Replicating and Re-evaluating the Theory of Relative Defect-Proneness

Mark D. Syer, *Member, IEEE,* Meiyappan Nagappan, *Member, IEEE,* Bram Adams, *Member, IEEE,* and Ahmed E. Hassan, *Member, IEEE*

Abstract—A good understanding of the factors impacting defects in software systems is essential for software practitioners, because it helps them prioritize quality improvement efforts (e.g., testing and code reviews). Defect prediction models are typically built using classification or regression analysis on product and/or process metrics collected at a single point in time (e.g., a release date). However, current defect prediction models only predict *if* a defect will occur, but not *when*, which makes the prioritization of software quality improvements efforts difficult. To address this problem, Koru et al. applied survival analysis techniques to a large number of software systems to study how size (i.e., lines of code) influences the probability that a source code module (e.g., class or file) will experience a defect at any given time. Given that 1) the work of Koru et al. has been instrumental to our understanding of the size-defect relationship, 2) the use of survival analysis in the context of defect modelling has not been well studied and 3) replication studies are an important component of balanced scholarly debate, we present a replication study of the work by Koru et al. In particular, we present the details necessary to use survival analysis in the context of defect modelling (such details were missing from the original paper by Koru et al.). We also explore how differences between the traditional domains of survival analysis (i.e., medicine and epidemiology) and defect modelling impact our understanding of the size-defect relationship. Practitioners and researchers considering the use of survival analysis should be aware of the implications of our findings.

Index Terms—Survival Analysis; Cox Models; Defect Modelling

1 Introduction

Modelling defects in software systems is essential for software maintenance and quality assurance. Practitioners must understand which metrics are good indicators of software defects to best allocate their limited resources to the most defect-prone source code modules (e.g., classes or files) in their systems [1], [2]. Existing defect modelling techniques typically use classification or regression analysis on product (e.g., the number of lines of code) and/or process (e.g., code churn) metrics associated with source code modules [3], [4].

Most modelling techniques collect their metrics at a single point in time, such as a major release, then model which source code modules are most likely to experience a defect in the near future. In doing so, these models ignore the aspect of time. Since the defect-proneness of modules changes over time as releases come and go, and requirements or features change, models get outdated relatively soon (i.e., "concept drift" [5], [6]) and must be re-built. Further, the models can only give a probability of the occurrence of a defect in the next period, they fail to model how much time it will take before a defect occurs. Yet, such information is critical for practitioners to schedule their quality assurance efforts.

The modelling of defects in source code modules can be formulated as a time-to-event problem. Taken from the field of medicine and epidemiology, time-to-event analysis or *survival* analysis aims to determine 1) what factors (i.e., covariates) affect the time-to-event and 2) who or what (i.e., subjects) will experience an event given an interval of time. At the heart of survival analysis are two interrelated functions. The *survival function* describes the probability that the subject will not experience an event before time t (i.e., the probability that the subject will *survive* at least until time t). The *hazard function* describes the instantaneous event occurrence rate, the hazard rate, at time t (i.e., the number of events per unit of time at time t).

Traditionally, survival analysis is used to determine how covariates (e.g., age, white blood cell count and frequency of treatment) affect the length of time before a medical condition is either contracted (e.g., the start of flu season to contracting the flu) or cured (e.g., contracting the flu to being cured).

Koru et al. formulated the modelling of defects in source code modules as a time-to-event problem. The authors were amongst the first empirical software engineering researchers to use survival analysis for defect modelling [7]–[9]. Koru et al. used survival analysis techniques to study how size (i.e., lines of code) influences the probability that a source code module will experience a defect at any given time. The authors found that the hazard rate increases at a slower rate than module size. This indicates that larger modules proportionally are less defect-prone (i.e., the number of defects per line of code is higher in smaller modules).

Although survival analysis has shown promising results for modelling defects, care should be taken when transferring approaches and ideas from different fields to software engineering. Major differences exist between the traditional domains of survival analysis (i.e., medicine and epidemiology), and defect modelling. We consider two of these differences.

First, defect fix data (i.e., data describing when defects are fixed) is much easier to obtain than defect introduction data (i.e., data describing when defects are introduced), therefore, software practitioners currently build models that predict when a defect will be fixed (being cured of the flu) instead of when a defect will be introduced (contracting the flu). However, in order to best prioritize quality improvement efforts, software practitioners need to understand when defects are introduced. In traditional defect models, defect fix data is a good approximation for defect introduction data because all time information is collapsed when building the model for a particular point in time (i.e., the time difference between defect introduction and fix becomes almost irrelevant). On the other hand, survival analysis explicitly takes time information into account, making it likely that the approximation of defect introduction data by defect fix data no longer holds.

Second, events tend to be modelled along a continuous time scale (i.e., defects can be fixed/introduced at any time). In practice, such events can only occur along a discrete time scale (i.e., defects can only be fixed/introduced when a revision is made). Survival analysis experts recommend a discrete time scale be used when observations can only be made at specific points in time [10], [11]. Discrete time models have many advantages. For example, they allow multiple events to occur simultaneously (i.e., multiple defects can be introduced/fixed in a revision), whereas continuous time models assume that only one event can occur at a given point in time.

The work of Koru et al. has been instrumental in the community's recent understanding of the relationship between size and defects [7]–[9]. Therefore, in this paper, we replicate the work of Koru et al. [8]. We also extend this work by examining the impact of the two important differences between the traditional domains of survival analysis and defect modelling.

This paper makes three contributions:

1) We replicate the results of Koru et al. [8] and provide details missing from the original paper.
2) We demonstrate the impact of modelling defect introduction (as opposed to defect fix) events along a discrete (as opposed to continuous) time-scale.
3) We provide a clear outline of how to use survival analysis for defect modelling such that other researchers can benefit from our experiences.

The paper is organized as follows: Section 2 presents previous work in defect modelling and provides an overview of survival analysis. Section 3 presents our replication of the original study by Koru et al. [8]. Section 3.4 builds on our replication study using larger projects and the diagnostics required for the proper application of survival analysis. Section 3.7 presents the results of our analysis using our new data formulation (i.e., defect introducing events along a discrete time-scale). Section 4 presents a discussion of our results. Section 5 outlines the threats to the validity of our work. Finally, Section 6 concludes the paper.

2 Background and Related Work

2.1 Defect Modelling

Defect modelling has received substantial attention from the empirical software engineering community [6], [12]. Researchers have studied the impact that product (e.g., number of lines of code and code complexity) [13]–[16], process (e.g., code churn) [15]–[17] and social (e.g., code ownership and developer experience) [18], [19] metrics have on defects in source code modules. Existing methods have used regression, data mining and machine learning techniques to model defects.

The importance of size (i.e., lines of code) in understanding software quality has been acknowledged by many researchers. Size has consistently been found to be one of the most important metrics when modelling defects in source code modules [13]–[15], [17]. However, there has not been a consensus on the functional form (i.e., the exact mathematical form) of the size-defect relationship.

Studies of defect density, a metric designed to account for the size of a module by dividing the number of defects by the number of lines of code, have typically shown a "U" shaped relationship between defect density and size [20]–[23]. The "U" shaped relationship between defect density and size indicates that defect density has a minimum value in medium-sized modules and is higher in both smaller and larges modules. These conclusions are generally referred to as the Goldilocks principle, where the ideal module size is "not too small or not too large" [24]. Software practitioners were recommended to produce source code modules in this ideal size range.

However, researchers have been critical of the Goldilocks principle and the defect density approach to studying the size-defect relationship [24]–[26]. These researchers claim that defect density masks the true size-defect relationship, resulting in artificial correlations and misleading conclusions. This claim is based on the notion that defect density is artificially high in smaller modules because the denominator (i.e., size) is small. These researchers concluded that the Goldilocks principle is an artifact produced by the analysis and not a result of the size-defect relationship.

Koru et al. identified gaps in the existing literature and applied survival analysis techniques to a large number of closed- and open-source software systems to study the size-defect relationship [7]–[9]. The authors use Cox Proportional Hazards (Cox) models, one of the most popular models in survival analysis, to determine that there is a power-law relationship between size and the number of defects and that smaller source code modules are proportionally more defect-prone than larger modules. These findings contradict the previous work that showed that defect density has a "U" shape (i.e., defect density has a minimum value in medium-sized modules). The work of Koru et al., the "Theory of relative defect proneness" [8] in particular, is the focus of our replication.

Wedel et al. also demonstrated how survival analysis techniques can be applied to enhance existing defect prediction techniques [27]. The authors found the same size-defect relationship as Koru et al. in Eclipse. However, Wedel et al., similar to Koru et al., failed to properly verify the underlying assumption of the Cox Proportional Hazards model (discussed in the Section 2.2.2). Further, Wedel et al. used simulated data by assuming that defects occur uniformly over time, as opposed to determining the actual timing of events. When formulating the modelling of defects in source code modules as a time-to-event problem, the actual timing of events is necessary.

Gehan et al. use survival models to study the defect-proneness of methods in two open-source projects using code predictors (e.g. lines of code) and clone predictors (e.g., number of clone siblings) to determine the impact of code clones on software defects [28]. However, the authors limited their analysis to source code files with file sizes within a particular range where the underlying assumptions of their Cox models were satisfied [28].

2.2 Survival Analysis

Survival analysis consists of a wide range of models and techniques for modelling the time-to-event. These techniques vary in their underlying statistical framework (i.e., parametric, semi-parametric and non-parametric models) and model of event occurrences (i.e., terminating and recurring events). However, these techniques share the same aims: 1) to model the time between a start event and another event of interest (i.e., the "survival time") and 2) to model the factors that affect this survival time. The following two sections discuss: 1) the data required for survival analysis and 2) one of the most popular models in survival analysis (i.e., the Cox Proportional Hazards model).

2.2.1 Survival Data and the Counting Process Format

The data required for survival analysis is composed of one or more observations for each subject in the study. Each observation describes the state of a subject during a particular time period that ends with an event occurrence. For example, an observation may describe the health (state) of a patient (subject) between patient exams (event occurrences). The state of a subject is described by one or more covariates. Covariates are variables that are collected at each event occurrence and may or may not have predictive power over the time to the event (the predictive power of these variables is often the focus of a study involving survival analysis). Depending on the study, there may be multiple observations over successive time periods for each subject. For example, one observation per annual patient exam. Each observation must include the following fields:

1) ID – a unique identifier for each subject (e.g., patients) in the study.
2) Start – the time of the start event/the start time of the observation period.
3) End – the time of the end event/the end time of the observation period.
4) Event – an indicator for whether this observation period ends with an event (e.g., whether the patient is alive at the End time).
5) Covariate(s) – one or more covariates that describe the state of the subject (e.g., white blood cell count) at the time of the start event/the start time of the observation period.

The above data format is the Counting Process Format. This format can easily accommodate multiple events of the same or differing types, time-dependent covariates and discontinuous observation periods.

There are two key choices that must be made when constructing survival data. First, we must define the events. Second, we must define how time is measured between the events.

Three types of events exist within survival data: starting events, terminating/recurring events and censoring events. These events are defined as:

1) Starting event – the first observation of a subject.
2) Terminating/Recurrent event – the event of interest (possibly occurring more than once if the event is recurrent).
3) Censoring event – an event that is not the event of interest.

The typical application of survival analysis attempts to model subjects who may experience a single terminating event, such as the time from a patient's diagnosis to his/her death. For example, in a clinical trial of a new drug designed to prevent fatal heart attacks, clinicians would follow up with each patient in the study to determine how they died. Some patients may suffer a fatal heart attack (i.e., the event) and some will die from other causes. However, it is impractical to continue the study until every patient has died. Therefore, the study will end after some time (e.g., five years). During the study some patients will die from causes other than a heart attack, some patients will withdraw from the study and some patients will still be alive at the end of the study. Despite the fact that we do not have the time-to-event (i.e., the time from the start of the study until a fatal heart attack) for these patients, their information is still useful as it provides a lower bound for their survival time. This partial information is called "censored" data and the last date for which we have information on these patients (e.g., the date the patient withdraws from the study) is called a "censored" event.

Unlike many applications of survival analysis that attempt to predict the time to biological death, which is a terminating event, survival analysis in the context of defect modelling typically needs to account for recurring events. For example, when predicting defects in source

code files, multiple bugs may be introduced into the file at multiple points in time. Defects are not necessarily fatal, since other defects may be introduced later. Survival models have been extended with counting process theory to model subjects with recurrent events [29].

After we have defined the starting, terminating/recurring and censoring events, we must define how the time between events is measured. Event occurrences may occur on one of three time scales:

1) Continuous – the event may occur at any time and the "exact" time of the event is known (e.g., the time of death).
2) Discrete – the event may occur at any time but the exact time is not known (e.g., contraction of a medical condition between patient exams).
3) Intrinsically discrete – the event may only occur at certain points in time (e.g., transmission of a genetic condition at birth).

