MODELLING SOFTWARE RELIABILITY USING HYBRID BAYESIAN NETWORKS

by

Ayşe Tosun Mısırlı

B.S., Computer Science and Engineering, Sabanci University, 2006

M.S., Computer Engineering, Boğaziçi University, 2008

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Graduate Program in

Boğaziçi University

2012

MODELLING SOFTWARE RELIABILITY USING HYBRID BAYESIAN NETWORKS

APPROVED BY:

Prof. Ayşe Başar Bener (Thesis Co-supervisor)	....
Prof. Oğuz Tosun (Thesis Co-supervisor)	....
Prof. Fikret Gürgen	....
Prof. Emin Anarım	....
Assist. Prof. A. Taylan Cemgil	....
Burak Turhan, Ph.D.	....

DATE OF APPROVAL: 4.4.2012

ACKNOWLEDGEMENTS

I would like to express my genuine gratitude to my thesis supervisor, Prof. Ayşe Bener, whose endless support and knowledge helped me to complete my research as well as to find my perspective in life. I will never forget how Prof. Bener has been my guidance as I engage in the completion of this work. I thank to Prof. Oğuz Tosun for being my co-advisor and his personal support, and Prof. Fikret Gürgen, Prof. Emin Anarım, Assist. Prof. Taylan Cemgil and Dr. Burak Turhan for kindly accepting to be in my defense jury. I owe special thanks to Assist. Prof. Taylan Cemgil and Dr. Burak Turhan for allocating their valuable time for discussions about my research.

My gratitude goes to Turgay Aytaç for helping me find industrial supporters for this research as well as sharing his extensive knowledge on computer science, engineering and management. I am indebted to my colleagues in SoftLab, Bora Çağlayan, Gül Çalıklı, Aslı Uyar Özkaya and Gülfem Alptekin, for unlimited kindness, friendship and support, and many of my colleagues at Department of Computer Engineering for being always ready to help.

I would like to thank Ayhan Inal, Tolga Aytekin from Logo Business Solutions and Reşat Kale from Turkcell Technology for providing data for my empirical research. This work could not have been completed without financial supports of Turkish Scientific Research Council (TUBITAK), Turkish State Planning Organization (DPT under the project number 2007K120610) and partial supports of Turkcell Akademi, BUVAK, ACM Graduate Student Support Program.

I am also grateful to everyone in my large family for providing psychological support, and especially for "food" support when I was desperately in need of care. Finally, I would like to dedicate this thesis to my dear parents, Fatma Tosun and Ziya Tosun, my dear sister, Merve Tosun, and my lovely husband Yiğit Mısırlı.

ABSTRACT

MODELLING SOFTWARE RELIABILITY USING HYBRID BAYESIAN NETWORKS

In this research, we analyse the problem of predicting software reliability from AI perspective. We observe that existing models are built based on expert knowledge including defining a set of metrics through surveys and causal relationships. We overcome their limitations by introducing new data collection, model construction and inference methodology. We propose a Hybrid Bayesian network that would estimate reliability of consecutive releases of software projects before a release decision, in terms of their residual (post-release) defects. We form this hybrid model by incorporating quantitative factors of development and testing processes into qualitative factors of requirements specification and documentation process without the need for any transformation. As quantitative factors, we select popularly used product, in-process and people metrics as well as introduce new ones depending on the availability of local data in the organizations. We also identify qualitative factors representing requirements specification process via surveys with development teams. Dependencies between software metrics and defects are determined according to correlation and independence tests and graphical dependence analysis with chi-plots. We utilize a Monte Carlo technique to approximate joint probability distribution of the model over conditionals by inferring unknown distribution parameters. Empirical analyses on two industrial datasets show that (i) Hybrid Bayesian networks are capable of estimating reliability in terms of residual defects, (ii) proposed way of defining causal relationships, chi-plots, decreases error rates significantly, (iii) expert judgement-based models may not achieve as good prediction performances as statistical models, (iv) local data are so valuable and representative as expert knowledge in software organizations that they should be used primarily and strengthened with expert knowledge in predicting software reliability.

ÖZET

YAZILIM GÜVENİLİRLİĞİNİN HİBRİD BAYES AĞLARI KULLANILARAK MODELLENMESİ

Bu tezde, yazılım güvenilirliğini tahmin etme problemi yapay zeka açısından incelenmektedir. Önceki çalışmalarde kurulan tahmin modellerinde, yazılım süreçlerine ait faktörler and aralarındaki sebep-sonuç ilişkileri tamamen uzmanlık bilgilerine bağlı kalınarak kurulmuştur. Bu sebeple, yeni bir veri toplama, model kurma ve çıkarsama metodu önerilmektedir. Önerilen yazılım güvenilirliği tahmin modeli, yazılım projelerinin sürüm sonrası hatalarını tahmin eden bir Hibrid Bayes ağları modelidir. Bu model, şirketlerin varolan veri ambarlarından çıkartılan, kodlama ve test süreçlerini tanımlayan nicel faktörler ile, yazılım ekibiyle yapılan anketlerden çıkartılan, gereksinim belirleme ve analiz sürecini tanımlamak için oluşturulan nitel faktörleri birleştirir. Böylece, önerilen hibrid model hem kategorik hem de sürekli değişkenleri, sürekli değişkenlere herhangi bir transformasyon uygulamadan, birlikte kullanmıştır. Yazılım ölçütleri ve hatalar arasındaki sebep-sonuç ilişkileri, istatistiksel bağımsızlık testlerine ek olarak, grafiksel bağımlılık analizleri ile belirlenmiştir. Bir Monte Carlo yöntemi ile, Bayes modelinin ortak olasılık dağılımı ve bilinmeyen parametreleri tahmin edildiği gibi, anketlerdeki eksik veriler de tamamlanmıştır. İki yazılım şirketi için yapılan ampirik çalışmaların sonuçları şöyle özetlenebilir: (i) Bayes ağları, nicel ve nitel faktörleri birlikte kullanarak, yazılım güvenilirliğini sürüm sonrası hatalar açısından, başarıyla tahmin edebilmektedir, (ii) Grafiksel bağımlılık analizleri sayesinde belirlenen sebep-sonuç ilişkileri, sürüm sonrası hata yakalama başarısını önemli ölçüde iyileştirmektedir, (iii) Sadece uzmanlık bilgilerine dayanarak kurulan modeller, istatistiksel analizlerle kurulan modellere göre daha düşük bir hata yakalama performansı göstermektedir, (iv) Şirket içi veri havuzları, uzman bilgi ve deneyimleri kadar değerlidir ve yazılım güvenililirliğini tahmin etmede öncelikli olarak kullanılmalıdır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS/ABBREVIATIONS

AI	Artificial Intelligence
BN	Bayesian Network
CPT	Conditional Probability Table
EM	Expectation Maximisation
LOC	Lines of Code
MCMC	Markov Chain Monte Carlo
SDLC	Software Development Life Cycle
SVN	Subversion
3P	People, Process, Product
3GL	Third Generation Programming Language
4GL	Forth Generation Programming Language

1. INTRODUCTION

With the rapid evolution in computer technology, hardware and software systems, we can see complex software systems in almost all industries, including telecommunications supporting phone operations, as well as aerospace producing space shuttles, or even in our personal computers running with complex operating systems with millions of lines of code. The demand for complex software systems has increased more rapidly than the ability to design, implement, test and maintain them [1]. As the requirements for software systems increase, failures also increase due to lack of sufficient advancement in productivity, quality, cost and performance issues. Various historical software failure stories, such as the loss of Mars Climate Orbiter (1999) [2] or radiation therapy miscalculation at National Cancer Institute (2000) [3], showed that failures occurred in the field caused many million dollars of budget loss as well as significant health problems and loss of human lives.

Reliability is considered as the most important aspect of software quality both from the engineer's and customer's points of view [4]. It is often quantified with failures (or post-release faults) in software systems. In many engineering disciplines, reliability engineering has been applied with complex but routine designs for years [1]. Software engineering discipline should also be well defined through standardized processes. It should be effectively applied to produce high quality products consistent with customer requirements.

Software managers in companies primarily look for reliable software which would function properly in accordance with the customer needs [5]. Due to critical project constraints such as time, budget and personnel, detecting all possible faults leading software failures prior to a release and, hence, producing reliable and high quality products is a real challenge for software industry. Software teams often face with a constant pressure of "Deliver Now!" during development process [6]. With this pressure, a quick short cut for the team is to ensure that the product satisfies minimum requirements, works as expected and, then, to deploy the software as soon as possible. However, defects are more costly if discovered and fixed in the later stages of the development life cycle or during production [7]. Therefore, testing is one of the most critical phases of the software development life cycle, which is responsible for 50% of the total cost of development [7]. On the other hand, investing a large amount into software testing can be difficult to justify, particularly for and time- and budget-conscious companies.

Decisions on “when to stop testing” and “how much reliable the software is” are very critical, and hence, they are very difficult to make. Testing is a complex process due to complexity of applications and interdependencies between systems. However, complete testing of a software product is almost impossible. Even though a complete testing is carried out before the product release, software cannot be 100% reliable. According to a research carried out more than 20 years ago [8], Adams observed the occurrence of rates of failures on large IBM software systems and he found that more than 60% of failures were “5000-year bugs”. This means there are bugs that would cause the failures of the systems, on average, every 5000 years. So, we cannot claim that our software is 100% fault-free and, hence, it is 100% reliable.

From the managerial point of view, it is still impossible to tell that our software is reliable enough before we have seen the operational usage. We need additional factors, i.e. metrics, predictions, or expert judgements, to make formalisms on the final reliability level for a specific project. There are common factors suggested to be monitored during software development to decide an acceptable reliability, i.e., confidence, level for a company: (i) percentage of test cases correctly executed, (ii) test budget and time, (iii) code coverage, functionality or requirements covered, (iv) bug rate. However, as software systems get more complex, managers do not actually monitor each factor individually; instead, they combine all these factors with other process related metrics. Furthermore, product reliability is strongly connected with software development life cycle (SDLC) activities and three major aspects affecting the quality of these activities: Product, Process, People (Resources) (3P). Hence, it is very complex to model the product reliability with a single factor or using only one aspect in 3P. Each phase in a SDLC has to be modelled by considering the relations of product, process and people related factors, as well as the relationships between these phases and their impacts on the final product reliability.

Predictive models are increasingly in demand to guide software developers and managers to help decision-making under uncertainty [9]. Such models can be applied during different processes to estimate the cost/effort of the development life cycle [10], to predict defect-prone parts of the software, either in pre-release [11] or post-release [12], and to assess the reliability of the system [9]. Our research has so far focused on building decision-making mechanisms for defect prediction [13] and cost estimation [14]. Similar to much research, we have dealt with these issues separately. However, in real life, software managers take simultaneous decisions under uncertainty. Although applying such models helps effective allocation of limited resources (e.g. early indication of defects during testing with defect predictors), software managers want to come up with a conclusion on the final reliability of software before the release. To accomplish this, it is necessary to observe not only single factor, such as residual defect density, but also various factors which may possibly affect the software reliability. After that, managers have to answer several questions, such as:

- When to stop testing?
- Is the software product reliable enough to release?
- Does the software need additional testing effort, requirements or design effort?
- Based on a reliability level, should software managers make a drastic change, delay or cancel the project?

1.1. Contributions

In this research, we build a Hybrid Bayesian network that would estimate reliability of consecutive releases of software projects before a release decision, in terms of the number of residual (potential post-release) defects. We build this causal network by modelling the relationship between major processes of a software development life cycle (SDLC) and residual defects. Contributions of this research can be summarized as follows:

- *Quantification of software reliability in terms of metrics*: Since this research is based on empirical analysis, we have to measure/quantify software reliability within the scope of software processes. Our initial step was to made an extensive literature survey on metrics used to quantify software reliability. We have seen that software reliability was often evaluated with the number of residual (post-release) defects in a software system. Furthermore, reliability was defined within the context of development process, with a very few empirical studies on predicting reliability using metrics from other processes.

  In this research, we have selected previously used metrics by adapting them to our problem as well as defined new ones to investigate potential effects of development and testing processes on final software reliability. During metric selection, we have filtered the most representative ones in terms of predicting residual defects by considering the mapping to Product, Process and People aspects. In order to complete SDLC, requirements process is represented through a survey due to difficulty of quantifying this process via metrics and lack of standard deliverables (like the source code in development, or test cases in testing) that could be used for metric extraction. Finally, we have managed to represent three major processes of SDLC in terms of quantitative and qualitative metrics.
- *Identification of causal relationships among metrics*: One of our research objectives is to gain more insight about how dependencies between software metrics affect the final reliability of software. In real life, software managers use this kind of information by combining their previous experience and projections with current situation at the time of decision making. However, expert predictions may include bias in their judgemental processes or they may be over-optimistic. Bayesian learning encodes probabilistic relationships among variables successfully so that we can gain an understanding about the problem domain and predict the consequences of different configurations. In this research, we have utilized Bayesian networks in conjunction with statistical tests in order to identify causal relationships between software metrics and residual defects.
- *Constructing a Hybrid BN using both qualitative and quantitative data*: During data collection steps, we have realized that collecting local data from all soft-

ware processes is not feasible, while quantifying processes with survey data also contains a bias due to subjective judgements of software team with varying experience levels and qualifications. A hybrid model is necessary when organizations have different levels of maturity in different processes of SDLC. In this research, we have applied a hybrid data collection approach by using quantitative data extracted from organizational repositories and qualitative data collected through surveys.

To handle mixture of continuous and categorical variables in a single model, we have proposed a Hybrid BN in which categorical variables can be parents of continuous variables. A Hybrid BN also requires to handle continuous variables without the need of any transformation, such as discretization, so that we can avoid possible biases that may occur due to the application of discretization in small state spaces, like software engineering datasets.

- *Utilization of Monte Carlo methods during BN inference*: In this research, our main concern is to predict software reliability by learning the probability of residual defects given observations from software metrics. This probability needs to be computed from BNs, which is called *inference* in short. Although inferring posterior probability of the output node in a BN is easier with categorical variables, it is not trivial when we have metrics with both continuous and categorical values. We have used a well known Monte Carlo technique, namely *Gibbs sampling*, on approximation of the joint probability distribution over conditional distributions. Gibbs sampling is well suited for inferring posterior distribution in a BN (including estimation of parameters), handling mixture of continuous and categorical variables, as well as handling missing data. In our problem, surveys sometimes have missing data, and hence there is not a one-to-one correspondence with quantitative software metrics. We have used Gibbs sampling to handle this situation in addition to estimating probability of residual defects.
- *Empirical analysis of Bayesian networks built by experts versus learned from local data*: Throughout this research, we have collected two industrial datasets with quantitative metrics extracted from organizations' data repositories and qualitative metrics extracted through surveys conducted with development teams. We

have performed statistical tests for identifying causal relationships and built a Hybrid BN that would estimate residual defects periodically (weekly or prior to a new release). In addition, we have built an expert opinion-based BN, whose causal relations are defined based on evaluations of experts in the organizations. Our objective is to understand whether a predictive model can imitate the way experts synthesize their knowledge with current state of software while making release decisions. Results obtained on two different datasets are very promising such that we managed to extract relationships between metrics and software reliability through Bayesian networks and statistics and make predictions better than results obtained from an expert-based model. These results also show that predictive models can be used effectively by software organizations in order to reduce the risk of wrong estimations and pressure over software managers.

2. BACKGROUND

The main interests in practical software engineering are as follows: (i) we are not sure about when to stop testing and release the software, and (ii) we do not have any information about final reliability of a software product. This section briefly summarizes previous studies on “optimal release time” and “software reliability” by listing available models built for predicting software reliability. Then, a broader research on Bayesian Networks and their applications in other domains are presented.

2.1. Optimal Software Release Problem

Yang *et al.* [15] defined “optimal release problem” as constrained optimization problem, where we expect to maximize the software reliability and minimize the project cost subject to a predefined cost limit. Software release time has been affected by various factors during development life cycle, some of which are the number of faults detected and removed during testing activities, the effort spent for fault removal activities, or time spent for each software process area. Authors discussed that it is impossible to decide on actual release time, since time spent for those factors mentioned above are completely random. They studied uncertainty in software cost and its impact on software release time using a linear model whose parameters are defined by non-homogeneous Poison distributions. Results of this study show that uncertainty in project life cycle should be taken into account seriously in order to avoid major false downs in project management. Other studies estimated optimal software release time using differential equations [16]. Estimation of optimal release times would, in turn, determine the software reliability in terms of residual defect densities in the end product. However, finding the optimal release time is not our major concern during this research.

2.2. Predicting Software Reliability

Software reliability is defined as “the probability that software will not cause the failure of a system for a specified time under specified conditions” [17]. Software reliability assessment is categorized into three classes by Eusgeld and Freiling [18]: *black box reliability analysis, software metric-based reliability analysis, architecture-based reliability analysis.* Black-box reliability approaches deal with time-dependent failure observations from testing or operation phases, and hence, they do not consider internal details of the software system. Software metric-based reliability analysis aims to reason about residual faults or fault frequencies expected to have during the software execution [18]. In this analysis, metrics from the source code or development processes can be used to estimate the reliability. Finally, architecture-based reliability analysis divides the software into smaller components, called blocks, and these blocks and their dependencies among each other are investigated based on the architecture of the software earlier in design phase.

Software reliability is also one of the system reliability concepts, especially a bottleneck of system reliability in many existing large scale systems [1], and it is critical to model software reliability and predicting its trend to provide crucial information for *reliability engineering. Software reliability engineering (SRE)* is defined as *the quantitative study of the operational behaviour of software-based systems with respect to user requirements concerning reliability* [17]. Therefore, it includes reliability measurement, prediction and estimation, as well as impacts of product design, development process, architecture and operational environment on reliability. Measurements of software reliability includes two types of activities: *Prediction* and *Estimation.*

Estimation: This activity aims to asses the *current* reliability in terms of failure data obtained from system tests or during system operation and determine whether a reliability growth model is a good fit in retrospect [1].

Prediction: This activity determines *future* software reliability based upon available software metrics and measures. In general, when failure data are available (after testing) reliability models can be parametrized and verified to perform future reliability prediction [1].

In this research, we focus on reliability measurement and predict software reliability (in terms of post-release defects) by using available software metrics and failure data from previous releases in order to perform predictions for future releases.

In empirical studies, researchers often evaluate the software reliability as predicting residual faults expected to have during operation, similar to the software metric-based reliability analysis. Today's verification, validation and testing (VV&T) strategies can be applied to detect and fix software faults earlier in development life cycle, such as manual code reviews [19, 20], inspections [21, 22] and automated defect[1] prediction models [11, 23, 24]. Defect predictors improve the efficiency of testing phase and help developers assess the quality and defect-proneness of their software product [25]. They also help managers in allocating resources. These models require historical data in terms of software metrics and actual defect rates. They combine metrics and defect information as training data to learn which modules are likely to be defective. Based on the knowledge from training data and software metrics acquired from a recently completed project, such tools can estimate defect-prone modules of that project. Generally, defect prediction is treated as a classification problem, where the aim is to predict defect proneness of a software module[2] by assigning 0 to defect-free modules and 1 to defect-prone modules.

Previous research on software defect prediction show that learning based defect predictors can detect 70% of all defects in a software system on average [11], while manual code reviews can detect between 35 and 60% of defects [20], and inspections can detect 30% of defects at most [22]. Furthermore, code reviews are labour-intensive, since depending on the review procedure, they require 8 to 20 LOC/minutes for each person in the software team to inspect the source code [11]. Learning based models

[1]In this research, we use *defect* and *fault* interchangeably to define a product anomaly or imperfections found during software product life cycle, which causes the software fail to perform its required function.

[2]A module can be a package, file, class or a method in a source code.

have been one of the widely used approaches in both empirical software engineering research and industrial practice [11, 13, 23, 24, 26, 27]. Our previous study conducted on a large telecommunication company showed that we were able to predict, 80% of pre-release defects on average, while reducing the testing effort by 30% [13].

Research on defect prediction has a history of more than 20 years with one of the most promising works based on predicting pre-release defects using static code metrics [11]. Static code metrics, such as lines of code measures, Mc Cabe's cyclomatic complexity measures, and Halstead's operand-operator count measures, are widely used and easily extracted from the source code through automated tools [28]. As new web and mainframe technologies, 3GLs, and 4GLs, are emerging, it is more challenging to extract code metrics due to lack of parsers for these new generation languages [29]. Although static code metrics are successful in predicting 71% defects on average, it has been shown that the performance of defect predictors using static code metrics was upper bounded due to the limited information content of these metrics [30]. Due to this fact, different algorithmic approaches such as Naive Bayes and rule based learners, could not go beyond the performance achieved so far. For this reason, recent research in defect prediction has been concentrated on increasing the information content of the model by adding various metrics from other aspects of software engineering. So far, churn metrics [23,24,31,32], network dependencies [33–36], and organizational network of developers [27, 34, 37] have been used to enhance the performance of prediction models. In addition, some researchers investigated the effect of requirements, design or testing processes on predicting defective parts of the software by proposing new metric sets from either of these processes [12, 38, 39].

Churn metrics represent the progress in development activities throughout a project's life cycle in terms of added/deleted lines of code, file age, or rank of committers who edit the source code. They can be extracted from the change history stored in version control systems through custom scripts. Studies in [24, 32] used churn metrics to predict defects prior to a release, whereas [23, 27, 31] used them to predict post-release defects. Caglayan *et al.* [32] showed that churn metrics were better indicators than code metrics in predicting pre-release defects for Eclipse releases. Our previous research also concluded that churn metrics were the best indicators of defects regardless of their categories in terms of the phases they were found (i.e., during functional tests, system tests, or from the customer side) [29].

Call graphs extracted from the dependency (static caller-callee) relations between software modules were used to re-weight metrics [35], increase the number of metrics [36], or as a complete alternative to existing metrics in defect prediction models [33, 34]. Nagappan *et al.* [33] found that network metrics predicted post-release defects in Windows binaries with 10% more accuracy compared to code metrics. Tosun *et al.* [35] also extracted network metrics from a large-scale project and three small-scale projects in order to validate the effects of network metrics on different software projects. Authors claimed that network metrics were valuable and significant indicators of pre-release defects only in large-scale complex systems.

On the other hand, Meneely *et al.* used social metrics from the collaboration network of developers, and obtained better prediction results than using static code metrics in a case study [37]. Similarly, a comparative study by Nagappan and Murphy [27] showed that organizational metrics were the most significant indicators of post-release defects compared to code, churn, and network metrics. In contrast to the results obtained by [27], Weyuker *et al.* [40] added new metrics for developers who accessed and edited individual code units to their prediction model. Experiments showed that number of developers was not a significant measure for pre-release defects.

Static code metrics and network metrics use the *product*, i.e. source code, to predict defective parts of a software system, whereas churn metrics use the development *process* through mining version control systems. Social/organizational metrics look at the *people* aspect of a software development life cycle by considering the effect of development team on defect proneness of software. To enrich the *process* aspect, several researchers also considered the requirements documents [38,41], class diagrams [39] and test execution records [12] as significant inputs to defect prediction models. In [38], requirements were mapped with defects by linking each requirement statement to one or more source files and then linking files to defects. Later, textual metrics from requirements documents were extracted to understand the weaknesses in requirement documents (e.g. # vague statements, incompleteness, conditionals). However, extracting textual metrics from requirement documents is a difficult task, and at most of the cases, researchers are unable to access requirements documents of software, or documents are informally written, or their language affects the metric extraction process.

Kpodjedo *et al.* [39] extracted design evolution metrics from class diagrams and concluded that when used in conjunction with known metrics(churn), design evolution metrics improved defect detection performance. Metrics were derived from basic information in class diagrams, such as number of added/deleted methods, number of incoming/outgoing relations, and number of modifications on these relations. Based on these information, authors defined *Class Rank* measuring the relative importance of the class in the system, while *Evolution Cost* quantifying the amount of changes in signature and relations (i.e. associations, aggregations) that class underwent [39].

Nagappan *et al.* [12] introduced testing in-process metrics, also called as STREW-J metric suite, in order to measure their effectiveness on predicting defects [12]. Their proposed metrics included but not limited to *number of test cases per lines of code/ per each requirement/ per each class* and *number of conditionals* used in a test case.

Most defect prediction models combine well-known methodologies such as statistical techniques [23, 24, 42] and machine learning algorithms [11, 13, 43]. A systematic literature review on fault prediction models have recently been conducted by Hall *et al.* [44]. Authors investigated how certain factors, such as the context of models, independent variables (metrics) used, the modelling techniques applied, influenced the performance of such models. Based on an analysis on 36 studies, it was found that most studies preferred simple modelling techniques such as Naive Bayes or Logistic Regression.

While some researchers prefer simple modelling techniques like regression models, others use more complicated models like Bayesian Networks that include causal relations between project and process features [45]. Simpler models like Nave Bayes effectively capture the relationship between defective modules and code metrics in various studies with the probability of detection (pd) and probability of false alarms (pf) as 71% and 25% respectively [11]. More comprehensive views onto software systems, such as Bayesian Networks (BN) [46–48], would, on the other hand, help decision-making and reasoning, rather than measuring only defect density. According to Fenton *et al.* [47], only metrics may omit “sometimes simple and obvious causal factors that can have major explanatory effect on what is observed and learned”. BNs, on the other hand, are causal models, which consist of variables as nodes with different probability distributions and cause-effect relationships between variables as directed edges [46]. The effect of one variable on others can be observed in many useful combinations so as to find the best cause-effect relationship for software defects and final reliability [45].

2.3. Applications of Bayesian Networks in Software Engineering

BN approach has been widely used in various risk assessment problems in software engineering domain, mainly for predicting software and system reliability as well as for predicting resources or effort to accomplish a project. One of the earliest studies on predicting system safety was proposed by [49] in which a Bayesian Belief Network was constructed to make reasoning between system variables and safety. Later, different authors modelled software failures in an operational profile using a time series (Markov chain BN) [50], as well as using simple BNs (such as [47, 51, 52]). In [53], anomalies in failing executions of programs were detected online using fully observable Markov models. BNs were also formed for sensitivity, uncertainty and trade off analysis (such as [54], [55]). There were also studies that built BNs to predict software project effort and schedule variance, and to predict testing effort in iterative development, some of which are listed in Tables 2.1 and 2.2.

