aalen's additive risk model. *Journal of the American Statistical Association*, 86(413):114–129, march 1991.

[13] Z. Jelinski and P. Moranda. Software reliability research. In W. Freiberger, editor, *Statistical Computer Performance Evaluation*. Academic Press, New York, 1972.

[14] Ian W. McKeague. Asymptotic theory for weighted least squares estimators in Aalen's additive risk model. *Contemporary Mathematics*, 80:139–152, 1988.

[15] Audris Mockus, Roy T Fielding, and James D Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(3):309–346, 2002.

[16] John D. Musa. Software reliability data. Technical report, Data & Analysis Center for Software, January 1980. http://www.dacs.dtic.mil/databases/sled/swrel.shtml [27 Januray 2003].

[17] John D. Musa, Anthony Iannino, and Kazuhira Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill, 1987.

[18] Hoang Pham. *Software Reliability*. Springer-Verlag, Singapore, 2000.

[19] Nozer D. Singpurwalla and Simon P. Wilson. *Statistical Methods in Software Engineering*. Springer Series in Statistics. Springer-Verlag, New York, 1999.

[20] Mark C. van Pul. A general introduction to software reliability. *CWI Quarterly*, 7(3):203–244, 1994.