Noname manuscript No.
(will be inserted by the editor)

The Impact of Code Review Measures on Post-Release Defects

Replications and Bayesian Networks

Andrey Krutauz · Tapajit Dey · Peter C. Rigby · Audris Mockus

Received: date / Accepted: date

Abstract Aim: In contrast to studies of defects found during code review, we aim to clarify whether code reviews increase long-term quality by reducing post-release defects.

Method: We replicate McIntosh *et al.*'s [45] study that uses additive regression to model the relationship between defects and code reviews. To increase external validity, we apply the same methodology on a new software project. We then investigate how to reduce the impact of correlated predictors in the variable selection process and how to increase understanding of the inter-relationships among the predictors by employing Bayesian Network (BN) models.

Context: As in the original study, we use the same measures authors obtained for Qt project in the original study. We mine data from version control and issue tracker of Google Chrome and operationalize measures that are close analogs to the large collection of code, process, and code review measures used in the replicated the study.

Results: Both the data from the original study and the Chrome data showed high instability of the influence of code review measures on defects with the results being highly sensitive to variable selection procedure. Models without

Concordia University
Montreal, QC, Canada
E-mail: andrey.krutauz@ensce.concordia.ca

University of Tennessee
Knoxville, Tennessee, USA
E-mail: tdey2@vols.utk.edu

Concordia University
Montreal, QC, Canada
E-mail: peter.rigby@concordia.ca

University of Tennessee
Knoxville, Tennessee, USA
E-mail: audris@utk.edu

code review predictors had as good or better fit than those with review predictors. Replication, however, confirms with the bulk of prior work showing that prior defects, module size, and authorship have the strongest relationship to post-release defects. The application of BN models helped explain the observed instability by demonstrating that the review-related predictors do *not* affect post-release defects directly and showed indirect effects. For example, changes that have *no review discussion* tend to be associated with files that have had many *prior defects* which in turn increase the number of post-release defects. We hope that similar analyses of other software engineering techniques may also yield a more nuanced view of their impact. Our replication package including our data and scripts is publicly available [1].

Keywords code review · inspection · statistical models · bayesian networks · visual models

1 Introduction

For decades code review has been seen as a cornerstone of quality assurance for software projects. The process evolved from a formal process with checklists and face to face meetings [20] to a lightweight and semi-formal review done via e-mails or specially designed collaboration tools [70]. The lightweight code review approach was originally used in open source software projects (OSS), because of their highly distributed nature [48,69] and has also become a common practice among commercial projects as well [67,5]. Recent studies suggest that the focus of review has shifted from early defect discovery to problem discussion and knowledge sharing [5,67,12,65,40]. It is perceived as a the major quality control mechanism to prevent defects from getting inside production code [5,13,51].

An obvious and important scientific question of whether or not the reviews actually improve software quality. To clarify such theoretical question, science resorts to replication to make it self-correcting system [77,15]. Replication helps establish if the phenomenon is dependable or idiosyncratic [71,76,4]. Our first aim is, therefore to conduct a similar-internal replication (a replication where only the experimenters varied [28,2]) of a highly reputable recent result investigating the effects code reviews have on software quality. We chose a commonly used quality measure: post-release defects. Such defects affect end-users (and vendor reputation) and are costly to repair [36].

To investigate a hypothesis-driven scientific question researchers typically use linear regression models [1] to examine the relation between code review (and other metrics) and software quality [63,41,49]. A recent award-winning work by McIntosh *et al.* [45] employed additive models to fit non-linear curves that are more suited for non-monotone or non-linear relationships than linear regression. We do an exact reproduction of the experiment reported in the work

[1] Machine learning methods are widely used for defect prediction, but such methods can not be used to test scientific hypothesis.

to determine if we can obtain the same conclusions using methods and data reported there, *i.e.* we construct OLS models with restricted cubical splines to model non linearity.

Our second goal is to increase external validity [28] of the results to avoid conclusions that are unique to the specific dataset reported in the paper. To accomplish that we apply exactly the same set of methods on a different software project: Chromium. This is sometimes referred as differentiated-external replication [2]. We chose the project due to its size and richness and quality of the associated data that allowed us to obtain measures highly similar to ones obtained in the Qt project of the replicated study. More specifically, we model software defects that are reported in a bug tracker. As control variables, we use many of the previously studied measures that have been shown to impact defects, including size, complexity, churn, authors, and file ownership [9,33]. The focus of this study is on code review measures many of which have been examined in past studies, including the number of reviewers, discussion length, and rushed review. [45,44,41,69,68].

Our third goal is to address methodological limitations of linear regression and additive models when applied to datasets that have high correlations among the predictors as is typical software engineering data in general and in code review data in particular [68,45]. The linear (or additive) models can not reliably determine which of the highly correlated predictors is affecting the response (quality). Best practices in empirical studies that employ regression models, therefore, dictate removal of highly correlated variables, or variables that do not contribute to the explanatory power of the model. The selection of variables is largely based on the subjective opinion of the researcher. Another shortcoming of such models is their inability to model the relations among predictors, which may reveal salient aspects of the development process by providing a rich picture of how the predictors may influence each other and the response. A Bayesian Network (BN) is a Probabilistic Graphical Model (PGM). PGM describes probabilistic relationships among variables that describe a problem domain [34]. This model has several advantages over linear or additive regression models. In particular, it allows for a natural representation of conditional dependence and independence using graph notation where variables are nodes and dependencies are edges. The removal of the notion of predictor and response variables disposes of the oversimplifying assumption that a single response variable is explained by a long list of predictors. Instead the edges in the Bayesian Network provide a meaningful structure based on collected data. Each variable in a graph can be interpreted as a predictor or a response variable based on the topology of the graph. The researcher can then inject information to understand the impact of an edge of interest [26].

Our main findings related to the first aim — the reproduction of the study by McIntosh *et al.* [45] — have demonstrated high sensitivity of the results to the subjective steps in the analyses when data contains highly correlated predictors. In particular, we found that even in exact reproduction we were unable to confirm the importance of code review measures on reduction of defects. Moreover, the results are inconsistent across software releases and heavily depend on the variables selected. We did, however, find several metrics not related to code reviews, such as churn or prior defects, that were reproduced reliably despite the subjectivity of the variable selection process.

The investigation of the second objective increased the external validity of the findings by confirming that a relatively small set of measures are related to post-release defects on a large and unrelated software project. As on the Qt dataset, the impact of review measures was inconsistent across software releases and heavily depend on the variables selected.

To reduce the subjectivity of variable selection process and to untangle the complex web of dependencies among the predictors we applied Bayesian Networks (BN) on both datasets. The approach revealed that there is no direct relation between review measures and defects. The graph shows, for example, that modules with more self-approved changes also have more changes with no discussion, more reviewers, and also more prior defects. Increase in review issues increases the share of the work done by the minor authors, which, in turn, is associated with increased number of defects.

This paper is organized as follows. In Section 2, we discuss the case study design, the systems under study, and data extraction process. We also give a brief overview of the Chrome code review process. In Section 3, we replicate McIntosh *et al.* [45] study, describe the model construction, results, and discussions. In Section 4, we describe BNs and discuss the findings from these models. Threats to validity are discussed in Section 6. The final section concludes the paper and suggests future work. Our replication package including our data and scripts is publicly available [1].

2 Case study design and data

In this section we discuss the case study design including the projects under study and reasons for their selection. We describe the data sources, steps in the data extraction, and analysis approach. We discuss the Bayesian Network modeling methodology in Section 4.