In Tables 2.1 and 2.2, we list existing studies in software engineering that have successfully applied BNs for solving different software engineering problems. Although the application of BNs in software engineering is not limited to these studies, we have filtered only the most representative ones in terms of (i) how the BN structure is formed, (ii) which techniques are used to estimate prior and conditional probability tables, (iii) what type of input variables (cat: categorical, cont: continuous) are used. In most of these studies, final BN structure, i.e. causal dependencies, was assumed to be already at hand, or it was inspired from another model (defect flow [56], activity based quality [57]). Experts opinions fed by historical data also played a critical role for defining causal relationships between variables (such as in [9, 45, 47, 49, 52]).

In terms of techniques for estimating the parameters, i.e. prior and conditional probability tables (in case of categorical variables) and distributions (in case of continuous variables), either the experts were used ([45, 50, 51]), or missing parameters were learned from historical data using structure learning algorithms employed in BN tools, such as Junction tree [9,45,47,52,54] or polytree propagation [49]. Alternatively, when continuous valued variables were needed to be modelled or in case of missing data, very few authors applied a Monte Carlo method (MC), such as *Gibbs sampling* and *Expectation maximization* [50, 58]. These MC methods are quite successful and accurate for estimating the unknown parameters of probability distributions or when inferring posterior probabilities in the case of missing data [59].

Most studies defined their input variables as representative of three different aspects, *process, product, people resources*, in a software life cycle. These variables were often collected via surveys, whose responses are in a categorical range in all studies except [50, 60]. In some studies, discretization techniques were employed to convert continuous valued variables, such as product size, complexity, into categorical values [9,45,55]. Recently, [55] built hybrid BNs by incorporating continuous and categorical variables in a single model using a dynamic discretization method. Discretization is a straightforward approach when modelling a BN to create prior and conditional probability tables [61]. Among various (supervised or unsupervised) discretization methods, Fayyad and Irani's discretization method that uses entropy minimization heuristic is the most commonly used [62].

Beside the inability to make inference over continuous variables, most studies were limited due to lack of a mechanism for structure learning. Expert judgement incorporated with the historical data for post-release defects were the most commonly used technique. However, there may be circumstances when a model is needed to be built based on the historical data [61], due to the fact that experts may also be uncertain about dependencies between variables. It is known that the problem of learning the best structure for a BN is NP-hard [59]. Thus, proposed techniques in other applications should be investigated to select the most appropriate technique for our problem.

2.4. Bayesian Networks: Theory and Applications in Other Domains

We have investigated software engineering domain to list available studies utilizing Bayesian networks and highlighted their limitations. In this section, several contributions by other researchers from other domains are proposed to solve these limitations.

Table 2.3 lists a sample of studies from medical diagnosis and machine learning. Since 1990s, there have been hundreds of studies employing Bayesian networks on a specific problem and proposing new techniques. Contributions of these studies are mainly focused on either *qualitative* or *quantitative* parts of BNs [61]. Qualitative part deals with finding causal dependencies between variables and building the final structure of the model. Quantitative part, on the other hand, deals with parameter estimations such as prior and conditional probability tables (CPT). Ideally, both parts of a BN can be inferred from data [59]. When data is not enough to make inference, expert judgment is widely used to fill quantitative parts (CPTs) [63,64]. Furthermore, software tools such as Hugin [65], Netica [66] are helpful, although they have their own limitations (e.g. handling continuous variables, inference algorithms for insufficient statistics, building the structure). Murphy reviewed some of the most popular and recent software packages for dealing with graphical models, i.e., Bayesian networks and undirected graphs [67].

In Table 2.3, we list a sample of studies from the literature and highlight how these papers have overcome three major issues: (i) Parameter learning, (ii) Structure learning, and (iii) Handling continuous variables.

In medical diagnosis domain, data taken from case studies are much larger than those in software engineering studies. Therefore, most of the studies (such as [63, 68–70]) computed parameters of prior and conditional probability distributions by inferring from data. Authors proposed different techniques for structure learning, one of which proposed a Mutual Information (MI) based inference to find causal dependencies between input-output variables and Chow-Liu's algorithm to learn interdependencies between input variables [69]. In all medical diagnosis studies, input variables were treated as categorical (when they had continuous values, an expert opinion based method was used to create pre-determined number of bins).

In the context of Bayesian Network theory and machine learning, parameter learning was applied with different variations of Monte Carlo (MC) methods, such as *Metropolis Hastings* with Hamiltonian dynamics [61], *Importance Sampling* with Hybrid Loop Belief Propagation [71], *Expectation Maximization (EM)* for parametric and semi-parametric conditional probabilities [72] and *accelerated EM* for factorized mixture of Gaussians [73].

Studies on structure learning, on the other hand, contain variations in terms of algorithms or heuristics used for finding the best structure. For instance, [74] applied genetic algorithms on continuous variables such that six different parameters of the algorithm were optimized during cross validation experiments. In [73], all pair-wise interactions were calculated with the help of a heuristic inspired from hill climbing technique. Furthermore, in [75], an independence based test was applied to both continuous and discrete/categorical variables, which does not have a distributional assumption (like in Chi-square test). In summary, depending on the size of data sets to find the parameters of search techniques and convergence times of the algorithms, structure learning technique should be carefully designed for every problem.

2.5. Research Questions

We align our research questions in line with the above-mentioned discussions about the limitations of previous models in software engineering. The rest of this research will look for empirical evidence to the answers of these research questions:

(i) What are the software factors affecting software reliability?

(ii) How do software processes affect software reliability?

(iii) How can we build a predictive model that would estimate software reliability prior to release?

(iv) How can software companies build predictive models with local data rather than based solely on expert knowledge?

Table 2.1. A summary of software engineering studies, which are based on faults/failures and software reliability, using BNs.

Aim	**Tool?**	**Parameter learning**	**Structure learning**	**Data (Free?)**	**Input description**	**Category**	**Type**
Safety assessment of nuclear computer-based systems [49]	Hugin & Polytree propagation	Expert judgement with simplified elicitation mechanisms	Expert judgement	No	Requirements, design, system specification document, personnel qualities	People, Process, Product	Cat
Modelling discrete time failure data [50]	No	Historical data & Gibbs sampler on Markov BN	Expert judgement & Historical data	Space program (No)	operational profile and failure rates	Process	Cont.
Predicting level of fault injection [51]	Netica & Bayes Net Assistant	Expert judgement & Historical data	Linear regression & PCA	Motorola software system (No)	Domain knowledge, problem complexity, stability	Product, People	Cat
On-line anomaly detection in failing executions [53]	No	Baum-Welch algorithm	Fully observable Markov model	NanoXML (No)	Predicate states obtained from program execution	Product	Cont
Predicting fault content & proneness [60]	No	Expert judgement & Historical data	Generalized Linear Model	NASA dataset (Yes)	Object oriented code metrics	Product	Cont
Predicting reliability [52]	AgenaRisk	Expert judgement & Junction tree	Expert judgement	Software project (No)	People and process quality, project complexity, novelty	People, Process, Product	Cat
Predicting reliability [47]	AgenaRisk	Weighted TNormal distribution using ranked nodes, then Junction tree	Expert judgement & Historical data	Software project (Partially)	People experience, domain knowledge, testing quality	Process, People	Cat
Predicting reliability [45]	AgenaRisk	Expert judgement for conditional and prior probability distributions, then Junction tree	Expert judgement & Historical data	Software project (Partially)	Testing, requirements, development quality factors, defects	People, Process, Product	Cat
Predicting reliability [9]	AgenaRisk	Dynamic discretization & Junction tree	Expert judgement & Historical data	Software project (Partially)	Testing, requirements, development quality factors, defects	People, Process, Product	Cat
Quality prediction [57]	AgenaRisk	Expert judgement	Activity based quality models to BN transition	NASA & Tomcat servlet (Yes)	Code attributes, effort	Product	Cat

Table 2.2. A summary of software engineering studies, which are based on project cost/schedule and effort estimation, using BNs.

Aim	**Tool?**	**Parameter learning**	**Structure learning**	**Data (Free?)**	**Input description**	**Category**	**Type**
Effort estimation [58]	Hugin & Powersoft	Historical data & Expectation Maximization	Expert judgement	Web projects (No)	size related metrics,survey responses	Product	Cat
Predicting time-cost-quality tradeoff [54]	AgenaRisk	Junction tree	Axioms to build BN	Software project (Partially)	Resources, quality, functionality	Product	Cat
Predicting software change effort [76]	No	Expert judgement & expert elicitation method	Expert judgement	Nordic manufacturing projects (No)	People, documentation, size, coupling	People, Product, Process	Cat
Predicting defect correction effort [56]	No	Expert judgement & Historical data	Defect flow model to BN transition	Bosch system (No)	Requirements volatility, complexity, effort	Process, Product	Cat
Sensitivity, uncertainty analysis [55]	AgenaRisk	Dynamic discretization & Junction tree	Hybrid BN	CPU module of a cardiac assist system (Partially)	Time to failure	Product	Cat
Predicting project schedule variance [77]	Netica	Expert judgement	K2 algorithm	Software from a Beijing corporation (No)	Coding and design complexity, programmer experience, staff turnover, requirements volatility	Product, People	Cat
Predicting test effort with dynamic BNs [78]	Agena	Expert judgement	Expert judgement	Two synthetic projects (No)	Test process effectiveness, test tool quality, faults, test team, effort	Process, People	Cat
Building basic blocks (idioms) for risk BN [79]	Serene	N/A	Expert judgement & idiom definitions	N/A	N/A	N/A	Cat

Table 2.3. A list of studies on Bayesian Networks and applications in medical diagnosis systems.

Aim	Parameter learning	Structure learning	Input variables
Medical Diagnosis			
Diagnosis of liver disorders [63]	Data	Expert judgment	Binary and discretized continuous features
Ovarian cancer diagnosis [64]	Expert judgment	Expert judgment	Discretized continuous features
Assessing the effects of penetrating trauma [68]	JavaBayes (junction tree)	Expert judgment	Categorical?
Surgery survival chance prediction [69]	Data	Mutual information (MI) based inference & Chow-Lius algorithm	Categorical
Structure learning (assuming data is complete) [70]	Data (All parameters are assumed to come from Dirichlet (Gamma) dist.)	Binarization of MI based matrix to prevent majority of structures	Categorical
Machine Learning			
Structure learning [74]	Data	Genetic algorithms to find optimal ordering of variables	Categorical
Handling continuous attributes using the best of two techniques [72]	Dicretization and EM (for parametric and semi-parametric cond. Prob.)	Learned from data	Continuous
Mix-net for both continuous and discrete variables [73]	Accelerated EM with factored mixture of Gaussians	Measuring all pairwise interactions between variables	Continuous & Categorical
Distribution free learning for continuous variables [75]	Not available	Conditional independence test based method (non-parametric) & recursive median alg.	Continuous
Handle Hybrid BNs [71]	HEPIS (Hybrid Loop Belief Propagation based importance function	Expert judgment	Categorical & Continuous
Building BNs [61]	1. Data and junction tree 2. MCMC methods	Expert judgment	1.Categorical 2.Continuous

3. PROPOSED MODEL: HYBRID BAYESIAN NETWORK

In this research, we propose a comprehensive Hybrid Bayesian Network containing sub-networks representing three major processes in a software development life cycle (Requirements specification and documentation, Development and Testing) to improve decision making for software managers by predicting the reliability in terms of residual defects. Each sub-network is represented with different factors, i.e. mixture of continuous and categorical variables, whose causal relationships would represent the reliability of the associated process. Combinations of these subnets build a local model for a software organization that can be used to decide when to release the software.

BNs are graphical models that encodes probabilistic relationships among a set of variables. They allow to learn causal relations, and they are capable of handling incomplete data [59]. BNs were previously employed to predict software reliability with residual (post-release) defects (For example, [9,45]). Similarly, three aspects of software engineering, *Process, Product* and *People* were treated as interconnected such that each factor corresponding to one of these aspects would eventually affect other factors as well as the final reliability. Therefore, proposed networks took into account causal relationships of these aspects and effects on software reliability. Despite similarities, our proposed model differs from previous approaches in a number of ways.

Hybrid Data Approach: BNs proposed in previous studies often required large amounts of data representing different processes in SDLC. Collecting large amounts of accurate/reliable data is very challenging when organizations do not follow a systematic measurement and quality assurance activities. Surveys are valid and cheaper methods used by many researchers [80–82], compared to other data collection techniques (e.g. mining repositories, or building a new repository [13, 29, 83]). Thus, models were fed with data collected via surveys conducted with project managers and development team. Fenton *et al.* [45] also used a questionnaire to build a BN including all major phases in a life cycle in order to predict residual defects in a software system. Authors claimed that such models were almost impossible to build using limited data, hence they could be consistently built based on subjective expert judgements.

However, in organizations that already have a mature measurement system, historical data collection may not be a concern. On the contrary, due to high turnover within development teams in certain industries, expert knowledge may be insufficient or non existent. In such cases, collecting data with surveys may not be feasible and/or reliable. Therefore, data dependent models would be more effective. In this paper, we employ a *hybrid* data collection approach: Metrics representing the development and testing phases are collected from historical data through mining version control systems, code bases and metrics databases. On the other hand, metrics characterizing requirements specification and documentation phase are collected via a questionnaire proposed by Fenton *et al.* [45]. We have interviewed project managers to capture qualitative data for specification and documentation and project management processes.

Forming Causal Relations: Previous studies in software engineering incorporated expert judgement with data to build the structure of their networks (Section 2.3). However, it is sometimes necessary to base the final model onto the local data, especially when experts are not experienced enough or uncertain about dependencies between variables of the network. Structure learning in BNs (finding the optimal structure for a problem) is an NP-hard problem and there is not a unique set of solutions for this. In our proposed model, we define causal relations between variables of the hybrid network by statistical correlation and independence tests as well as graphical dependence analysis. Then, we formalize these relations in the network with the help of expert judgement.

Model Inference: Once the final model is formed, we need to determine various probabilities of interest from the model. In our problem concerning predicting reliability, we want to know the probability of residual defects given observations from other variables. This probability is not stored directly in the model, and hence needs to be computed. In general, the computation of the probability given the model is known as *probabilistic inference* [59]. Inferring posterior probability of the final node in a BN is easier with categorical variables, and hence, researchers in software engineering often handle continuous variables with discretization. Although discretization works well in small state spaces, it can clearly cause loss of information about the variables [72]. Furthermore, in case of missing data, it is inaccurate to learn discretized intervals from that data.

We utilize Monte Carlo methods, specifically the most commonly used one, *Gibbs sampling*, during inference, since this technique approximates the joint probability distribution of the model when it is not known explicitly, but the conditional distribution of each variable is known or easier to sample from [59]. In our problem, the joint probability of residual defects (with all input variables and parameters) is not known explicitly, but conditional distributions formed by causal relations between variables are set via statistical tests. Gibbs sampling is well suited for inferring the posterior distribution of a BN (in our case, it is $P(Defects \mid Metrics, \theta)$ where θ represents unknown parameters), since BNs are typically a collection of conditional distributions. Gibbs sampling also helps us to use mixture of continuous and categorical variables without the need of any transformation and estimate unknown parameters during inference.

4. METHODOLOGY

The proposed model requires a dataset construction step, a network construction step and a performance evaluation step.

4.1. Dataset Construction

Dataset construction step consists of two different techniques: (i) mining software data repositories to collect quantitative data, and (ii) in case of lack of data related to a software process, conducting a survey with technical leads and managers on software processes and principles applied in software organizations.

4.1.1. Quantitative Data

To collect raw data from data repositories, we have implemented automated scripts and queries that would access to version control systems, source code repositories and testing monitoring systems and to extract product and process related factors. An illustration showing data collection procedure can be seen in Figure 4.1. In a version control system, each entry shows a commit done by development team including the *file names* edited before this commit and the reason of the commit in *comment* part. We have also collected statistics such as *added/ deleted lines of code* by parsing commit logs. The most important information contained in a commit is the *key* part corresponding to an issue. An issue can be a feature request or a defect which is entered to an issue management system. We collected defects whose status is “Resolved” and classified them based on the “Phase” defect was found (as *test defects* that were found during testing, and *post-release defects* that were found externally –by users). Test execution records are also kept in a separate system on a timely basis. Tests are often collected in small packages (test suites) depending on the functionality each test corresponds to. For each test suite run, results with an explanation of “success/fail/error” are recorded in the system.

After raw data is formed, we define a set of software metrics for each software process in the organization. This metric set should be revised for each organization specifically based on the availability of required data. Data collection procedure and the final set of metrics for both organizations that we worked with are presented in Section 5. In this section, we will describe metrics that we collected from two organizations by investigating their effects on software quality in previous studies.

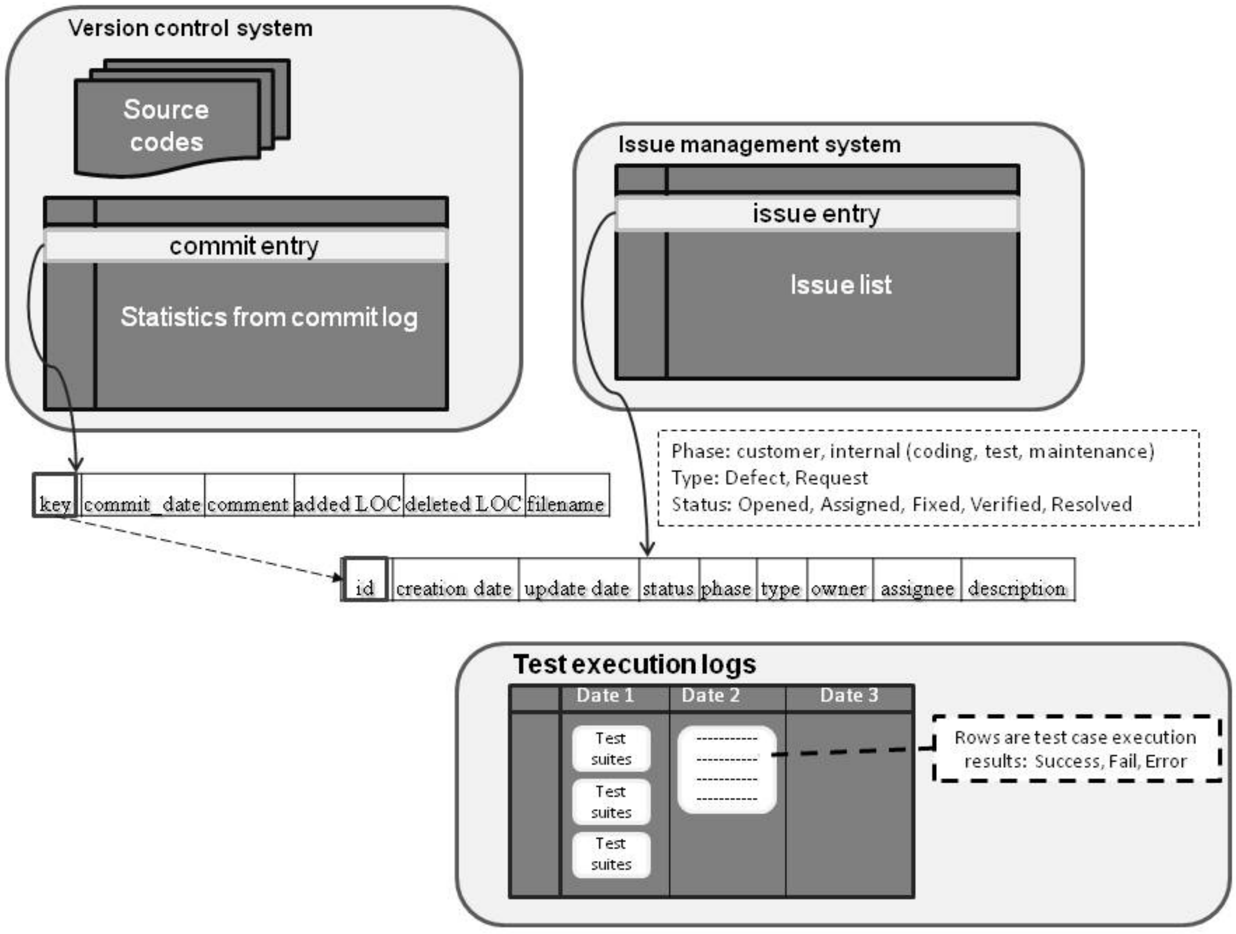


Figure 4.1. Data collection procedure from available resources of the software organizations.

4.1.1.1. Software Metrics Representing Development Phase. We have extracted 6 different metrics from the version history and code base in software organizations and calculated the average values of file-level metrics for each release to aggregate them to release-level. These metrics are *age of source files*, *average churn (added and deleted LOC)*, *number of committers*, *number of commits*, *average time between edits*, and *Mc-*

Cabe's cyclomatic complexity. Our aim is to capture product (i.e. complexity), process (i.e. age, churn, time between edits) and people (i.e. committers) aspects of the development life cycle through different kind of metrics. Depending on the availability of raw data required to extract these metrics, we extracted all or a subset of them for the organizations that we collaborated with.

Age: We selected source files created before a release and calculated the age in days since their creation date until the release. Previously, age was investigated along with size and complexity measures. While [84] reported that young and small files were more defect prone, [25] found no correlation between defects and size. In a recent study by Seifert and Peach [85], age, churn and unique committers to a file were reported as good indicators of software quality, i.e., defects. Thus, we have also included Age as a metric into our network.

Average Churn (Churn): We collected edited (in Dataset 1), added and deleted (in Dataset 2) lines of code in all files of a release and took the average lines of edited code for each release. Churn metrics were also previously used by various researchers to predict pre-/post-release defects in commercial and open-source projects. From [12, 27, 32], it is seen that churn metrics are significant indicators of defects in software systems. Thus, we have included Churn metric to our network.

# Committers (People): Previous studies done on commercial systems showed that organizational metrics such as the number of developers (i.e. committers) were statistically significant indicators of post-release defects with the best prediction accuracy, compared to churn and code metrics [27]. Therefore, we counted the number of unique committers who edited source files in a release and took the average number of committers for each release.

# Commits: This metric is also known as "churn count" in previous studies done by Nagappan and Ball [23] and counts the number of times an edit has been made on a source file. Commit count is generally the base metric to calculate other churn metrics, such as time between edits, average amount of edited/added/deleted LOC. But in one

of our datasets, we use this term as one of the in-process metrics in addition to others.

Average Time between Edits: This metric aims to measure the independence between consecutive edits (i.e. commits) on a source file, or conversely, dependencies between changes. As this metric value increases, this may indicate that there are several independent tasks completed on that file. We calculated this metric to add the extent of edits (changes) and distinguish independent tasks inside a development process. It is computed by measuring average number of days between consecutive edits in a release.

Cyclomatic Complexity (Complexity): Lastly, we computed McCabe's cyclomatic complexity metric for each file in a release using our open source metric extraction tool, Prest, [28] and computed average complexity values for each release. This metric, as the churn metric, is among the most popular indicators of defects in software systems and it is widely used in defect prediction studies (such as [11, 13]).

<u>4.1.1.2. Software Metrics Representing Testing Phase.</u> Similar to the metrics collected from development process, we have mined test monitoring systems employed in organizations and extracted several metrics representing the activities during testing phase. These metrics are *# test cases*, *# test cases with errors*, *# test cases with failures*, *% test cases with a status change*, *# test defects*, *test case quality* and *average # of executions per test case.*

While selecting testing metrics, we have investigated previous studies which measured and analysed testing phase in software organizations [12,86,87]. A previous study proposed by Kan *et al.* [86] provided a detailed discussion on key in-process metrics for managing testing process based on their experience in IBM Rochester software development laboratory. Nagappan *et al.* also introduced 7 in-process testing metrics, also called as STREW-J metric suite, in order to measure their effectiveness on predicting software reliability [12]. Their proposed metrics included but not limited to *number of test cases per lines of code, per each requirement, per each class* and *number of* *conditionals* used in a test case.

We have also used several of these metrics proposed by Kan *et al.* [86] and by Talby *et al.* [87] as our testing in-process metrics, but we set the granularity as release-level for the purpose of this study. Furthermore, we have added a test case quality metric to our first dataset in order to measure how logic and data flows in a test case are related to defect proneness of a software release. This metric was previously defined as # conditionals per lines of code in [12]. We did not have access to test source codes in both organizations, but the first organization had already measured test case quality metric for each test case and stored it in their databases. Thus, we also included that into our study.

# Test Cases: This metric is the vital component of testing progress curves defined by [86], to track the progress of a testing process. It is calculated by counting the number of executed test cases in a release or product. We have collected total number of test cases executed in every release. The data may be cumulative over time, but this situation does not happen in our datasets due to the fact that we focus on # test cases executed during user-acceptance, performance and regression testing phases.

# Test Cases with Errors: Similar to total # of test cases, test cases completed with errors are also essential to track testing progress and to take action upon early indications that testing activity is falling [86]. Generally, test cases completed with errors indicate that a step in a test case has failed during a run, but it does not cause a termination. We counted the number of executed test cases which were completed with errors in a release.

# Test Cases with Failures: Failures and unexpected terminations of test executions are not desirable scenarios and need careful examination during testing process. We also counted total number of test cases whose termination status is “failed” in a release.

# Test Defects: This metric is also considered as a common feature of many

testing tools [86, 87]. Although defect arrival and fix patterns have significant quality indications for the final product, we could only measure the number of test defects reported during testing phase of a release due to unavailability of these patterns.

Average # Executions per Test Case (Test Run): Test execution records, especially test cases which are continuously failed or completed with errors, are significant indicator for a testing process [86]. However, these records were not available in all organizations. As an alternative, we reported average execution times of test cases in a release. Having higher execution times in a release show defect tracking and fixing progress, since multiple runs of a test case are most probably mapped with one or more defects. Several test case runs for a specific defect may also catch other defects.