Careful thought is required when selecting a model for survival events (i.e., terminating versus recurrent), the specification of the event types (i.e., start, censored and terminating/recurring) and the measurement of time between events (i.e., continuous, discrete or intrinsically discrete).

2.2.2 Cox Proportional Hazard Models

One of the most popular models for survival analysis is the Cox Proportional Hazards (Cox) model. It is a semi-parametric model, i.e., the model has two components: 1) a nonparametric baseline function and 2) a parametric function. Cox models assume that the hazard function (i.e., the instantaneous event occurrence rate at some time t for a particular subject i) has the following form:

$$\lambda_i(t) = \lambda_0(t) \times \exp(X(t) \times \beta) \tag{1}$$

where λ_0 is some unspecified baseline hazard function that describes the instantaneous risk of experiencing an event at some time, t, when the values of all covariates are zero. $X(t)$ is a vector of possibly time-varying covariates that are collected at each event occurrence that may or may not have predictive power over the time to the event. β is a vector of regression coefficients (i.e., one coefficient for each covariate).

From Equation 1, we can determine that the relative hazard between two subjects, i and j, depends only on their covariate values. That is to say:

$$\frac{\lambda_i(t)}{\lambda_j(t)} = \frac{\exp(X_i(t) \times \beta)}{\exp(X_j(t) \times \beta)} \tag{2}$$

$$= \exp(((X_i(t) - X_j(t)) \times \beta) \tag{3}$$

We can rewrite Equation 2 as the log-relative hazard:

$$\log(\frac{\lambda_i(t)}{\lambda_j(t)}) = ((X_i(t) - X_j(t)) \times \beta \tag{4}$$

Equation 4 is called the Cox Proportional Hazards assumption because the Cox model assumes that the log-relative hazard between two subjects is linearly dependent on the difference between their covariate values and holds for all time. If a covariate tends to violate this assumption, then the covariate needs to be transformed using a link function to satisfy the assumption. A link function, $f(X(t))$, transforms Equation 4 so that the following relation will hold:

$$\log(\frac{\lambda_i(t)}{\lambda_j(t)}) = (f((X_i(t)) - f(X_j(t))) \times \beta \tag{5}$$

In the event of multiple covariates, then $f(X(t))$ becomes a vector of link functions (i.e., one link function for each covariate). When a link function is not needed then $f(X(t)) = X(t)$. A commonly used link function is the natural logarithm. For defect modelling, Koru et al. [8] used the natural logarithm as the link function for the "number of lines of code" covariate.

Link functions are useful when a covariate of interest violates the Cox Proportional Hazards assumption. However, if the covariate is not of interest, but it is a source of nonproportionality, we can *stratify* our models across these factors. In a stratified Cox model, covariates that are not of interest and may have nonproportional effects are included in the baseline hazard function as opposed to being used as a covariate. Stratifying a Cox model by a covariate removes the nonproportional effects the covariate had on the model without adding an additional coefficient to the model. As a result, the Cox model actually has multiple baseline hazards (i.e., one for each level of stratification).

3 The Replication Study

In the next three sections we present our three research questions.

> *RQ1:* Can we replicate the results in the original study by Koru et al?
>
> *RQ2:* Can we generalize the approach of Koru et al. to additional software projects?
>
> *RQ3:* What is the impact of using a different data formulation?

Figure 1 provides a graphical overview of the process for building and verifying Cox models that is used in each of our research questions. We will examine different aspects of this process in each of our research questions.

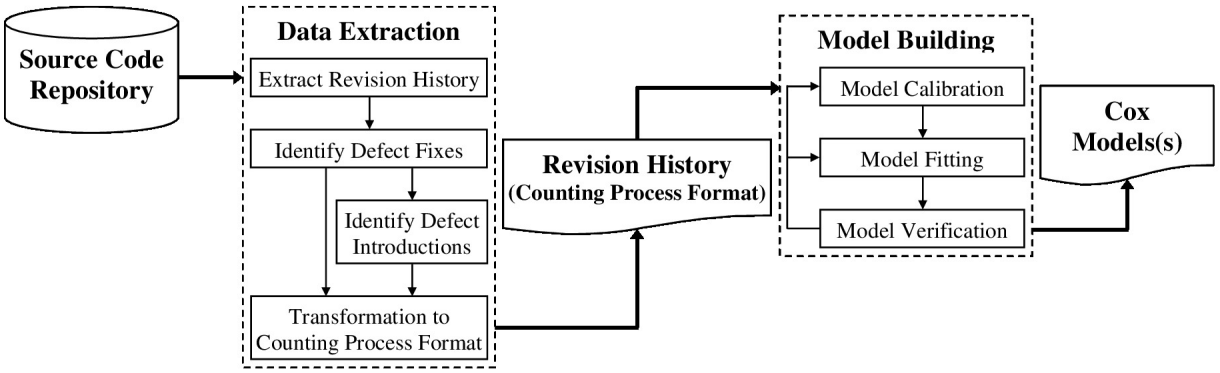


Fig. 1: Overview of Event Modelling With Cox Models.

RQ1: Can we replicate the results in the original study by Koru et al?

3.1 Motivation

Module size (i.e., lines of code) is one of the most important metrics used by software practitioners in defect modelling. Researchers have found that module size is one of the best predictors of defect-proneness, where larger source code modules typically have a higher number of defects. However, the functional form of the relationship between module size and defect-proneness is not well understood [24], [25], [30].

In an earlier study of defect-proneness in the Mozilla project, Koru et al. found that a one unit increase in the natural logarithm of size led to a 44% increase in the rate of defect fixes [7]. These findings suggest that defect-proneness monotonically increases with module size, but at a slower rate. Consequently, smaller modules are proportionally more defect-prone (i.e., the number of defects per line of code is higher in smaller source code modules). However, the exact functional form of this relationship was not uncovered.

Building upon their earlier results and observations [7], Koru et al. investigated the functional form of the relationship between module size and defect-proneness [8]. In particular, the authors tested their hypothesis that smaller modules are proportionally more defect prone. Koru et al. collected the history of each class in ten open-source software projects. The authors fit a Cox model to each project to model the relationship between module size and the time-to-defect. Finally, the authors extracted the relationship between module size and defect-proneness from each of the Cox models.

Koru et al. performed their analysis in R, a software environment for statistical computing, using two standard R packages [31]. The majority of their analysis was performed using the Design package [32] (now known as the rms package). However, the survival package had to be used to obtain robust error estimates of the Cox model coefficients ($\hat{\beta}$) [33].

Many of the implementation details necessary for replicating or extending the work of Koru et al. was missing from the original paper. Therefore, based upon the methodology and results presented in the original paper [8] and our understanding of survival analysis and defect modelling, we recovered the implementation details that were missing from the original paper. We use these implementation details, combined with the functionality provided by the survival analysis packages referenced by Koru et al., to reproduce the R scripts used by Koru et al. We then use these R scripts to reproduce the tables and figures from the original paper (i.e., to verify that we have correctly recovered the missing implementation details), which are reported in the remainder of this research question. Finally, we provide a replication package in the appendix so that other researchers may be able to use these techniques.

This research question allows the remainder of our replication study to build upon a common foundation with the original study by Koru et al.

3.2 Approach

Based upon the methodology and results presented in the original paper by Koru et al. [8] and our understanding of survival analysis and defect modelling, we recovered the implementation details that were missing from the original paper. We exhaustively enumerate each possible combination of implementation details and compare the results with those presented in the original paper by Koru et al. The results presented below are based upon the implementation details which correctly reproduce the results in the original paper.

3.2.1 Data Source

The dataset used in the original study by Koru et al. is the KOffice dataset. The KOffice dataset consists of ten open-source C++ projects that form a suite of productivity software (e.g., word processor, spreadsheet and presentation applications). Koru et al. collected the history of each class in each project between April 18, 1998 (i.e., the date of the initial commit to the KOffice source code repository) and January 19, 2006. A distinct dataset was created for each of the ten KOffice projects. Koru et al. have generously made the KOffice dataset available through the PROMISE repository [34].

3.2.2 Data Extraction

Extract Revision History: Koru et al. extracted the revision history for each class in the project. The revision history of a particular class contains the list of revisions, including the date and time of the revision and the commit log message (i.e., a description of the revision). Koru et al. also measured the size (i.e., lines of code) of the class at the time of the revision.

Identify Defect Fixes: Koru et al. identified defect fixes by searching for the keywords "bug," "fix" and "defect" in the commit log messages of each revision.

Transformation to Counting Process Format: The revision history extracted in the preceding sub-steps records the following information for each revision of each class: 1) the date and time of the revision, 2) the size of the class at the time of the revision and 3) a binary indicator for whether this revision is a defect fix. Recall that in Section 2.2.1, we described how survival data was composed of one or more observations, with each individual observation composed of a specific set of fields. Koru et al. analyzed the history of each class in the source code repository and created one observation for each revision of each class. Each individual observation was composed of the following fields:

1) ID – A unique identifier for each class in the study.
2) Start – The number of minutes between the previous revision to the class and this revision. The Start time of the first revision is set to zero.
3) End – The number of minutes between this revision and either 1) the next revision or 2) the end of the study, whichever occurs first.
4) Event – An indicator (one or zero) of whether this revision was a defect-fixing revision. Koru et al. identified defect-fixing revision by mining the commit log message of the revisions for a specific set of keywords (i.e., "bug," "fix" and "defect").
5) Size – The covariate of interest, i.e., the number of lines of code (excluding blank and comment lines) in the class at the start time of this revision (i.e., the class size after the revision was made). Size can, and often does, change during each revision.

These rules easily allows for the modelling of class deletions and changes in class size, something that is quite difficult in traditional regression modelling. These rules also allow for classes to be moved or renamed. Table 1 shows the history of a hypothetical class formatted according to these rules, including the creation, modification and deletion of the class.

3.2.3 Revision History (Counting Process Format)

The preceding steps produced a distinct dataset in the Counting Process Format for each project in the KOffice dataset. Table 2 lists each project in the KOffice dataset and contains a brief description of the functionality, size (total number of classes and lines of code) and activity (total number of revisions and defect-fixes) of each project. Each of these datasets has been formatted in the Counting Process Format described above.

3.2.4 Model Building

3.2.4.1 Model Calibration

When building a Cox model, consideration must be given to ensure that the Cox Proportional Hazards assumption (Equation 4) is satisfied with respect to covariates and confounding factors. This can be done using link functions and stratification.

Link function: Recall that from Equation 4, we expect to see a linear relationship between log-hazard and each covariate. A linear relationship between log-hazard and each covariate indicates that the Cox Proportional Hazards assumption has been satisfied. However, if this is not the case, than we must use link functions, as in Equation 5, to transform (or maintain) the relationship between log-hazard and the covariate to a linear relationship. Care must be taken to specify the link function because an incorrectly specified link function may become a source of nonproportionality [29].