2.1 Systems under study

McIntosh *et al.* [45] mined code review data from Android, LibreOffice, QT, ITK, and VTK. They did not conduct an analysis on Android and LibreOffice because they found that many of the reviews were not linked to bug reports which did not allow them to study the impact of review on bugs. In total, they studied two QT releases and one release for VTK and ITK. For the reproduction, McIntosh provided the Qt and ITK data and was used in their work [45]. The ITK data had only 24 defective components and 344 commits with reviews. We feel that this dataset is too small to produce meaningful statistical models. Although we show the ITK results in our replication package [1], we only present the Qt results in this work. To improve external validity, we replicate the study on the Google Chrome project, because like QT, it is large and

Table 1 Description of Measures

	Measure	Description
Product	Size	Number of lines of executable code in component
	Complexity	The McCabe cyclomatic complexity.
Process	Prior defects	Number of defects fixed in component prior to considered period
	Effective tests(Chrome only)	Total number of times a test found an issue during the review process
	Churn	Sum of added or removed lines of code per component during considered period of time
	Change entropy	Distribution of changes among files within a component
Human factors	Minor authors	Number of unique contributors that contribute less than 5% of code changes to component
	Major authors	Number of unique contributors that contribute at least 5% of code changes to component
	All authors	Number of unique contributors to component
	Author ownership	Proportion of changes to component done by major authors
Participation	Rushed reviews	Number of reviews that were concluded faster than acceptable review rate (200 loc per hour)
	Changes without discussion	Changes that were integrated without discussion comments
	Self approved changes	Changes that were approved for integration only by submitter himself
	Typical discussion length	Discussion length typical for that specific component measured in number of discussion comments. Normalized by size of change(churn)
	Typical review window	Typical amount of time that passes between patch was uploaded till it is approved for integration. Normalized by size of change(churn)
	All reviews	Total number of times the component was reviewed
	All reviewers	Total number of of reviewers that reviewed a component
	Review issues	Total number of patch revisions created during review process
	Effective reviews(Chrome only)	Number of revisions that led to a code change during a single review per component.
Expertise	Lacking subject matter expertise	Number of changes that were not authored or approved by major author.
	Typical reviewer expertise	Total number of changes to the component authored or reviewed by this reviewer prior to this change

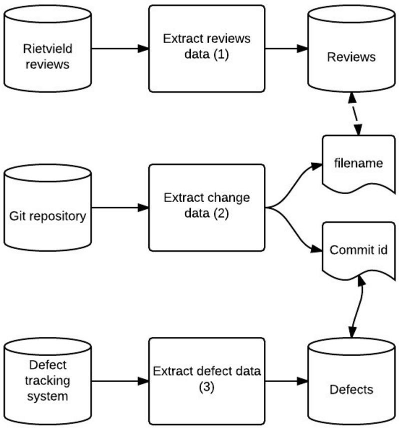

Fig. 1 Extraction methodology and data sources

primarily written in C++. A further reason for studying Chrome is that it is an open source web browser, that is mostly developed by paid Google developers and its development practices mirror those used internally at Google. Chrome developers are required to perform code review on each change and use Reitvield, the precursor to Gerrit, to improve traceability of bugs, changes, and reviews.

For completeness, we briefly describe Chrome's code review process which resembles other modern review practices [66]. A review begins with the change author submits a patch to invited reviewers. A reviewer examines a change and either approves it by replying with special keyword *lgtm* (looks good to me) or proposes improvements. The author addresses all comments either by fixing an issue in code or replying to reviewer comments. Subsequent modifications to the original patch appear in the same review and are called *patchsets*. The new patchset triggers a new cycle of review and revision. The process continues until all issues are fixed and the reviewers are satisfied with the patch. The code can then be merged to the trunk.

2.2 Chrome data extraction

In order to be able to predict the influence of code review on defects we need to create a link between code review conducted on a specific system component and future defects. To this end, we collect data from three data sources: Reitvield, Git repository, Chrome bug tracker (figure 1). Data extraction is divided into three major steps as described below.

Extracting review data: We leverage an API provided by Reitvield to download code review tickets in JSON format and extract the data into a database. For each code review patch revision we extract the unique identifier and the set of files modified by this revision. For every file and revision we also capture the number of added/removed lines to calculate the size of a change. We process the reviewers comments. We ignore comments that were added automatically or provided by the author.

Extracting Git repository information: We extract commit information *i.e.* the commit hash and list of files related to the change from the Git repository. We use the Understand static analysis toolkit[2] to extract source code measures from the files.

Extracting defect data: We mine the defects from the Chrome issue tracker. We extract the submission date, type of the issue, review ID and commit ID for the fix.

Post-release defects: We consider a defect to be the post-release defect of the current release if it was submitted during the time period between the release dates of the current and the following releases. We use Chrome release calendar website for release dates information.[3] Following McIntosh [45], we associate the post-release defects with the pre-release reviews and other source measures using first the file level and then sum the measures to the component, *i.e.* directory level. This is done to reduce the fraction of zero observations, because the majority of the files in the system do not have any defects.

2.3 Collected Measures

The measures we collect to evaluate the impact of code review on post-release defects are well known and have been used in multiple past studies [45,44,67, 41] and are described in Table 1. We divide them into four categories: product, process, human factors, review participation, and reviewer expertise. [4] The code review measures are the number of reviewers, discussion length, rushed reviews, typical reviewer expertise, etc. The control variables in our model are also well known and widely used with defect prediction models [48,33,9,30]. The control variables are the size of the file, the number of prior defects, the churn, etc.

3 Code Review Reproduction and Replication Study

We replicate the study published by McIntosh *et al.* [45]. We strictly follow the steps of the model construction described in the original paper [45]. We

[2] https://scitools.com/

[3] https://www.chromium.org/developers/calendar

[4] We do not include review code coverage measures as these variables were determined to not consistently predict post release defects and were excluded from the study under reproduction

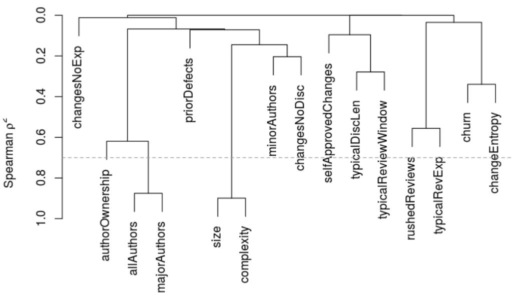

Fig. 2 Hierarchical Correlation Analysis for Qt 5.0.

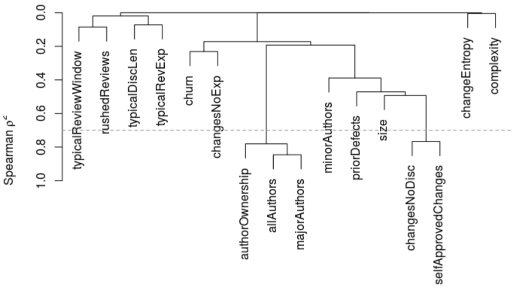

Fig. 3 Hierarchical Correlation Analysis for Chrome 40

fit an Ordinary Least Squares (OLS) regression model. Since the dependent variable is the number of post-release defects and it is highly skewed, we log transform it. We also create regression models that take non-linear effects into account. We then compare the goodness of fit among models and discuss the contribution of each independent variable.

Correlated variables can distort the contribution of a variable to a model and must be removed. We use the hierarchy clustering analysis (Figures 2 and 3) with the threshold of $|\rho| \geq 0.7$ suggested by McIntosh [45] to identify the highly correlated variables. Then we decide which variables will be discarded using drop one analysis and parsimony principles. The results of this step are summarized in the Tables 2 and 3.

Redundant variables can be explained by the other variables, that is they do not contribute to the explanatory power of the model and should be removed. Such variables may be overlooked by the pairwise correlation analysis, therefore we use *redun* function from *rms* R package [32]. For each independent variable a regression model is created using the the other variables. If the model has a R^2 greater than 0.9, then the current variable is redundant as the other variables clearly explain its contribution to a statistical model.

Non-linear effects and degrees of freedom. Traditional defect prediction models assume linear dependencies between the dependant and independent variables. McIntosh *et al.* [45] showed that for some code review measures this relation has a non-linear shape. To identify variables that may be nonlinear, we calculate the Spearman multiple ρ^2 scores for each independent variable. Variables with higher scores are more likely to be non-linear. To fit a non-linear curve we use restricted cubic splines in the *rms* R package [32]. Using this approach we assign knots, which are points where slope changes, to potentially non-linear variables. The more knots that added the greater the curve complexity. Every additional knot requires a degree of freedom. If we use all of the degrees of freedom, then there will be a knot for each data point and the fit will be perfect, but the model will be over fitted to the data. As a result, the degrees of freedom are budgeted to avoid over fitting while still allowing variables that have a high ρ^2 score to be modelled non-linearly.

To *assess model fitness*, we report the adjusted R^2 to compensate for a large number of variables [45]. To assess the individual contributions of each variable, we report its statistical significance and the Wald χ^2 maximum likelihood test value. The larger the value the greater the impact the variable has on the model. All the results are summarized in the tables 2, 3.

3.1 Variable selection and model construction

In our summary table we have approximately 1.3k reviews for Qt 5.0 and 1.4k for Chrome 40. We start with 16 measures. This gives us 81 and 87 degrees of freedom for Qt and Chrome respectively. Following previous works, we discard measures with correlation at or above 0.7. We use clustering analysis to identify these measures, see Figures 2 and 3. We also perform *drop one* analysis to determine which measures should be discarded from each cluster. *Major authors*, *author ownership* and *complexity* were removed from Qt dataset. *Major authors*, *author ownership* and *changes without discussion* were removed from Chrome. Using a redundancy test *minor authors* was removed from both datasets. We perform a non-linearity analysis. Variables that exhibit a higher degree of non-linearity require additional degrees of freedom to model their curved line. The results of the variable selection process and the number of allocated degrees for each variable can be found in in Tables 2 and 3.