% Test Cases with a Status Change (Tests Change): This metric is also related to execution records. The percentage of test cases whose status went from (i) pass to fail/error, (ii) fail to error or (iii) error to fail is also an indicator of the degree to which inadequate or incorrect code changes were made [86]. It is also defined as *defect injection rate*, since when the status of a test case changes, it is an indication that a code change was made and there is a possibility that this change causes a defect. To measure this metric, we monitored the status of test cases and calculated percentage of those whose status changed as (i), (ii) and (iii).

Test Case Quality: Quality of test cases is also as critical as their execution patterns. Therefore, we have defined this composite metric based on (i) test runs and (ii) the number of test cases executing in only 1 step and more than 1 steps in a release, as to measure how many independent paths a test case can cover inside the code. Having only 1 execution step in a test case may indicate that its capability of catching defects in the code is lower compared to other test cases with more than 1 execution steps (branches or if-else conditionals). We calculated this metric based on the ratio between average test runs for test cases with 1 step ($TestRun_{Step=1}$) and more than 1 steps ($TestRun_{Step>1}$). Smaller values of this metric are preferred since 1-step test cases should be avoided in a company.

4.1.2. Qualitative Data

The reason of conducting surveys for some of the software processes in this research is that there is not always historical data characterizing certain processes in the organizations. For instance, quantifying requirements specification and project management processes with metrics is a hard task. Furthermore, there may be a more stable expert knowledge with senior managers and analysts having years of experience and knowledge about these processes.

Fenton *et al.* [45] published a questionnaire consisting of 5 major topics representing different phases of a development life cycle. In this questionnaire, each topic is represented with qualitative factors that authors and other partners in their project believed had a significant influence on the outcome of a project. Each factor is described by a question to be answered. These descriptive questions were specifically tailored for the organizations that authors worked with so far.

Although the original questionnaire has 5 major topics representing different phases of a development life cycle (i) specification and documentation process, (ii) design and development process, (iii) testing and rework process, (iv) project management, (v) new functionality), we only collected responses for the topics (i) and (iv) to incorporate requirements specification process into our network. Specification and documentation subnet proposed in [45] includes 4 factors from the same process and 2 factors from project management process. Authors claimed that these subnets were built based on expert knowledge incorporated with empirical analysis in the literature. Hence, we also used the same factors while building the requirements specification subnet in our models. These factors are described in Table 4.1. Each factor is explained with additional questions and evaluated with a 5-scale point by participants: Very Low, Low, Medium, High, Very High.

4.1.2.1. Missing Data Imputation. Since we have two different sources of data (qualitative based on surveys and quantitative based on historical data), it is often the

Table 4.1. Specification and Documentation Factors used in the network.

	Factor Name	Descriptive question
S1	Relevant experience of spec and doc staff	How would you rate the experience and skill set of your team members for executing this project during the requirements and specifications phase?
S2	Quality of documentation inspected	How would you rate the quality of the requirements given by the client or other groups?
S3	Regularity of spec and doc reviews	Have all the Requirements, Design Documents and Test Specifications been reviewed in the project?
S4	Standard procedures followed	In your opinion, how effective was the review procedure?
S7	Requirements stability	How stable were the requirements in your project?
P2	Configuration management	How effective is the project's document management and configuration management?
P5	Stakeholder involvement	To what extent were the key project stakeholders involved?

case when a one-to-one mapping from qualitative responses to quantitative data is not achieved. Surveys conducted with technical leads and managers usually represent a general evaluation of the processes rather than a scoring for a specific release. Therefore, even though we work to map each response or a set of responses to a specific release, there exists missing rows in qualitative part of our datasets.

Imputations are general methods for handling missing-data problems. There are explicit and implicit modelling approaches both of which require a method of creating a predictive distribution for the imputation based on the observed data [88]. For instance, in explicit modelling, *mean imputation* is the simplest technique, where mean of the observed records is assigned to missing values. *Hot deck imputation* is popular in survey practice, such that missing values are replaced by similar observed values (with random sampling with replacement). It is an implicit modelling method which is useful when missing values are relatively few. In short, as we impute a missing value (y_i^*) by randomly selecting an observed sample ($Y_{1..i-1}$) from the dataset, we treat the new instance as observed (y_i) and impute the next missing instance (y_{i+1}^*) by selecting from new set of observed values ($Y_{1..i-1} \cup y_i$).

We can also think filling missing data from the perspective of Bayesian inference. If we knew the parameters of the data, then it would be possible to obtain predictions for missing values. Suppose data come from a normal distribution with parameters μ and σ. Missing values can be generated from this distribution if we estimate the mean parameter (μ) using Bayesian inference. The critical part in this approach is the choice of prior distributions. Monte Carlo methods, such as Gibbs sampling, can be used to infer parameters based on full posterior distribution, by integrating the conditional distribution of the unknown parameter (e.g. $p(\mu \mid Y, \sigma^2)$ where Y are known values) over the posterior distribution of σ^2 [88].

In this research, we apply several imputation techniques: In dataset 1, where the ratio of missing values is small enough (11%), we apply mean imputation. In dataset 2, we use Bayesian inference to estimate parameters of the underlying distribution and generate samples from that using estimated parameters. We explain the details of both techniques in Section 6.1.

4.2. Network Construction

Bayesian networks help to make reasoning under uncertainty by structuring any situation using causal relations between events [59]. In a Bayesian network, there are a set of variables (represented as nodes) and a set of directed edges between variables. Each variable has a finite set of mutually exclusive states, or has a conditional probability table when it has parents (in discrete case). In continuous case, each variable is represented with a prior probability distribution, or when it has parents it is represented with a conditional probability distribution [59].

We draw an arrow from a variable A, to another variable B, ($A \rightarrow B$) if A is one of the parents of B. In this case, conditional probability distribution of B is represented as $p(B \mid ..., A)$. To solve Bayesian networks, joint probability distributions are used where we benefit from the product of all potentials, i.e. conditional and prior probability distributions of the variables in the network. For N variables $x_1, x_2, ..., x_N$ with discrete values, joint distribution can be expressed as [89]:

$$p(x_1, x_2, ..., x_N) = \prod_{i=1}^{N} p(x_i \mid parents(x_i)) \tag{4.1}$$

Whenever new information about a variable comes, i.e. evidence, we can update the joint probability table using the chain rule from Bayes Theorem [90]. There are automated software tools for building Bayesian network such as Agena [91], Bayes net toolbox [92], Hugin [65] and Netica [66].

In this research, we aim to use Bayesian learning for complex models using *Monte Carlo*(MC) methods [59]. In case of *insufficient statistics*, i.e. unknown parameters of distributions, or *incomplete data*, these methods work remarkably well in practice and are widely used in machine learning, bioinformatics, medical domain, and other engineering disciplines. The basic idea behind Gibbs sampling, which is one of the most popularly used MC method, is to successively sample from posterior distribution of each node in a Bayesian model given all the others as full conditionals. This Monte Carlo technique is quite successful when estimating the unknown parameters of probability distributions or when making empirical analysis to infer true values of a given sample [59]. An example of Gibbs sampler showing its iterative procedure is explained in [93] as follows:

Suppose independent variables s_1, s_2 are parents of a variable x in a network. If s_1, s_2 do not have parents, the joint distribution of this network can be written in terms of conditional distributions, such as $p(x, s_1, s_2) = p(x \mid s_1, s_2)p(s_1)p(s_2)$. Gibbs sampler starts by assigning initial values to $s1, s2$ based on their prior distributions $(q(s_1), q(s_2))$, whereas x has the evidence (set of known values). Afterwards, samples for $s1, s2$ are generated from their conditionals, as in Equation 4.2, until the model converges. Convergence may take longer especially for more complex models.

$$s_2^i \sim q(s_2) \tag{4.2}$$

$$s_1^{i+1} \sim p(s_1 \mid s_2^i, x = \hat{x}) \tag{4.3}$$

$$s_2^{i+1} \sim p(s_2 \mid s_1^{i+1}, x = \hat{x}) \tag{4.4}$$

Finally, to build the final structure of the network, we apply independence tests derived from well-known theories [94] as well as graphical dependence analysis [95], since the problem of finding the best structure among all possible configurations is not tractable.

The general network is illustrated in Figure 4.2. Gray rectangle nodes represent the subnets of our proposed model: Requirements Specification, Development and Testing. Output variable 'Software Reliability' is fed by all subnets. Rectangles with dashed lines represent requirements specification, development and testing subnets as illustrative purposes. In this figure, subnets are considered as independent, however they may have dependencies among each other, which is also considered while building the model. Output variable represents the final reliability with residual defects.

4.3. Performance Evaluation

To assess the performance in terms of predicting residual defects, we use *Mean Relative Error (MRE)*, *Mean Magnitude of Relative Error (MMRE)*, *Median Magnitude of Relative Error (MdMRE)* and *Prediction at level k (Pred)* which are commonly used and suggested to validate such type of models especially in software cost estimation [45,96,97]. Equations for all measures are presented in Equation 4.5. In MRE formula, y_i represents the actual number of defects, while $\hat{y}_i$ represents the estimated number of defects. MMRE was criticized due to the fact that it was very sensitive to outliers and it might be "unreliable". Since MMRE is dependent on the variance of MRE, an alternative measure, *Pred*, is proposed. According to recent studies [96, 98], *Pred*

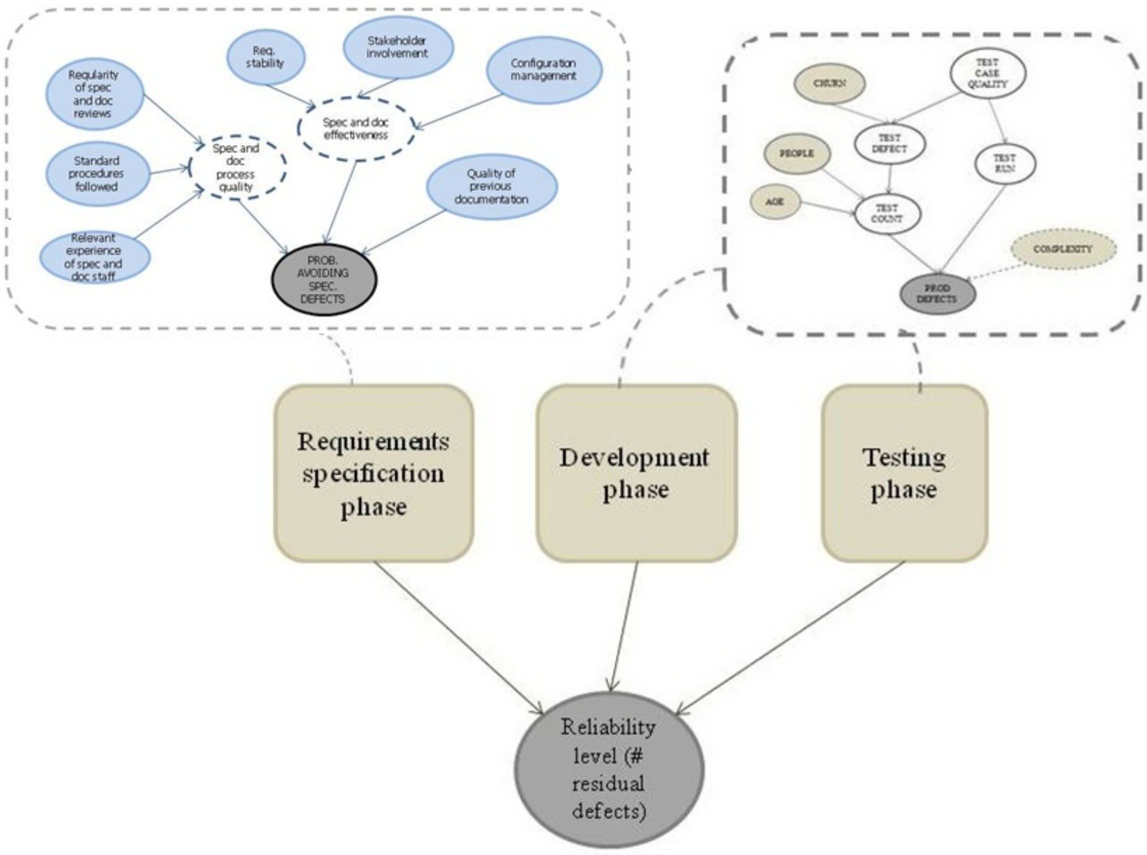

Req. stability
Stakeholder involvement
Configuration management
Regularity of spec and doc reviews
Spec and doc effectiveness
Spec and doc process quality
Standard procedures followed
Quality of previous documentation
Relevant experience of spec and doc staff
PROB. AVOIDING SPEC. DEFECTS
CHURN
TEST CASE QUALITY
TEST DEFECT
PEOPLE
TEST RUN
AGE
TEST COUNT
COMPLEXITY
PROD DEFECTS
Requirements specification phase
Development phase
Testing phase
Reliability level (# residual defects)

Figure 4.2. An illustration of the proposed network.

has been more consistent and robust than MMRE, since it is simply the number of instances which the model produces an MRE less than a pre-defined level k. Hence it is not dependent on the variance of MREs or the dataset size. We also report MdMRE which reports the median of MREs as an alternative to MMRE which is the mean of MREs.

$$MRE = \mid y_i - \hat{y}_i \mid / y_i \tag{4.5}$$

$$MMRE = \frac{1}{N}\sum_{i=1}^{N} MRE_i \tag{4.6}$$

$$MdMRE = Median(MRE) \tag{4.7}$$

$$Pred(k) = \frac{1}{N}\sum_{i=1}^{N} MRE_i \quad \text{if } MRE_i \leq k \tag{4.8}$$

In Bayesian networks, a natural way to compare two models is to use criterion based on trade-off between the fit of the data to the model and the complexity of the model [99]. Deviance Information Criterion (DIC) is one of these criteria whose quantities are easier to run in Monte Carlo chains. It is calculated as follows:

$$D(\phi) = -2logL(data|\phi) \tag{4.9}$$

$$\bar{D} = E_{\phi|y}[D] \tag{4.10}$$

$$D(\bar{\phi}) = D(E_{\phi|y}[\phi] \tag{4.11}$$

$$p_D = \bar{D} - D(\bar{\phi}) \tag{4.12}$$

$$DIC = \bar{D} + p_D \tag{4.13}$$

In 4.9, the first term is the deviance calculated from the log likelihood of data. Complexity in terms of number of parameters (pD) in the model is calculated as "posterior mean deviance" minus "deviance evaluated at the posterior mean of the parameters" [99]. Models with smaller DIC are better supported by the data. We report DIC values for all models and evaluate how they fit to the data.

5. DATASETS

We have collected data from two large-scale software companies in Turkey working at two different domains: Telecommunications and ERP.

5.1. Dataset 1 from Telecommunications Domain

We mined the version control system, issue management system and testing monitoring system of a large scale software product developed and maintained by the leading telecommunications company in Turkey. Research & development centre of the company hosts more than 200 developers, testers and architects developing software products and solutions for mobile operators all over the world. Some of these solutions are network solutions, value added services, subscriber identity module (SIM card) related solutions, terminal based solutions, billing and charging solutions, data mining, data warehouse, customer and channel management systems and applications.

Their legacy system contains millions of lines of code that are maintained by the local development teams. The majority of their software is implemented with Java, Jsp, PL/SQL and other new technologies such as Service Oriented Architecture (SOA).

Every 2 weeks, a new release –on the main branch of their version control system– with 10 to 15 work packages (smaller units, such as projects) and more than 400 interfaces of their major components (i.e. solutions summarized above) goes into production. Since the release period is short, each release package contains at most a single functionality, such as a new campaign, or two new functionalities and the rest is modifications/ upgrades for different components of the current system. Design and development activities are engaged in the company such that there is not a well defined design process applied prior to coding, but architects and developers work in cooperation. The development activities are stored consistently in a popular version management system. We mined historical logs of the version control system through scripts and extracted 4 in-process metrics, namely *average churn*, *average # commit-*

ters, *average time between edits* and *age*. Furthermore, the latest stable version of the product prior to each release was checked out from source repository and *cyclomatic complexity* metric was calculated using Prest [28]. Test execution records and schedules are stored in a testing monitoring system, whose database records were used to extract 4 additional metrics and aggregate at release level: *# test cases executed*, *# test defects found during testing*, *test case quality metric* and *average # of executions per test cases*.

Finally, defect data is stored in the company's issue management system where post-release defects are specifically monitored for their implications on cost control and resource allocation. Every defect that is reported after a new release is labelled as a *post-release defect*. Finally, we collected number of post-release defects and 9 in-process and product metrics associated with 44 releases between 2009 and 2011.

As the qualitative part of the dataset, we conducted interviews with 15 team leads and project managers to complete the survey presented in Section 4.1.2. We applied this survey twice in the company: (i) to evaluate the first 20 releases, and (ii) to evaluate the next 19 releases. Each release was evaluated by 3-5 people, depending on which teams actively worked on that release. Most popular response (rating with the highest number of responses) for each question was added to final dataset. In total 39 releases were evaluated for the first dataset.

5.2. Dataset 2 from ERP Domain

To validate the success of hybrid bayesian networks on predicting post-release defects and release confidence, we added another dataset to our study collected from the biggest independent software vendor in Turkey with more than 150000 customers. The company produced comprehensive ERP solutions and business applications for SMEs (small and medium size enterprises) including material management, purchasing, accounting, sales management and more.

We selected one of the major products of the company developed by a total of 15 people, including developers, testers and analysts. Additionally, the company has an on-site maintenance team located at different customers to solve problems in the software and answer custom requests. The development team is well aware of the importance of measurement and analysis, and hence, they have a mature data repository incorporated with open source management solutions, such as SVNStat for version control system management, and Hudson for testing monitoring and scheduling. The release period includes 5 weeks for development and unit testing, continued with 2 weeks for system, regression and user-acceptance tests. Every night, a build is made and automated test scripts are run. However, to increase data points in our dataset, we extracted 43 weekly metrics between January 2011 and December 2011 and predicted post-release defects on a weekly basis.

We mined historical logs of available tools and extracted 6 in-process metrics related to development process, namely *# commits, # unique committers, # added LOC, # deleted LOC* and *average time between edits*. In addition, we implemented a Java program to extract 6 test metrics: *# test cases executed*, *# test cases completed with errors*, *# test cases completed with failures*, *% test cases with a status change*, *# test defects found during testing* and *# test suites/packages executed*. Different from the first dataset, we had access to test execution logs, and hence we extracted more detailed metrics including number of test cases completed with errors, or with failures and percentage of test cases whose status has changed from error to fail, success to error/fail. We also defined a new metric, similar to number of test cases executed, *# test suites/packages executed*, in order to monitor how many different functionalities have been tested in a week. Test suites are large packets with several test cases specifically written for a specific functionality or scenario. Thus, we also added this metric to increase variability. Furthermore, we computed the edits in lines of code as *added* and *deleted* separately. However, we could not calculate the *age* metric in this company.

As for the qualitative part of this dataset, we interviewed with 15 people including developers, testers and analysts and asked them to evaluate the maturity of their software processes. The final dataset consists of only 15 instances with 7 different responses for requirements specification and documentation process, since we only had one chance for these interviews and not all releases were evaluated separately. We applied data imputation techniques for the qualitative part of this dataset in order to combine it with quantitative data.

6. EXPERIMENTS AND RESULTS

6.1. Descriptive Statistics on Software Metrics

6.1.1. Dataset 1

In Table 6.1, we report minimum, maximum, mean, standard deviation of metrics collected from 44 releases. As Turhan stated in [100], in datasets with large variations and possibility of outliers, reporting median instead of mean should be preferred as it is more robust to outliers. Thus, we also added median values of metrics to Table 6.1. Box plots representing lower/upper quantiles and medians for each metric can also be seen in Appendix E.

Table 6.1 shows a large variation in data such that source files are around 3 years old (930 days) on average, while there are almost 10 years old files (3144 days) as well as 2.5 months old files (75 days) in the software. On average, 67 LOC are edited in every release, which means a large portion of LOC is maintained with small modifications done in every release. Table 6.1 also shows that very few people (around 2 committers on average) edits the source code in a release, which may indicate that releases contain small modifications/ updates for a specific application in the software rather than large updates on the entire system. It may also indicate the code ownership in the company [101]. Values of churn support our prior hypothesis: Files are edited in every 4 months on average, due to independent and small modifications in every release. However, some files, which may contain close dependencies to various components in the system, are edited in every 9 days. Average complexity of source files in a release is around 4.8; indicating that majority of files are not so complex in contrast to their age. Finally, there are almost same number of test cases with 1-step as test cases with more than 1-step in the dataset, which may bring the question of "how successful are these test cases in catching defects?". Based on average number of post-release defects reported for 44 releases, it is seen that the quality of test cases should also be further investigated. We left this issue as another research topic.

Table 6.1. Dataset 1: Software metrics regarding development and testing phases.

Metrics	Min.	Max.	Mean	Stdev	Median
Age	74.7	3144.5	930.0	652.7	863.9
People	1.0	4.0	1.6	1.0	2.0
Churn	10.0	230.0	67.0	55.0	53.0
Time between edits (EDITFR)	9.3	499.3	120.2	114.9	86.1
Complexity (COMPL)	0.5	11.1	4.8	2.6	4.3
Test cases (TC)	1.0	101.0	52.2	21.5	51.0
Test defects (TD)	1.0	1303.0	179.0	261.6	116.5
Test case quality (TCQ)	0.2	1.8	1.1	0.4	1.1
Test run (ATR)	1.0	7.7	1.9	1.0	1.7
Post-release defects (DEFECT)	12.0	91.0	46.9	21.2	48.0

Values in Table 6.1 also highlight some exceptional cases, such as 1303 defects reported during testing, and 91 post-release defects, which may have occurred due to miscalculations or inaccurate raw data. We do not filter these extreme values during model construction, but we applied outlier detection techniques on both datasets and discussed their advantages as well as limitations in predictive studies in Section 6.5.

Responses to two major processes of the survey conducted in both software organizations can be seen in Appendix C. Since there are 39 responses in total for the first dataset, we applied mean imputation to fill missing evaluations of the final 5 releases. Mean imputation assigns the mean of observed values to the missing value, however in our case, since the data is categorical, we calculate the mode instead of mean, by counting the most frequent response given for each question, and assign the mode to missing rows. For instance, in Table C.1 for Dataset 1, S1 has 4 responses of '2', 9 responses of '3', 10 responses of '4', and 16 responses of '5'. Thus, the mode for S1 is '5'. In the final set, missing 5 rows are filled with '5, 3, 3, 2, 2/3, 3, 3' in which S7 has equal number of rows with responses of '2' and '3'. Hence, with 50% probability, a row is filled with 2 or 3.

Table 6.2. Dataset 2: Software metrics regarding development and testing phases.

Metrics	Min.	Max.	Mean	Stdev	Median
Commits	52.0	662.0	178.5	123.5	149.0
People	5.0	12.0	8.5	1.5	9.0
Added LOC (ADDLOC)	11.0	670.0	78.8	108.7	43.0
Deleted LOC (DELLOC)	2.0	202.0	32.7	40.3	17.0
Time between Edits (EDITFR)	0.2	2.0	0.8	0.4	0.7
Complexity (COMPL)	0.2	4.3	1.8	0.8	1.8
Test Cases (TC)	49.0	11067.0	2470.3	2713.7	1236.0
Test Cases with Errors (TCE)	0.0	1735.0	281.9	367.9	157.0
Test Cases with Failures (TCF)	0.0	475.0	87.0	113.5	40.0
Tests with Status Change (TCST)	0.0	29.0	4.4	7.8	0.0
Test Suites (TS)	1.0	224.0	41.8	52.4	20.0
Test Defects (TD)	12.0	121.0	58.3	21.2	60.0
Post-release Defects (DEFECT)	2.0	29.0	12.1	7.0	11.0

6.1.2. Dataset 2

In Table 6.2, we report minimum, maximum, mean, standard deviation of metrics collected from 43 releases. This dataset shows a higher amount of churn compared to the first dataset, even though metrics are collected on a weekly basis rather than on every two weeks. On average, 8 to 9 developers work on the source code and they edited the code approximately in every 3 hours (0.7 days between edits). Table 6.2 also shows that more than 1000 test cases are executed every week, of which 3.5% on average (87 out of 2470 test cases) is failed and nearly 4% of their status has changed to fail/error. We checked the correctness of very high and very low values with the test and development managers in the company and concluded that major commit and churn activities happened near to release dates, and hence, they are in acceptable ranges. Box plots representing lower/upper quantiles and medians for each metric can also be seen in Appendix E.

Responses to two major processes of the survey conducted in the second organization are also presented in Appendix C. As it is seen, dataset 2 contains only 15 responses which should ideally be 43 to make a complete inference from requirements, development and testing processes. Since the ratio of observed values are smaller than the ratio of missing values in this dataset, mean or hot deck imputation forms a synthetic dataset with majority of the instances equal to the mean of observed values. Therefore, we have used Gibbs sampling to infer parameters of the underlying Bayesian model that can be seen in Section A.6. However, different from the mean imputation used in Dataset 1, instead of filling missing responses, we have generated samples from the posterior distribution of the final node, "probSpecDefect" by estimating its mean (reqFinal) given unknown parameters (coefficients) and known values (15 responses to 7 questions). Then we have used generated samples of the final node in Hybrid networks that we built.

6.2. Empirical Analysis on Software Metrics

During model construction, we have focused on two important issues: (i) Defining prior/conditional distributions, and (ii) Defining causal relations. To investigate the first issue, we have applied statistical tests (Lilliefors significance test [102]) on all metrics including defects, which check whether samples come from a specific distribution (such as *normal*, *exponential*, *weibull*). Lilliefors significance test is the same as Kolmogorov-Smirnov goodness of fit test for checking the null hypothesis (H_0): Samples come from a distribution from the normal family. Lilliefors provides better solutions when sample size is less than 50 and it is also possible to check null hypothesis for other distributions, such as exponential [102]. Hence, we also used Lilliefors to check whether the metrics come from the specified distribution. Results of this test (H) is 0 if the null hypothesis could not be rejected with 95% significance. However, if null hypothesis is rejected (H=1), this indicates that samples do not come from the specified distribution with 95% significance.