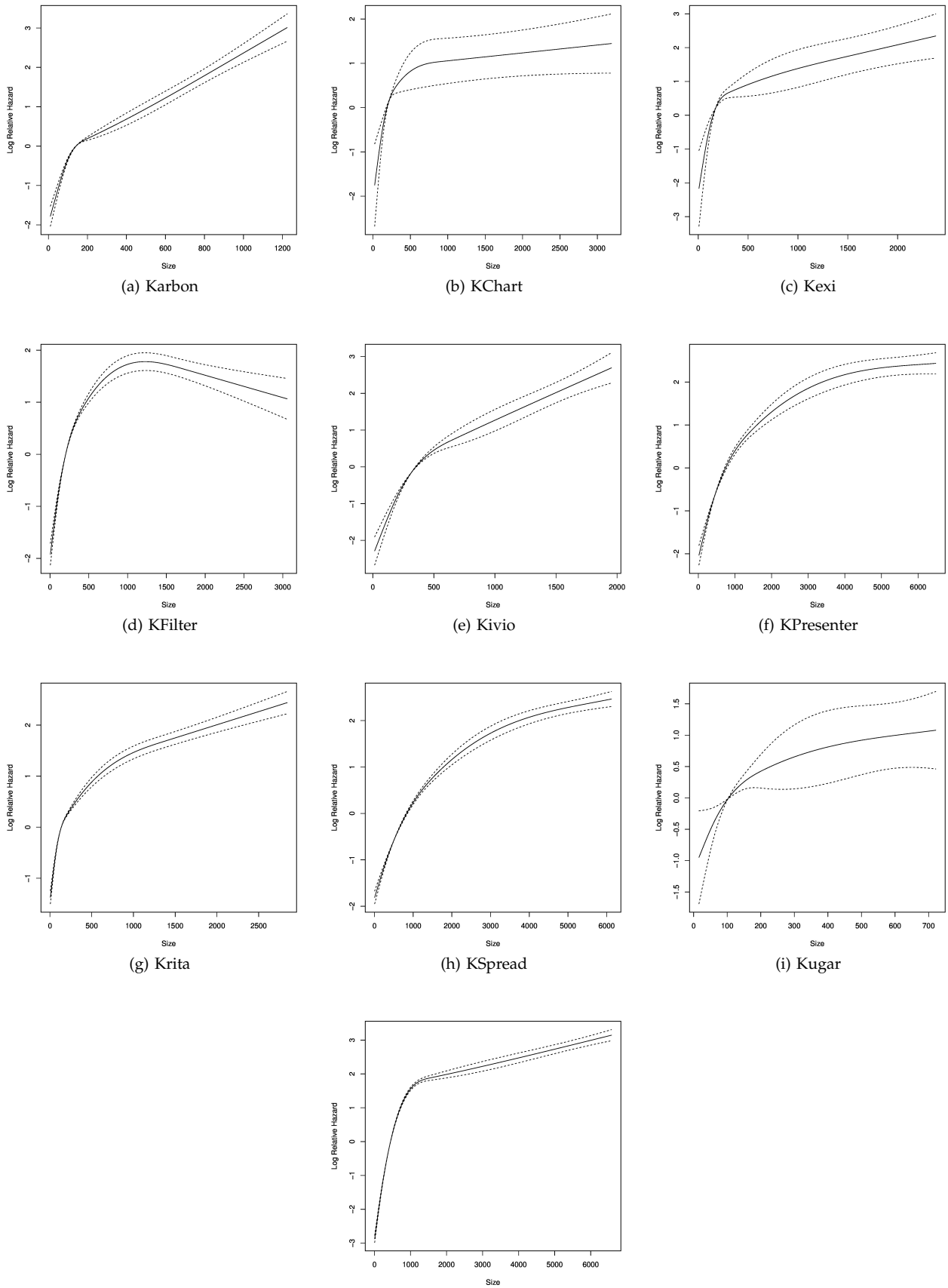
Within the traditional application domains of survival analysis, medical and biological sciences, these link functions are often well known from a large body of previous work [29]. However, when the link function is unknown, we must determine the link function ourselves.

Koru et al. identified their link function by plotting size against the log relative hazard [29], [35]. Koru et al. used a fitting function (i.e., restricted cubic splines) to visualize this relationship. Restricted cubic splines were used to divide size into multiple ranges that are identified by knots (four knots placed at quartile size values are recommended [35]). A cubic polynomial was then fit to each range. This approach relaxes the assumption that the relationship between the log relative hazard and size is linear. Further, cubic splines produce a better fit than linear splines because cubic splines curve at the knot points [35].

Figure 2 shows the relationship between the log relative hazard and size for each of the KOffice projects. The dashed line indicates the 95% confidence interval. The log relative hazard is relative to the average class size (i.e., the log relative hazard for a class with the average class size is zero). To remove potential outliers, the smallest and largest ten observations were removed. This analysis is performed automatically by the rms package (formerly the Design package), a R package that was developed by Harrell [35].

Figure 2 clearly shows that the relationship between the log relative hazard and size is not linear. Therefore, a link function is required to transform size. Koru et al. manually examined these figures and concluded that the general shape of the figures in Figure 2 is logarithmic. Therefore, Koru et al. used the natural logarithm of size to build their Cox model.

Stratification: In addition to applying a link function, Koru et al. also stratified their models based on the number of previous defects to control for the inherent "defect-proneness" of the class. The specific stratification levels were determined empirically by 1) examining the distribution of the number of defects per class across



(a) Karbon (b) KChart (c) Kexi

(d) KFilter (e) Kivio (f) KPresenter

(g) Krita (h) KSpread (i) Kugar

(j) KWord

Fig. 2: Identifying the link function for size for each of the KOffice projects.

TABLE 1: Formatted History of a Hypothetical Class

Class	Start	End	Event	Size	Note
foo.c	0	5	0	50	foo.c was created at time 0 (Start = 0) with a class size was 50 (Size = 50) and 3 lines were added to foo.c (Size = 50 + 3 for the next event) 5 minutes after creation (End = 5).
foo.c	5	9	1	53	a defect was fixed in foo.c (Event = 1) 9 minutes (End = 9) after creation and the last event was at 5 minutes after creation (Start = 5). 3 lines were added to foo.c in the last revision (Size = 50 + 3) and 6 lines were deleted from foo.c in this revision (Size = 53 - 6 for the next event).
foo.c	9	18	0	47	foo.c was modified (Event = 0) 18 minutes (End = 18) after creation and the last event was at 9 minutes after creation (Start = 9). 6 lines were deleted from foo.c in the last revision (Size = 53 - 6) and 51 lines were added to foo.c in this revision (Size = 47 + 51 for the next event).
foo.c	18	48	0	98	foo.c was deleted (Event = 0) 48 minutes (End = 48) after creation and the last event was at 18 minutes after creation (Start = 18). 51 lines were added to foo.c in the last revision (Size = 47 + 51) and 98 lines were deleted from foo.c in this revision (Size = 98 - 98, foo.c no longer exists).
foo.c no longer exists					

TABLE 2: Projects in the KOffice Dataset

Project	Functionality	#Classes	#LOC	#Revisions	#Defect-Fixes
Karbon	Vector graphics editor	382	30,749	5,072	1,242
KChart	Creating tool	112	22,719	406	98
Kexi	Data management tool	250	47,441	613	106
KFilter	File format converter	1,131	141,398	5,045	1,142
Kivio	Diagramming tool	191	29,869	1,431	377
KPresenter	Presentation tool	409	108,299	2,380	608
Krita	Graphics painting	1,210	112,422	9,149	2,961
KSpread	Spreadsheet tool	587	151,375	5,339	1,789
Kugar	Report generation tool	129	15,701	602	112
KWord	Word processor	802	83,731	5,953	1,932

each project in the KOffice dataset and 2) testing which stratification levels satisfy the Cox Proportional Hazards assumption. Each class was assigned to one of the following states: state 1 (no prior defects), state 2 (1-5 prior defects), state 3 (6-25 prior defects) and state 4 (more than 25 prior defects). Classes begin in an initial state (state 1) and are limited to making certain state transitions (state 1 to state 2, state 2 to state 3 and state 3 to state 4) as they experience defects. Classes cannot skip a state (i.e., defects are measured by the number of defect-fixing revisions and two defect-fixing revisions cannot simultaneously occur) or return to a previous state (i.e., the number of prior defects cannot decrease).

3.2.4.2 Model Fitting

Once link functions have been identified and stratification levels have been specified, a Cox model can be built from the data collected in Section 3.2.2. Koru et al. built one Cox model for each KOffice project. These models are summarized in Table 3. Table 3 shows the coefficient estimate ($\hat{\beta}$), the robust standard error estimate of $\hat{\beta}$ and the nonproportionality test statistic (explained in the next Section).

The default standard error estimate for $\hat{\beta}$ in a fitted Cox model assumes that each observation is independent. However, this is not the case in recurrent event analysis because subjects can have multiple events. Therefore, Koru et al. used a *robust standard error estimate* which systematically recomputes the covariates leaving out one or more subjects (which may have multiple events) at a time from the sample set. In this manner, the bias and variance in the coefficient is estimated.

TABLE 3: Cox Models for the KOffice Projects

Project	$\hat{\beta}$	Robust Standard Error	Nonproportionality Test Statistic (p-value)
Karbon	0.592	0.069	0.817
KChart	0.656	0.109	0.339
Kexi	0.843	0.100	0.691
KFilter	0.583	0.040	0.770
Kivio	0.786	0.075	0.780
KPresenter	0.590	0.051	0.402
Krita	0.414	0.026	0.061
KSpread	0.474	0.033	0.738
Kugar	0.555	0.091	0.492
KWord	0.740	0.037	0.285

The $\hat{\beta}$s may be interpreted as follows: one unit increase in the natural logarithm of class size multiplies the rate of experiencing defects (i.e., the hazard) by $e^{\hat{\beta}}$.

From Table 3, Koru et al. found that $0 < \hat{\beta} < 1$. This indicates that the relationship between class size and defect-proneness is consistent across the ten KOffice project. The implications will be further explored in Section 3.3. However, prior to interpreting the results of their models, Koru et al. validated the Cox Proportional Hazards assumption and investigated overly influential observations.

3.2.4.3 Model Verification

Valid Cox models will satisfy the Cox Proportional Hazards assumption and will not be overly influenced by any single observation. Koru et al. verified these two conditions before interpreting the results of their models.

The Cox Proportional Hazards assumption states that the log-relative hazard between two subjects is linearly dependent on the difference between their covariate values and holds for all time (Equation 4). This assumption can be evaluated using several graphical and/or numeric techniques. Using Equation 6, the *nonproportionality test statistic* tests the null hypothesis that $H_o : \hat{\Theta} = 0$, given some time dependent function $g(t)$. If the hypothesis that $\hat{\Theta}$ is zero cannot be rejected, then $\hat{\beta}$ has a statistically significant interaction with time (a violation of the Proportional Hazard assumption).

$$\hat{\beta}(t) = \hat{\beta} + \hat{\Theta} \times g(t) \tag{6}$$

Although not specifically stated by Koru et al., the results in Table 3 are consistent with the identity transform (i.e., $g(t) = t$). The identity transform, combined with Equation 6, tests whether $\hat{\beta}$ has a statistically significant *linear* interaction with time.

From Table 3, Koru et al. found that the nonproportionality test statistic indicates that $\hat{\beta}$ does not have a statistically significant interaction with time (i.e., p $\geq$ 0.05). Therefore, the Cox Proportional Hazards assumption was satisfied.

Overly influential observations may skew the coefficients of the final model and affect the validity of the Cox Proportional Hazards assumption. Koru et al. analyze the impact that each observation has on the model using dfbeta residuals. The dfbeta residual calculates the influence of each observation by fitting a Cox model to the dataset with and without the observation. The difference between the $\hat{\beta}$s of both models (i.e., one model with the observation and one model without the observation) is the influence of that observation. Figure 3 shows the dfbeta residuals for each of the KOffice projects.

From Figure 3, Koru et al. identified outliers from each KOffice project. However, upon further analysis of each outlier, Koru et al. chose not to remove any observations as they were found to be valid observations.

3.3 Results

With a set of models, Koru et al. defined the Relative Defect-Proneness (RDP) of two classes, i and j, as the log-relative hazard between class i and j. This is based solely on the differences between class i and j and regardless of any baseline defect-proneness. In this case study, the difference between class i and j is based solely on the difference in class size (i.e., lines of code).

The relationship between size and defect-proneness was derived from the Cox model. From Equation 2, the risk that class i experiences a defect relative to the risk that class j experiences a defect depends on their size (i.e., X_i(t) and X_j(t) respectively) and a link function f. Koru et al. found that the link function was logarithmic, therefore Equation 2 simplifies to:

$$\frac{\lambda_i(t)}{\lambda_j(t)} = \exp((f(X_i(t)) - f(X_j(t))) \times \beta) \tag{7}$$

$$= \exp((log(X_i(t)) - log(X_j(t))) \times \beta) \tag{8}$$

$$= \exp(log(X_i(t)/X_j(t)) \times \beta) \tag{9}$$

$$= (X_i(t)/X_j(t))^{\beta} \tag{10}$$

From Equation 10, the relationship between $\hat{\beta}$ and 1) the number of defects and 2) defect density (i.e., the number of defects divided by class size) can be derived, as shown in Table 4. The relationships in Table 4 are well supported if $\hat{\beta}$ and the 95% confidence interval fall within one of these interpretation ranges.

TABLE 4: Interpretation of the $\hat{\beta}$ Coefficient.