To represent our models, we use the R language notation [61]. For example, the formula $y \sim a + b$ means that the response y is modelled by explanatory variables a and b. McIntosh used the following model for Qt:

$$\begin{aligned}
&log(defects+1) \sim rcs(size,5) + rcs(all\ authors,5)\\
&\quad + complexity + churn + rcs(change\ entropy,3)\\
&\quad + rcs(changes\ w/o\ discussion,3)\\
&\quad + rcs(self-approved\ changes,5)\\
&\quad + rcs(typcal\ discussion\ length,5)\\
&\quad + rcs(typical\ reviewer\ expertise,5) + rcs(lacking\ subject\ matter\ expertise,5)
\end{aligned}$$

Our final formulas for Qt and Chrome respectively:

$$\begin{aligned}
&log(defects+1) \sim rcs(size,5) + rcs(prior\ defects,5)\\
&\quad + rcs(churn,3) + rcs(changeentropy,3)\\
&\quad + rcs(all\ authors,5) + rcs(changes\ w/o\ discussion,5)\\
&\quad + self-approved\ changes + typcal\ discussion\ length\\
&\quad + typcal\ review\ window + rcs(lacking\ subject\ matter\ expertise,3)\\
&\quad + rcs(typical\ reviewer\ expertise,5) + rcs(lacking\ subject\ matter\ expertise,5)\\
&\quad + typical\ reviewer\ expertise
\end{aligned}$$

$$\begin{aligned}
&log(defects+1) \sim rcs(size,5) + rcs(prior\ defects,5)\\
&\quad + complexity + rcs(churn,3) + rcs(change\ entropy,3)\\
&\quad + rcs(all\ authors,5) + rcs(self-approved\ changes,5)\\
&\quad + typical\ discussion\ length + rushed\ reviews\\
&\quad + typcal\ review\ window + rcs(lacking\ subject\ matter\ expertise,3)\\
&\quad + rcs(typical\ reviewer\ expertise,5) + rcs(lacking\ subject\ matter\ expertise,5)\\
&\quad + typical\ reviewer\ expertise
\end{aligned}$$

We fit OLS model using formulas from above and calculate adjusted R^2 to assess goodness of fit. Tables 2 and 3 summarize the results.

3.2 Model results and model comparisons

In this section we compare our reproduction and replication results with those from McIntosh *et al.* [45] original study. We highlight differences and discuss their possible causes.

Table 2 Post-release defects prediction model for Qt. Original and reproduction study results.

Release		5.0 McIntosh		5.0		5.1 McIntosh		5.1	
Nonlinear model adjusted R^2		0.69		0.62		0.46		0.66	
Linear model adjusted R^2				0.61				0.63	
Nonlinear model w/o codereview variables adjusted R^2				**0.61**				**0.64**	
Overall D.F.		80		81		78		81	
Allocated D.F.		22		22		24		22	
		Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear
Size	D.F.	4	3	4	3	2	1	4	3
	χ^2	110***	76***	60***	14**	10**	5*	47***	35***
Complexity	D.F.	1	na	†	†	1	na	†	†
	χ^2	1°	na	†	†	< 1°	na	†	†
Prior defects	D.F.	‡	‡	2	1	2	1	3	2
	χ^2	‡	‡	48***	45***	9*	< 1°	90***	77***
Churn	D.F.	1	na	2	1	1	na	2	1
	χ^2	1°	na	15***	7***	< 1°	na	5°	3°
Change entropy	D.F.	2	1	1	na	2	1	2	1
	χ^2	8*	7**	< 1°	na	6*	6*	3°	2°
All authors	D.F.	‡	‡	3	2	2	1	2	1
	χ^2	‡	‡	274***	63***	30***	15***	193***	13***
Minor authors	D.F.	†	†	‡	‡	1	na	‡	‡
	χ^2	†	†	‡	‡	2°	na	‡	‡
Major authors	D.F.	†	†	†	†	†	†	†	†
	χ^2	†	†	†	†	†	†	†	†
Author ownership	D.F.	†	†	†	†	†	†	†	†
	χ^2	†	†	†	†	†	†	†	†
Self-approved	D.F.	2	1	1	na	1	na	1	na
	χ^2	22***	1°	7*	2°	< 1°	na	< 1°	na
Rushed reviews	D.F.	†	†	2	1	2	1	2	1
	χ^2	†	†	4°	< 1°	48***	23***	3°	< 1°
Changes w/o disc.	D.F.	2	1	2	1	2	1	2	1
	χ^2	6°	4*	18**	3°	3°	1°	36***	29***
Typical review window	D.F.	†	†	1	na	†	†	1	na
	χ^2	†	†	< 1°	na	†	†	< 1°	na
Typical disc. length	D.F.	4	3	1	na	2	1	1	na
	χ^2	26***	24***	< 1°	na	32***	21**	3°	na
Lacking subject matter expertise	D.F.	2	1	2	1	4	3	1	na
	χ^2	80***	70***	33***	3°	34***	22**	< 1°	na
Typical reviewer expertise	D.F.	4	3	1	na	2	1	1	na
	χ^2	26***	24***	< 1°	na	32***	21**	7**	na

Discarded during: † - Removed during correlation analysis; ‡ - Removed during redundancy analysis
Statistical significance: $***\rho < 0.001$; $**\,\rho < 0.01$; $*\rho < 0.05$; $\circ\rho >= 0.05$
Other: na - not used

Table 3 Post-release defects prediction model for Chrome

Release		39		40		41		42		43		44	
Nonlinear model adjusted R^2		0.61		0.58		0.59		0.59		0.51		0.53	
Linear model adjusted R^2		0.59		0.54		0.56		0.57		0.50		0.49	
w/o codereview variables adjusted R^2		**0.60**		**0.58**		**0.59**		**0.59**		**0.49**		**0.53**	
Overall D.F.		62		87		90		84		83		80	
Allocated D.F.		26		21		23		20		19		20	
		Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear
Size	D.F.	4	3	2	1	4	3	2	1	2	1	2	1
	χ^2	28***	3°	1°	1°	10*	< 1°	1°	< 1°	3°	2°	8*	6*
Complexity	D.F.	1	na	1	na	1	na	1	na	1	na	1	na
	χ^2	< 1°	na	3°	na	< 1°	na	< 1°	na	1°	na	< 1°	na
Prior defects	D.F.	4	3	2	1	4	3	4	3	4	3	2	1
	χ^2	43***	42***	37***	34***	74***	71***	61***	55***	21***	20***	3°	3°
Churn	D.F.	2	1	2	1	2	1	2	1	2	1	2	1
	χ^2	14**	8*	23***	22***	31***	30***	26***	24***	< 1°	< 1°	11**	2°
Change entropy	D.F.	2	1	1	na	1	na	1	na	1	na	1	na
	χ^2	< 1°	< 1°	< 1°	na	< 1°	na	< 1°	na	3*	na	< 1°	na
All authors	D.F.	3	2	4	3	4	3	3	2	3	2	3	2
	χ^2	49***	2°	43***	8*	42***	2°	33***	< 1°	51***	4°	24***	1°
Minor authors	D.F.	‡	‡	‡	‡	‡	‡	‡	‡	‡	‡	‡	‡
	χ^2	‡	‡	‡	‡	‡	‡	‡	‡	‡	‡	‡	‡
Major authors	D.F.	†	†	†	†	†	†	†	†	†	†	†	†
	χ^2	†	†	†	†	†	†	†	†	†	†	†	†
Author ownership	D.F.	†	†	†	†	†	†	†	†	†	†	†	†
	χ^2	†	†	†	†	†	†	†	†	†	†	†	†
Self-approved	D.F.	4	3	4	3	4	3	4	3	4	3	4	3
	χ^2	8*	7°	12*	5°	2°	2°	3°	2°	13**	9**	9*	3°
Rushed reviews	D.F.	1	na	1	na	1	na	1	na	1	na	1	na
	χ^2	1°	na	19***	na	2°	na	14**	na	6*	na	< 1°	na
Changes w/o disc.	D.F.	†	†	†	†	†	†	†	†	†	†	†	†
	χ^2	†	†	†	†	†	†	†	†	†	†	†	†
Typical review window	D.F.	1	na	1	na	1	na	1	na	1	na	1	na
	χ^2	1°	na	1°	na	< 1°	na	< 1°	na	< 1°	na	< 1°	na
Typical disc. length	D.F.	1	na	1	na	1	na	1	na	1	na	1	na
	χ^2	< 1°	na	< 1°	na	< 1°	na	< 1°	na	< 1°	na	< 1°	na
Lacking subject matter expertise	D.F.	2	1	2	1	2	1	2	1	1	na	2	1
	χ^2	7*	3°	3°	< 1°	32***	7***	20***	5*	7**	na	19***	10**
Typical reviewer expertise	D.F.	1	na	1	na	1	na	1	na	1	na	1	na
	χ^2	< 1°	na	< 1°	na	< 1°	na	< 1°	na	7*	na	10*	na