To investigate the second issue (causal relations), we have used various correlation and independence tests as well as graphical dependence analysis (chi plots used in copula modelling [103]).

Correlations, similarities of distributions & significance of equal medians: First, we have used Spearman's correlation test to check whether metric pairs have a statistical relation between each other. We also have computed Kolmogorov-Smirnov significance tests to check whether there are metric pairs which come from identical continuous distributions and Wilcoxon rank-sum tests to check whether two metrics come from distributions with equal medians. These additional tests help us understand the statistical relations between metrics in terms of distributions and medians.

Brownian covariance independence test: We know that correlation tests does not necessarily indicate causality. Thus, as a second approach, we have used a new dependence measure on the metrics of development and testing phases. This new measure was introduced by Szekely and Rizzo [94] as a new approach to the problem of measuring dependence and testing the joint independence of two random vectors. It is easy to calculate based on pairwise Euclidean distances between samples such that A_{kl} is an NxN matrix where each index a_{kl} contains pairwise Euclidean distance between k^{th} and l^{th} samples of a variable. Distance covariance between two vectors, X and Y, is computed by using these pairwise distance matrices. Distance covariance is computed as in Equation 6.1. Brownian distance correlation ratio, which is zero if and only if two metrics are independent, i.e., when $dCov(X,X) * dCov(Y,Y)$ is zero.

Authors showed that the definitions of the new dependence coefficients have theoretical foundations based on characteristic functions and on the new concept of covariance with respect to Brownian motion [94]. Hence its independence test is proposed as Brownian distance correlation test. Distance covariance and correlation also provide a natural extension of Pearson product-moment covariance and correlation. If the correlation coefficient of a metric pair is greater than 0 with p-value less than 0.05, it indicates that this pair has any type of dependence. We have selected metric pairs whose correlation coefficients are greater than 0.65 with 95% significance.

$$dCor(X,Y) = dCov(X,Y)/\sqrt{dCov(X,X) * dCov(Y,Y)}$$
$$dCov(X,Y) = 1/N^2 \sum_{k,l=1}^{N} A_{kl} B_{kl} \qquad (6.1)$$

Graphical dependence analysis with Chi-plots: As a third approach to dependence analysis, we have used a graphical tool, namely *chi-plot*, that has been recently proposed in the literature for detecting dependence between two variables. Chi-plots are additional graphical tools in addition to K-plots [104] both are used to assess the dependence between variables during copula modelling. We have plotted pairwise relations between development and testing metrics and post-release defects using chi-plots. Before explaining chi-plots, we will briefly explain the idea behind copula modelling.

A traditional view of dependence between two variables, X and Y, is to look at scatter plot of the pairs (X_1, Y_1),...,(X_n, Y_n). Even though scatter plots incorporate information about the dependence between X and Y, they also incorporate information about their marginal distributions [103]. Copula modelling provides a transformation of data with marginal distributions ($H(x,y) = C\{F(X), G(Y)\}$) where $F(X)$ and $G(Y)$ are marginal distributions and C is copula) so that the transformed data can be investigated for the possibility of dependence independently from the choice of marginal distributions. The important thing in this modelling is the choice of F,G, and C from one of the suitable parametric families. Once marginal distributions, F and G, are selected, the selected copula would uniquely characterize the joint dependence [103]. In the simplest case, if X and Y are independent, $C(u_1, u_2) = u_1 * u_2$ where u_1 and u_2 are transformations of X and Y to standard uniform marginal distributions [105]. U_1 and U_2 are based on *ranks* of X and Y divided by $1/n + 1$ to scale the values to $[0,1]^2$.

Copula models with different families (e.g. Farlie-Gumbel-Morgenstern family of copulas, Achimedean copula [105]) can be applied to model the dependence between two variables. Prior to modelling, it is necessary to check the presence of dependence through visual plots. Chi-plots are one of these tools proposed by Fisher and Switzer [95] and again in [106]. Authors argue that a graphical view of bivariate dependence is richer than various non-parametric statistical tests. Scatter plots are primary data analysis tool, however in case of independence, it is very difficult to judge the randomness through human eye. Hence, chi-plots is designed to address this problem, by providing characteristic patterns depending on whether variables (i) are independent, (ii)have some degree of monotone relationship, or (iii) have more complex dependence [106].

These plots exclusively depend on the ranks of variables by introducing new axes χ and λ based on the ranks of X and Y. Specifically,

$$\begin{aligned} H_i &= 1/(n-1)\#\{j \neq i : X_j \leq X_i, Y_j \leq Y_i\} \\ F_i &= 1/(n-1)\#j \neq i : X_j \leq X_i \\ G_i &= 1/(n-1)\#j \neq i : Y_j \leq Y_i \end{aligned} \tag{6.2}$$

In 6.2, each data pair (X_i, Y_i) is transformed to (F_i, G_i). Fisher and Switzer [106] propose to plot the pairs (λ_i, χ_i), where

$$\begin{aligned} \chi_i &= (H_i - F_i G_i)/(\sqrt{F_i(1-F_i)G_i(1-G_i)}) \\ \lambda_i &= 4sign((F_i - 1/2)(G_i - 1/2))max((F_i - 1/2)^2(G_i - 1/2)^2) \end{aligned} \tag{6.3}$$

To avoid outliers, authors recommended that what should be plotted are only the pairs for which $\lambda_i \leq 4(1/(n-1) - 1/2)^2$ [106]. Both λ_i and χ_i take values in the range of [-1,1]. In Figure 6.1, additional control limits are drawn at $\pm c_p/\sqrt{n}$ where c_p is selected as 1.78 for defining that 95% of values lie between control limits. Figure illustrates the chi-plot of the relation between *Test Cases* and *Post-release Defects* for Dataset 1. Association (dependence) between two metrics should be revealed by departures from zero-centered vertical scatter plot on (λ, χ). In Figure 6.1, we can see that there is a positive monotone association between test cases and post-release defects, which is significantly validated with the independence test based on Spearman's correlation coefficient.

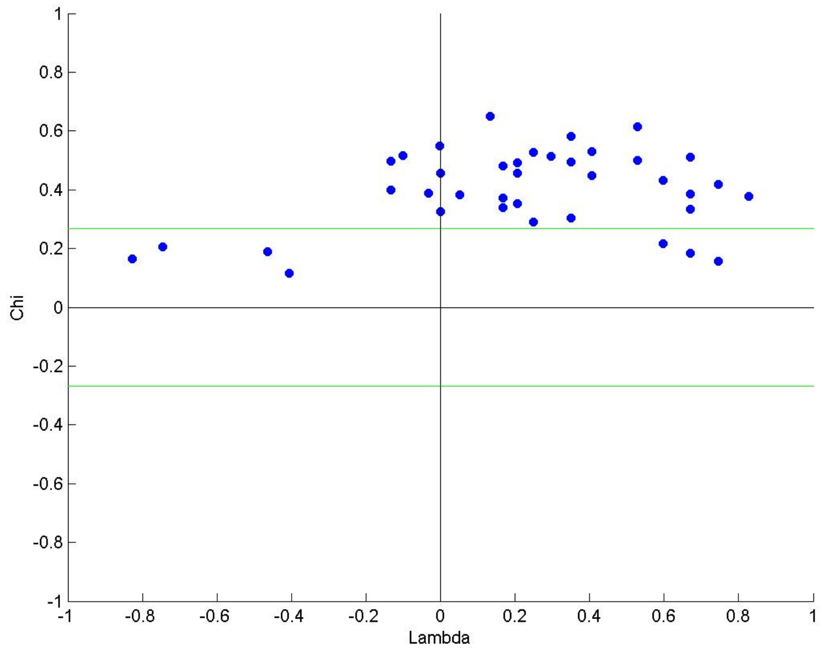
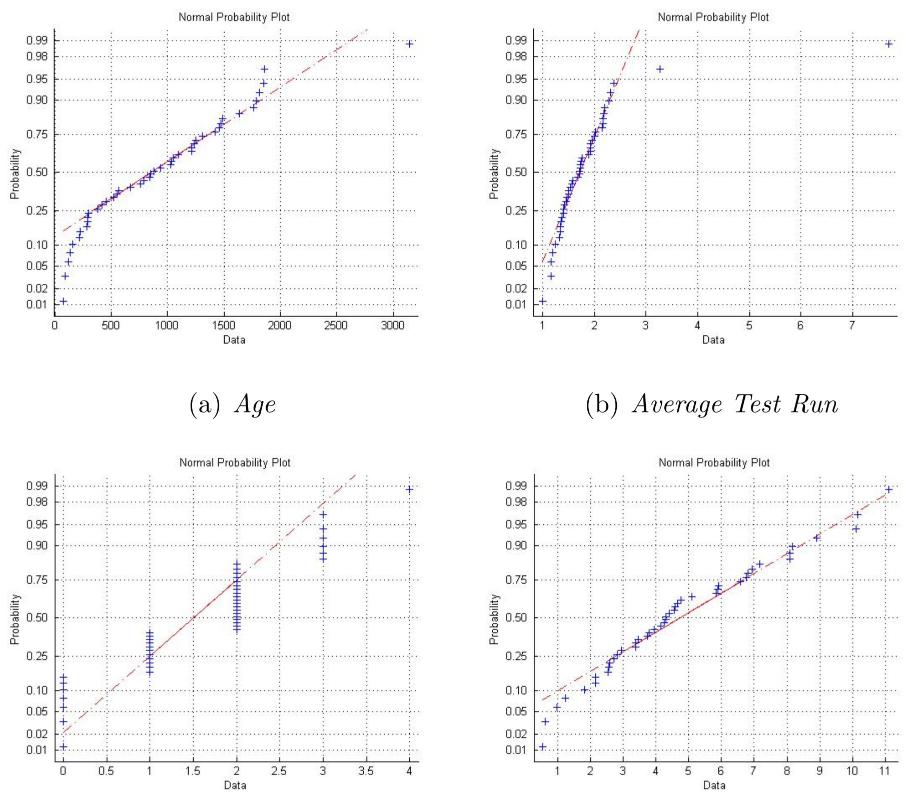


Figure 6.1. Chi-plot of two variables.

6.2.1. Analysis for Dataset 1

We have reported the results (H values) of Lilliefors test for normal and exponential distributions in Table 6.3. Based on these results, null hypothesis checking the normality assumption is rejected for all metrics in Dataset 1 except *Test Cases, Test Case Quality, Age* and *Post-release Defects*. For *Age* metric, the null hypothesis checking the goodness of fit for exponential distribution cannot be rejected, too. Thus, we can decide that values of *Test Cases, Test Case Quality, Post-release Defects* metrics may come from normal distributions. Furthermore, values of *Test Defects, Churn, Time between Edits* metrics may come from exponential distributions.



(a) *Age* (b) *Average Test Run*

(c) *People* (d) *Complexity*

Figure 6.2. Normality plots for the metrics in Dataset 1.

For the metrics, namely *Complexity, People, Average Test Run*, significance tests show that they do not come from both normal and exponential distributions, whereas for *Age* metric, either normal or exponential distributions may be good fit. To dissolve ambiguities for these four metrics, we have used graphical tests (normal probability plots [107]). Figures 6.2 show normal probability plots which are graphical techniques for assessing whether a data is normally distributed [107]. Although we could not decide on four metrics based on statistical tests, we can see that data points (+) for *Average Test Run* and *Complexity* metrics form a nearly linear pattern except few outliers and they overlap with the red line. On the other hand, *Age* metric seems to be very skewed for a normal distribution. Thus, we have applied different distributions to this metric during model construction in order to find the best fit in terms of DIC. Finally, *People* is also very different from a normal distribution due to discrete values. It may be more appropriate to assign a discrete distribution, such as Poisson, to this metric during our analysis.

Table 6.3. Lilliefors goodness of fit test results for Dataset 1.

	Test for normal distribution (H)	**Test for exponential distribution (H)**
Age	0	0
People	1	1
Churn	1	0
Edit Frequency	1	0
Complexity	1	1
Test Cases	0	1
Test Defects	1	0
Test Case Quality	0	1
Average Test Run	1	1
Post-release Defects	0	1

Results of the correlation test are presented in Table 6.4. Short abbreviations are mapped to metric explanations in Tables 6.1 and 6.2. Bold cells indicate significance correlations among metric pairs. From Table 6.4, except between TC and DEFECT (with correlation coefficient being 0.6), none of the correlations are strong enough to represent a relation. Furthermore, results of Kolmogorov-Smirnov and Wilcoxon rank-sum significance tests also show that TC and DEFECT may come from similar distributions with equal medians. We have also seen that CHURN-COMPL, TD-EDITFR, ATR-PEOPLE pairs may have a relationship in terms of distributions and parameters (medians of the distributions). We have used these results while forming causal links between metrics of three BNs that we propose.

We have run Brownian distance correlation test using R energy package and found correlation coefficients for metric pairs which may be dependent with 95% confidence. Results show that TC-TD, TC-AGE, TC-PEOPLE, TC-DEFECT, TD-TCQ,TD-CHURN, ATR-TCQ and ATR-DEFECT pairs have significant ($\alpha = 0.05$) dependency relations between each other. Therefore, we have formed an additional BN whose causal relations between metrics are formed by considering distance correlations.

Table 6.4. Spearman's correlation coefficients for Dataset 1.

	TD	**ATR**	**TCQ**	**AGE**	**PEOPLE**	**CHURN**	**EDITFR**	**COMPL**	**DEFECT**
TC	**0.36**	-0.13	-0.16	**0.34**	**-0.33**	-0.11	0.08	0.01	**0.60**
TD		0.11	-0.25	0.16	-0.12	-0.22	0.18	-0.01	**0.35**
ATR			-0.19	-0.22	0.06	0.07	-0.09	0.26	**-0.41**
TCQ				-0.20	-0.09	0.10	-0.07	-0.24	-0.17
AGE					-0.04	-0.10	0.07	0.00	0.23
PEOPLE						0.02	-0.15	0.18	-0.16
CHURN							-0.07	0.01	-0.05
EDITFR								-0.07	0.29
COMPL									-0.02

Finally, we have observed chi-plots between all variables by drawing them according to the specifications in [95]. Pairs of variables whose points fall outside of the "control limits" of the chi-plot are considered as if there exists a presence of an association. Figures of chi-plots for Dataset 1 can be seen in D.1 and D.2. Pairs with a dependence are listed as follows: TC-DEFECT, TC-TD, TC-AGE, TD-DEFECT, TCQ-AGE, ATR-TCQ, ATR-DEFECT, EDITFR-DEFECT, CHURN-COMPL. It can be observed that some of these pairs are already correlated in Table 6.4, while some of them are also confirmed with Brownian tests. But there are still new relations found via chi-plots. Therefore, these new pairs are used to form causal relations of another BN in order to observe whether there is a significant difference between a network built with statistical tests and a network built based on graphical assessment of dependence.

6.2.2. Analysis for Dataset 2

Same significance and correlation tests were applied on our second dataset collected from an ERP product. First, we have reported the results (H values) of Lilliefors test for normal and exponential distributions in Table 6.5. For three metrics namely, *Time between edits, Complexity, Test defects*, null hypothesis for the normal assumption could not be rejected. So we used normal distributions for these metrics when they are included into the model. For five metrics namely, *added LOC, deleted LOC, Test cases, Test cases with Errors, Test suites*, null hypothesis for the exponential distribution could not be rejected. However, using exponential distribution during inference with Gibbs sampling caused our model not to converge. Therefore, for these metrics,

Table 6.5. Lilliefors goodness of fit test results for Dataset 2.

	Test for normal distribution (H)	**Test for exponential distribution (H)**
Commits	1	1
People	1	1
Added LOC	1	0
Deleted LOC	1	0
Time between Edits	0	1
Complexity	0	1
Test Cases	1	0
Test Cases with Errors	1	0
Test Cases with Failures	1	1
Test Cases with Status Change	1	1
Test Suites	1	0
Test Defects	0	1
Post-release Defects	1	1

we also computed negative log-likelihood of the parameters for different type of distributions, such as "log-normal", "gamma", and selected a more suitable distribution.

Similar to results obtained in Dataset 1, some of the metrics (*Commits, People, Test cases with Failures, Test cases with Status Change, Post-release defects*) rejected both distributions in Dataset 2. *People* takes discrete values between [5,12] so that it should have a different distributional form. Furthermore, *Test cases with Status Change* contains large amount of 0's (43%) and hence it is very hard to fit a distribution to this metric (see its histogram for 43 instances in Figure 6.3). We decided not to remove this metric from the dataset but to use it as a constant in our models.

In addition, normal probability plots for *Commits*, *Test cases with Failures* and *Post-release defects*, are presented in Figure 6.4. These plots show that *Post-release Defects* and *Commits* almost follow a normal distribution with a few outliers on the right corner. However, *Test cases with failures* has a left skewed curve (points bend down and right of the normal line) and when outliers are removed, it looks like a long tailed distribution with more variance than a normal distribution. Therefore, we

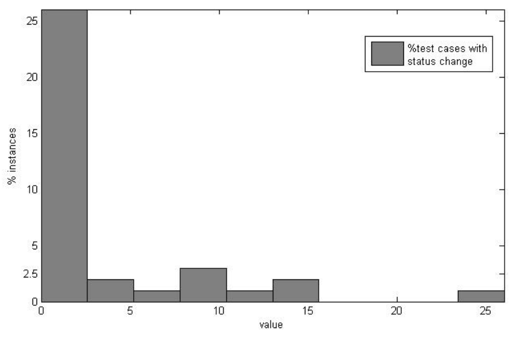
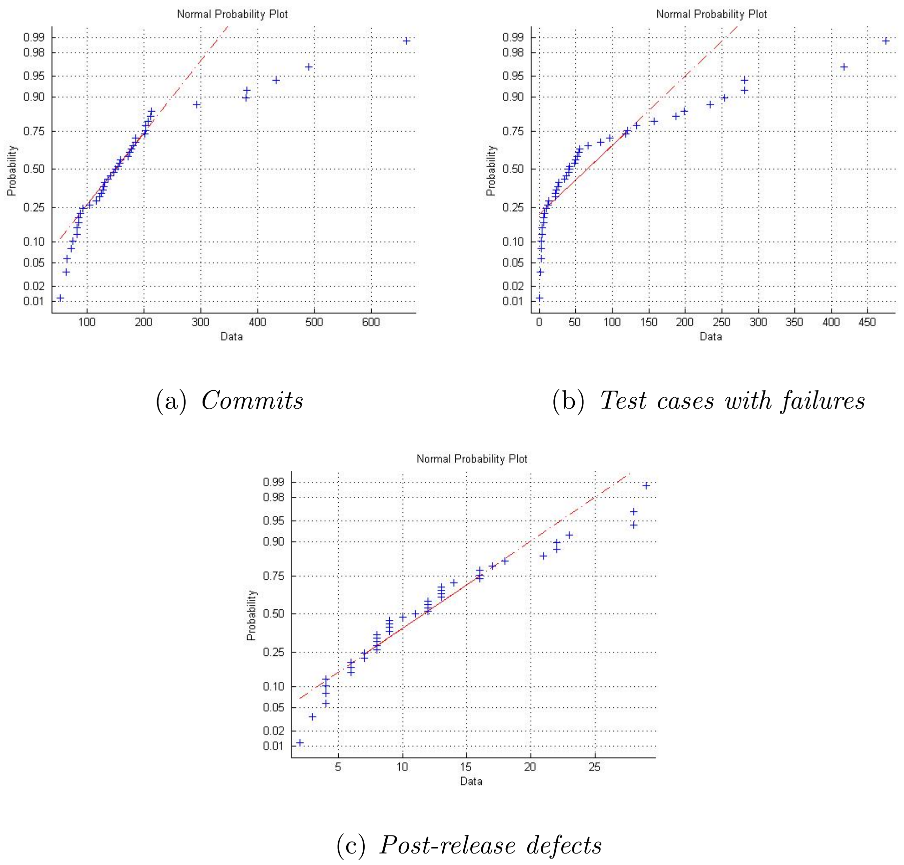


Figure 6.3. Histogram plot for Test Cases with Status Change.

assigned t-distribution to *Test cases with failures*, which is more useful for modelling distributions with long tails.



(a) *Commits*

(b) *Test cases with failures*

(c) *Post-release defects*

Figure 6.4. Normality plots for metrics in Dataset 2.

Correlations between metrics are also presented in Table 6.6. Bold cells indicate significant correlations with 95% confidence. Strong correlations with coefficients greater than 75% are COMMITS-EDITFR, TC-TCE, TC-TCF, TC-TS, TCE-TCF, TCE-TS, TCF-TS, all of which are due to measurement relations between metric pairs. We considered these relations while forming our statistical tests based model. Addi-

Table 6.6. Spearman's correlation coefficients for Dataset 2.

	PEOPLE	ADDLOC	DELLOC	EDITFR	COMPL	TC	TCE	TCF	TCST	TS	TD	DEFECT
COMMITS	0.29	0.17	0.16	**-0.97**	**-0.49**	-0.27	-0.30	-0.16	-0.23	-0.19	0.21	0.23
PEOPLE		**0.33**	**0.31**	-0.27	0.03	0.03	0.06	0.08	0.10	0.08	0.18	0.18
ADDLOC			**0.60**	-0.17	-0.15	-0.12	-0.16	-0.27	0.17	-0.08	0.24	-0.11
DELLOC				-0.15	-0.03	-0.30	-0.23	**-0.34**	0.07	-0.24	0.29	-0.07
EDITFR					**0.49**	0.29	**0.32**	0.14	0.16	0.22	-0.20	-0.21
COMPL						0.18	0.17	0.22	0.28	0.15	0.04	-0.15
TC							**0.91**	**0.80**	0.08	**0.93**	0.23	0.02
TCE								**0.73**	0.06	**0.93**	0.24	0.07
TCF									0.13	**0.76**	0.09	0.20
TCST										0.04	0.04	-0.14
TS											0.24	0.08
TD												-0.10

tionally, Kolmogorov-Smirnov and Wilcoxon rank-sum tests identified relations between COMMITS-TCE, ADDLOC-TCF, ADDLOC-TD and TCF-TS such that they may come from identical distributions with equal medians.

For independence tests, we have run Brownian distance correlation test using R energy package and found correlation coefficients for metric pairs which may be dependent with 95% confidence. Results show that COMMITS-EDITFR, TC-TCE, TC-TCF, TC-TS, TCE-TCF, TCE-TS and TCF-TS pairs have a dependency relation with correlation coefficients greater than 65%. There are also correlation whose coefficients are below 65% for pairs: PEOPLE-EDITFR, EDITFR-TC, EDITFR-TCE, EDITFR-TCF, TC-DEFECT and TCF-DEFECT. Results of this test show a very similar pattern with the results of Spearman's correlation test.

Finally, chi-plots between pairs of metrics are investigated and pairs whose joint distribution show a relation between λ and χ, are represented in Figures D.3, D.4 and D.5. Pairs with a possibility of dependence are listed as follows: COMMITS-EDITFR, COMMITS-COMPL, COMMITS-TCST, PEOPLE-DELLOC, TC-TS, PEOPLE-TCST, PEOPLE-EDITFR, ADDLOC-DELLOC, ADDLOC-TCF, DELLOC-TCST, DELLOC-TD, EDITFR-COMPL, TC-TCE, TC-TCF, TCE-TS, TCE-TCF, TCF-DEFECT, TCF-TS. As in Dataset 1, we can see similar relations obtained from both Spearman's correlation and Brownian independence tests, however in chi-plots, we can see additional relations with the help of transformation of data into ranks. For instance, distribution

Table 6.7. Proposed models for both datasets.

Model	**Description**
Model #1 (GLM)	A generalized linear model in which the output variable (post-release defects) is a parametric functional form of input variables (metrics) which are considered as independent from each other.
Model #2 (Idiom)	A model built with basic building blocks (idioms) defined by Neil *et al.* [79] based on expert judgement.
Model #3 (Stats)	A model whose causal relationship among metrics are defined based on correlations and significance tests.
Model #4 (DistCov)	A model whose causal relationships are defined based on Brownian distance covariance independence test.
Model #5 (ChiPlot)	A model whose causal relationships are defined based on graphical dependence analysis with chi-plots.
Model #6 (Hybrid)	Model whose development & testing subnets are in the form of either *Stats*, *DistCov* or *ChiPlot*, and requirements subnet is taken from [45].

of *% test cases with a status change* metric could not be estimated due to a lot of zero values. Converting real values to ranks helps to see the relation between this metric and others. We have also considered these new pair relations and built another BN during model construction.

6.3. Proposed Models

Based on our empirical analysis, we have built 6 different BNs for both datasets: The first five model is built on quantitative data, while the last one is built as a Hybrid model with both qualitative and quantitative data. Hybrid model has three variations in terms of the model used for building development and testing subnet: Stats, DistCov or ChiPlot. Short explanations for each model can be seen in Table 6.7.

We have implemented all models in WinBugs [108] which is a powerful framework for inference using Gibbs Sampling. WinBugs codes for all models can be found in Appendix A and B. We present 12 different models for both Dataset 1 and 2 in the next sections.

6.3.1. Models for Dataset 1

This subsection explains 6 different models built on Dataset 1. WinBugs codes for all models of Dataset 1 can be found in Appendix A. In graphical models, white nodes represent test metrics, while the rest represent development metrics. *Post-release defects* node is represented as *DEFECT*. Metrics which does not have a significant relationship with others are represented with dashed circles. Hidden nodes are also represented with blue nodes to distinguish them from metrics.