Range	Relationship Between Size and:	
	Number	Density
$\hat{\beta}(t) < 0$	Decreases	Decreases
$\hat{\beta}(t) = 0$	No Impact	No Impact
$0 < \hat{\beta}(t) < 1$	Increases	Decreases
$\hat{\beta}(t) = 1$	Increases	No Impact
$\hat{\beta}(t) > 1$	Increases	Increases

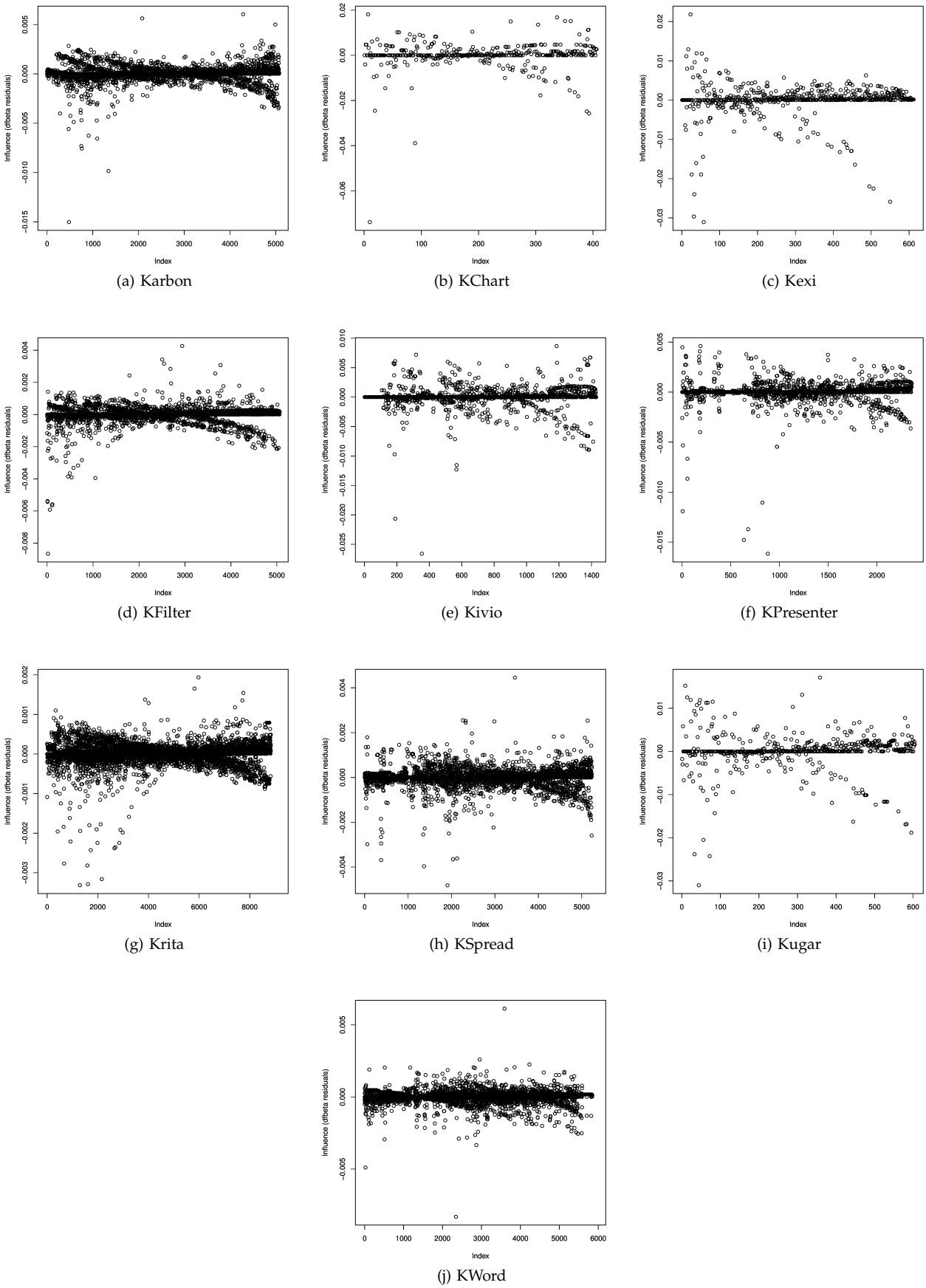
Figure 4 shows the $\hat{\beta}$s and the 95% confidence intervals for each of the KOffice projects. From this, Koru et al. found that, with one exception, $\hat{\beta}$ and the 95% confidence interval lie between zero and one. Therefore, their hypothesis that smaller classes are proportionally more defect-prone is well supported. Specifically, there is a power-law relationship between class size and defect-proneness where defect-proneness increases at a slower rate compared to class size.

3.4 Lessons Learned

Table 5 shows the implementation details that were missing from the original paper by Koru et al., but required for our replication study. A description of these implementation details is available in the documentation of the rms and survival packages [32], [33]. In addition to these details, we also provide the scripts that we used to replicate the work in Koru et al. in the appendix.

In addition to the missing parameters listed in Table 5, several other implementation details were missing. For example, two distinct models were fit to each dataset: 1) one model was used to calculate the nonproportionality test statistic and 2) one model was used to calculate the robust standard error estimate.

Despite the difficulty in reverse engineering the implementation details in the original study by Koru et al., the results of this research question indicate that we were successful in recovering these details. In addition, we were able to determine whether the implementation details chosen be Koru et al. were appropriate. This insight will be leveraged in our next two research questions.



(a) Karbon (b) KChart (c) Kexi

(d) KFilter (e) Kivio (f) KPresenter

(g) Krita (h) KSpread (i) Kugar

(j) KWord

Fig. 3: Identifying overly influential observations using dfbeta residuals for each of the KOffice projects.

TABLE 5: Missing Implementation Details.

Function	Missing Parameter	Possible Values	Actual Value
coxph	ties	efron, breslow or exact	efron
	cluster	either null or id	id
cph	method	efron, breslow, exact, model.frame or model.matrix	efron
cox.zph	transform	km, rank or identity	identity

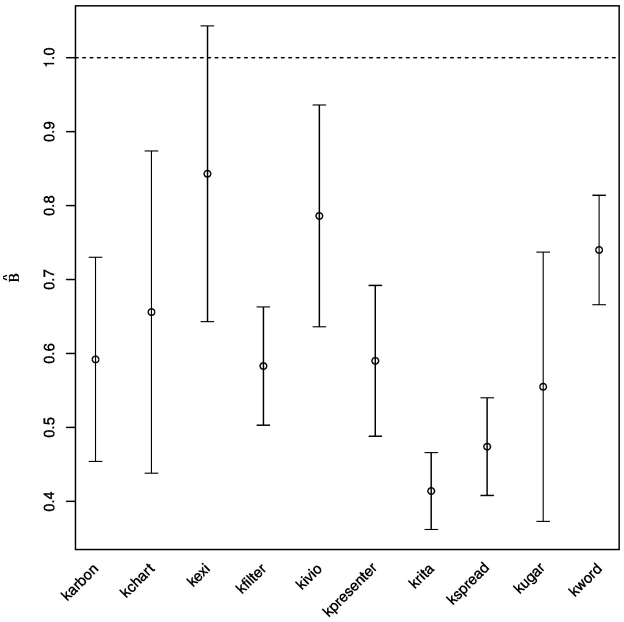


Fig. 4: $\hat{\beta}$ and the 95% Confidence Interval.

RQ2: Can we generalize the approach of Koru et al. to additional software projects?

3.5 Motivation

Our second research question addresses the generalizability of the methodology and results of Koru et al. We used the methodology and data formulation of Koru et al. (i.e., defect fix and continuous time scale), described in Section 3, to fit a Cox model to Chrome, Eclipse, Firefox and Netbeans. We then compare the results derived from these models with the results of Koru et al. (presented in Section 3).

Koru et al. have studied Eclipse in their previous work [9]. They found that smaller classes are proportionally more defect-prone in Eclipse (i.e., the same relationship they found in the KOffice dataset). However, Koru et al. failed to properly verify the Cox Proportionals Hazards assumption: the authors use the nonproportionality test statistic, a numerical technique, that is widely considered to be insufficient [29], [36], [37]. Therefore, we use scaled Schoenfeld residuals, a graphical technique, that is widely used to verify the Cox Proportional Hazards assumption [29], [36], [37].

3.6 Approach

3.6.1 Data Source

The dataset used in our replication study consists of four large-scale, widely-used software projects (i.e., Chrome, Eclipse, Firefox and Netbeans). The revision history of each source code file for each of the four projects was collected. The data was collected between the date of the initial commit to each source code repository and June 24, 2010. A distinct dataset was created for each of the four projects.

3.6.2 Data Extraction

Extract Revision History: We extracted the revision history for each file in the project. The revision history of a particular file contains the list of revisions, including the date and time of the revision and the commit log message. We also measured the size (i.e., lines of code) of the file after the revision was made.

Identify Defect Fixes: We identified defect fixes by searching for the keywords "bug," "x," "defect" and "patch" in the commit log messages of each revision. In addition to one or more of these keywords, we searched for a unique numeric identifier (i.e., a defect identifier). We then cross-referenced the defect identifier with the issue tracking system (e.g., Bugzilla) to confirm that the revision was a defect fix.

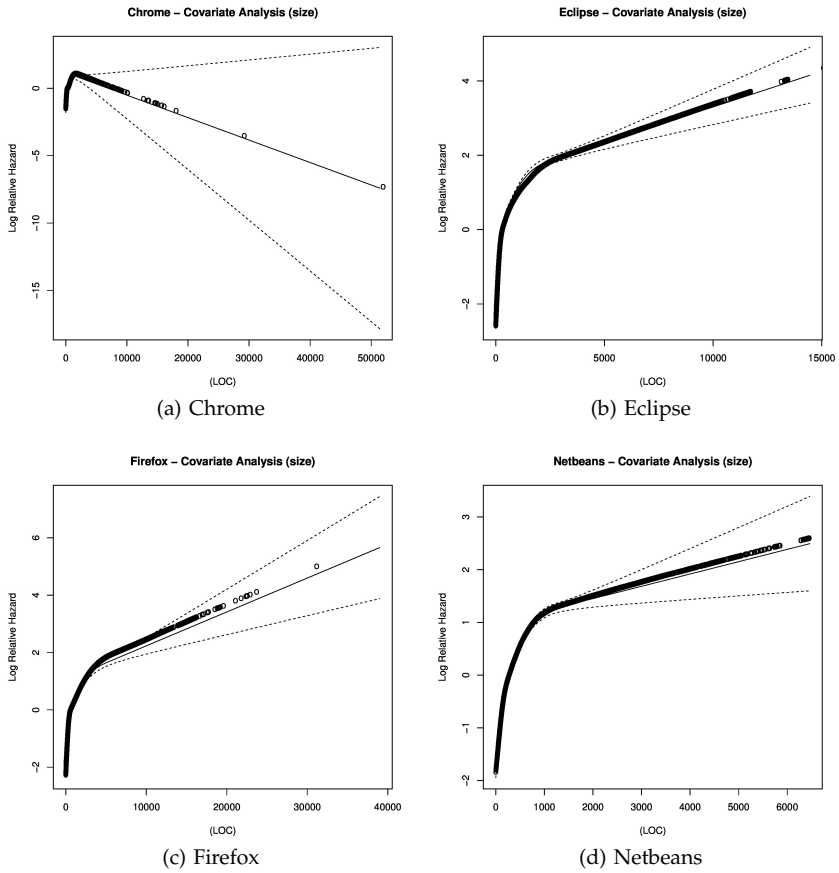
Transformation to Counting Process Format: The revision history extracted in the proceeding sub-steps was formatted this data in the same manner as Koru et al., outlined in Section 3.2.2.

3.6.3 Revision History (Counting Process Format)

Table 6 presents an overview of the four projects in our dataset. We calculate the total number of 1) source code files, 2) lines of code, 3) revisions and 4) defect fixes for each project. Similar information for the KOffice projects was presented in Table 2 in Section 3.2.3. Table 6 also presents the date of the first revision for each project.

TABLE 6: Descriptive Statistics for Chrome, Eclipse, Firefox and Netbeans

Project	#Files	#LOC	#Revisions	#Defect-Fixes	First Revision
Chrome	8,034	2,277,598	114,019	629	7/26/2008
Eclipse	9,318	1,977,825	227,802	33,561	5/2/2001
Firefox	11,697	3,478,150	270,351	12,166	27/3/1998
Netbeans	9,760	1,847,668	119,725	23,577	5/1/1999



(a) Chrome (b) Eclipse

(c) Firefox (d) Netbeans

Fig. 5: Plots of the log-relative hazard against size to identify the link function. The dashed line indicates the 95% confidence interval. The confidence intervals diverge at the larger end of the scale where there are fewer files.

3.6.4 Model Building

3.6.4.1 Model Calibration

Link function: First, we identify the link function using the same technique as Koru et al. presented in Section 3.2.4. Figure 5 shows the relationship between the log relative hazard and size for Chrome, Eclipse, Firefox and Netbeans.