Discarded during: † - Removed during correlation analysis; ‡ - Removed during redundancy analysis
Statistical significance: $***\rho < 0.001$; $**\rho < 0.01$; $*\rho < 0.05$; $\circ\rho >= 0.05$
Other: na - not used

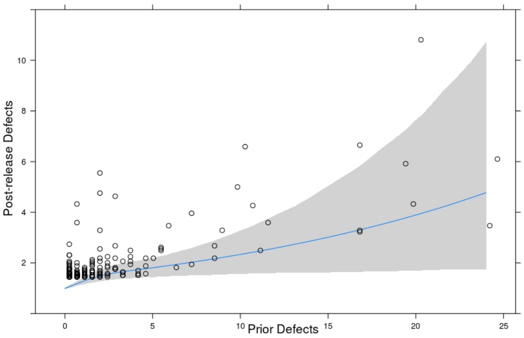

Fig. 4 A wide margin of error for nonlinear predictions, for example, prior defects in Qt 5.0

3.3 Comparing linear and non-linear models

To illustrate the nonlinear effect we select an independent variable with the highest potential of nonlinearity from the model and calculate predicted number of post-release defects as the function of this variable, using *Predict* function from R rms package. The rest of the variables are fixed at their median values. As an illustration, we choose *prior defects* for Qt it has the highest Spearman squared value among independent variables participating in the model. We then plot the results in Figures 4. Although the shape of the plot may suggest some nonlinearity, the grey funnel, which is the error margin, is too wide to claim with confidence that these variables have a nonlinear relation with the response variable. The goodness of fit R^2 also shows that nonlinear models do not yield better results than regular linear models and do not justify the additional complexity associated with nonlinear models.

3.4 Models with and without review measures

The results show that although many of the code review measures are statistically significant, they usually tend to have lower values of Wald χ^2 test than other measures, suggesting their lower contribution to explanatory power of the model. Even the most prominent measures, like *typical discussion length* and *rushed reviews* are repeatedly outperformed by measures like *size*, *prior defects*, and *all authors*. As a further investigation, we fit a model *without* review measures and record the values of adjusted R^2 (shown in bold in the section of R^2 values in the Tables 2, 3). The decrease in the values of adjusted R^2 in both datasets is minimal, meaning that overall contribution of the review measures to the explanatory power of the model is nominal. In addition to low contribution to the model the performance of review variables is inconsistent between datasets and releases. For instance, in McIntosh *et al.* *rushed reviews* was discarded from the model in Qt 5.0 during the correlation analysis, however, in Qt 5.1 this measure is one of the strongest variables of the model. The *typical discussion length* is one of the most influential variables for both Qt releases in both studies, but in Chrome dataset the contribution of this variable is insignificant.

Conclusion 1: *The review measures contributed little to the performance of the model, with the R^2 remaining practically unchanged from the model that included only the traditional predictors, such as the number of prior-defects, size, and authors.*

3.5 Impact of individual variables

Size of component is a well-known predictor in empirical software studies. McIntosh *et al.* show that in the Qt project *size* provides significant contribution to the explanatory power of the model. Our result is similar for Qt dataset. However, in Chrome dataset the contribution of the *size* measure is quite small (Table 3).

Prior defects and all authors have been shown to be a good predictors of future defects [30,9]. McIntosh *et al.* discard *prior defects* in Qt 5.0 due to redundancy. In our study, the redundancy analysis on the Qt 5.0 dataset does not indicate that *prior defects* are redundant. For the Qt 5.1 release, both McIntosh *et al.* and our model keep *prior defects* but find it to be a poor predictor. The *all authors* measure is redundant in Qt 5.0 release in our study contrary to the McIntosh *et al.*. For Qt 5.1 the *all authors* is the most influential predictor in the model. This result is consistent for both studies. In Chrome dataset these two variables are repeatedly found to be the most influential variables of the model. A possible explanation for this inconsistency could be that these two variables share a common cause. Defects are not always fixed by the owner of the module, especially in big teams. That means that more developers are touching the file, and the more developers modifying a file the higher the risk of the future defects. Intuitively, the growth in these two measures should be related, but our correlation and redundancy analysis fails to find this. These inconsistent results suggest that traditional variable selection techniques are lacking and indicate the need for a different approach that can deal with complex interactions between variables.

Review measures . The important measures in the Qt dataset are similar to what McIntosh *et al.* found. The *self-approved changes* has low impact on post-release defects. The *rushed reviews* variable was discarded in 5.0 release, but in 5.1 it appears as one of the most influential variables. The *typical discussion length* variable has moderate to strong influence in both releases. For the Chrome dataset the review measures are not statistically significant in most cases. When they are, such as in *lacking subject matter expertise* the result is inconsistent across releases. The overall performance of review variables is inconsistent in both studies. We continue to discuss the issues with use of review variables in the following section.

Conclusion 2: *The inconsistent performance across projects and releases of strong predictors, like* all authors*,* prior defects *and others, suggests a possible issue with the traditional variable selection approach and indicates the need for an approach that is capable of dealing with a more complex inter-variable relations.*

4 Application of Bayesian Networks models

To address the concern of the surprising absence of the relationship between code reviews and post-release defects demonstrated in previous models, and the lack of reproducibility due to the subjectivity in variable selection approaches that are necessary in a traditional model, such as the one used in Section 3, we use Bayesian Networks(BN) as an alternative modeling approach. Our goal in using the BN model is not to create the best predictive model for post-release defects, instead, we focus on understanding the complex interaction between the variables described by the data, and finding which variables directly impact the number of post-release defects in such a generative model[5].

4.1 Bayesian Networks

A Probabilistic Graphical Model is a mathematical way to encode the probabilistic relationships such as conditional independence and dependence among the random variables. The nodes of PGM are the random variables and the edges represent probabilistic dependence. PGM tells us how we believe the variables are related in our (probabilistic) generative model. Such a model, with all variables of importance, can provide an understanding of the underlying mechanics, and help to discover if the review measures are related to defects, and to understand which variables play a major role in software quality.

PGMs come in two varieties, Bayesian networks (which we use in this paper) representing directed acyclic graphs (DAGs) and Markov random fields representing undirected graphs. They differ in the set of independencies they can encode and the factorization of the distribution that they induce [39].

One important concept related to the BNs is the concept of *Markov Blanket* [57]. The Markov Blanket for a node in a Bayesian Network is the set of

[5] A generative model specifies a joint probability distribution over all observed variables, whereas a discriminative model provides a model only for the target variable(s) conditional on the predictor variables. Thus, while a discriminative model allows only sampling of the target variables conditional on the predictors, a generative model can be used, for example, to simulate (i.e. generate) values of any variable in the model, and consequently, to gain an understanding of the underlying mechanics of a system, generative models are essential.

nodes composed of its parents, its children, and its children's other parents (co-parents). The Markov blanket of a node contains all the variables that shield the node from the rest of the network *i.e.* for a node A, its Markov Blanket MB_A, and a node $B : B \neq A, B \notin MB_A$, we have the property that:

$$Pr(A|MB_A, B) = Pr(A|MB_A)$$

This means that the Markov blanket of a node is the only knowledge needed to predict the behavior of that node.

There are two primary ways of constructing BN models. In the first approach the graph represents dependencies is obtained from domain experts. The graph may include prior distributions about the parameters of the overall model. The data is then used to calculate the posterior distribution and to make inference. The second approach puts minimal a-priori assumptions about the model and focuses on the search for the best graphical representation for a given dataset (structure learning). This is an NP-hard problem [16], but a number of different heuristic structure learning algorithms are available.

Bayesian Networks models have several advantages over regression models. To be precise, regression analysis is a very simple BN where there is one directed link from each independent variable to dependent variable. BNs, therefore, can help with multicollinearity by linking independent variables. In process of BN construction we can control the number of edges (relations) by specifying a connection strength threshold. Once the Bayesian Network is constructed we can use the graphical representation to learn about less obvious interactions among variables and infer how the injection of specific facts affects variables of interest. We use BN to investigate the lack of consistency in the replication in the previous sections.

For simplicity, we will assume no latent or hidden variables, i.e., we assume that the set of variables we use fully describes our problem domain (discussed further in Section 6).

4.2 Graphical model construction

Here we describe the procedures used to create our model and the reasoning behind them. Despite the promises of BNs, they tend to be quite sensitive to data, and operational data is often problematic [47,86]. Careful preprocessing, therefore, is needed to ensure a reliable and reproducible result. We found that all our variables have a long-tailed distribution that could not be corrected even by a log-transformation. Since BN structure learning methods for continuous data require a normal distribution, we discretize the data as is often done in prediction that involves classifiers as in this case the prediction whether or not the file will have a post-release defect fixed.