6.3.1.1. Model #1 (GLM). In a GLM, the output variable is defined as a parametric functional form of input variables [109],

$$E(Y) = g^{-1}(X\beta) \tag{6.4}$$

where g is the link function determined according to the distribution of Y, and $X\beta$ is the linear predictor with X being input variables and β being coefficients. The link function can change depending on the distributions as shown in Equation 6.5. We use identical link function, *(E(Y))*, since we assume that post-release defects are normally distributed (see Table 6.3).

$$E(Y) = X\beta \qquad \text{if Y} \sim \text{N}(X\beta, \sigma) \tag{6.5}$$

$$ln(E(Y)) = X\beta \qquad \text{if Y} \sim \text{P}(e^{X\beta}) \tag{6.6}$$

$$logit(E(Y)) = X\beta \qquad \text{if Y} \sim \text{B}(e^{X\beta}/(1 + e^{X\beta})) \tag{6.7}$$

In this model, we used all software metrics as independent variables regardless of the phase they represent. Graphical representation of Model # 1 can be seen in Figure 6.5.

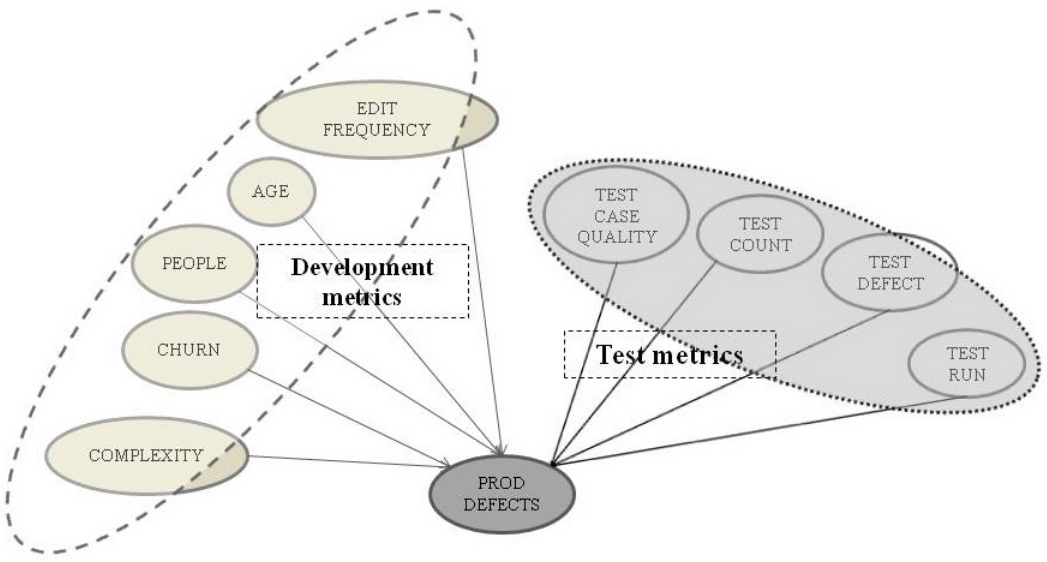


Figure 6.5. Graphical representation of Model #1 - Dataset 1.

6.3.1.2. Model #2 (Idiom). Neil *et al.* inspired from bottom-up approach in system and software engineering and fragments of Object-Oriented Bayesian Networks to form a complete Bayesian network [79]. They proposed basic building blocks called *idioms*, which were small and reusable components of a network in such a way that complexity can be easily managed after combining them together. Authors worked on the process model of a development life cycle by forming idioms.

We have used some of these idioms defined by [79] while constructing our BN: *Definitional idiom, cause-consequence idiom* and *measurement idiom.* Furthermore, we have validated these relations with senior managers in the company to avoid potential biases during model construction. Final representation of this model using these idioms (and agreed by senior management) can be seen in Figure 6.6.

Definitional idiom is used in BNs when there is a definitional relationship between variables, i.e. a synthetic node is defined by other nodes. We have used this idiom to define the relationship between a synthetic node H and three development process metrics: *Churn, Complexity* and *Edit Frequency.* Furthermore, we have defined *Post-*

release Defects by three test process metrics: *Test Case Quality, Test Defects, Average Test Run.*

Cause-consequence idiom is used to model a process in terms of its relationship between its causes (input to process) and consequences (output of process). We have used this idiom to define the relationship between *Age-Complexity*, *People-Churn*, and *Test Cases-Test Defects*. All relationships were previously investigated by other researchers in empirical studies (see 4.1.1). Based on these works, we have selected the inputs to the prediction process as *Age, People, Test Cases* and the outputs of the process as *Complexity* (due to Age), *Churn* (due to People) and *Test Defects* (due to Test Cases).

Finally, measurement idiom is used in BNs to measure one event based on the other. We have used this idiom to define the relationship between *Test Case Quality* and *Test Cases*, since the value of the first variable in fact depends on the second variable due to the calculation made for finding *Test Case Quality*.

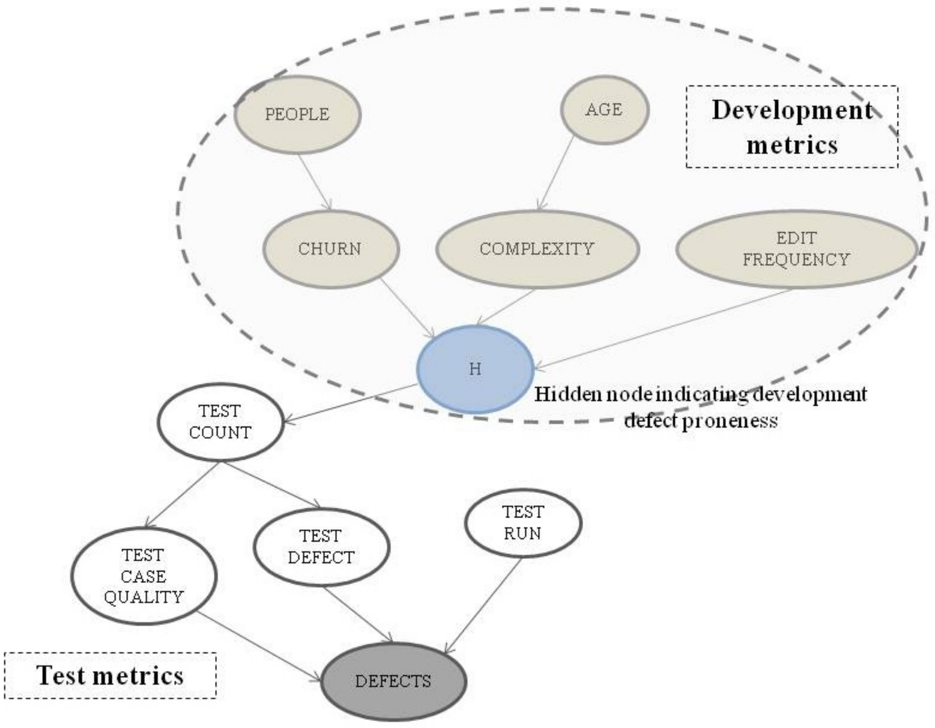


Figure 6.6. Graphical representation of Model #2 - Dataset 1.

6.3.1.3. Model #3 (Stats). In this model, we have used the correlations among metric pairs as well as significance tests measuring similarities in terms of distributions and their medians to define causal links between metrics and post-release defects. We also modified the model by adding two unused metrics into the model (*Test case quality, Age*), defining new causal links based on quantile plots (*People-Churn*, *Complexity-Test Defects, Test case quality-Test Defects*, *Age-Test Cases, Age-Average test run*) as well as removing one metric (*Edit frequency*). The output node is defined as a normal distribution whose mean is defined as a linear function of *Test Cases, Test Defects, Churn* and *Average test run.* This linear function incorporates four metrics as well as their pairwise interaction effect parameters and a higher-order interaction effect (quad). A graphical representation can be seen in Figure 6.7.

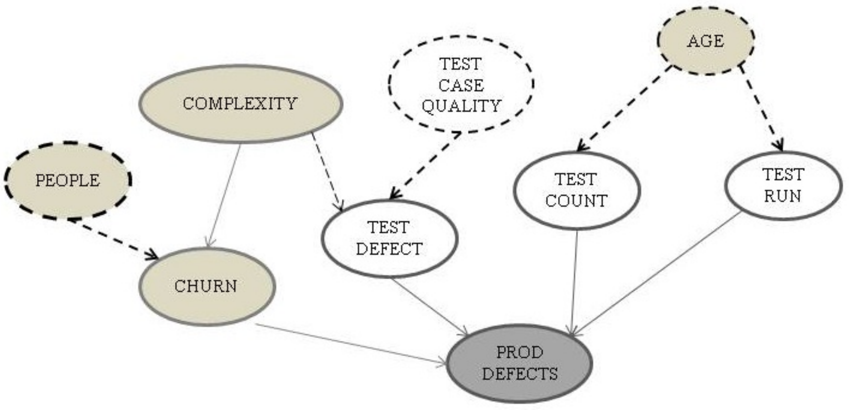


Figure 6.7. Graphical representations of Model #3 - Dataset 1.

6.3.1.4. Model #4 (DistCov). We have taken into account the results of Brownian distance covariance independence test while forming causal links among metrics, rather than correlation results used in the previous model. Results of the independence test highlight 8 significantly "correlated" metric pairs. Graphical representation of Model # 4 can be seen in Figure 6.8. As it is seen from the figure, *Edit frequency* could not be related to any metric, and hence, it is not included into the model. *Complexity*, on the other hand, is included although it is also uncorrelated with other metrics, since it helps to improve the performance of this model.

6.3.1.5. Model #5 (ChiPlot). This model is built based on manual interpretation of chi-plots to assess the dependence among metrics. In Section 6.2.1, we have listed 9

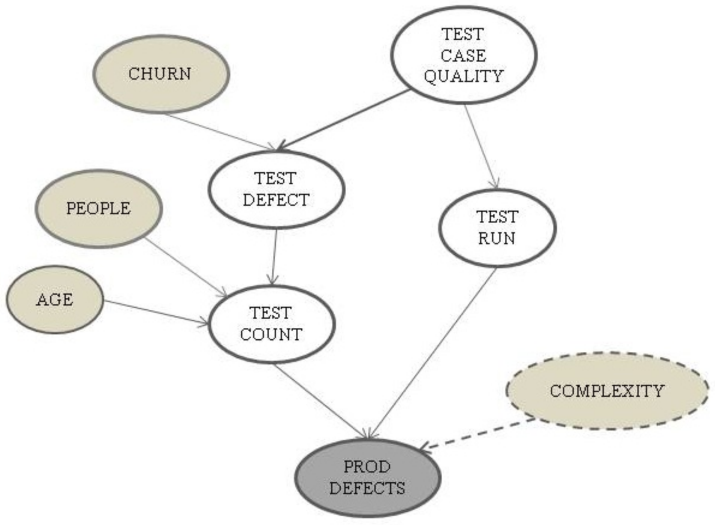


Figure 6.8. Graphical representation of Model #4 - Dataset 1.

metric pairs with dependence relations according to specifications of chi-plots. We have considered these relations while building Model #5 (Figure 6.9).

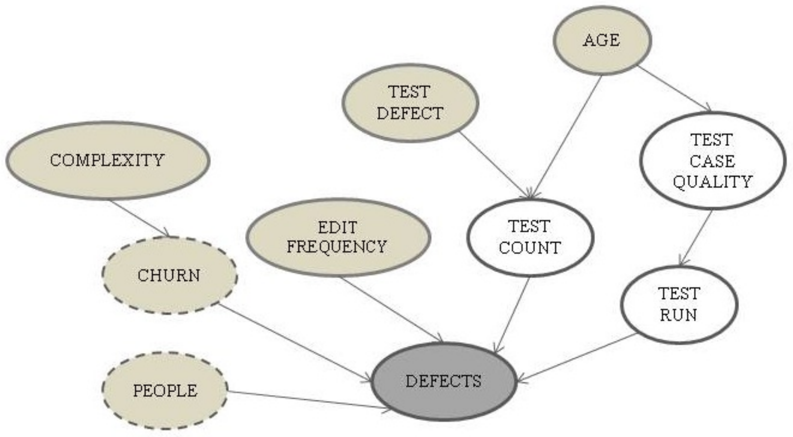


Figure 6.9. Graphical representation of Model #5 - Dataset 1.

6.3.1.6. Model #6 (Hybrid). The first five models were built based on quantitative data collected from development and testing activities. In order to incorporate requirements specification subnet into existing models, we investigated the model proposed by Fenton *et al.* [45] that was built based on expert judgement (experienced project managers). Each subnet consists of qualitative factors and their causal relationships, which were elicited from experts based on their discussions on the impact of these factors on the final outcome of software product. For specification and documentation subnet, we list qualitative factors that are collected through survey in Section 4.1.2.

We have used qualitative factors while building our hybrid network due to the fact that there is not available data for specification and documentation process in software organizations. Therefore, we have also adapted the model proposed in [45] for our final model. Figure 6.10 shows a graphical representation of specification and documentation subnet, with blue nodes as factors filled with survey results and dashed nodes as hidden nodes. Distributions and equations of the hidden nodes are modified according to our model. Final node of this subnet is assumed to follow a Gaussian distribution, whose mean is a weighted linear equation of two hidden node (*Spec and doc effectiveness* and *Spec and doc process quality*). WinBugs code for this subnet can also be found in Appendix A.

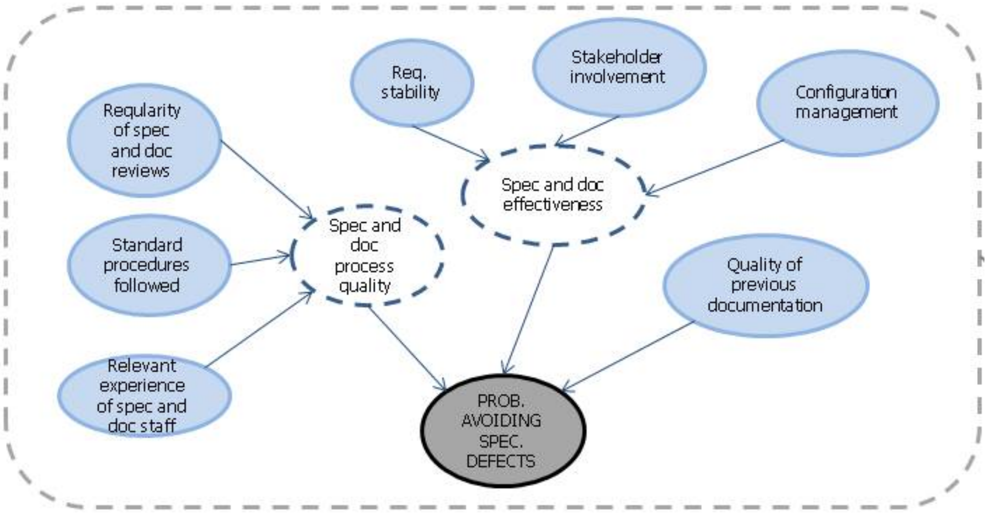

Req. stability
Stakeholder involvement
Configuration management
Regularity of spec and doc reviews
Spec and doc effectiveness
Standard procedures followed
Spec and doc process quality
Quality of previous documentation
Relevant experience of spec and doc staff
PROB. AVOIDING SPEC. DEFECTS

Figure 6.10. Graphical representation of Specification and Documentation Subnet.

While building the Hybrid network, we use Stats, DistCov and ChiPlot models shown in Sections A.3, A.4, and A.5 and added a new node (indicating spec. and doc. subnet's defect proneness) directly connected to the final node (post-release defects) on each of three models. During inference, Hybrid model uses a combined version of the Winbugs code of either of three models (see Sections A.3, A.4, A.5) and Figure A.6 for representing requirements subnet and revise the linear equation of y as follows:

$$
\begin{aligned}
y[i] \quad & < -\beta_1 + \beta_2 * TD[i] + \beta_3 * ATR[i] + \\
& \beta_4 * TC[i] + \beta_5 * CHURN[i] + \\
& (\beta_6 * TD[i] * ATR[i] * TC[i] * CHURN[i]) + \\
& (\beta_7 * TD[i] * ATR[i]) + \\
& (\beta_8 * TD[i] * TC[i]) + \\
& (\beta_9 * CHURN[i] * TD[i]) + (\beta_{10} * probSpecDefect[i])
\end{aligned}
\tag{6.8}
$$

The final node estimating post-release defects uses this y value as the mean of a normal distribution and Gibbs sampler generates estimated values of y during several iterations of Winbugs runs.

6.3.2. Proposed Models for Dataset 2

Similar to Dataset 1, we built 6 different Bayesian networks for Dataset 2. In this section, we presented their graphical representations and explained how we built the causal relations as well as formulations of these causal relations. WinBugs codes for Dataset 2 can be seen in Appendix B. Similar to graphical models for Dataset 1, testing metrics are represented with white nodes, while development metrics are represented with gray nodes. Hidden nodes are also coloured with blue to distinguish them from metrics. We have also put dashed rectangles to some of the models to highlight the metrics which do not have a significant association with others.

6.3.2.1. Model #1 (GLM). In the generalized linear model (GLM), each metric is assumed to be independent from each other and their weighted linear equation corresponds to the mean of the final node. Graphical representation can be seen in Figure 6.11.

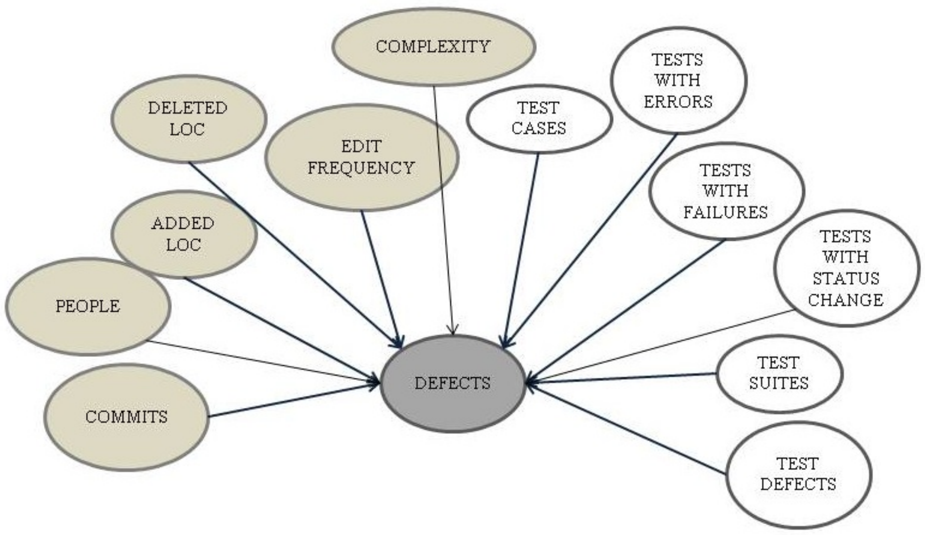


Figure 6.11. Graphical representation of Model # 1 - Dataset 2.

6.3.2.2. Model #2 (Idiom). Cause-consequence, measurement and definition idioms are used to build the final model. For development subnet, cause-consequence idiom is defined between *People* and *Added/Deleted LOC*: As more *People* (i.e., Committers in Winbugs code) work on the source code, the more lines of code is changed in terms of *Added LOC and Deleted LOC*.

Measurement idiom is defined between *Edit frequency* and *Commits*: *Average time between edits (Edit frequency)* are measured by taking the days between each commit on the source code and divided by total number of *Commits*.

Definition idiom is also used to define a hidden node representing defect density of development subnet in terms of *Complexity, Time between Edits* and *Changed LOC* (another hidden node defined with a cause-consequence idiom).

For testing subnet, cause-consequence idiom is defined between *Test cases with Failures, Test cases with Errors, Test suites* and *Test defects*. Since there are strong correlations between these three metrics and *Test cases*, we did not include the latter into Idiom model. We also defined a hidden node representing defect density of testing subnet in terms of *Test defects* and *Test cases with status change*.

Final node representing *Post-release defects* is also a weighted linear form of two hidden nodes from two subnets. Graphical representation can be seen in Figure 6.12.

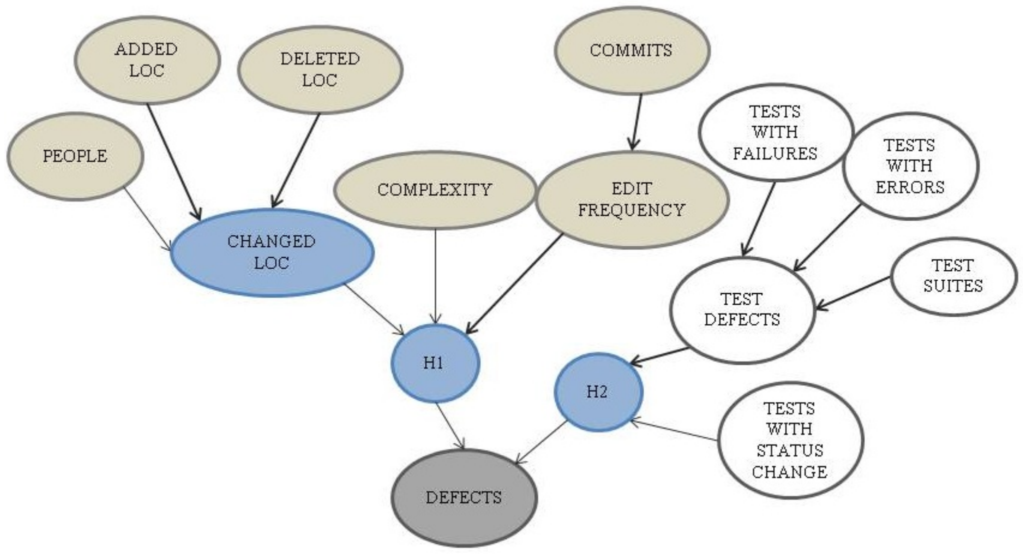


Figure 6.12. Graphical representation of Model # 2 - Dataset 2.

6.3.2.3. Model #3 (Stats). This model is formed based on the results of statistical tests and correlation results. In Dataset 2, correlations are very weak between majority of metrics and only a few relations are defined after statistical tests. Thus, the third model includes 6 metrics with causal relationships among each other and 3 additional metrics, namely *People, Complexity, Tests with status change*, as independent variables. Final node is again defined with a Gaussian distribution whose mean is expected as a weighted linear equation between 3 independent metrics and *Test suites. Test defects* and *Time between edits (Edit frequency)* are taken out from the Stats model, whose graphical representation is in Figure 6.13.

6.3.2.4. Model #4 (DistCov). This model is formed based on the results of Brownian distance correlation tests. 7 metrics are used in relation with each other, while the rest 5 of them are also included into the model as independent variables affecting the final node. We also excluded these 5 metrics during model construction and compared the performance with or without these metrics. Results show that including them as independent variables increases the performance. The final graphical representation can be seen in Figure 6.14.

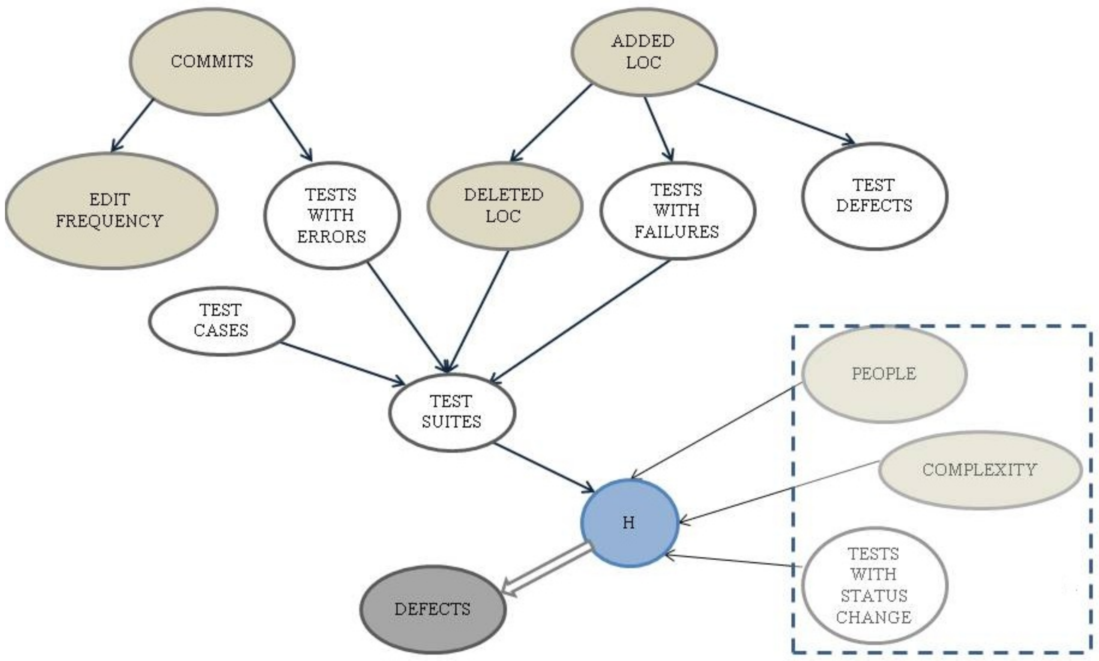


Figure 6.13. Graphical representation of Model # 3 - Dataset 2.