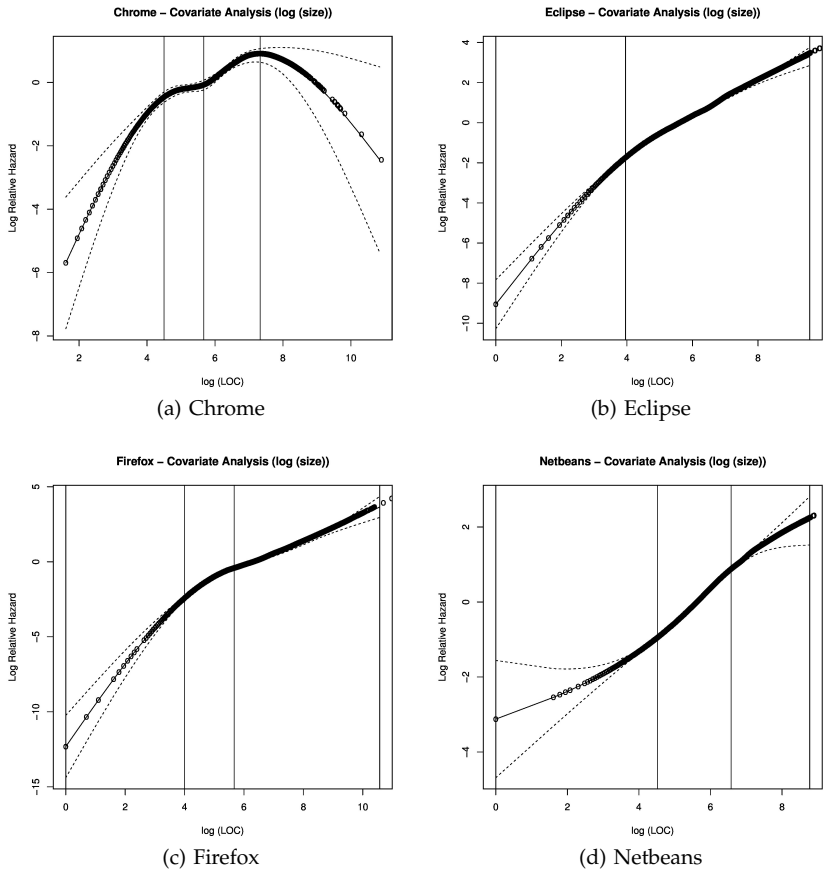
Figure 5 clearly shows that the relationship between the log relative hazard and size is not linear. Therefore, a link function is required. Similar to Koru et al., our link function is the natural logarithm. To determine whether the link function is sufficient to satisfy the Cox Proportional Hazards assumption (Equation 4), we plot the relationship between the log relative hazard and the natural logarithm of size for Chrome, Eclipse, Firefox and Netbeans. Figure 6 shows this relationship.

We expect to see a linear relationship between the log relative hazard and the natural logarithm of size if the Cox Proportional Hazards assumption is satisfied over the entire size range. However, from Figure 6, we find that this relationship is not linear. Therefore, a link function alone is not sufficient to satisfy the Cox Proportional Hazards assumption and the approach of Koru et al. is not sufficient to build valid Cox models for Chrome, Eclipse, Firefox and Netbeans. However, the reason for this is unclear.

One potential reason may be that the link function used by Koru et al. (i.e., the natural logarithm) is an approximation of the actual link function. This approximation may break-down at larger module sizes and this break-down is exasperated by the presence of much larger modules. Although we can only validate this reason by determining the actual link function, which requires exhaustive evaluation of all possible link functions, we should expect much larger modules in Chrome, Eclipse, Firefox and Netbeans if this break-down occurs. Table 7 shows the minimum, median and maximum module (i.e., classes for the KOffice projects and files for Chrome, Eclipse, Firefox and Netbeans) sizes for the projects in our replication study.

From Table 7, we find that the maximum module size in Chrome (73,480), Eclipse (14,450) and Firefox (39,000) is more than twice as large as the maximum module size in any of the KOffice projects.

Another potential reason that a link function alone is not sufficient to satisfy the Cox Proportional Hazards assumption and the approach of Koru et al. was not sufficient to build valid Cox models for Chrome, Eclipse, Firefox and Netbeans may be that the size-defect relationship is influenced by the development community. Therefore, the link function may also differ between projects because the development community



(a) Chrome (b) Eclipse

(c) Firefox (d) Netbeans

Fig. 6: Plots of the log-relative hazard against the natural logarithm of size to verify the link function. The dashed line indicates the 95% confidence interval. The confidence intervals tend to diverge at the smaller and larger ends of the scale where there are fewer files.

TABLE 7: Module size statistics for the KOffice Projects, Chrome, Eclipse, Firefox and Netbeans

Project	Minimum	Median	Maximum
Karbon	4	146	1,280
KChart	7	219.5	6,686
Kexi	4	175	2,858
KFilter	2	201	3,377
Kivio	5	318	1,984
KPresenter	9	457	6,511
Krita	2	126	4,270
KSpread	1	513	6,312
Kugar	5	102	844
KWord	3	399	6,591
Chrome	1	281	73,480
Eclipse	1	279	14,450
Firefox	1	1,341	39,000
Netbeans	1	422	6,462

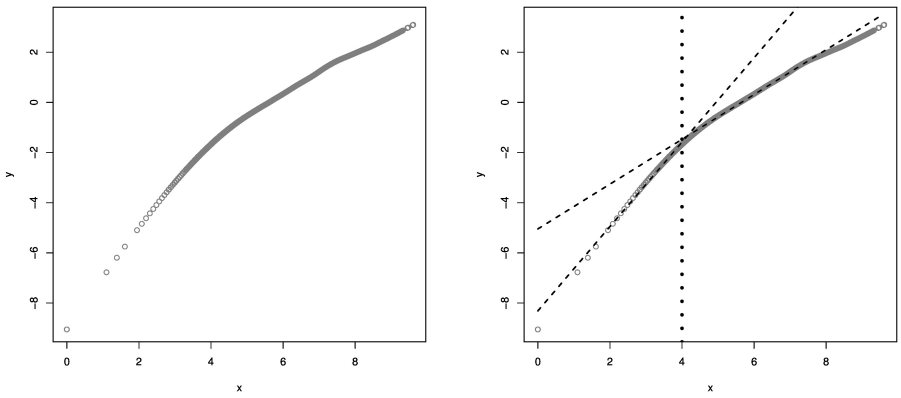
differs from project to project.

Regardless of the underlying cause, changes to the approach of Koru et al. are required to satisfy the Cox Proportional Hazards assumption and build a valid Cox model. One such change is partitioning. Instead of fitting a single Cox model over the entire size range, we fit multiple models by partitioning size into subsets where the log-relative hazard is piecewise linear. Hence, partitioning produces multiple models for a single project, where each model explains the size-defect relationship in a subset of the files (i.e., files with a specific file size). Such an approach to modelling (i.e., building models on subsets of the files) is becoming more common in empirical software engineering [38].

Figure 7a shows an example of a non-linear curve and Figure 7b show how that curve can be partitioned so that we have two linear subsets. From Figure 7b, we see that partitioning the curve at 4.0 produces two linear sections.

Each partition must contain enough subjects (e.g., files) and events (e.g., defect fixes) to fit a Cox model because each partition acts as an independent dataset. In a stratified Cox model, each partition must contain between 5 and 7 events per stratum [39].

Partitioning size has many advantages over other techniques (e.g., complex link functions) for dealing with non-proportionality. First, we can easily interpret the resulting models, whereas interpretation can become difficult when complicated link functions are used. Second, we can specifically test whether the relationship between lines of code and defect-proneness is constant over the entire range of lines of code.



(a) Curve (b) Partitioned Curve

Fig. 7: Partitioning a Non-Linear Curve.

Stratification: Similar to Koru et al., we stratify our models based on the number of previous defects to control for the inherent "defect-proneness" of the file. The specific stratification levels were determined empirically by 1) examining the distribution of the number of defects per file across Chrome, Eclipse, Firefox and Netbeans and 2) testing which stratification levels satisfy the Cox Proportional Hazards assumption for all Cox models. Table 8 shows the resulting stratification levels for each project. Although the specific stratification levels differ between projects, they do not hamper our interpretation of the resulting Cox models.

TABLE 8: Stratification Levels for Chrome, Eclipse, Firefox and Netbeans

Project	List of Stratification Levels
Chrome	0, 1-5, 6+
Eclipse	0, 1, 2, 3-4, 5-6, 7-9, 10-15, 16-37, 38+
Firefox	0, 1, 2, 3-4, 5-6, 7-9, 10-17, 18+
Netbeans	0, 1, 2, 3, 4-5, 6-8, 9-14, 15-30, 30+

3.6.4.2 Model Fitting

Once we have identified the link function, partition points and stratification levels, we build one or more Cox models (based on the number of partition points) for Chrome, Eclipse, Firefox and Netbeans. These models are summarized in Table 9. Table 9 presents the partition points (i.e., the minimum and maximum file size in the partition), the coefficient estimate ($\hat{\beta}$), the robust standard error estimate of $\hat{\beta}$ and the nonproportionality test statistic for each model.

From Table 9, we find that anywhere between two and four models are needed to explain the size-defect relationship in any one project.

3.6.4.3 Model Verification

Valid Cox models will satisfy the Cox Proportional Hazards assumption and will not be overly influenced by any single observation. We verify these two conditions before interpreting the results of their models.

The Cox Proportional Hazards assumption states that the log-relative hazard between two subjects is linearly dependent on the difference between their covariate values and holds for all time. Significant departures from the Cox Proportional Hazards assumption can invalidate a Cox model and lead to incorrect conclusions. We assess the Cox Proportional Hazards assumption using numerical and graphical techniques.

The numerical technique tests whether size has a statistically significant interaction with time. We use the same technique as Koru et al. to test whether $\hat{\beta}$ has a statistically significant interaction with time. However, we use a different transform (i.e., the Kaplan-Meier transform) because the Kaplan-Meier transform is far less influenced by outliers than the identity transform [40] that Koru et al. used in their original study [8]. Despite the usefulness of the nonproportionality test statistic, the numerical technique by itself is not adequate alone, because a violation of the Cox Proportional Hazards assumption does not necessarily invalidate the model [29], [36], [37]. First, while the interaction of time with a particular covariate may be statistically significant, the effect of this nonproportionality on the model may in fact be small. Second, the nonproportionality may have been introduced by a small number of overly influential subjects. To determine if either of these situation is the reason of the nonproportionality, we must also use a graphical technique.

The graphical technique requires plotting the scaled Schoenfeld residuals. The Cox Proportional Hazards assumption is supported by a random pattern of residuals against time. From Table 9, we find that, for all models, the nonproportional test statistic does not have a statistically significant interaction with time (i.e., $p \geq 0.05$). Figure 8 shows the scaled Schoenfeld residuals for Eclipse (similar results were found for Chrome, Firefox and Netbeans). We find that the Cox Proportional Hazards assumption is supported, because the residuals show a random pattern against time.

TABLE 9: Cox Models for Chrome, Eclipse, Firefox and Netbeans

Project	min(size)	max(size)	$\hat{\beta}$	Robust Standard Error	Nonproportionality Test Statistic (p-value)	Interpretation (Does Defect Density Increase or Decrease with Size?)
Chrome	1	90	1.17	0.232	0.863	Increases
Chrome	90	290	0.329	0.131	0.598	Decrease
Chrome	290	1525	0.157	0.0958	0.581	Decrease
Chrome	1525	73479	-0.185	0.224	0.927	Decrease
Eclipse	1	52	1.08	0.114	0.65	Increases
Eclipse	52	14448	0.401	0.0089	0.0677	Decrease
Firefox	1	55	2.5	1.26	0.934	Increases
Firefox	55	290	0.216	0.0322	0.167	Decrease
Firefox	290	38998	0.151	0.0154	0.11	Decrease
Netbeans	1	92	0.445	0.0723	0.126	Decrease
Netbeans	92	718	0.288	0.0168	0.054	Decrease
Netbeans	718	6463	0.291	0.0805	0.752	Decrease

Overly influential subjects (files) may skew the coefficients of the final model and affect the validity of the Cox Proportional Hazards assumption. We mark these overly influential subjects for removal in the next iteration of model fitting.

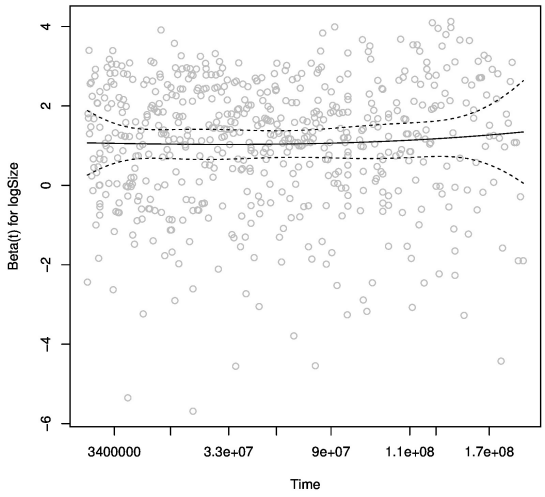
We use dfbetas residuals to identify and remove overly influential files. Overly influential files will have dfbetas values greater than twice the inverse of the square root of the number of files [41], [42]. Overly influential files are removed from the dataset and the model is refit without these files. This is a similar technique that was used by Koru et al., however, we assess overly influential *files*, as opposed to overly influential *revisions*. We also remove all outliers, as opposed to manually determining whether a file is an outlier because this approach is not biased by human interpretation.

Figure 9 shows the dfbetas residuals for Eclipse. Data points above the upper dashed line or below the lower dashed line represent files that are considered outliers [41], [42].