4.2.1 Discretization

Discretizing variables while preserving relationships among them is an NP-hard problem [17], but several heuristics exist. To obtain a generative model, we had to use an unsupervised discretization method. The added benefit is that the discretization was totally response-variable agnostic unlike supervised discretization methods that are commonly used. This prevents any bias towards better fit that may accompany supervised methods. The commonly used supervised methods optimize discretization to improve explanatory power for a single response variable, as for example, Chi-square, or MDLP. This is not suitable for a BN structure search, because we do not know *a-priori* which variables will be responses (have arrows pointing to them) and which will be independent (have no incoming arrows). While some research on multidimensional discretization methods exist [59] we are not aware of any such method that has a robust implementation. We, therefore, use unsupervised discretization methods. Based on the recommendations in [27]), we chose the Equal Frequency method, implemented in the *arules* package [31]. For ease of understanding, the "defects" node was discretized to a binary no-defect/defect variable, since ~75% of the data has no defects. Based on the distribution of the data, three levels were deemed appropriate for the remaining variables, except variables representing minor authors, rushed reviews, typical review window, and Lacking subject matter expertise which were discretized into two levels, since more than 70% of entries were 0. We present the distribution of the variables (for combined Chrome and Qt data) in our replication package [1] as additional evidence for the choice of our discretization levels.

4.2.2 Structure Learning

To learn our structure we chose a well-performing and widely used Hill-Climbing (HC) algorithm, which is the best BN structure learning algorithm in terms of accuracy [18] and runtime. We used the implementation of the algorithm as available in the *bnlearn* R package. Implementation details for this algorithm are outside the scope of this paper.

The HC algorithm attempts to maximize network score with several scoring functions available in *bnlearn* package: *e.g.,* BIC, AIC, BDE. A detailed study [14] examining how well different scores performed concluded that in general all scores perform similarly and for large data sets Bayesian scores more suitable. Since our dataset is not particularly large, at least for the individual releases, we decided not to use Bayesian scores *e.g.,* BDE, instead we chose to focus on the Information theoretic scores *e.g.,* AIC, BIC. We finally used the BIC score because it is more appropriate for constructing explanatory models, while AIC is better suited for building predictive models [79,75].

Hill-Climbing has the known limitation of finding a local maxima, and there are several enhanced versions of the algorithm that deal with this shortcoming. For example, Stochastic Hill-Climbing, Random Walk, Hill-Climbing with Simulated Annealing. The R implementation provides the number of restarts and perturbations as tuning parameters. Restarts represent the number of random restarts, and the number of attempts to randomly insert/remove/reverse an arc on every random restart is specified by the perturb option.

The results, structure and parameters are often noisy, meaning that different settings induce slightly different networks. To mitigate this effect, we use the non-parametric bootstrap model averaging method described in [26], which provides confidence levels for both the existence of an edge and its direction. This enables us to select a model based a confidence threshold. Friedman *et al.* [26] argue that threshold is domain specific and needs to be determined for each domain. To identify a suitable threshold we performed a simulation study, by generating data for the same number of nodes. The result of the simulation showed that a threshold of 0.85 was suitable to accurately recover the original structure. We also investigated alternative thresholds to assess the stability of the results as described in Section 6.

4.2.3 Fitted BN model

We first discovered the structure of Chrome and Qt datasets separately for each release. As expected, there were some differences between the models for the different releases. However, the Markov blankets of the "defects" node were consistently devoid of the review measures. We were interested in finding which variables affect the "defects" node across releases, so we decided to combine the datasets, creating an aggregate dataset for all Chrome releases, a similar aggregate dataset for all Qt releases, and a combined dataset with all Chrome and Qt releases.

We had different variables appearing in the Markov Blanket of the "defects" node in each release and the combined releases, although some variables were appearing more frequently (*e.g. priordefects*) than others, as can be seen from Table 4. Although review related measures like *allreviews*, *allreviewers* appeared in the Markov Blanket of the "defects" node for Chrome release 39 and 40 respectively, they appeared only in those instances, and their effect wasn't seen in the aggregated datasets, indicating their effects are comparatively small and didn't affect the number of post-release defects across the releases. It is also to be noted that the "defects" node appeared without any child node in all of our models automatically, which in turn reinforces our belief in the generated models and acts as additional evidence for the suitability of our our selected structure learning algorithms and choice of discretization method.

To conserve space in Figure 5 we present the most relevant part of the BN model for the combined Chrome and Qt datasets containing only links touching the Markov Blanket for the "defects" node. The BN models for individual Chrome and Qt releases, the aggregate Chrome and Qt datasets, and the full model of the combined dataset are available in our replication package [1] repository.

The dotted edges indicate that the coefficient is negative for that edge, *i.e.* increasing the value of the parent node decreases the value of the child node and vice versa. The immediate parents of the "defects" node (consequently, the Markov blanket for the "defects" node in this case) are colored in light blue and the "defects" node has a rectangular shape.

Table 4 Variables in Markov Blanket of "defects" for different releases

Release	Variables in Markov Blanket of "defects"	Precision	Kappa
Qt_50	priordefects, changesnodisc	0.89	0.46
Qt_51	allauthors, size	0.85	0.46
Qt_combined	priordefects, changesnodisc	0.87	0.42
Chrome_39	allreviews, priordefects	0.76	0.5
Chrome_40	allreviewers	0.78	0.49
Chrome_41	allchangescount, priordefects	0.77	0.51
Chrome_42	priordefects	0.74	0.43
Chrome_43	minorauthors, reviewissues	0.76	0.42
Chrome_44	allchangescount, minorauthors	0.8	0.48
Chrome_combined	minorauthors, priordefects	0.75	0.39
All	minorauthors, priordefects, size	0.81	0.46

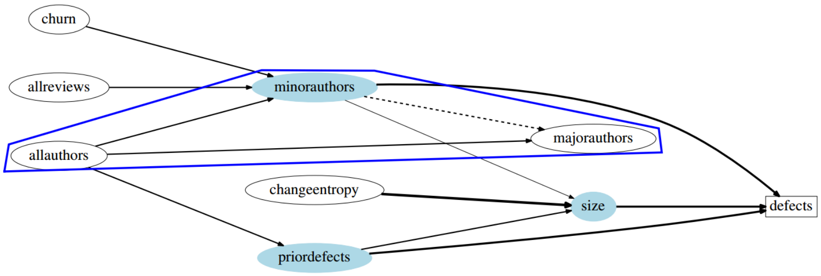

Fig. 5 Snapshot of Resultant Bayesian Network from combined Chrome and Qt data

4.2.4 Model fit

Models can be evaluated based on their predictive and explanatory power and these two objectives are radically different [75]. Our main goal is to develop a model that explains the data, in contrast to finding the most accurate predictor of defects. Explanatory power (the magnitude of variance explained by the model) is, therefore, a more salient measure of fit. The proportion of the log-likelihood score our model explains (relative to the baseline model of an empty graph that assumes all variables to be independent) is $\sim 31.5\%$ (76930.93 in absolute value, over 55 degrees of freedom, since we have 55 edges). The model explains a substantial portion of the observed variation.

Although we are not trying to develop a predictive model for defects, we wanted to test the predictive power of our models for comparison. To evaluate predictive power we use a confusion matrix. To minimize effects of anomalies in the data, we implement a ten-fold cross-validation on reshuffled data and then average the results. We present raw accuracy, percentage of correctly predicted values, and Kappa measure to address the issue of model guessing the right results by accident. We use Kappa, Cohen's kappa coefficient, because it is considered to be more robust than simple percentage of agreement. The values of precision and kappa for each release is presented in Table 4. Although we were not trying to develop a predictive model for post-release defects,

Table 5 CPT of *defects* with the variables in its Markov blanket for BN model of all Chrome and Qt releases combined

size	defects		minorauthors	defects		priordefects	defects	
	0	1		0	1		0	1
[0, 114)	0.839	0.161	0	0.767	0.233	[0, 2)	0.926	0.074
[114, 571)	0.735	0.265	[1,108]	0.583	0.417	[2, 8)	0.776	0.224
[571,106595]	0.655	0.345				[8,1702]	0.439	0.561

we present these performance metrics here to show that our model offers a moderately well predictive performance as well,thus improving our trust in the results we derive from our model.

5 Discussion

Several key insights are evident from the BN modeling approach.