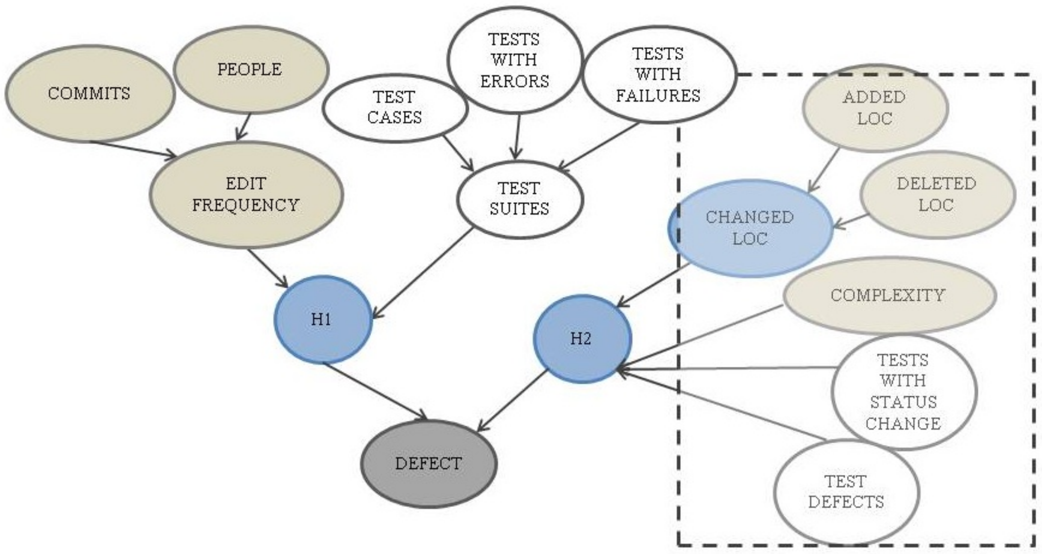


Figure 6.14. Graphical representation of Model # 4 - Dataset 2.

<u>6.3.2.5. Model #5 (ChiPlot).</u> As an alternative to statistical tests investigating dependence between variables, this model is formed based on graphical assessment of chi-plots. In total of 18 metric pairs are listed in Section 6.2.2. However, we have used the most significant ones, between *Test cases*, *Test cases with failures* and *Test cases with errors*, and between *Commits, Edit frequency* and *Complexity.* Furthermore, we have included metrics, namely *Test defects*, and *Test cases with status change*, which has no specific relationship with other metrics, since adding them improved the performance. We have considered adding hidden node, *H1* in Figure 6.15, on top of three mostly correlated metrics, based on the idea that there might be a hidden nodes affecting all metrics and leading to such significant associations.

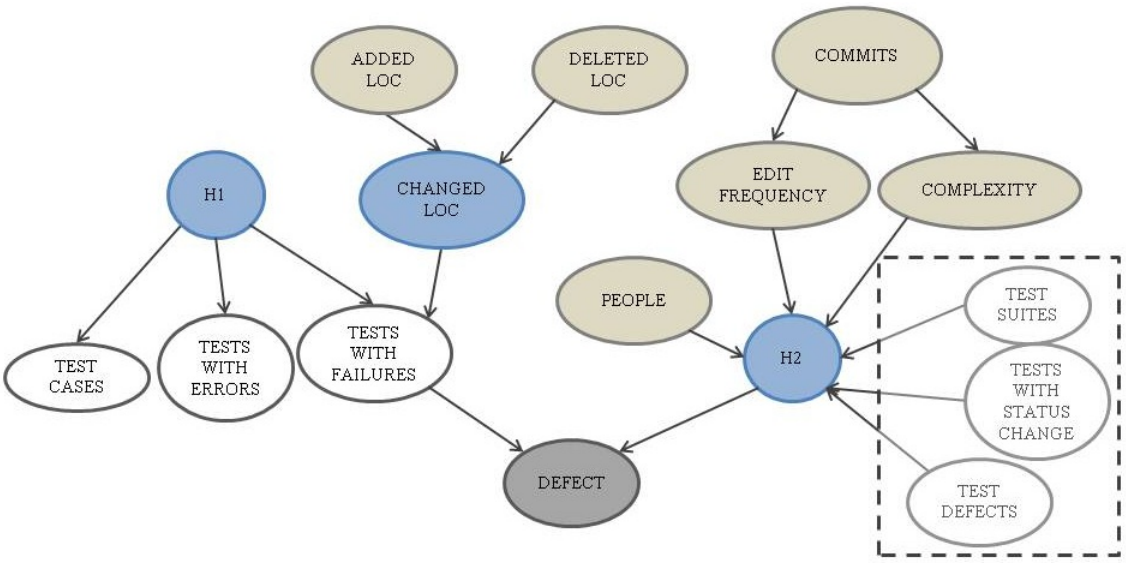


Figure 6.15. Graphical representation of Model # 5 - Dataset 2.

<u>6.3.2.6. Model #6 (Hybrid).</u> Similar to the approach used on Dataset 1, we built specification and documentation subnet according to a previous study by Fenton *et al.* [45] and we combined this subnet with Stats, DistCov and ChiPlot models respectively to build a complete model. During inference of parameters, Hybrid model uses a combined version of Winbugs codes from either of Figures B.3,B.4, or B.5 and A.6 and revise the linear equation of y as follows:

$$\begin{aligned} y[i] \quad & < -\beta_1 + \beta_2 * TS[i] + \beta_3 * PEOPLE[i] + \\ & \beta_4 * COMPL[i] + \beta_5 * TCST[i] + \beta_6 * probSpecDefect[i] \end{aligned} \tag{6.9}$$

6.4. Results

Tables 6.8 and 6.9 summarizes the performance of 8 different BNs built on Datasets 1 and 2, in terms of MMRE, MdMRE, Pred(25), Pred(30). The first 5 models are already described in Table 6.7, while the last three of them are Hybrid models (see the last row in Table 6.7) whose development and testing subnet vary between Stats, DistCov and Chiplot models. We have also reported DIC values for each model to find the best fit to data. Bold cells highlight models with the best performance significantly (with 95% according to Mann-Whitney U-test) in terms of performance measures.

Results of Dataset 1 show that Hybrid models are very successful at predicting post-release defects as they incorporate two different type of data (qualitative and quantitative) into the model. In Table 6.8, Hybrid models manage to reduce prediction error in terms of MdMRE down to 11%, with 84% of predictions having MRE lower than 25%. DIC is also reported as 216.5 in the first hybrid model and 278.1 in the third, indicating that Hybrid models fit data better than others.

(Idiom) model produces the worst prediction performance although expert judgement was used while building these idioms and links. This shows that statistical relations between metrics may reveal other types of relations that idioms could not cover. When we observe Hybrid models built over three major statistical models, it is seen that (ChiPlot) incorporated with requirements subnet manages to predict 84% of releases in Dataset 1 with less than 25% MRE. Median of MRE values also decreases down to 0.11; showing that the success of Hybrid BNs for capturing relationships between metrics and residual (post-release) defects.

Hybrid models built with (Stats) and (DistCov) are worse than (ChiPlot), which validates the motivation of Fisher and Switzer [95] that a graphical plot may sometimes worth 100 different tests. The reason for this difference between models with tests and the model with graphical assessment can be as follows: Considering (Stats) model, the applied techniques are Spearman's correlation test and Kolmogorov-Smirnov tests for testing independence of medians. Even though these are rank based tests without any limitation of distribution assumption, certain relations may not be observed due to dependence on the marginal distributions of metrics. As described in copula modeling [103], additional transformations may be necessary to discover relations between two metrics, independently from their marginal distributions. Since chi-plots are successful in assessment of dependence and easier to interpret with control limits, we have seen that in Dataset 1, we manage to identify additional dependencies that were not highlighted in (Stats) and (DistCov) models.

Table 6.9 also presents prediction performance of five models on Dataset 2. The success of Hybrid models could not be captured easily in this case which may be due to the fact that requirements specification subnet was not well represented. We have used parameter estimates via Gibbs sampling to fill missing values of requirements specification subnet. Unfortunately, actual data is quite a few (15 instances), from which unknown parameters were inferred, and hence, they could not represent this subnet as good as in Dataset 1. If we had more data representing the requirement specification subnet, we would improve the prediction performance as we did in Dataset 1.

In summary, Hybrid models achieve to get quite low MdMRE rates (32% with ChiPlot) with 47% of predictions have MRE lower than 30%. Similar in Dataset 1, (Chiplot) model is better than (Stats) and (Distcov) in terms of DIC, as the best fit to data. In addition, it reduces the variance of predictions and achieves 30% MdMRE, 40% Pred(25) and 51% Pred(30).

Table 6.8. Prediction performance of models on Dataset 1.

	MMRE	**MdMRE**	**Pred(25)**	**Pred(30)**	**DIC**
GLM	0.38	0.24	0.50	0.59	393.6
Idiom	0.47	0.33	0.39	0.46	417.0
Stats	0.37	0.21	0.57	0.59	391.8
DistCov	0.41	0.27	0.46	0.57	388.0
ChiPlot	0.39	0.27	0.48	0.57	384.7
Hybrid (with Stats)	0.31	0.16	0.62	0.66	**216.5**
Hybrid (with DistCov)	0.36	0.18	0.60	0.66	292.7
Hybrid (with ChiPlot)	**0.16**	**0.11**	**0.84**	**0.84**	278.1

Table 6.9. Prediction performance of models on Dataset 2.

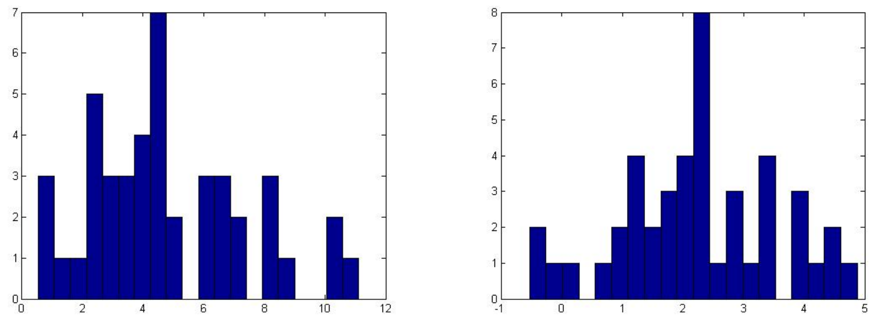
	MMRE	**MdMRE**	**Pred(25)**	**Pred(30)**	**DIC**
GLM	0.57	0.31	0.40	0.49	299.41
Idiom	0.66	0.39	0.33	0.40	326.91
Stats	0.67	0.36	0.33	0.37	293.20
DistCov	0.60	0.36	0.33	0.40	293.79
ChiPlot	0.69	**0.30**	0.44	**0.51**	**280.08**
Hybrid (with Stats)	0.68	0.38	0.30	0.33	293.99
Hybrid (with DistCov)	0.60	0.36	0.33	0.37	294.50
Hybrid (with ChiPlot)	0.69	0.32	0.42	0.47	284.42

6.5. Discussion on Data Quality

In all studies working on real industrial settings, there may exist data anomalies, such as exceptional data points affecting dataset structure and model parameters, or outliers resulted due to a miscalculation during measurement [100]. Outliers are defined as abnormal data objects that do not follow a generating mechanism, i.e. statistical distribution [110]. Turhan discussed that in order to avoid false generalizations, researchers should be aware of outliers' existence while they should be careful during the choice of removing or keeping them [100]. There are several outlier detection techniques, some of which are probability tests based on statistical models, depth-based, distance-based, density-based or deviation-based methods [110]. In software engineering studies, outliers or data quality are often discarded with a few exceptions. Turhan summarized a number of publications explicitly addressed outlier detection using techniques, such as Cook's distance for ordinary least square regression models, jackknifed cross-validation and visual inspections with box-plots [100].

Removing outliers may significantly increase the prediction performance of a model (e.g. in predicting residual defects), but it may also cause that the model is limited to represent certain type of defects, while it misses to capture the others [100]. It is challenging to define which instances in a dataset are “outliers”, since removing extreme values from the data may also form a narrowed distribution, so that previously “normal” instances would look like “outliers” in the current filtered data. Remove outliers when dataset size is small or we are not sure about the distribution of data under investigation is also very risky. Before considering removing these points from data, we should understand why this value appears during the process and whether it is likely that similar values may exist in the future.

Traditional statistical tests consider outliers as strong deviations from a distribution [111]. The simple rule behind identifying outliers is to look at data points deviating more than z times standard deviation from the mean [111]. Value of z is often assigned as 3 with varying between 2.5 to 4.0 depending on the sample size and skewness of dataset. Box plots and inter-quartile ranges (IQR) are also applied during outlier de-



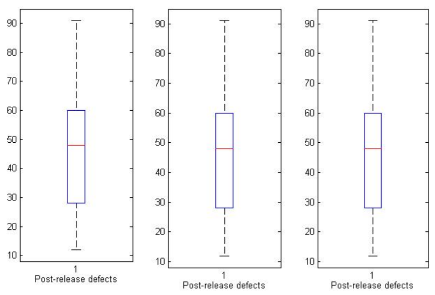
(a) *Histogram of original data* (b) *Histogram of transformed data*

Figure 6.16. Effect of Box-cox transformation on complexity metric in Dataset 1.

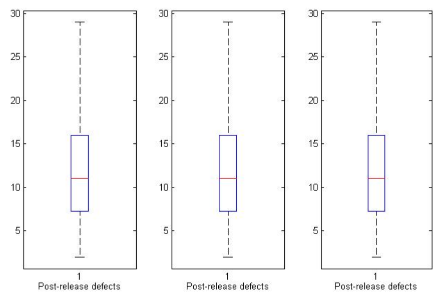
tection, however the assumption that data follow a Gaussian distribution, limits the usage of this technique. The region between $Q_1 - 1.5IQR$ and $Q_3 + 1.5IQR$, where Q_1 represents lower quartile, Q_3 represents upper quartile and $IQR = Q_3 - Q_1$, contain 99.3% of observations for a Gaussian distributed data [112]. Possible transformation techniques, such as Box-Cox [113] can be used when the distribution of a variable may not be known or there is a possibility of high skewness on dataset. Box-Cox transformation is applied to data (Y) as follows:

$$T(y_i) = \begin{cases} \frac{(y_i^\lambda - 1)}{\lambda} & \text{if } \lambda \neq 0 \\ log(y_i) & \text{otherwise} \end{cases} \tag{6.10}$$

An illustration of the effect of Box-Cox transformation on complexity metric from the first dataset can be seen in Figure 6.16. We observe box plots of post-release defects of both datasets after applying statistical tests and IQR limits to identify possible outliers. Figure 6.17 shows that none of the data points is considered as outlier for post-release defects, and hence box plots are the same after both techniques. This may be due to the fact that post-release defects come from a distribution (Gaussian according to our normality tests) with no indication of skewness/ heavier or thinner tails. Therefore, we do not replicate our experiments on a filtered data. We can also conclude that removing outliers based on an outlier detection technique is very risky



(a) *Dataset 1*



(b) *Dataset 2*

Figure 6.17. Box plots of post-release defects for both datasets. First sub figure represents the original data, while the second represents filtered data after statistical tests and the final sub figure represents the filtered data based on IQR.

while building predictive models, such that data should be investigated deeply with a bunch of outlier detection techniques considering their distributional assumptions.

7. THREATS TO VALIDITY

Many types of validity can be invoked when trying to develop a framework to understand experiments in complex field settings. Campbell and Stanley invoked two validity types, “internal” and “external” [114]. According to Cook and Campbell [115], “statistical conclusion validity” should be defined as a special case of internal validity in order to consider sources of random error and appropriate use of statistics and statistical tests. Furthermore, “construct validity” is further defined in order to consider “confounding”, as well as the generalizability of experiments across different settings (external validity). Confounding relates to the representation of the relationship between constructs and requires naming measures in generalizable terms [114].

In this research, we have also highlighted possible threats to internal, external, construct and conclusion validity of our experiments. Before observing measures and their relationships for internal validity, we have considered possible biases during data collection. Collection of data, required to build prediction models, is a challenging task in software organizations. In order to avoid bias during data collection, we have been supported by release management team of the organizations and scripts written to extract required metrics were validated by them. Furthermore, we have investigated possible outliers in datasets which may occur due to miscalculation of metrics. Extreme values were manually traced and discussed with the management to understand their reasons and the possibility of having similar values in future releases. We have also applied two popular outlier detection techniques during our discussion on data quality (Section 6.5) and concluded that removing extreme values from the datasets may distort the underlying distributions, even though it may improve the prediction performance.

Our research objectives include quantifying software processes by considering their impacts on software reliability. We have defined post-release (residual) defects as a measure of software reliability, since quality is the major attribute of software reliability, and measuring software quality is quite often defined as predicting defects/faults or fault density in software systems [17].

During quantification of software processes, all development and testing metrics were selected based on previous empirical research investigating unique characteristics of post-release defects in software systems. Although there is not a study observing all processes at once in terms of local data, we have gone through the literature and formed a larger set of metrics representing both development and testing processes. We have also discussed prior research on most of these metrics in Section 2.2, emphasizing that they were very good at predicting post-release defects in software systems. Qualitative factors, on the other hand, were previously validated as effective when predicting post-release defects of several commercial software systems [45].

We have stated that causal relationships among major constructs of our research should be encoded in order to identify possible effects of these constructs, i.e., measures, on software reliability. Simple regression models would not identify causal relations between metrics representing different processes and residual defects, even though they were popularly used in previous studies (see [12,24]). Bayesian networks are well suited to our problem, where the aim is to encode causal relationships from data and predict posterior distributions using prior evidence. In fact, if we consider independence of software metrics and build a simple regression model, we have seen that software metrics can only explain 32% of variability (R^2) in the first dataset, and 36% of variability in the second dataset. However, taking causal relationships into account helps to achieve very good prediction rates in addition to understanding the dynamics of a development life cycle. We have seen that Bayesian networks help us identify that the changes in our independent variables (metrics) cause the observed changes in our dependent variable (post-release defects). We have considered causal relationships between metrics from development and testing processes based on statistical and graphical independence tests, but we could not identify relationships between metrics representing requirements process and other processes.

Bayesian statistics in conjunction with Bayesian networks also offer an efficient approach to avoid over-fitting [59]. It is not necessary to build the network with separate training and test data as in learning-based prediction models, since Monte Carlo techniques avoid sampling bias by inferring posterior distributions from the model. In addition, causal relations between variables were found based on different statistical tests, including correlation statistics and independence tests, as well as graphical dependence analysis. We did not consider one method superior to the others, but we modelled the network with all approaches respectively.

To overcome threats to construct validity in terms of selecting the right performance measures, we have considered popularly used performance measures in various effort estimation studies (e.g. [98]): MMRE, MdMRE, Pred(k). MMRE metric computes the average relative error between actual and predicted values in a typical regression problem; however it has been criticized as being sensitive to extreme values in data. Pred captures the variation among predictions and hence, it also supports MdMRE which is less sensitive to large variations in data, compared to MMRE. Therefore, we have also reported MdMRE, Pred(k) values in our experiments.

To overcome possible threats to statistical conclusion validity, we have checked the statistical significance of our predictions using a non-parametric test, Mann-Whitney U-test. Other tests such as ANOVA or t-tests make Gaussian assumptions that may not hold in every domain or dataset. We have also used Spearman's correlation test and Kolmogorov-Smirnov test which are also computed over ranks as Mann-Whitney U-test. Brownian covariance independence test is applied on pairwise Euclidean distances, and hence, it does not have a distribution assumption. Finally, chi-plots are also based on ranks of variables and it is more informative than scatter plots, due to independence of a relation between two variables from marginal distributions of data. Thus, we have considered selecting appropriate tests during our experiments in order to avoid assumptions implied by several tests.

The software engineering community has begun to emphasize empirical research methods to improve the validity and generalizability of research results [116]. Even though experiments in many studies give precise measurements under controlled conditions, they may suffer from a lack of generalizability if they are not carefully designed. Drawing general conclusions from an empirical study is very difficult, since each dataset contains unique characteristics, in terms of software processes and the product, of the organization it belongs to. However results should be transferable to other researchers in terms of the methodology behind. In this research, we have proposed an application of Bayesian Networks in software engineering domain by proposing solutions to limitations in previous studies. Methods used during model construction can be easily transferable to other studies in order to repeat or refute our conclusions.

8. CONCLUSIONS

8.1. Summary of Results

Due to the nature of empirical studies in software engineering, there are limited number of empirical findings each tested on different settings and it is harder to verify your theories [116]. Since software engineering is an applied science, theories should be useful to academia, as well as to the software industry. Therefore, we need to be accurate while summarizing and generalizing our results. This research proposes an application of Bayesian networks by selecting to build a Hybrid network containing both quantitative data regarding development and testing processes and qualitative data regarding requirements specification process. Our empirical findings on two different software systems show that the effects of software processes and their relationships on software reliability are worth the investigation. In addition, we have observed that local data is very valuable as expert knowledge such that in software organizations with a mature measurement/ metrics data repository, local data should be used primarily and strengthened with expert knowledge while predicting software reliability. We have also linked our research questions in Section 2.5 with our findings on two different datasets.

8.1.1. What are the software factors affecting software reliability?

We have quantified software metrics representing development and testing processes to measure software reliability. After an extensive literature survey, we have evaluated software reliability in terms of *post-release/residual defects* and then, we have used previously well-known metrics from development process, which were defined as significant indicators of defects in software projects [12, 27]. We have also investigated in-process metrics that are effective for managing software testing [86]. Three major aspects of SDLC (People, Product, Process) are covered with these metrics, namely *cyclomatic complexity* represents the product, whereas *age, number of commits, added LOC, deleted LOC, average time between edits* represent development process and *number of committers* represents the people aspect. Similarly, we have selected *number of test cases, number of test cases with errors, number of test cases with failures, number of test defects* represent the product, whereas *percentage of test cases with a status change, test case quality, average number of executions per test case* represent the testing process.

For quantifying requirements specification and documentation process, we have used a survey proposed in [45] capturing factors such as people experience, quality of documentation, review process and configuration management. Based on previous research, these factors were successful at representing requirements process and predicting post-release defects [45]. We have also incorporated expert judgement into the model via surveys due to lack of local data regarding this process in both organizations.

8.1.2. How do software processes affect software reliability?

We have believed that dependencies between software processes affect final reliability and validated our research question by building (i) a generalized linear model in which each metric affects final reliability independently, and (ii) causal models in which cause-effect relationships among software processes are represented based on statistical analysis. Results show that causal models are better at predicting software reliability than linear model.

8.1.3. How can we build a predictive model that would estimate software reliability prior to release?

Bayesian networks are well suited to our problem since (i) they encode causal relationships between input and output variables, (ii) they can be built using both continuous and categorical variables in a hybrid model, (iii) predicting software reliability can be formalized as inferring posterior probability of post-release defects given observations from software metrics.

We have stated that a hybrid model is necessary when organizations have different levels of maturity in different processes of SDLC. We have also applied a hybrid data collection approach by using quantitative data extracted from organizational repositories and qualitative data collected through surveys. Hence, we have proposed a Hybrid Bayesian Network that incorporates both continuous and categorical variables without any transformation, such as discretization. Gibbs sampling is used for approximation of the joint probability distribution over conditional distributions, estimation of unknown parameters and for handling missing data points in qualitative part. Results show that proposed Bayesian networks are able to predict upto 84% of post-release defects with less than 25% MRE in the first dataset, while 51% of defects with less than 30% MRE in the second dataset.

8.1.4. How can software companies build predictive models with local data rather than based solely on expert knowledge?

We have seen that a predictive model can estimate post-release defects in most of the releases with less than 25% MRE rate, when causal relationships are modelled through statistical tests and graphical analysis. In software organizations with a mature measurement process, accessing local data can be easier than communicating with senior managers to conduct surveys, especially when development teams are allocated at different geographical locations. In the second dataset, we have seen that incorporating requirements subnet into other processes could not improve the prediction performance. One reason for this may be limited survey data. However, it may also indicate that in software companies with small and cohesive development teams having highly experienced staff and almost no turnover, local data may represent expert behaviour very successfully. Therefore, survey data may not add new information into the model. This also supports our claim that local data can be successfully used in such predictive models instead of depending solely on expert judgement.

8.2. Contributions

We have achieved theoretical and methodological contributions as well as highlighted their practical implications to the industry.

8.2.1. Theoretical and Methodological Contributions

In this research, we have applied well known AI techniques to a software engineering problem, i.e., predicting software reliability prior to a release decision. Recent research in predicting software reliability have also utilized Bayesian networks, but proposed models were fed by qualitative factors collected via surveys and causal relations were also defined by experts. We have proposed a new methodology in data collection by incorporating local data with expert knowledge as well as a new network model (*hybrid*) handling mixture of these data without any transformation. Cause-effect relationships are also defined based on statistical tests strengthened with graphical analysis, rather than using expert judgement.

In addition to methodological contributions, a well known Monte Carlo method, namely Gibbs sampling, has been applied during Bayesian inference in software engineering domain in order to estimate unknown parameters of probability distributions and predict joint distribution of the final model over conditionals. Furthermore, we have imputed missing data in surveys via Gibbs sampling, specifically by estimating unknown parameters of the underlying distribution.

Similar models using BNs were previously proposed in the literature [9, 45, 47, 52, 79]. Some authors also built a tool that would help building a BN [45]. We have improved previously proposed models by applying statistical and graphical independence tests to encode causal relationships between variables, rather than depending solely on expert judgement during network construction. We have also managed to build a *hybrid* network that can incorporate categorical variables into continuous variables without appyling any transformation technique, like discretization. Inference in this hybrid network is also handled by using Monte Carlo methods and working on a small sample of local data. However, our model is a bit more complex to construct than previous models, since we did not apply any transformation on continuous variables to ease probabilistic inference. But, there are freely available inference engines, like WinBugs [108], that would help inference in complex models.

We have performed a set of well-defined experiments during model construction which allows other researchers to replicate our methodology. Reproducibility of empirical studies is a very important property in software engineering domain. Unfortunately, building complex models like ours, involving both a hybrid data collection and a network construction with various statistical tests, is very difficult and interpret, even though we have defined all steps precisely. An automated mechanism makes this process significantly easier and more reliable. Therefore, we have started developing a tool, namely Dione, which is not only for predicting software reliability through Bayesian networks, but also a metric collection, analysis and reporting tool for the use of both academicians and practitioners.