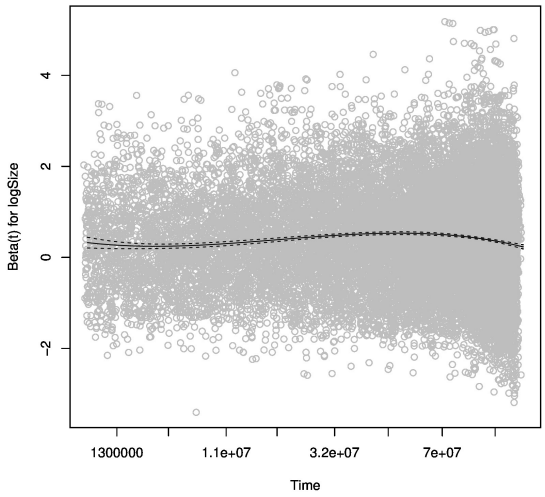
3.7 Results

Finally, we interpret the size-defect relationship. By choosing the same link function (i.e., the natural logarithm) as Koru et al., we can make the same interpretation regarding the size-defect relationship captured by a Cox model. The relationship between $\hat{\beta}$ and 1) the number of defects and 2) defect density is shown in Table 4 in Section 3.3. When multiple partitions are present, we can make the same interpretation regarding the size-defect relationship for each partition.

From Table 9, we find that the relationship between size and defects in the four projects is not consistent. In three of the four projects we find that defect density *increases* in smaller files, peaks in the largest small-sized files/smallest medium-sized files, then *decreases* in medium and larger files. This shows that defect density has an inverted "U" shaped pattern (i.e., medium-sized files have a higher defect density than small or large files). However, this conclusion is not well supported because $\hat{\beta}$ and the 95% confidence interval overlap with

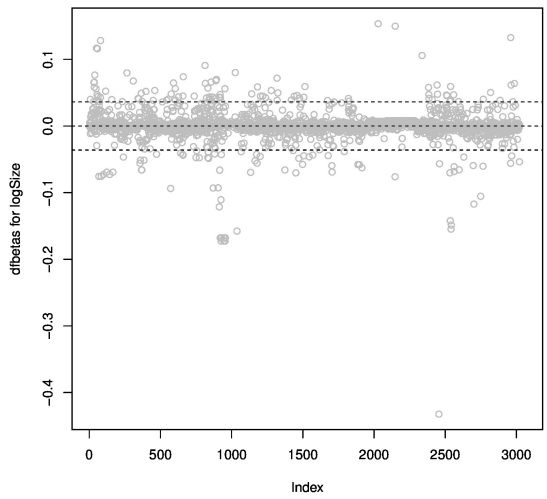


(a) Small Partition (1-52LOC)

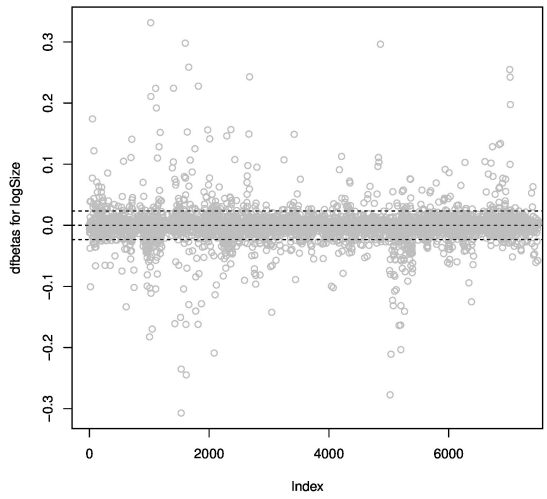


(b) Large Partition (52-14448LOC)

Fig. 8: Scaled Schoenfeld Residuals for Eclipse. The values in parenthesis indicate the range in the lines of code for each partition (e.g., the small partition includes files between 1 and 52 lines of code).



(a) Small Partition (1-52LOC)



(b) Large Partition (52-14448LOC)

Fig. 9: Dfbetas Residuals for Eclipse. The values in parenthesis indicate the range in the lines of code for each partition (e.g., the small partition includes files between 1 and 52 lines of code)

two of the interpretation ranges from Table 4 (i.e, $0 < \hat{\beta} < 1$ and $\hat{\beta} > 1$). For the fourth project, Netbeans, the size-defect relationship indicates that defect density continually decreases as file size increases. This is the same result that was found by Koru et al. in the KOffice project and in our first research question.

In general, our results do not indicate a well supported, consistent relationship between size and defects.

> We find that the conclusions of Koru et al. are not generalizable.

RQ3: What is the impact of using a different data formulation?

3.8 Motivation

Koru et al. formulated the modelling of defects in source code modules as a time-to-event problem and used survival analysis to study the size-defect relationship. They found that smaller modules are proportionally more defect-prone (i.e., the number of defects per line of code is higher in smaller modules). Koru et al. demonstrated the power of survival analysis in modelling defects, however care should be taken when transferring approaches from other fields to software engineering. Major differences exist between the traditional domains of survival analysis and defect modelling.

First, Koru et al. modelled defect *fix* data, as opposed to defect *introduction* data. However, to prioritize software quality improvement efforts, software practitioners must model when a defect will be introduced, as opposed to when a defect will be fixed because defect introductions are the true event of interest. In traditional defect models, defect fix data is a good approximation for defect introduction data because all time information is collapsed when building the model for a particular point in time, therefore, the time difference between defect introduction and fix becomes almost irrelevant. However, survival analysis explicitly takes into account time information, making it likely that the approximation of defect introduction by defect fix no longer holds.

Second, Koru et al. modelled events along a continuous time scale (i.e., a defect can be fixed or introduced at any time). However, defects can only be fixed or introduced along a discrete time scale (i.e., when a revision occurs). Software practitioners can modify the source code at any point in time, however, we can only observe these changes when revisions are made to the source code repository. Therefore, defect fixes or introductions, occur along a discrete time scale. Survival analysis experts recommend that a discrete time scale be used when observations can only be made at specific points in time [10], [11].

Therefore, we formulate a new dataset using a discrete time-scale that uses defect introductions as the event of interest.

3.9 Approach

3.9.1 Data Source

The dataset used in our replication study consists of the same four software projects (i.e., Chrome, Eclipse, Firefox and Netbeans) as our previous research question.

3.9.2 Data Extraction

Extract Revision History: We extracted the revision history for each file in the project. The revision history of a particular file contains the list of revisions, including the date and time of the revision and the commit log message. We also measured the size (i.e., lines of code) of the file after the revision was made.

Identify Defect Fixes: We used the SZZ algorithm presented in [43] by Śliwerski et al. to determine which revisions are defect introducing.

The SZZ algorithm is currently the state-of-the-art algorithm for automatically identifying defect introducing revisions. First, SZZ identifies defect fixing revisions by searching for the keywords "bug," "x," "defect" and "patch" in the commit log messages of each revision and matching defect identifiers in the revisions' commit messages to defect reports in the issue tracking system that are marked as FIXED. Second, each modified code snippet (fixed code) in a fixing revision is mapped back to the most recent revision in which it was modified. Finally, the defect-introducing revisions are identified based on how much fixed code maps back to these revisions. Therefore, we are able to identify each revision as 1) defect introducing, 2) defect fixing or 3) neither.

Transformation to Counting Process Format: The revision history extracted in the proceeding sub-steps records the following information for each revision of each file: 1) the revision number, 2) the size of the file after the revision was made and 3) a binary indicator for whether this revision is a defect introduction. We analyzed the history of each file in the source code repository and created one observation for each revision of each file. Each individual observation was composed of the following fields:

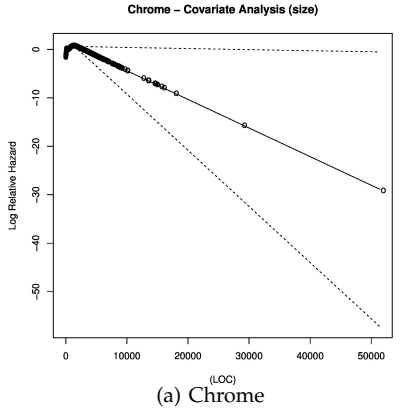
1) ID – A unique identifier for each file in the study.
2) Start – The End time of the previous revision plus one. The Start time of the first revision is set to zero. A discrete time scale normalizes the time between two revisions. Regardless of the number of minutes between two revisions, we always measure one unit of time.
3) End – The Start time plus one.
4) Event – An indicator (one or zero) of whether this revision was a defect introducing revision.
5) State – The current stratification level (state) of the file. Table 11 shows the resulting stratification levels for each project.
6) Size – The covariate of interest, the number of lines of code in the file at the Start time (i.e, the file size after the revision was made).

3.9.3 Revision History (Counting Process Format)

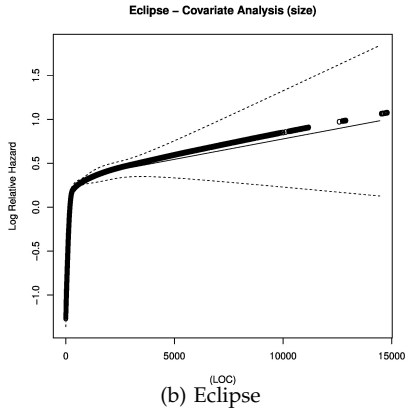
Table 10 presents an overview of Chrome, Eclipse, Firefox and Netbeans. Similar information for the KOffice projects was presented in Table 2 in Section 3.2.3.

TABLE 10: Descriptive Statistics for Chrome, Eclipse, Firefox and Netbeans

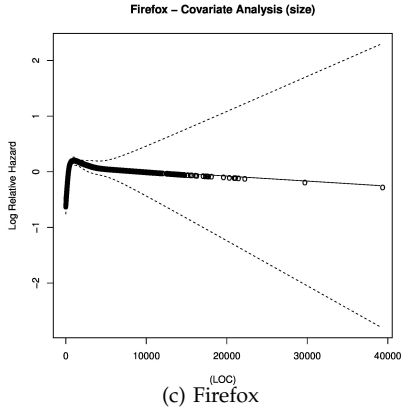
Project	#Files	#LOC	#Revisions	#Defect-Introductions
Chrome	8,034	2,277,598	114,019	467
Eclipse	9,318	1,977,825	227,802	21,508
Firefox	11,697	3,478,150	270,351	9,164
Netbeans	9,760	1,847,668	119,725	16,267



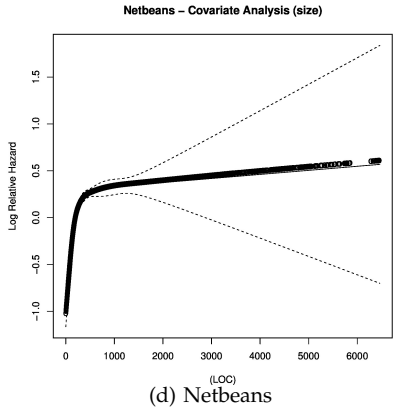
(a) Chrome



(b) Eclipse



(c) Firefox



(d) Netbeans

Fig. 10: Plots of the log-relative hazard against size to identify the link function. The dashed line indicates the 95% confidence interval. The confidence intervals diverge at the larger end of the scale where there are fewer files.

3.9.4 Model Building

3.9.4.1 Model Calibration

Link function: Similar to our previous research question, we identify the link function using the technique presented in Section 3.2.4. Figure 10 shows the relationship between the log relative hazard and size for Chrome, Eclipse, Firefox and Netbeans.

Similar to our previous research question, Figure 10 clearly shows that the relationship between the log relative hazard and size is not linear. Again, a link function is required and our link function is the natural logarithm. To determine whether the link function alone is sufficient to satisfy the Cox Proportional Hazards assumption (Equation 4), we plot the relationship between the log relative hazard and the natural logarithm of size for Chrome, Eclipse, Firefox and Netbeans. Figure 11 shows the relationship between the log relative hazard and the natural logarithm of size for Chrome, Eclipse, Firefox and Netbeans.

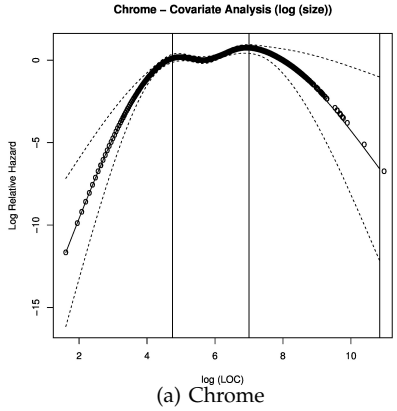
Similar to our previous research question, Figure 11 shows that the relationship between the log relative hazard and the natural logarithm of size is not linear. Therefore, a link function alone is not sufficient to satisfy the Cox Proportional Hazards assumption and, again, we must also partition size. We used the technique presented in Section 3.6.4 to identify the partition points. The solid lines in Figure 11 indicate these partitions points.