5.1 Variables affecting post release defects.

The BN models we obtained show that review related measures have little direct effect on post-release defects. By examining the conditional probability tables (CPT) of *defects* and the variables in its Markov Blanket we can understand the extent of effect these variables have on post-release defects. We show the CPTs for the BN model constructed with all the Qt and Chrome releases in Table 5. The CPT were trained with the *gRain* package in R using Junction Tree belief propagation [39] method. The CPTs for other models can be constructed in a similar way. The code for constructing the CPTs is available in our replication package [1] repository.

5.2 Comparing the result with the result from traditional modeling approach in Section 3.

The variables indicated as the most important in the traditional modeling approach are also the ones indicated as most important in BN modeling approach, except in the traditional model the *minor authors* variable was discarded as being redundant, and *all authors* was used instead, while in BN approach *minor authors* have a direct influence on *defects*. This illustrates the ability of BNs to address some of the issues posed by correlated predictors, a situation common in software engineering.

The results from the two approaches are largely consistent in terms of indicating which variables are most significant in explaining the post release defects, and both approaches show that review related measures have no direct influence over the post release defects variable. Having the same result by two completely independent modeling approaches increases our confidence in the result obtained.

Conclusion 3: *Only prior defects, module size, and minor authors have direct effects on (form a Markov blanket for) post-release defects in both projects.*

5.3 Addressing the issue of highly correlated variables — problem of subjectivity in variable selection.

We have claimed before that BN modeling approach is not affected by the presence of highly correlated variables, and that can be seen in our BN model as well. The three author related variables: *all authors*, *minor authors*, and *major authors* were highly correlated in our data. Therefore, in the traditional modeling approach only one of them, *all authors* was used in the final model.

In the BN model, the three variables appear connected to each other, as can be seen in Figure 5 (the three nodes are inside the blue dotted polygon). The relationship among the nodes from the BN model is easily interpretable: more *allauthors* implies more *minor authors* and *major authors*, while increase in *minor authors* inevitably decreases *major authors* as *all authors* is the sum of minor and major authors. The BN model also suggests that the *minor authors* variable has substantially more influence over *defects* than *major authors* or *all authors*, thus it resolves the subjectivity in the variable selection problem.

To illustrate the usefulness of BNs it is worth making a few additional observations. The size of the module tends to be associated with code smells, effort, and defects (see, e.g., [78,52,3,46]). Not surprisingly, size affects both prior defects and defects, since relative module size tends to be stable release to release.

More interestingly, the BN model in Figure 5 suggests that, for example, the presence of *minor authors* both, increases the size of the module (perhaps via unnecessary code bloat), and also has a direct effect on the number of post-release defects (perhaps due to lack of understanding of the module). Thus it has a double effect on defects: direct, and mediated via module size.

The variable *minor authors* is, in turn, affected by the total number of authors, the number of changes made to the module, and the number of review issues. Arguably, the arrow should be pointing towards the review issues from minor authors as it is the minor authors that are likely to submit problematic code or be screened more vigorously during the review (see more discussion on incorporation prior knowledge in Section 6). However all these three relationships (except the direction of the third, which can be addressed by introducing a suitable prior) are rather intuitive.

Finally, it is worth considering the most important predictor of defects: prior defects. Apart from size, it is also related to the proportion of changes with no discussion, suggesting less aggressive reviews, the typical number of reviewers, and, surprisingly, is better for more complex modules. As noted earlier, the arrows should arguably be reversed: modules with prior defects probably invite more scrutiny with a larger review team. Why the modules with larger review teams tend to have higher proportion of changes with no discussion may be worth a further investigation.

Conclusion 4: *BN's can help address some difficulties posed by correlated predictors and the generative models help articulate potential mechanisms of how development process and product measure interact.*

5.4 Role of Expertise Measures in the Bayesian Network model.

We found that the two expertise related variables, *changes no expertise* (representing changes by authors lacking subject matter expertise) and *typical reviewer expertise* (representing typical reviewer expertise) are not part of the Markov blanket of the *defects* variable in our BN model. Thus, as expected, we found that adding these two variables do not change the results in any way.

6 Limitations

In this section, we discuss factors that in our opinion may pose a threat to validity of the results we present. We inherit validity threats from the study we replicate and discuss new threats related to BNs.

6.1 Latent variables.

We assume no latent variables. Dealing with hidden variables in Bayesian Networks remains an open research question. Our assumption to exclude potentially relevant unobserved variables is ameliorated by the use of prominent predictors of software defects used in extensive prior research on the subject.

6.2 Prior knowledge in BN structure search.

We do not use any prior knowledge of the problem domain while learning the BN structure. For instance, we have some prior knowledge about the directions: *e.g.,* the *defects* node should not have any outgoing edges since it is measured after the release, and the *prior defects* node should not have any incoming edges since this information is known a-priori. This knowledge can be incorporated into the search process by providing the initial partial structure as a parameter for the search function. Our unrestricted structure search yielded a model where the first assumption does hold, but second one does not. There is room for an argument that incorporating this prior knowledge will result in a more realistic model, but a counter-argument may be made as well. For example, in our model *prior defects* might represent a proxy measure for the inherent defectiveness of the module, and using the assumed prior knowledge would have excluded this possibility. Since this analysis was primarily concerned with direct effects on defects and all the discovered links were pointing inward (rendering the question moot), the ways to specify and incorporate expert knowledge while being important by itself is beyond the scope of this analysis.

6.3 Discretization.

We transform our count variables to discrete variables using the Equal Frequency method, due to reasons discussed before, and use two or three levels, based on the distribution of the original variables, for our discretized variables for the sake of simplicity in our final model. However, we acknowledge the fact that our choice of the method might not be optimal, and our choice of the number of levels is subjective as well.

6.4 Threshold.

In order to obtain the final structure from averaged model we use an arbitrary threshold of confidence. Obviously, selection of this value directly affects the structure and as a result the whole outcome of this study subjected to selection of this value. We verify the robustness of the network by gradually reducing the threshold and plotting the new structure. The conclusion of such sensitivity analysis was that the overall structure remains stable. And in particular, the Markov Blanket of *defects* variable remains unchanged even for a threshold value of 0.45 for most of the BN models. Although our analysis show that structure has quite low sensitivity to changes in threshold we still list this as a threat because we do not have a good theoretical reason behind selecting that or another threshold value.

6.5 External validity.

We conduct our study on OSS projects. Although Chrome is developed by Google developers and uses the same practices as Google uses internally, and Qt is developed as open source software now, we acknowledge that our results may not generalize to other development settings beyond these two projects.

7 Discussion of Related Work

In 1976 Fagan published the first empirical evaluation of software review, *i.e.* inspection [21]. The work quantified the defect finding effectiveness of inspection based on the number of defects found per thousand lines of source code

(KLOC) and percentage of total defects found by inspection. On the IBM system under study 38 defects per KLOC were found by inspection vs 8 per KLOC found by unit tests. Inspection found 82% of the total defects found for the released product. In the intervening 40 years, code review has changed dramatically from the rigid inspection process that Fagan introduced.

Most of the early work on inspection focused on minor variations in the inspection process but kept the formality, measurability, and rigidity intact[43, 37,38,42,84]. The most important finding was that the inspection meeting need not be held in person to find a substantial number of defects [83,19,60]. This lead the way to online review tools that ultimately lead to the currently popular and widely studied Gerrit [50,45] and the pull request mechanism of GitHub [85,64,29].

There is also a long history of examining the factors that make peer review effective. Porter *et al.* [62] examined both the process and the inputs to the process (*e.g.,* reviewer expertise, and artifact complexity). In terms of the number of defects found during review, Porter *et al.* concluded that the best predictor was the level of expertise of the reviewers. Varying the processes had a negligible impact on the number of defects found. This finding is echoed by others (*e.g.,* [73,38]).

Rigby *et al.* [69,66,70,68] examined open source software based review on multiple projects including the Linux kernel, the Apache server, and KDE. They created regression models with the number of defects found during review and the amount of time take for review. They found remarkably similar practices across project that had very little process, but relied on expert reviewers frequently reviewing each commit. In a study at Microsoft and AMD, Rigby and Bird [67] found that these lightweight review practices were also used in industry. They also found that the focus had shifted from a defect finding activity to a problem solving one.

Recent works have focused on the non-defect finding benefits of code review. For example, interviews of Microsoft and OSS developers have been conducted to understand developer motivations for code review [5,10]. They found that while developers want to find defects, they were also interested in spreading knowledge and discussing alternative solutions. Indeed, code review has also been shown to be effective at spreading knowledge and reducing the impact of code ownership [67,41,81]. Other works focused on the types and utility of feedback provided by developers [12,7,40] and on the ability of code reviews to identify security vulnerabilities [11,51]

Despite these additional benefits of code review, the primary goal is still defect finding [5,10]. The literature abounds with papers that use product and process metrics to predict where defects will occur, for example, [24,54, 74]. These models have also been used to understand changes in development practices, such as co-location vs remote developers [9], the impact of developer turnover [46] and much more. As far as we know, McIntosh *et al.*'s [45,44] is the first to examine to include peer review measures into a defect model. Earlier works [62,68] measured how many defects where found during the review, but did not look at the long-term impact of review on defects. As a result, our work first replicates McIntosh *et al.*'s work that covered only releases (two Qt releases, one release from ITK, and one from VTK), we expand the study to include six releases of the Chrome project.