8.2.2. Practical Implications

In practice, software organizations need predictive models while making release decisions in order to reduce the risk of over-/under-estimations and dependence on experts. Using estimated post-release (residual) defects and the company's pre-defined thresholds for software reliability, it is possible to estimate a *release readiness level* for each release. This level informs software managers earlier in development life cycle in order to be prepared for the following scenarios:

(i) *Release the software:* Its readiness level in terms of residual defects is above a pre-defined threshold. Managers are aware of potential failures after the release and they are well prepared for this scenario.

(ii) *Delay the release:* Its readiness level is within the acceptable ranges, but the software still needs improvement: Additional requirements, design or testing efforts may be necessary through code reviews or inspections.

(iii) *Cancel the release:* Its readiness level is below the acceptable ranges, and failures after the release may cost much more than making drastic changes right now.

Furthermore, such predictive models help software managers observe maturity of their processes in terms of their impacts on software reliability. Measuring/quantifying software processes also helps software organizations build a measurement repository and monitor trends of development practices in a systematic manner. An automated tool like Dione also provide tangible benefits to companies such as collecting and storing local data easily and invisible to the staff, building a complex model based on historical data and estimating release readiness periodically. We aim to extend Dione with a questionnaire support to conduct surveys as well.

In this research, we have seen that there are not a specific set of metrics that should be used in any software organization to predict reliability in terms of post-release defects. However, it would be useful to extract a common set of metrics (as the ones we extracted during this research) which were proved as significant indicators of post-release defects. Causal relationships also change from one model to another based on local data collected from the organization. Hence, we suggest practitioners to follow our methodology rather than using same metric sets with same causal relationships among each other.

As a performance evaluation measure, we proposed 4 different measures, MdMRE, MMRE, Pred(25) and Pred(30) and assessed the performance of our models by discussing all of them respectively. However, in real life, practitioners may require using a single measure in order to judge their model in terms of its prediction performance. In that case, we suggest them using *Pred(k)* measure, since Pred would evaluate the overall success of a model in terms of variance of its predictions'. It is also possible to set success thresholds for a model, such as "the model should predict at least 75% of projects with 25% error or less", and evaluate how the model meets the requirements using *Pred* measure.

8.3. Future Directions

Popular applications of BNs on predicting software reliability lack a generalizable methodology for model construction due to dependence on expert judgement. To consider generalizability of our results, we have proposed well known statistical correlation and independence tests as well as graphical analysis to infer causal relationships from local data. We have also defined a set of software metrics, that are characteristics of residual defects, which can be extracted from organizational repositories. As a future work, this research can be expanded to different settings by adding new set of metrics depending on available local data and encoding causal relationships with proposed techniques. The objective of this research in the long run is to build robust predictive models and integrate them into daily routines of software development activities so that software managers would use such models to define policies.

APPENDIX A: CODES OF PROPOSED MODELS FOR DATASET 1

A.1. Winbugs code for the implementation of GLM

```
model{
for(i=1:N){
p[i] <- beta[1] + beta[2]*TEST_CASE[i] +
        beta[3]*TEST_DEFECT[i] + beta[4]*AVG_TEST_RUN[i] +
        beta[5]*TEST_CASE_QUALITY[i] + theta[1]*AGE[i] +
        theta[2]*PEOPLE[i] + theta[3]*CHURN[i] +
        theta[4]*EDIT_FREQUENCY[i] + theta[5]*COMPLEXITY[i]
DEFECT[i] ~ dnorm(p[i], t)
}
for(j=1:5){
beta[j] ~ dnorm(1,0.001)
theta[j] ~ dnorm(1,0.001)
}
t ~ dgamma(0.5,0.5)
}
```

A.2. Winbugs code for the implementation of Idiom

```
model{
for(i=1:N){
COMPLEXITY[i] ~ dnorm(AGE[i], a)
CHURN[i] ~ dexp(PEOPLE[i])

e[i] <- theta[1] + theta[2]*COMPLEXITY[i] +
        theta[3]*CHURN[i] + theta[4]*EDIT_FREQUENCY[i]
h[i] ~ dnorm(e[i], s)

x[i]  <- TEST_CASE[i]*h[i]
TEST_DEFECT[i] ~ dlnorm(x[i], t)
TEST_CASE_QUALITY[i] ~ dnorm(TEST_CASE[i], b)
y[i] <- h[i] * K
AVG_TEST_RUN[i] ~ dnorm(y[i], t)
p[i] <- beta[1] + beta[2]*TEST_DEFECT[i] +
        beta[3]*AVG_TEST_RUN[i] +
        beta[4]*TEST_CASE_QUALITY[i]

 DEFECT[i] ~ dnorm(p[i], t)
}
for(j=1:4){
beta[j] ~ dnorm(1,0.001)
theta[j] ~ dnorm(1,0.001)
}
 t ~ dgamma(0.5,0.5)
 s ~ dgamma(0.5,0.5)
 a ~ dgamma(0.5,0.5)
 b ~ dgamma(0.5,0.5)
 K ~ dgamma(1,1)
}
```

A.3. Winbugs code for the implementation of Stats

```
model{
for(i=1:N){
 COMPLEXITY[i] ~ dnorm(e[i], t)
 PEOPLE[i] ~ dpois(lambda[i])
 AGE[i] ~ dnorm(o[i], p)
 TEST_CASE_QUALITY[i] ~ dnorm(k[i], w)
 h[i] <- theta*COMPLEXITY[i]*PEOPLE[i]
 CHURN[i] ~ dexp(h[i])

 v[i] <- scale*AGE[i]
 AVG_TEST_RUN[i] ~ dnorm(v[i], s)

 a[i] <- alpha * TEST_CASE_QUALITY[i] * COMPLEXITY[i]
 TEST_DEFECT[i] ~ dexp(a[i])

 y[i] <- beta[1] + beta[2]*TEST_DEFECT[i] +
         beta[3]*AVG_TEST_RUN[i] + beta[4]*TEST_CASE[i] +
         beta[5]*CHURN[i] +
         (beta[6] * TEST_DEFECT[i] * AVG_TEST_RUN[i]
         * TEST_CASE[i] * CHURN[i]) +
         (beta[7] * TEST_DEFECT[i] * AVG_TEST_RUN[i]) +
         (beta[8] * TEST_DEFECT[i] * TEST_CASE[i]) +
         (beta[9]* CHURN[i] * TEST_DEFECT[i])

 DEFECT[i] ~ dnorm(y[i], g)
}
for(j=1:9){
 beta[j] ~ dnorm(1,0.001)
}
for(f=1:N){
 e[f] ~ dnorm(5, 1)
 lambda[f] ~ dexp(2)
 o[f] ~ dgamma(1.7, 550)
 k[f] ~ dgamma(10,10)
}
}
```

A.4. Winbugs code for the implementation of DistCov

```
model {
for(i=1:N){
CHURN[i] ~ dexp(w[i])
TEST_CASE_QUALITY[i] ~ dnorm(a[i], k)
s[i] <- alpha[1] + alpha[2]*TEST_CASE_QUALITY[i] +
        alpha[3]*CHURN[i]

COMPLEXITY[i] ~ dnorm(m[i],c)
TEST_DEFECT[i] ~ dexp(s[i])
PEOPLE[i] ~ dpois(lambda[i])
AGE[i] ~ dnorm(ageMean[i],e)

t[i] <- beta[1] + beta[2]*PEOPLE[i] +
        beta[3]*AGE[i] + beta[4]*TEST_DEFECT[i]

TEST_CASE[i] ~ dnorm(t[i], h)
AVG_TEST_RUN[i] ~ dnorm(TEST_CASE_QUALITY[i], f)

y[i] <- theta[1] + theta[2]*TEST_CASE[i] +
        theta[3]*AVG_TEST_RUN[i] + theta[4]*COMPLEXITY[i] +
        (theta[5]*TEST_CASE[i]*AVG_TEST_RUN[i])
DEFECT[i] ~ dnorm(y[i],g)
}
}
```

A.5. Winbugs code for the implementation of ChiPlot

```
model {
for(i=1:N){
 TEST_DEFECT[i] ~ dt(muTD[i], tauTD[i], d)
 AGE[i] ~ dnorm(muAGE[i], varAGE)

 h[i] <- theta*COMPLEXITY[i]
 CHURN[i] ~ dnorm(h[i], varC)
 PEOPLE[i] ~ dpois(lambda[i])

 funcTC[i] <-  cTC * (TEST_DEFECT[i] + AGE[i])
 TEST_CASE[i] ~ dnorm(funcTC[i] , varTC)

 temp[i] <- cTCQ * AGE[i]
 TEST_CASE_QUALITY[i] ~ dnorm(temp[i], w)

 v[i] <- scale *TEST_CASE_QUALITY[i]
 AVG_TEST_RUN[i]  ~ dnorm(v[i], s)

 H[i ] <- Hbeta[1] + Hbeta[2]*PEOPLE[i] +
          Hbeta[3]*CHURN[i]

 y[i] <- beta[1] + beta[2] * TEST_CASE[i] +
         beta[3] * AVG_TEST_RUN[i] +
 beta[4]*EDIT_FREQUENCY[i] +
 (beta[5] * TEST_CASE[i] * AVG_TEST_RUN[i] *
 EDIT_FREQUENCY[i]) + beta[6] * H[i]

 DEFECT[i] ~ dnorm(y[i],g)
}
}
```

A.6. Winbugs code for the implementation of Specification and Documentation Subnet

```
model {
for(i=1:N){
reqA[i] <- coefR[1] + coefR[2]*S1[i] +
           coefR[3]*S3[i] + coefR[4]*S4[i]
A[i] ~ dnorm(reqA[i], reqVA)

reqC[i] <- coefRC[1] + coefRC[2]*S7[i] +
           coefRC[3]*P2[i] + coefRC[4]*P5[i]
C[i] ~ dpois(reqC[i])

reqF[i] <- coefRF[1] + coefRF[2]* A[i] +
           coefRF[3]* C[i] + coefRF[4]* S2[i]
probSpecDefect[i] ~ dnorm(reqF[i], reqVF)
}
for(b=1:4){
coefR[b] ~ dnorm(2, 0.01)
coefRC[b] ~ dnorm(4, 0.01)
coefRF[b] ~ dnorm(3, 0.01)
}
reqVA ~ dgamma(1,1)
reqVF ~ dgamma(1,1)
}
```

APPENDIX B: CODES OF PROPOSED MODELS FOR DATASET 2

B.1. Winbugs code for the implementation of GLM

```
model {
for(i=1:N){
y[i] ~ dnorm(p[i],t)
p[i] <- beta[1] + beta[2]*commits[i] +
beta[3]*committers[i] + beta[4]*addedLOC[i] +
beta[5]*delLOC[i] + beta[6]*commitFreq[i] +
beta[7]*complexity[i] + beta[8]*testCases[i] +
beta[9]*errorTest[i] + beta[10]*failTest[i] +
beta[12]*testSuites[i] + beta[13]*testDefects[i]
}
for(j=1:13){
beta[j] ~ dnorm(1,0.001)
}
t ~ dgamma(0.5,0.5)
}
```

B.2. Winbugs code for the implementation of Idiom

```
model {
for(i=1:N){
cMu[i] <- abs(addedLOC[i] - delLOC[i]) * committers[i]
changedLOC[i] ~ dt(cMu[i], tau, d)

scale[i] <- (1/ theta) * commits[i]
commitFreq[i]  ~ dnorm(scale[i], var1)
H1[i] <- h1beta[1] + h1beta[2]*changedLOC[i] +
  h1beta[3]*commitFreq[i] + h1beta[4]*complexity[i]

testSum[i] <- (1/theta2) * (errorTest[i] + failTest[i] +  testSuites[i])
testDefects[i]  ~ dnorm(testSum[i], var2)
H2[i] <- h2beta[1] + h2beta[2]*testDefects[i] +
  h2beta[3]*testsChange[i]

p[i] <- beta[1]*H1[i] + beta[2]*H2[i] +
(beta[3]*H1[i]* H2[i])
y[i] ~ dnorm(p[i], t)
}
for(j=1:3){
h2beta [j ] ~ dnorm(1,0.01)
beta[j] ~ dnorm(1, 0.01)
}
for(k=1:4){
h1beta[k] ~ dnorm(1, 0.01)
}
t ~ dgamma(2,2)
var1 ~ dgamma(1,1)
theta ~ dnorm(1,1)
theta2 ~ dnorm(50, 0.1)
var2 ~ dnorm(20,0.1)
tau ~ dnorm(70, 1)
d<- 2
}
```

B.3. Winbugs code for the implementation of Stats

```
model {
for(i=1:N){
testCases[i] ~ dlnorm(tcMu[i], tcVar)
scale[i]<- commits[i] * theta
errorTest[i]  ~ dexp(scale[i])
delLOC[i] <- addedLOC[i] * scale2
failTest[i] ~ dt(addedLOC[i] , fTau[i] , d)
sumTest[i] <- (coeff[1]*testCases[i]) +
      (coeff[2]*errorTest[i]) +
      (coeff[3]*failTest[i]) + (coeff[4]*delLOC[i])

testSuites[i]  ~ dt(sumTest[i], sTau[i] , d2)
p[i] <- beta[1] + beta[2]*testSuites[i] +
beta[3]*committers[i] + beta[4]*complexity[i] +
beta[5]*testsChange[i]
y[i] ~ dnorm(p[i], t)
}
for(e=1:6){
beta[e] ~ dnorm(1,0.01)
coeff[e] ~ dnorm(1, 0.001)
}
for(k=1:N){
tcMu[k] ~ dnorm(7, 0.001)
sTau[k] ~ dnorm(5,5)
dvar[k] ~ dnorm(1, 0.01)
fTau[k]  ~ dnorm(30, 0.1)
}
theta ~ dnorm(1, 0.1)
scale2 ~ dnorm(0.1, 0.01)
t ~ dgamma(2,2)
tcVar  ~ dnorm(1, 0.001)
d2 <- 2
d <- 2
}
```

B.4. Winbugs code for the implementation of DistCov

```
model {
for(i=1:N){
scale[i] <- theta * commits[i] * committers[i]
commitFreq[i]  ~ dnorm(scale[i], var1)

testCases[i] ~ dlnorm(tcMu[i], tcVar)
errorTest[i]  ~ dexp(muError[i])
failTest[i] ~ dt(fMu[i] , fTau[i] , d)

sumTest[i] <- (var2 * testCases[i ]) +
(var3 * errorTest[i]) +
(var4 * failTest[i])
testSuites[i]  ~ dt(sumTest[i], sTau[i] , d2)

p[i] <- beta[1] + beta[2]*commitFreq[i] +
 beta[3]*testSuites[i] +
 beta[4]*testSuites[i]*commitFreq[i] +
 beta[5]*(addedLOC[i] + delLOC[i]) +
 beta[6]*complexity[i] + beta[7]*testDefects[i] +
 beta[8]*testsChange[i]

y[i] ~ dnorm(p[i], t)
}
for(j=1:8){
```

```
beta[j] ~ dnorm(1, 0.001)
}
for(k=1:N){
tcMu[k] ~ dnorm(7, 0.001)
muError[k] ~ dgamma(250,1)
fMu[k]  ~ dgamma(15, 2)
fTau[k]  ~ dnorm(30, 0.1)
sTau[k] ~ dnorm(5,5)
}
}
```

B.5. Winbugs code for the implementation of ChiPlot

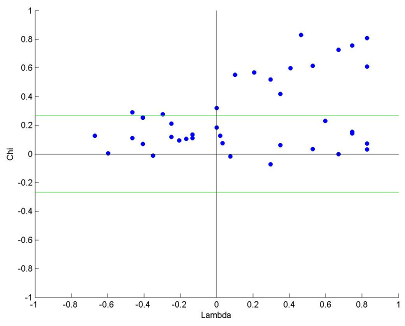
```
model {
for(i=1:N){
changedLOC[i] <- addedLOC[i] + delLOC[i]

tcMuF[i] <- HtestConstant * tcMu[i]
muErrorF[i ] <- muError[i] + HtestConstant
fMuF[i] <- coeff[1] +
(coeff[2] * pow(fMu[i], HtestConstant)) +
(coeff[3] * changedLOC[i])
testCases[i] ~ dlnorm(tcMuF[i], tcVar)
errorTest[i]  ~ dexp(muErrorF[i])
fTauOrg[i ] <- 1/ fTau[i]
failTest[i] ~ dt(fMuF[i] , fTauOrg[i] , d)

scale[i] <- theta * commits[i]
scale2[i] <- theta2 * commits[i]
commitFreq[i] ~ dnorm(scale[i], var1)
complexity[i] ~ dnorm(scale2[i], var2)
H[i] <- coeff2[1] + (coeff2[2]*committers[i]) +
(coeff2[3]*commitFreq[i]) + (coeff2[4]*complexity[i]) +
(coeff2[5]*testsChange[i]) + (coeff2[6]*testSuites[i]) +
(coeff2[7]*testDefects[i])

p[i] <- beta[1] + (beta[2]*failTest[i]) + (beta[3]*H[i])
y[i] ~ dnorm(p[i], t)
}
}
```

APPENDIX C: SURVEY RESPONSES

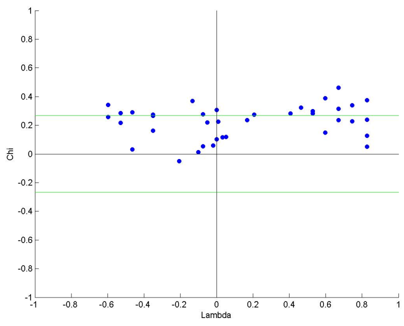
Table C.1. Responses for Specification, Documentation & Project Management in Datasets 1 & 2.

Response No.	Dataset 1							Dataset 2						
	S1	S2	S3	S4	S7	P2	P5	S1	S2	S3	S4	S7	P2	P5
1	5	4	2	1	3	5	3	5	3	2	3	4	4	5
2	3	2	4	2	1	2	5	4	3	5	3	3	4	4
3	4	3	4	3	1	3	3	5	2	3	1	3	4	4
4	3	3	5	3	4	3	4	4	3	5	3	3	4	4
5	4	3	3	2	4	3	5	4	3	4	1	3	3	4
6	5	3	2	4	2	5	3	3	2	1	1	2	3	3
7	4	3	3	2	4	3	5	3	2	1	1	2	3	1
8	5	2	1	1	3	3	3	4	2	1	1	2	3	3
9	3	2	4	2	1	2	5	4	3	2	1	3	4	3
10	3	2	4	2	1	2	5	5	4	5	5	4	4	4
11	2	2	1	4	1	1	2	4	3	4	3	3	4	3
12	3	3	2	4	2	4	3	3	4	5	2	3	4	3
13	4	3	4	3	1	3	3	5	4	2	3	3	4	5
14	3	3	5	3	4	3	4	5	4	5	4	3	4	3
15	5	5	2	4	5	2	3	3	3	2	2	4	4	3
16	5	3	2	4	2	5	3							
17	5	4	5	5	2	5	4							
18	5	5	2	4	5	2	3							
19	2	2	1	1	3	2	3							
20	5	3	3	2	3	3	3							
21	4	1	2	3	3	1	4							
22	5	4	5	5	2	5	4							
23	3	3	3	3	1	5	3							
24	5	3	3	2	3	3	3							
25	4	3	3	2	4	3	5							
26	5	3	3	2	3	3	3							
27	3	2	4	2	1	2	5							
28	2	2	1	1	2	5	5							
29	4	2	3	4	2	4	3							
30	5	4	5	5	2	5	4							
31	3	2	3	2	3	1	3							
32	5	3	2	4	2	5	3							
33	4	3	3	2	4	2	2							
34	5	2	1	1	3	3	3							
35	2	2	1	1	2	5	5							
36	5	4	2	1	3	5	3							
37	4	3	1	1	2	3	3							
38	4	3	3	2	4	2	2							
39	5	4	2	1	3	5	3							

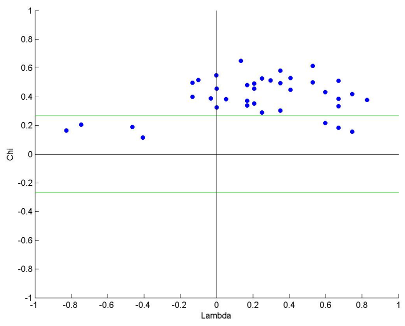
APPENDIX D: CHI-PLOTS FOR DEPENDENCE ANALYSIS



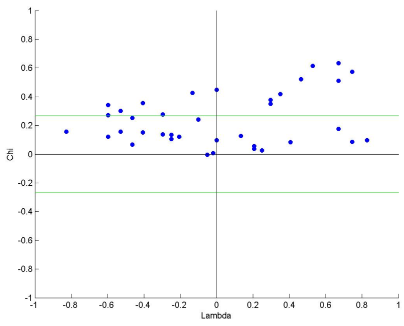
(a) *TC-TD*



(b) *TC-AGE*

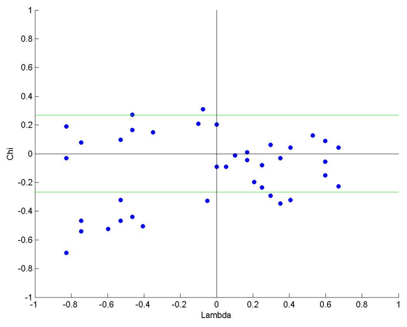


(c) *TC-DEFECT*

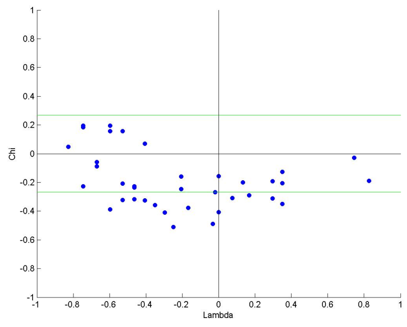


(d) *TD-DEFECT*

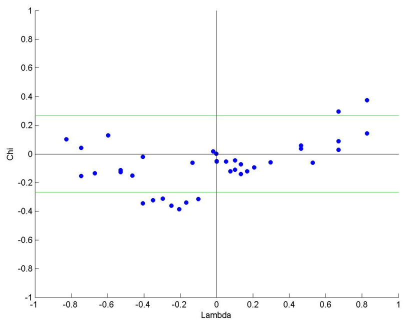
Figure D.1. Chi-plots showing an association between metrics in Dataset 1.



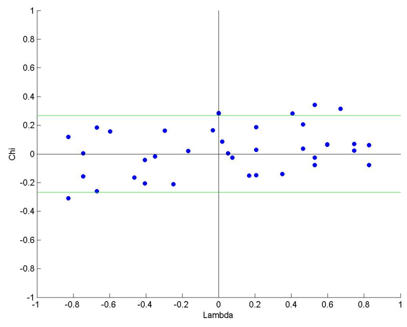
(a) *ATR-TCQ*



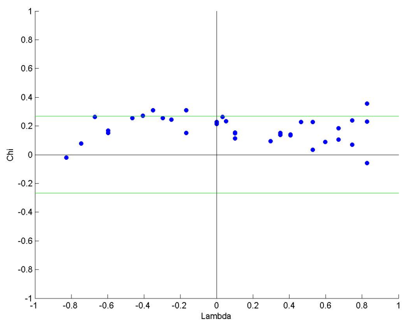
(b) *ATR-DEFECT*



(c) *TCQ-AGE*

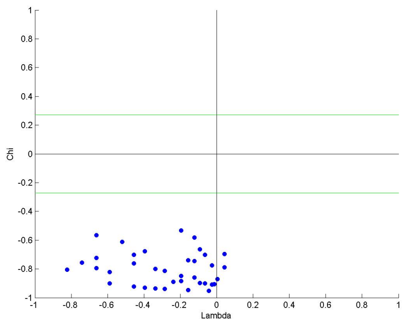


(d) *CHURN-COMPL*

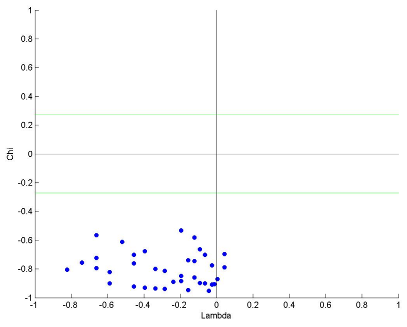


(e) *EDITFR-DEFECT*

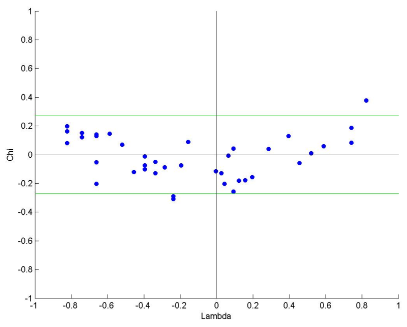
Figure D.2. Chi-plots showing an association between metrics in Dataset 1 (continued).