From Figure 11, we find that each project has three partitions. These partitions can broadly be classified into small files, medium files and large files. A Cox model will be fit to each of the three partitions of Chrome, Eclipse, Firefox and Netbeans (i.e., three Cox models per project). However, prior to fitting the Cox model and interpreting the size-defect relationship within each partition, we must ensure that the models we have fit are valid models.

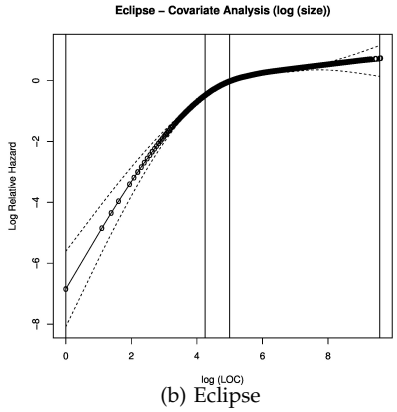
Stratification: Similar to Koru et al., we stratify our models based on the number of previous defects to control for the inherent "defect-proneness" of the file. Table 11 shows the resulting stratification levels for Chrome, Eclipse, Firefox and Netbeans.

TABLE 11: Stratification Levels for Chrome, Eclipse, Firefox and Netbeans

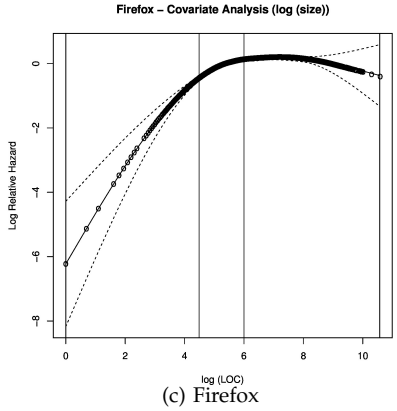
Project	List of Stratification Levels
Chrome	0, 1-5, 6+
Eclipse	0, 1, 2, 3-4, 5-6, 7-9, 10-15, 16-37, 38+
Firefox	0, 1, 2, 3-4, 5-6, 7-9, 10-17, 18+
Netbeans	0, 1, 2, 3, 4-5, 6-8, 9-14, 15-30, 30+



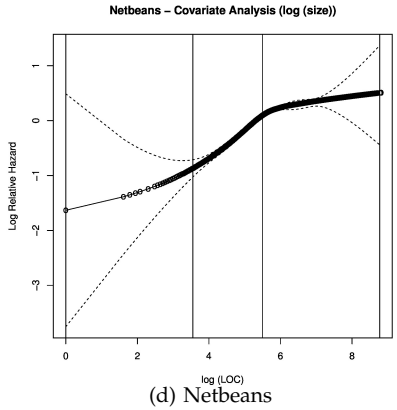
(a) Chrome



(b) Eclipse



(c) Firefox



(d) Netbeans

Fig. 11: Plots of the log-relative hazard against the natural logarithm of size to verify the link function. The dashed line indicates the 95% confidence interval. The confidence intervals tend to diverge at the smaller and larger ends of the scale where there are fewer files.

TABLE 12: Cox Models for Chrome, Eclipse, Firefox and Netbeans

Project	min(size)	max(size)	$\hat{\beta}$	Robust Standard Error	Nonproportionality Test Statistic (p-value)	Interpretation (Does Defect Density Increase or Decrease with Size?)
Chrome	1	116	1.66	0.145	0.968	Increases
Chrome	116	1097	0.371	0.0865	0.917	Decreases
Chrome	1097	51386	-1.33	0.442	0.433	Decreases
Eclipse	1	70	1.62	0.0891	0.597	Increases
Eclipse	70	148	0.388	0.0816	0.866	Decreases
Eclipse	148	14448	0.0878	0.0165	0.558	Decreases
Firefox	1	89	1.82	0.201	0.921	Increases
Firefox	89	403	0.114	0.0503	0.879	Decreases
Firefox	403	38998	-0.0141	0.0294	0.477	Decreases
Netbeans	1	35	1.74	0.686	0.708	Increases
Netbeans	35	245	0.614	0.035	0.108	Decreases
Netbeans	245	6463	0.0675	0.0218	0.260	Decreases

3.9.4.2 Model Fitting

Once we have identified the link function, partition points and stratification levels, we build one or more Cox models (based on the number of partition points) for each of the projects in our dataset. These models are summarized in Table 12. Table 12 presents the partition points (i.e., the minimum and maximum file size in the partition), the coefficient estimate ($\hat{\beta}$), the robust standard error estimate of $\hat{\beta}$ and the nonproportionality test statistic.

From Table 12, we find that three models are used to explain the size-defect relationship in any one project. These partitions broadly correspond to small files, medium files and large files.

3.9.4.3 Model Verification

Valid Cox models will satisfy the Cox Proportional Hazards assumption and will not be overly influenced by any single observation. We verify these two conditions before interpreting the results of our models.

We assess the **Cox Proportional Hazards assumption** using the nonproportionality test statistic and the scaled Schoenfeld residuals. From Table 12, we find that, for all models, the nonproportionality test statistic indicates that $\hat{\beta}$ does not have a statistically significant interaction with time (i.e., $p \geq 0.05$). This is evidence that the Cox Proportional Hazards assumption is satisfied.

However, Koru et al. failed to properly verify the Cox Proportionals Hazards assumption. Koru et al. used the nonproportionality test statistic, a numerical technique, that is widely considered to be insufficient [29], [36], [37]. Therefore, we also use scaled Schoenfeld residuals, a graphical technique, that is widely used to verify the Cox Proportional Hazards assumption [29], [36], [37].

To confirm that the Cox Proportional Hazards assumption is satisfied, we plot the scaled Schoenfeld residuals. Figure 12 shows the scaled Schoenfeld residuals for Firefox (similar results were found for Chrome, Firefox and Netbeans). From Figure 12, we find that the Cox Proportional Hazards assumption is supported, because the residuals show a random pattern against time.

We identify **overly influential subjects (files)** for removal in the next iteration of model fitting using dfbetas residuals. Figure 13 shows the dfbetas residuals for Firefox (similar results were found for Chrome, Eclipse and Netbeans). Points above the upper dashed line or below the lower line represent files that are considered outliers [41], [42].

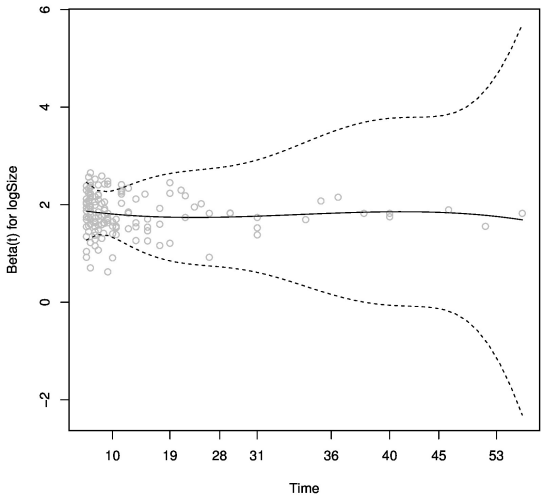
3.10 Results

Finally, we interpret the size-defect relationship. From Table 12, we find that defect density *increases* in smaller files then *decreases* in medium and larger files. This conclusion is well supported because $\hat{\beta}$ and the 95% confidence intervals of the small partitions of each project are greater than one, while $\hat{\beta}$ and the 95% confidence intervals of the medium and large partitions of each project are less than one. Therefore, defect density has an inverted "U" shaped pattern (i.e., medium-sized files have a higher defect density than small or large files).

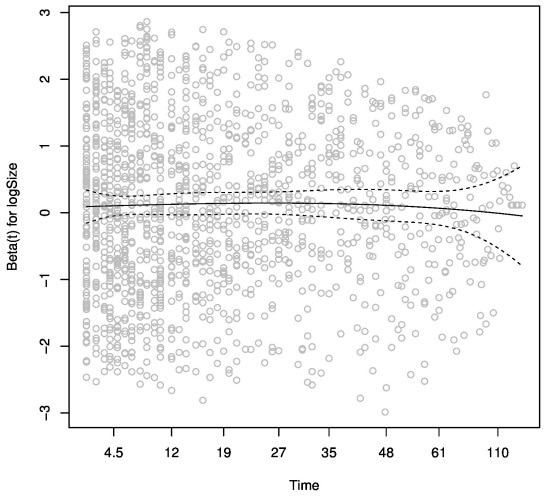
> We find that defect density has an inverted "U" shaped pattern (i.e., medium-sized files have a higher defect density than small or large files).

4 Discussion

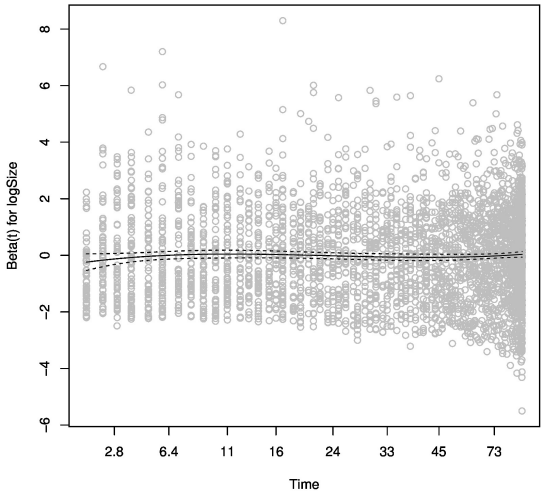
The work of Koru et al. has been instrumental in the software engineering research community's recent understanding of the size-defect relationship [7]–[9]. Therefore, in this paper, we have replicated the "theory of relative defect proneness" [8]. We have also re-evaluated the "theory of relative defect proneness" by 1) reformulating the problem to better reflect the problem of defect modelling and by 2) properly validating the underlying assumptions of the Cox Proportional Hazards model. Table 13 provides a summary of our three research questions and briefly describes our main findings. In particular, Table 13 outlines the differences between our three research questions and the original study by Koru et al. (our first research question is an exact replication of the original study).



(a) Small Partition (1-89LOC)

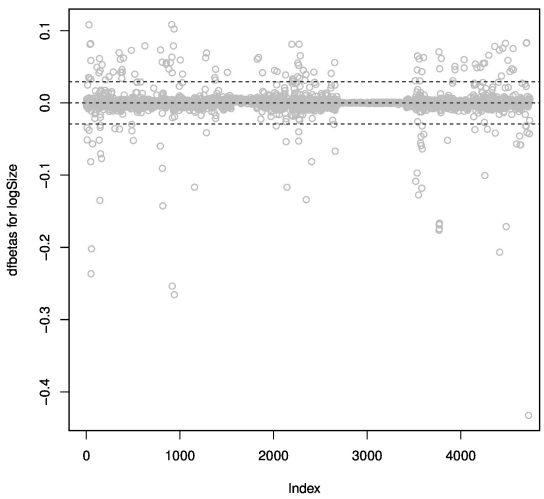


(b) Medium Partition (89-403LOC)

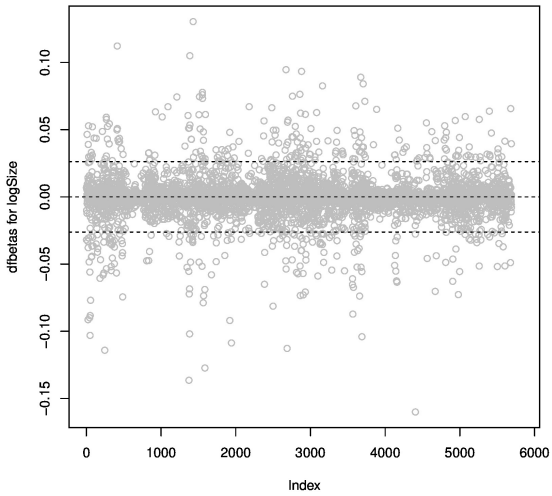


(c) Large Partition (403-38998LOC)

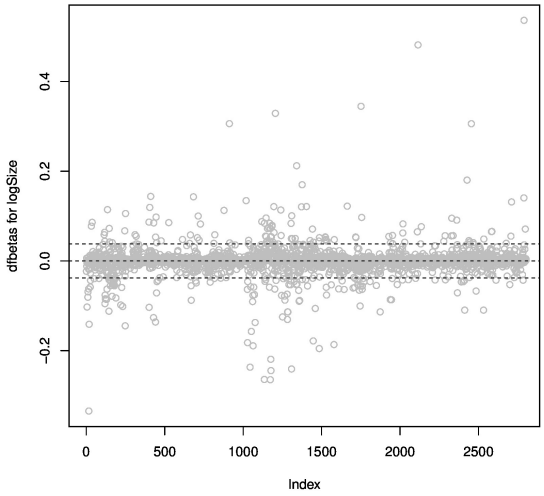
Fig. 12: Scaled Schoenfeld Residuals for Firefox.