A case for use of BNs in the context of Software Engineering was made by Fenton et.al. [25,22], while the earliest publications utilizing BNs we could find [35] constructed search of the structure based on the statistical significance of partial correlations in the context of modeling delays in globally distributed development. [80,58] considered the application of Bayesian networks to prediction of effort, [23,53,55] used Bayesian networks to predict defects, and [56] used BN approach for an empirical analysis of faultiness of a software. In a similar work, [6] used modified BNs (Markov Bayesian network) for software reliability prediction. [82] used BNs for predicting maintainability of Object Oriented software, and [8] used BNs as a software productivity estimation tool. We are not aware of prior applications of Bayesian Networks for modeling software reviews. On the other hand, Bayesian structure learning is a big domain in itself with a wide range of algorithms, but its use in software engineering context is not very common.

7.1 Conclusion

Prior works have shown that the defects are both effectively and efficiently found during code review [20,63,68]. Recent works provided qualitative evidence that reviews provide benefits beyond defect detection, such as knowledge sharing [72,5,70,12,40,65]. In contrast, the goal of this work is to quantify the *longterm* impact of peer review on post-release defects.

Conclusion 1: Reproduction and Replication

McIntosh *et al.* [44,45] were the first to study the impact of code review measures on post-release defects. We replicated their study using data they provided and as well as on the Chrome data we extracted. McIntosh *et al.* found that review participation had an influence on post-release defects, but we were unable to replicate these results. Instead we found that review measures contributed little to the performance of the model. The R^2 values with and without review measures were almost identical. In agreement with existing defect prediction work [48,33,9,30], our results show that prior defects, the module size, and the number of authors are the strongest predictors of post-release defects. Review measures are neither necessary nor sufficient to create a good defect prediction model.

Conclusion 2: Inconsistent Models

It is extremely difficult to replicate an empirical software study that involves both mining operational data and statistical modelling. Despite using exactly the same data and modelling approach we obtain substantially different results. In both our study and that of McIntosh *et al.* [45] a key problem is the need to select an uncorrelated set of variables. The variable selection process is inherently subjective because differences in expert opinions may lead to different sets of variables.

Furthermore, in both studies, the models were performed per project and per release. Even strong predictors, such as prior-defects varied substantially in their predictive power between project releases. This result suggests an issue with the traditional variable selection used in regression models.

Conclusion 3: Direct effects

Regression models require the researcher to define a response and a set of predictors. This approach lacks tools to distinguish between an actual relationship and the effect of a shared confound. In contrast, Bayesian Networks remove the need for variable selection and shows the Bayesian relationships among variables. The term "direct effect" is meant to quantify an influence that is not mediated by other variables in the model or, more accurately, the sensitivity of Y to changes in X while all other factors in the analysis are held fixed. Indirect effects can manifest themselves on the response only through affecting the value of predictors that gave direct effects on the response.

According to our BN, only three measures directly impact post-release defects: the number of prior defects, the number of minor authors, and the size of the module. The code review measures, such as *rushed reviews*, *number of review participants*, and *discussion length*, did not directly impact the number of post-release defects.

Conclusion 4: Generative models and indirect effects

The use of BN provides a way to evaluate the indirect effects that code reviews have on defects through the influence on other variables. Such indirect effects bedevil traditional analysis methods that use observational data. If the set of observed variables is complete, it is possible to calculate an impact of intervention akin to the results that could be obtained only in randomized experiments. For example, changes that have *no review discussion* tend to be associated with files that have had many *prior defects* which in turn increase the number of post-release defects. A further example from our BN model shows that having 5 or more reviewers is seen to increase chance of having post-release defects from 20% to 33% through mediating variables *allauthors* and *minorauthors.*

We have demonstrated the difficulties in using traditional models on observational data. Although individual code reviews find defects, we were unable to find any direct effect of review measures on post-release defects. By using BN we found that code review measures indirectly effect post-release defects. We hope that other researchers will use the approaches presented here to untangle the relationships among software measures. These indirect effects should provide a more nuanced understanding of software engineering. We make our scripts and data available in our replication package [1].

References

1. Replication package, 2018. Our scripts and data are available: `https://github.com/CESEL/ReviewPostReleaseDefectsReplication`.
2. J. P. F. Almqvist. Replication of controlled experiments in empirical software engineering-a survey. 2006.
3. E. Arisholm and L. C. Briand. Predicting fault-prone components in a java legacy system. In *International Symposium on Empirical Software Engineering*, pages 8 – 17, 2006.
4. R. Axelrod. Advancing the art of simulation in the social sciences. In *Simulating social phenomena*, pages 21–40. Springer, 1997.
5. A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 712–721. IEEE Press, 2013.
6. C.-G. Bai. Bayesian network based software reliability prediction with an operational profile. *Journal of Systems and Software*, 77(2):103–112, 2005.
7. M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 202–211, New York, NY, USA, 2014. ACM.
8. S. Bibi, I. Stamelos, and L. Angelis. Bayesian belief networks as a software productivity estimation tool. In *1st Balkan Conference in Informatics, Thessaloniki, Greece*, 2003.
9. 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 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14. ACM, 2011.
10. A. Bosu, J. C. Carver, C. Bird, J. Orbeck, and C. Chockley. Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft. *IEEE Transactions on Software Engineering*, 43(1):56–75, Jan 2017.
11. A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni. Identifying the characteristics of vulnerable code changes: An empirical study. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 257–268, New York, NY, USA, 2014. ACM.
12. A. Bosu, M. Greiler, and C. Bird. Characteristics of useful code reviews: An empirical study at microsoft. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 146–156, May 2015.
13. F. Camilo, A. Meneely, and M. Nagappan. Do bugs foreshadow vulnerabilities? a study of the chromium project. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 269–279, May 2015.
14. A. M. Carvalho. Scoring functions for learning bayesian networks. *Inesc-id Tec. Rep*, 12, 2009.
15. R. Carver. The case against statistical significance testing. *Harvard Educational Review*, 48(3):378–399, 1978.
16. D. M. Chickering. Learning bayesian networks is np-complete. *Learning from data: Artificial intelligence and statistics V*, 112:121–130, 1996.
17. B. S. Chlebus and S. H. Nguyen. On finding optimal discretizations for two attributes. In *International Conference on Rough Sets and Current Trends in Computing*, pages 537–544. Springer, 1998.

18. T. Dey and A. Mockus. Modeling relationship between post-release faults and usage in mobile software. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 56–65. ACM, 2018.
19. S. G. Eick, C. R. Loader, M. D. Long, L. G. Votta, and S. V. Wiel. Estimating software fault content before coding. In *Proceedings of the 14th International Conference on Software Engineering*, pages 59–65, 1992.
20. M. Fagan. A history of software inspections. In *Software pioneers*, pages 562–573. Springer, 2002.
21. M. E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15(3):182–211, 1976.
22. N. Fenton, P. Krause, and M. Neil. Software measurement: Uncertainty and causal modeling. *IEEE software*, 19(4):116–122, 2002.
23. N. Fenton, M. Neil, W. Marsh, P. Hearty, D. Marquez, P. Krause, and R. Mishra. Predicting software defects in varying development lifecycles using bayesian nets. *Information and Software Technology*, 49(1):32–43, 2007.
24. N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, Sep 1999.
25. N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on software engineering*, 25(5):675–689, 1999.
26. N. Friedman, M. Goldszmidt, and A. Wyner. Data analysis with bayesian networks: A bootstrap approach. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pages 196–205. Morgan Kaufmann Publishers Inc., 1999.
27. S. Garcia, J. Luengo, J. A. Sáez, V. Lopez, and F. Herrera. A survey of discretization techniques: Taxonomy and empirical analysis in supervised learning. *IEEE Transactions on Knowledge and Data Engineering*, 25(4):734–750, 2013.
28. O. S. Gómez, N. Juristo, and S. Vegas. Understanding replication of experiments in software engineering: A classification. *Information and Software Technology*, 56(8):1033–1048, 2014.
29. G. Gousios, A. Zaidman, M.-A. Storey, and A. van Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 358–368, Piscataway, NJ, USA, 2015. IEEE Press.
30. T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *Software Engineering, IEEE Transactions on*, 26(7):653–661, 2000.
31. M. Hahsler, S. Chelluboina, K. Hornik, and C. Buchta. The arules r-package ecosystem: Analyzing interesting patterns from large transaction datasets. *Journal of Machine Learning Research*, 12:1977–1981, 2011.
32. F. E. Harrell Jr. rms: Regression modeling strategies. r package version 4.0-0. *City*, 2013.
33. A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.
34. D. Heckerman. A tutorial on learning with bayesian networks. In *Learning in graphical models*, pages 301–354. Springer, 1998.
35. J. D. Herbsleb and A. Mockus. An empirical study of speed and communication in globally-distributed software development. *IEEE Transactions on Software Engineering*, 29(6):481–494, June 2003.
36. L. Huang and B. Boehm. How much software quality investment is enough: A value-based approach. *IEEE software*, 23(5):88–95, 2006.
37. J. C. Knight and E. A. Myers. An improved inspection technique. *ACM Communications*, 36(11):51–61, 1993.
38. S. Kollanus and J. Koskinen. Survey of software inspection research. *Open Software Engineering Journal*, 3:15–34, 2009.
39. D. Koller and N. Friedman. Probabilistic graphical models: principles and techniques. 2009.
40. O. Kononenko, O. Baysal, and M. W. Godfrey. Code review quality: How developers see it. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1028–1038, May 2016.

41. O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey. Investigating code review quality: Do people and participation matter? In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 111–120. IEEE, 2015.
42. O. Laitenberger and J. DeBaud. An encompassing life cycle centric survey of software inspection. *Journal of Systems and Software*, 50(1):5–31, 2000.
43. J. Martin and W. T. Tsai. N-Fold inspection: a requirements analysis technique. *ACM Communications*, 33(2):225–232, 1990.
44. S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 192–201. ACM, 2014.
45. S. Mcintosh, Y. Kamei, B. Adams, and A. E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Softw. Engg.*, 21(5):2146–2189, Oct. 2016.
46. A. Mockus. Organizational volatility and its effects on software defects. In *ACM SIGSOFT / FSE*, pages 117–126, Santa Fe, New Mexico, November 7–11 2010.
47. A. Mockus. Engineering big data solutions. In *ICSE'14 FOSE*, pages 85–99, 2014.
48. A. Mockus, R. T. Fielding, and J. Herbsleb. A case study of open source software development: the apache server. In *Proceedings of the 22nd international conference on Software engineering*, pages 263–272. Acm, 2000.
49. R. Morales, S. McIntosh, and F. Khomh. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 171–180. IEEE, 2015.
50. M. Mukadam, C. Bird, and P. C. Rigby. Gerrit software code review data from android. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 45–48, May 2013.
51. N. Munaiah, F. Camilo, W. Wigham, A. Meneely, and M. Nagappan. Do bugs foreshadow vulnerabilities? an in-depth study of the chromium project. *Empirical Software Engineering*, 22(3):1305–1347, Jun 2017.
52. N. Nagappan, B. Murphy, and V. R. Basili. The influence of organizational structure on software quality: an empirical case study. In *ICSE 2008*, pages 521–530, 2008.
53. M. Neil and N. Fenton. Predicting software quality using bayesian belief networks. In *Proceedings of the 21st Annual Software Engineering Workshop*, pages 217–230. NASA Goddard Space Flight Centre, 1996.
54. S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 529–540. ACM, 2007.
55. A. Okutan and O. T. Yıldız. Software defect prediction using bayesian networks. *Empirical Software Engineering*, 19(1):154–181, 2014.
56. G. J. Pai and J. B. Dugan. Empirical analysis of software fault content and fault proneness using bayesian methods. *IEEE Transactions on software Engineering*, 33(10):675–686, 2007.
57. J. Pearl. Probabilistic reasoning in intelligent systems: networks of plausible inference. 2014.
58. P. C. Pendharkar, G. H. Subramanian, and J. A. Rodger. A probabilistic model for predicting software development effort. *IEEE Transactions on software engineering*, 31(7):615–624, 2005.
59. A. Perez, P. Larranaga, and I. Inza. Supervised classification with conditional gaussian networks: Increasing the structure complexity from naive bayes. *International Journal of Approximate Reasoning*, 43(1):1–25, 2006.
60. D. Perry, A. Porter, M. Wade, L. Votta, and J. Perpich. Reducing inspection interval in large-scale software development. *Software Engineering, IEEE Transactions on*, 28(7):695–705, 2002.
61. J. Pinheiro, D. Bates, S. DebRoy, and D. Sarkar. R development core team. 2010. nlme: linear and nonlinear mixed effects models. r package version 3.1-97. *R Foundation for Statistical Computing, Vienna*, 2011.

62. A. Porter, H. Siy, A. Mockus, and L. Votta. Understanding the sources of variation in software inspections. *ACM Transactions Software Engineering Methodology*, 7(1):41–79, 1998.
63. A. Porter, H. Siy, A. Mockus, and L. G. Votta. Understanding the sources of variation in software inspections. *ACM Transactions on Software Engineering and Methodology*, January 1998.
64. M. M. Rahman and C. K. Roy. An insight into the pull requests of github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 364–367, New York, NY, USA, 2014. ACM.
65. M. M. Rahman, C. K. Roy, and R. G. Kula. Predicting usefulness of code review comments using textual features and developer experience. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 215–226, May 2017.
66. P. Rigby, B. Cleary, F. Painchaud, M.-A. Storey, and D. German. Contemporary peer review in action: Lessons from open source development. *IEEE software*, 29(6):56–61, 2012.
67. P. C. Rigby and C. Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 202–212. ACM, 2013.
68. P. C. Rigby, D. M. German, L. Cowen, and M.-A. Storey. Peer Review on Open-Source Software Projects: Parameters, Statistical Models, and Theory. *ACM Transactions on Software Engineering and Methodology*, 23(4):35:1–35:33, September 2014.
69. P. C. Rigby, D. M. German, and M.-A. Storey. Open Source Software Peer Review Practices: A Case Study of the Apache Server. In *ICSE '08: Proceedings of the 30th International Conference on Software engineering*, pages 541–550, New York, NY, USA, 2008. ACM.
70. P. C. Rigby and M.-A. Storey. Understanding broadcast based peer review on open source software projects. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 541–550. ACM, 2011.
71. P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131, 2009.
72. C. Sauer, D. R. Jeffery, L. Land, and P. Yetton. The effectiveness of software development technical reviews: a behaviorally motivated program of research. *IEEE Transactions on Software Engineering*, 26(1):1–14, Jan 2000.
73. C. Sauer, D. R. Jeffery, L. Land, and P. Yetton. The Effectiveness of Software Development Technical Reviews: A Behaviorally Motivated Program of Research. *IEEE Transactions Software Engineering*, 26(1):1–14, 2000.
74. S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering*, 39(4):552–569, 2013.
75. G. Shmueli. To explain or to predict? *Statistical science*, pages 289–310, 2010.
76. F. Shull, V. Basili, J. Carver, J. C. Maldonado, G. H. Travassos, M. Mendonça, and S. Fabbri. Replicating software engineering experiments: addressing the tacit knowledge problem. In *Empirical Software Engineering, 2002. Proceedings. 2002 International Symposium n*, pages 7–16. IEEE, 2002.
77. F. J. Shull, J. C. Carver, S. Vegas, and N. Juristo. The role of replications in empirical software engineering. *Empirical software engineering*, 13(2):211–218, 2008.
78. D. I. Sjoberg, A. Yamashita, B. Anda, A. Mockus, and T. Dyba. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, 2013.
79. E. Sober. Instrumentalism, parsimony, and the akaike framework. *Philosophy of Science*, 69(S3):S112–S123, 2002.
80. I. Stamelos, L. Angelis, P. Dimou, and E. Sakellaris. On the use of bayesian belief networks for the prediction of software productivity. *Information and Software Technology*, 45(1):51–60, 2003.
81. P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 1039–1050, New York, NY, USA, 2016. ACM.

82. C. Van Koten and A. Gray. An application of bayesian network for predicting object-oriented software maintainability. *Information and Software Technology*, 48(1):59–67, 2006.
83. L. G. Votta. Does every inspection need a meeting? *SIGSOFT Softw. Eng. Notes*, 18(5):107–114, 1993.
84. K. E. Wiegers. *Peer Reviews in Software: A Practical Guide*. Addison-Wesley Information Technology Series. Addison-Wesley, 2001.
85. Y. Yu, H. Wang, V. Filkov, P. Devanbu, and B. Vasilescu. Wait for it: Determinants of pull request evaluation latency on github. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 367–371, May 2015.
86. Q. Zheng, A. Mockus, and M. Zhou. A method to identify and correct problematic software activity data: Exploiting capacity constraints and data redundancies. In *ESEC/FSE'15*, pages 637–648, Bergamo, Italy, 2015. ACM.