(a) Small Partition (1-89LOC)



(b) Medium Partition (89-403LOC)



(c) Large Partition (403-38998LOC)

Fig. 13: Dfbetas Residuals for Firefox.

TABLE 13: Summary of Our Research Questions

	RQ1	RQ2	RQ3
Case Study Subjects	• KOffice projects	• Chrome • Eclipse • Firefox • Netbeans	• Chrome • Eclipse • Firefox • Netbeans
Data Formulation	• Defect fixes • Continuous time-scale	• Defect fixes • Continuous time-scale	• Defect introductions • Discrete time-scale
Model Calibration	• Link function (natural logarithm) • Stratification by the number of previous defects	• Link function (natural logarithm) • Partitioning by file size • Stratification by the number of previous defects	• Link function (natural logarithm) • Partitioning by file size • Stratification by the number of previous defects
Model Verification	• Nonproportionality test statistic (identify transform) • Dfbetas residuals	• Nonproportionality test statistic (Kaplan-Meier transform) • Scaled Schoenfeld residuals • Dfbetas residuals	• Nonproportionality test statistic (Kaplan-Meier transform) • Scaled Schoenfeld residuals • Dfbetas residuals
Results	• Defect density is highest in smaller files and decreases with file size	• The size-defect relationship was not consistent across Chrome, Eclipse, Firefox and Netbeans	• Defect density has an inverted "U" shaped pattern (i.e., medium-sized files have a higher defect density than small or large files)

5 Threats to Validity

5.1 Threats to Construct Validity

Threats to construct validity describe concerns regarding the measurement of our metrics.

The number of defects in each source code file was measured by identifying the files that were changed in a defect fixing revision. Although this technique has been found to be effective [44], [45], it is not without flaws. We identified defect fixing changes by mining the commit logs for a set of keywords (i.e., "bug," "fix," "defect" and "patch"). Therefore, we are unable to identify defect fixing revisions (and therefore defects) if we failed to find a specific keyword, if the committer misspelled the keyword or if the committer failed to include any commit message. We are also unable to determine which source code files have defects when defect fixing modifications and non-defect fixing modifications are made in the same revision. However, such problems are common when mining software repositories [46].

The data used in our third research questions (i.e., Chrome, Eclipse, Mozilla and Netbeans) was extracted from the source code repository of each project and the SZZ algorithm was used to identify defect introducing changes. Although the SZZ algorithm is currently the best algorithm for automatically identifying defect introducing changes, it is likely that not all of the defect fixing changes were mapped to defect introducing changes.

5.2 Threats to Internal Validity

Threats to internal validity describe concerns regarding alternate explanations for our results.

Our second and third research questions were addressed using file-level data (i.e., we modelled defects at the file-level), whereas the original study by Koru et al. used class-level data [8]. However, from Table 7, we find that the median class sizes in the KOffice projects and the median files sizes in Chrome, Eclipse, Firefox and Netbeans are similar. This is to be expected as most files have only one class. Therefore, measures of size at the class-level are comparable to those at the file-level. Further, Koru et al. have used the same approach to studying the functional form of the size-defect relationship (i.e., the approach we replicated in Section 3) with both class-level and file-level measures of size [47]. The authors found the same size-defect relationship regardless of whether class-level or file-level measures of size were used to build their models. Therefore, we do not believe that using file-level, as opposed to class-level, measures of size has impacted our results.

The results of our replication study, as well as the results of all survival analysis studies, depend upon how we satisfy the Cox Proportional Hazards assumption (i.e., the link functions and stratification levels). The natural logarithm link function simplifies our interpretation of the $\hat{\beta}$ coefficients in our models (i.e., the functional form of the size-defect relationship), however, that does not necessarily indicate that it is the best link function. We stratified our models across the number of previous defects, however, we may have failed to stratify our models across all confounding factors. Finally, we manually specified the partition ranges based on the plots of link functions, however, it is possible that we have not made the best choice of partition points.

5.3 Threats to External Validity

Threats to external validity describe concerns regarding the generalizability our results.

The studied projects represent a small subset of the total number of software projects available. We have also limited our replication study to open-source projects. Therefore, our results may not generalize to other projects, in particular closed-source projects. Although our replication study included a relatively small number of projects, we attempted to mitigate this issue by choosing a diverse set of projects. In particular, we chose projects from different domains (web browsers and integrated development environments) with different end users (software developers and consumers).

6 Conclusions

Our paper presented a replication study of the work of Koru et al. In particular, we paid close attention to the role of event selection (i.e., modelling defect introductions as opposed to defect fixes) and time scale specification (i.e., discrete time as opposed to continuous time scales) in determining the size-defect relationship. Although survival analysis has shown to be a promising approach to modelling defects, care should be taken when formatting defect data for such analysis. Our second and third research questions, demonstrate how using different formulations of defect data impact the results of a Cox model.

Interestingly, our findings show that defect density has an inverted "U" shaped pattern (i.e., defect density *increases* in smaller files, peaks in the largest small-sized files/smallest medium-sized files, then *decreases* in medium and larger files). This is the opposite of the Goldilocks principle, which states that the medium-sized files have the lowest defect density [24].

Our findings generally agree with the results of Koru et al., who found that defect density decreases as module size increases. Although we found that this relationship holds in medium and large-sized modules, we found that defect density increases in small modules as module size increases, whereas Koru et al. found that defect density always decreases as file size increases. Our results arise from interpreting Cox models after rigorously verifying that they satisfied the Cox Proportional Hazards assumption.

In the future, we intend to further explore the size-defect relationship by using finer grained partitions to pinpoint where defect density peaks. In our current work, we have found that defect density peaks in medium-sized files, however, we do not know exactly where this peak occurs.

ACKNOWLEDGMENT

We would like to thank Dr. Yasutaka Kamei for extracting the data used in our second and third research questions.

We would like to thank Koru et al. for contributing the KOffice dataset to the PROMISE repository [34].

REFERENCES

[1] J. Moad, "Maintaining the competitive edge," *Datamation*, vol. 4, no. 36, Feb 1990.
[2] L. Erlikh, "Leveraging legacy system dollars for e-business," *IT Professional*, vol. 2, no. 3, pp. 17–23, May 2000.
[3] A. Tosun, A. Bener, B. Turhan, and T. Menzies, "Practical considerations in deploying statistical methods for defect prediction: A case study within the turkish telecommunications industry," *Information and Software Technology*, vol. 52, no. 11, pp. 1242–1257, Nov 2010.
[4] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *Transactions on Software Engineering*, vol. 38, no. 6, pp. 1276–1304, Nov 2012.
[5] J. Ekanayake, J. Tappolet, H. C. Gall, and A. Bernstein, "Tracking concept drift of software projects using defect prediction quality," in *Proceedings of the International Working Conference on Mining Software Repositories*, May 2009, pp. 51–60.
[6] U. Raja, D. P. Hale, and J. E. Hale, "Modeling software evolution defects: a time series approach," *Journal of Software Maintenance and Evolution*, vol. 21, no. 1, pp. 49–71, Jan 2009.
[7] G. Koru, D. Zhang, and H. Liu, "Modeling the effect of size on defect proneness for open-source software," in *Proceedings of the International Workshop on Predictor Models in Software Engineering*, May 2007, pp. 115–224.
[8] G. Koru, K. E. Emam, D. Zhang, H. Liu, and D. Mathew, "Theory of relative defect proneness," *Journal of Empirical Software Engineering*, vol. 13, no. 5, pp. 473–498, Oct 2008.
[9] G. Koru, D. Zhang, K. E. Emam, and H. Liu, "An investigation into the functional form of the size-defect relationship for software modules," *Transactions on Software Engineering*, vol. 35, no. 2, pp. 293–304, Mar 2009.
[10] S. P. Jenkins, "Survival ananlysis," 2004.
[11] F. Steele, "NCRM methods review papers, NCRM/004. event history analysis," 2005.
[12] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," *Journal of Automated Software Engineering*, vol. 17, no. 4, pp. 375–407, Dec 2010.
[13] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *International Workshop on Predictor Models in Software Engineering*, May 2007, p. 9.
[14] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, Jun 1994.
[15] E. Shihab, Z. M. Jiang, W. M. Ibrahim, B. Adams, and A. E. Hassan, "Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, Sep 2010, pp. 29–39.
[16] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the International Conference on Software Engineering*, May 2008, pp. 181–190.
[17] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the International Conference on Software Engineering*, May 2005, pp. 284–292.
[18] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code! examining the effects of ownership on software quality," in *Proceedings of the Symposium and the European Conference on Foundations of Software Engineering*, Sep 2011, pp. 4–14.
[19] E. Weyuker, T. Ostrand, and R. Bell, "Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models," *Empirical Software Engineering*, vol. 13, no. 5, Oct 2008.
[20] B. T. Compton and C. Withrow, "Prediction and control of ada software defects," *Journal of Systems Software*, vol. 12, no. 3, pp. 199–207, Jul 1990.
[21] L. Hatton, "Reexamining the fault density-component size connection," *Software*, vol. 15, no. 3, pp. 89–97, Mar 1997.
[22] ——, "Does OO sync with how we think?" *Software*, vol. 15, no. 3, pp. 46–54, May 1998.
[23] C. Withrow, "Error density and size in ada software," *Software*, vol. 7, no. 1, pp. 26–30, Jan 1990.
[24] K. E. Emam, S. Benlarbi, N. Goel, W. Melo, H. Lounis, and S. N. Rai, "The optimal class size for object-oriented software," *Transactions on Software Engineering*, vol. 28, no. 5, pp. 494–509, May 2002.
[25] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *Transactions on Software Engineering*, vol. 25, no. 5, pp. 675–689, Sep 1999.
[26] J. Rosenberg, "Some misconceptions about lines of code," in *Proceedings of the international symposium on software metrics*, Nov 1997, pp. 137–142.
[27] M. Wedel, U. Jensen, and P. Göhner, "Mining software code repositories and bug databases using survival analysis models," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, Oct 2008, pp. 282–284.
[28] G. M. K. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, and Y. Zou, "Studying the impact of clones on software defects," in *Proceedings of the Working Conference on Reverse Engineering*, Oct 2010, pp. 13–21.
[29] T. M. Therneau and P. M. Grambsch, *Modelling Survival Data: Extending the Cox Model*, 1st ed. Springer, 2010.
[30] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *Transactions on Software Engineering*, vol. 26, no. 8, pp. 797–814, Aug 2000.
[31] The R project of statistical computing. [Online]. Available: http://www.r-project.org
[32] Cran - package rms. [Online]. Available: http://cran.r-project.org/web/packages/rms/index.html
[33] Cran - package survival. [Online]. Available: http://cran.r-project.org/web/packages/survival/index.html
[34] PROMISE. [Online]. Available: http://promisedata.org/
[35] F. E. Harrell, *Regression Modeling Strategies: With Applications to Linear Models, Logistic Regression, and Survival Analysis*, 1st ed. Springer, 2010.
[36] K. R. Hess, "Graphical methods for assessing violations of the proportional hazards assumption in cox regression," *Journal of Medicine*, vol. 14, no. 15, pp. 1707–1723, Aug 1995.
[37] D. W. Hosmer and S. Lemeshow, *Applied Survival Analysis: Regression Modeling of Time to Event Data*, 1st ed. John Wiley & Sons, 1999.
[38] T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, and D. Cok, "Local vs. global models for effort estimation and defect prediction," in *Proceedings of the International Conference on Automated Software Engineering*, Nov 2011, pp. 343–351.
[39] E. Vittinghoff, D. V. Glidden, S. C. Shiboski, and C. E. McCulloch, *Regression Methods in Biostatistics: Linear, Logistic, Survival, and Repeated Measures Models*, 2nd ed., 2012.
[40] T. M. Therneau, "A package for survival analysis in S," Mayo Foundation, Tech. Rep., Feb.
[41] T. V. der Meera, M. T. Grotenhuisb, and B. Pelzerb, "Influential cases in multilevel modeling: A methodological comment," *American Sociological Review*, vol. 75, no. 1, pp. 173–178, Apr 2006.
[42] J. Cohen, P. Cohen, S. G. West, and L. S. Aiken, *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*, 3rd ed., 2002.
[43] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the International Workshop on Mining software Repositories*, May 2005, pp. 1–5.
[44] A. E. Hassan, "Automated classification of change messages in open source projects," in *Proceedings of the Symposium on Applied Computing*, Mar 2008, pp. 837–841.
[45] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *Proceedings of the International Conference on Software Maintenance*, Oct 2000, pp. 120–130.
[46] A. E. Hassan, "The road ahead for mining software repositories," Oct 2008, pp. 48–57.
[47] G. Koru, H. Liu, D. Zhang, and K. E. Emam, "Testing the theory of relative defect proneness for closed-source software," *Empirical Software Engineering*, vol. 15, no. 6, pp. 577–598, Dec 2010.