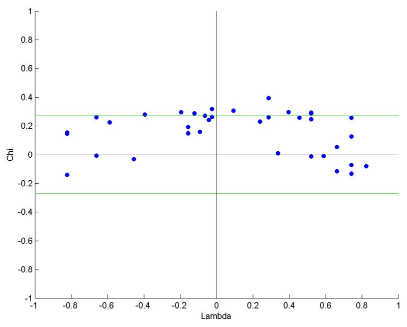
(a) *COMMITS-EDITFR*



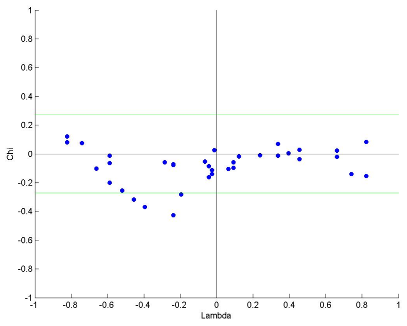
(b) *COMMITS-COMPL*



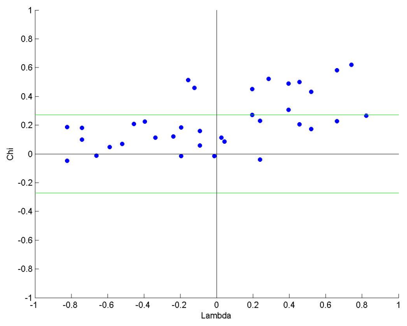
(c) *COMMITS-TCST*



(d) *PEOPLE-DELLOC*

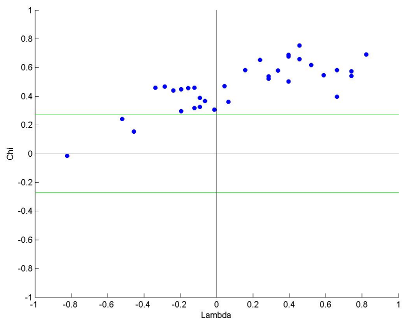


(e) *PEOPLE-EDITFR*

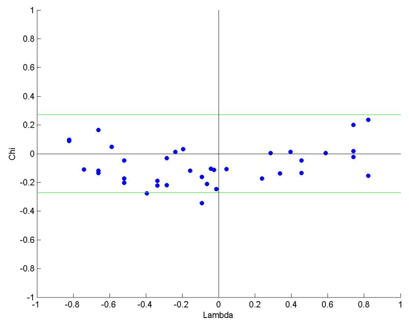


(f) *PEOPLE-TCST*

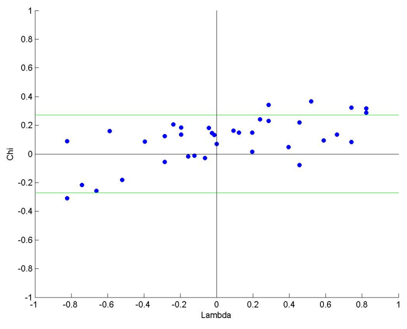
Figure D.3. Chi-plots showing an association between metrics in Dataset 2.



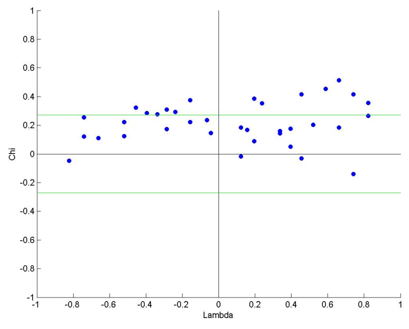
(a) *ADDLOC-DELLOC*



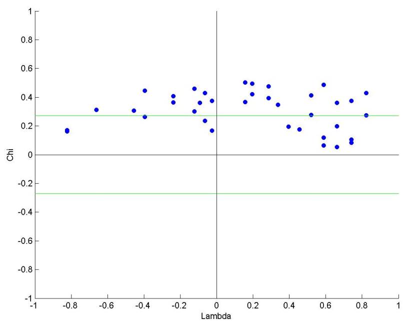
(b) *ADDLOC-TCF*



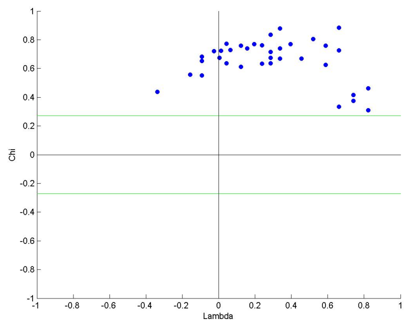
(c) *DELLOC-TCST*



(d) *DELLOC-TD*

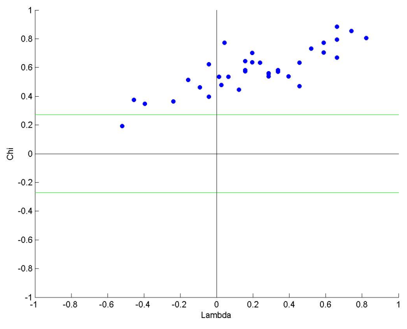


(e) *EDITFR-COMPL*

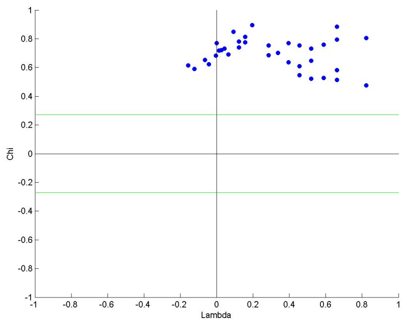


(f) *TC-TCE*

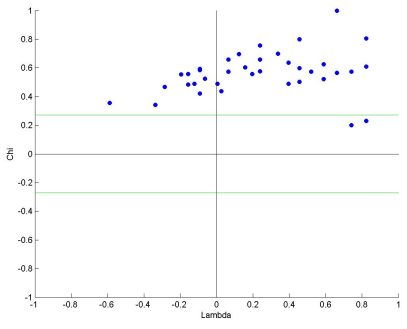
Figure D.4. Chi-plots showing an association between metrics in Dataset 2 (continued).



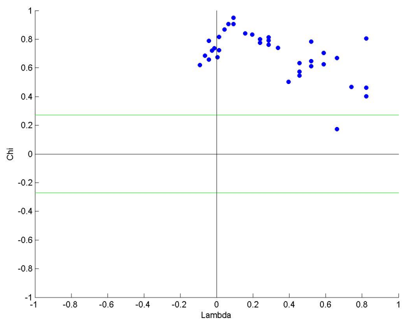
(a) *TC-TCF*



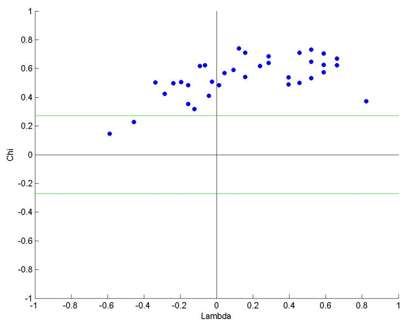
(b) *TC-TS*



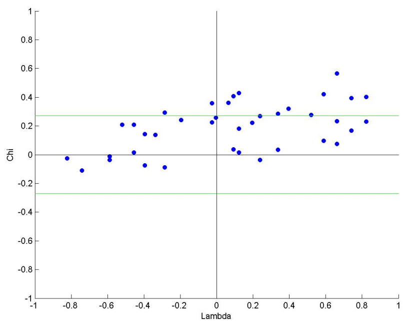
(c) *TCE-TCF*



(d) *TCE-TS*



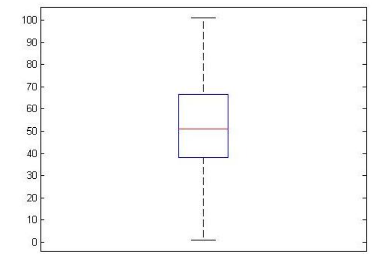
(e) *TCF-TS*

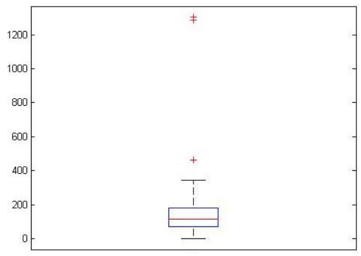


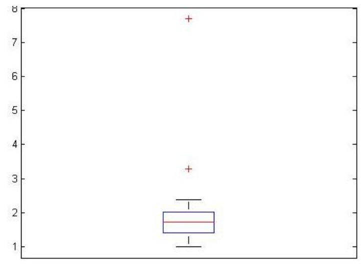
(f) *TCF-DEFECT*

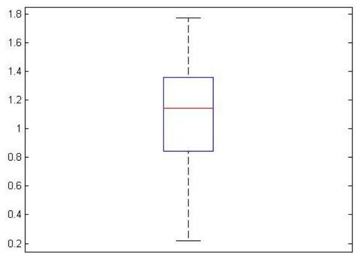
Figure D.5. Chi-plots showing an association between metrics in Dataset 2 (continued).

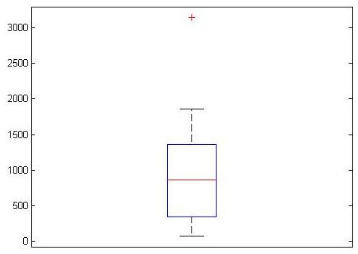
APPENDIX E: BOX-PLOTS OF SOFTWARE METRICS

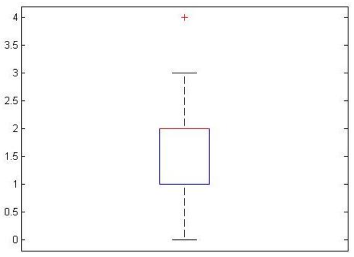

(a) *Test cases*

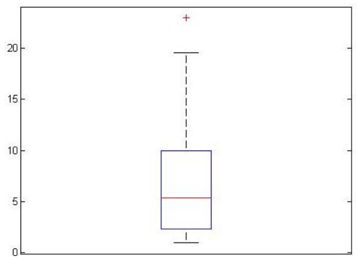

(b) *Test defects*

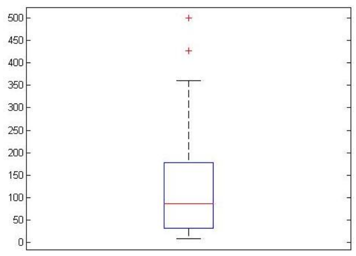

(c) *Average test run*


(d) *Test case quality*


(e) *Age*


(f) *People*


(g) *Churn*


(h) *Edit frequency*

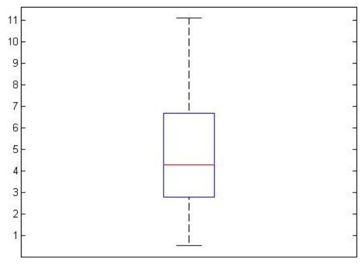
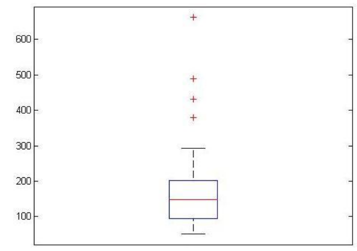
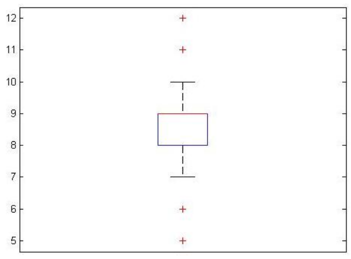

(i) *Complexity*

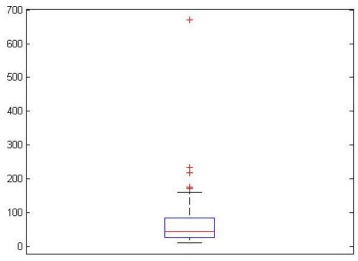
Figure E.1. Box-plots of software metrics for Dataset 1.



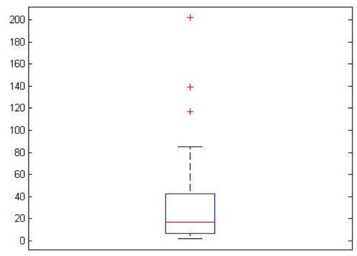
(a) *Commits*



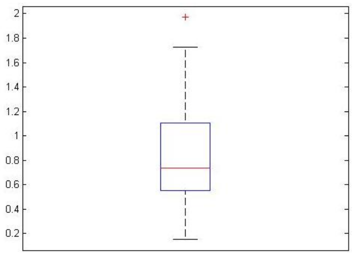
(b) *People*



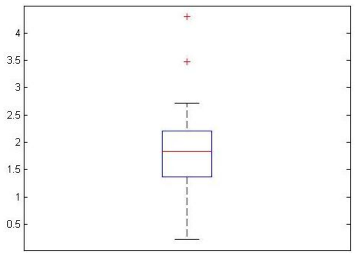
(c) *Added LOC*



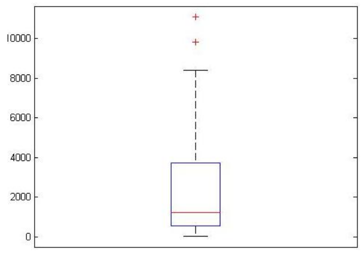
(d) *Deleted LOC*



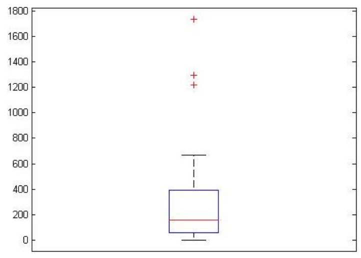
(e) *Edit frequency*



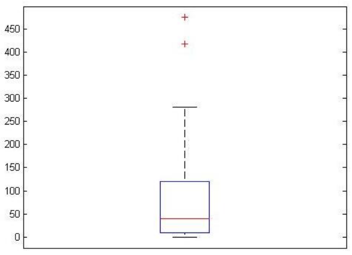
(f) *Complexity*



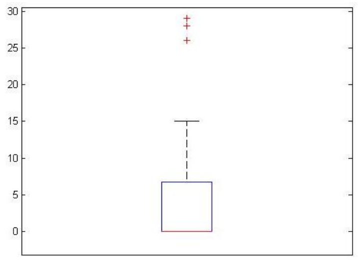
(g) *Test cases*



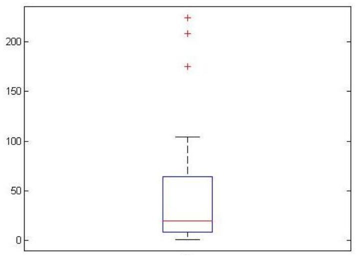
(h) *Test cases with errors*



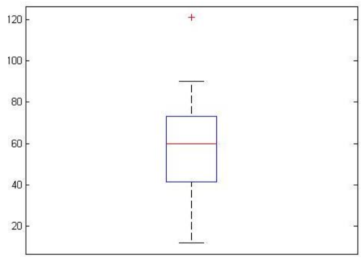
(i) *Test cases with failures*



(j) *Tests with status change*



(k) *Test suites*



(l) *Test defects*

Figure E.2. Box-plots of software metrics for Dataset 2.

REFERENCES

1. Lyu, M., “Handbook of Software Reliability Engineering”, chap. Introduction, IEEE Computer Society Press & McGraw-Hill, New York, NY, USA, 1996.

2. *Why Projects Fail: Nasaś Mars Climate Orbiter Project*, Tech. rep., JSC Centre of Expertise in the Planning & Implementation of Information Systems, 2003.

3. Germain, J., *Can Software Kill You?*, Tech. rep., Technology Special Report, 2004.

4. Farr, W., “Handbook of Software Reliability Engineering”, chap. Software Reliability Modeling Survey, IEEE Computer Society Press & McGraw-Hill, New York, NY, USA, 1996.

5. Rosenberg, L., T. Hammer and J. Shaw, “Software Metrics and Reliability”, *Proceedings of the International Symposium on Software Reliability Engineering*, pp. 361–370, 1998.

6. Ruben, O., “How Much Software Testing is Enough?”, *Communications of the ACM*, 2010.

7. Brooks, A., *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley Professional, 1995.

8. Littlewood, B., “Limits to Dependability Assurance–A Controversy Revisited”, *Companion to the proceedings of the 29th International Conference on Software Engineering*, ICSE Companion ’07, p. 6, IEEE Computer Society, Washington, DC, USA, 2007.

9. Fenton, N. E., M. Neil and D. Marquez, “Using Bayesian Networks to Predict Software Defects and Reliability”, *Journal of Risk and Reliability*, Vol. 222, pp.

701–712, 2008.

10. Boehm, B. W. and R. Valerdi, “Achievements and Challenges in Cocomo-Based Software Resource Estimation”, *IEEE Software*, Vol. 25, No. 5, pp. 74–83, 2008.

11. Menzies, T., J. Greenwald and A. Frank, “Data Mining Static Code Attributes to Learn Defect Predictors”, *IEEE Transactions on Software Engineering*, Vol. 33, No. 1, pp. 2–13, 2007.

12. Nagappan, N., L. Williams, M. Vouk and J. Osborne, “Using In-Process Testing Metrics to Estimate Post-Release Field Quality”, *ISSRE ’07: Proceedings of the The 18th IEEE International Symposium on Software Reliability*, pp. 209–214, IEEE Computer Society, Washington, DC, USA, 2007.

13. Tosun, A., A. B. Bener, B. Turhan and T. Menzies, “Practical Considerations in Deploying Statistical Methods for Defect Prediction: A Case Study within the Turkish Telecommunications Industry”, *Information & Software Technology*, Vol. 52, No. 11, pp. 1242–1257, 2010.

14. Tosun, A., B. Turhan and A. B. Bener, “Feature Weighting Heuristics for Analogy-Based Effort Estimation Models”, *Expert Systems with Applications*, Vol. 36, No. 7, pp. 10325–10333, 2009.

15. Yang, B., H. Hu and L. Jia, “A Study of Uncertainty in Software Cost and Its Impact on Optimal Software Release Time”, *IEEE Transactions on Software Engineering*, Vol. 34, No. 6, pp. 813–825, 2008.

16. Ho, J. W., C. C. Fang and Y. S. Huang, “The Determination of Optimal Software Release Times at Different Confidence Levels with Consideration of Learning Effects”, *Software Testing Verification and Reliability*, Vol. 18, No. 4, pp. 221–249, 2008.

17. *IEEE Standard Dictionary of Measures to Produce Reliable Software*,

http://ieeexplore.ieee.org/servlet/opac?punumber=2752, accessed at February 2012.

18. Eusgeld, I., F. C. Freiling and R. Reussner (Editors), *Dependability Metrics: Advanced Lectures*, Springer-Verlag, Berlin, Heidelberg, 2008.

19. Adrion, W. R., M. A. Branstad and J. C. Cherniavsky, “Validation, Verification, and Testing of Computer Software”, *ACM Computing Survey*, Vol. 14, No. 2, pp. 159–192, 1982.

20. Shull, F., V. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero and M. Zelkowitz, “What We Have Learned About Fighting Defects”, *Proceedings of the 8th International Symposium on Software Metrics*, METRICS ’02, pp. 249–258, IEEE Computer Society, Washington, DC, USA, 2002.

21. Wohlin, C., A. Aurum, H. Petersson, F. Shull and M. Ciolkowski, “Software Inspection Benchmarking - A Qualitative and Quantitative Comparative Opportunity”, *Proceedings of the 8th International Symposium on Software Metrics*, METRICS ’02, p. 118, IEEE Computer Society, Washington, DC, USA, 2002.

22. Fagan, M., “Software pioneers”, chap. Design and Code Inspections to Reduce Errors in Program Development, pp. 575–607, Springer-Verlag New York, Inc., New York, NY, USA, 2002.

23. Nagappan, N., T. Ball and B. Murphy, “Using Historical In-Process and Product Metrics for Early Estimation of Software Failures”, *ISSRE 2006*, pp. 62–74, 2006.

24. Ostrand, T. J., E. J. Weyuker and R. M. Bell, “Predicting the Location and Number of Faults in Large Software Systems”, *IEEE Transactions on Software Engineering*, Vol. 31, No. 4, pp. 340–355, 2005.

25. Fenton, N. E. and M. Neil, “A Critique of Software Defect Prediction Models”,

IEEE Transactions on Software Engineering, Vol. 25, No. 5, pp. 675–689, 1999.

26. *Nasa/Wvu IV&V Facility, Metrics Data Program*, `http://mdp.ivv.nasa.gov`, accessed at September 2010.

27. Nagappan, N., E. Maximilien, T. Bhat and L. Williams, "Realizing Quality Improvement Through Test Driven Development: Results and Experiences of Four Industrial Teams", *Empirical Software Engineering*, Vol. 13, No. 3, pp. 289–302, 2008.

28. Kocaguneli, E., A. Tosun, A. Bener, B. Turhan and B. Caglayan, "Prest: An Intelligent Software Metrics Extraction, Analysis and Defect Prediction Tool", *Proceedings of SEKE*, pp. 637–642, 2009.

29. Misirli, A. T., B. Caglayan, A. V. Miranskyy, A. Bener and N. Ruffolo, "Different Strokes for Different Folks: A Case Study on Software Metrics for Different Defect Categories", *Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics*, WETSoM '11, pp. 45–51, ACM, New York, NY, USA, 2011.

30. Menzies, T., B. Turhan, A. Bener, G. Gay, B. Cukic and Y. Jiang, "Implications of Ceiling Effects in Defect Predictors", *Proceedings of the 4th international workshop on Predictor models in software engineering*, PROMISE '08, pp. 47–54, ACM, New York, NY, USA, 2008.

31. Nagappan, N. and B. Thomas, "Use of Relative Code Churn Measures to Predict System Defect Density", *Proceedings of the International Conference on Software Engineering*, pp. 15–21, 2005.

32. Caglayan, B., A. Bener and S. Koch, "Merits of Using Repository Metrics in Defect Prediction for Open Source Projects", *Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, FLOSS '09, pp. 31–36, IEEE Computer Society, Washington, DC, USA, 2009.

33. Nagappan, N. and T. Ball, "Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study", *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, p. 364373, IEEE Computer Society, 2007.

34. Zimmermann, T. and N. Nagappan, "Predicting Defects Using Network Analysis on Dependency Graphs", *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pp. 531–540, ACM, New York, NY, USA, 2008.

35. Tosun, A., B. Turhan and A. Bener, "Validation of Network Measures as Indicators of Defective Modules in Software Systems", *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, PROMISE '09, pp. 1–9, ACM, New York, NY, USA, 2009.

36. Shin, Y., R. Bell, T. Ostrand and E. Weyuker, "Does Calling Structure Information Improve the Accuracy of Fault Prediction?", *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pp. 61–70, IEEE Computer Society, Washington, DC, USA, 2009.

37. Meneely, A., L. Williams, W. Snipes and J. Osborne, "Predicting Failures with Developer Networks and Social Network Analysis", *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pp. 13–23, ACM, New York, NY, USA, 2008.

38. Jiang, Y., B. Cukic and T. Menzies, "Fault Prediction Using Early Lifecycle Data", *Proceedings of the The 18th IEEE International Symposium on Software Reliability*, ISSRE '07, pp. 237–246, IEEE Computer Society, Washington, DC, USA, 2007.

39. Kpodjedo, S., F. Ricca, P. Galinier, Y.-G. Guéhéneuc and G. Antoniol, "Design Evolution Metrics for Defect Prediction in Object Oriented Systems", *Empirical Software Engineering*, Vol. 16, No. 1, pp. 141–175, 2011.

40. Weyuker, E., 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, pp. 539–559, 2008.

41. Tosun, A., *Defect Prediction in Embedded Software Systems: Cascading Naive Bayes Algorithm with Cross- vs. Within Company Data*, M.S. Thesis, Department of Computer Engineering, Bogazici University, Turkey, 2008.

42. Zimmermann, T., A. Zeller, P. Weissgerber and S. Diehl, "Mining Version Histories to Guide Software Changes", *IEEE Transactions on Software Engineering*, Vol. 31, No. 6, pp. 429–445, 2005.

43. Turhan, B., T. Menzies, A. B. Bener and J. Di Stefano, "On the Relative Value of Cross-Company and Within-Company Data for Defect Prediction", *Empirical Software Engineering*, Vol. 14, No. 5, pp. 540–578, 2009.

44. Hall, T., S. Beecham, D. Bowes, D. Gray and S. Counsell, "A Systematic Literature Review on Fault Prediction Performance in Software Engineering", *IEEE Transactions on Software Engineering*, pp. 1–31, 2011.

45. Fenton, N., M. Neil, W. Marsh, P. Hearty, L. Radliński and P. Krause, "On the Effectiveness of Early Life Cycle Defect Prediction with Bayesian Nets", *Empirical Software Engineering*, Vol. 13, No. 5, pp. 499–537, 2008.

46. Fenton, N. and M. Neil, *Combining Evidence in Risk Analysis Using Bayesian Networks*, Tech. Rep. W0707-01, Agena, 2004.

47. Fenton, N. E., M. Neil and J. G. Caballero, "Using Ranked Nodes to Model Qualitative Judgments in Bayesian Networks", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 19, No. 10, pp. 1420–1432, 2007.

48. Fenton, N., W. Marsh, M. Neil, P. Cates, S. Forey and M. Tailor, "Making Resource Decisions for Software Projects", *Proceedings of the 26th International*

Conference on Software Engineering, ICSE '04, pp. 397–406, IEEE Computer Society, Washington, DC, USA, 2004.

49. Courtois, P. J., N. Fenton, B. Littlewood, M. Neil, L. Strigini and D. Wright, *Examination of Bayesian Belief Network for Safety Assessment of Nuclear Computer-Based Systems*, Tech. Rep. DeVa ESPRIT Pro, City University, Centre for Software Reliability, 1998.

50. Bai, C. G., Q. P. Hu, M. Xie and S. H. Ng, "Software Failure Prediction Based on a Markov Bayesian Network Model", *Journal of Systems and Software*, Vol. 74, No. 3, pp. 275–282, 2005.

51. Pérez-Miñana, E. and J. J. Gras, "Improving Fault Prediction Using Bayesian Networks for the Development of Embedded Software Applications: Research Articles", *Software Testing Verification and Reliability*, Vol. 16, No. 3, pp. 157–174, 2006.

52. Radliński, L., N. Fenton, M. Neil and D. Marquez, "Improved Decision-Making for Software Managers Using Bayesian Networks", *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications*, SEA '07, pp. 13–19, ACTA Press, Anaheim, CA, USA, 2007.

53. Baah, G. K., A. Gray and M. J. Harrold, "On-line Anomaly Detection of Deployed Software: A Statistical Machine Learning Approach", *Proceedings of the 3rd international workshop on Software quality assurance*, SOQUA '06, pp. 70–77, ACM, New York, NY, USA, 2006.

54. Fineman, M., N. Fenton and L. Radlinski, "Modelling Project Trade-Off Using Bayesian Networks", *International Conference on Computational Intelligence and Software Engineering*, pp. 1–4, 2009.

55. Marquez, D., M. Neil and N. Fenton, "Improved Reliability Modeling Using Bayesian Networks and Dynamic Discretization", *Reliability Engineering & Sys-*

tem Safety, Vol. 95, No. 4, pp. 412–425, 2010.

56. Schulz, T., L. Radliński, T. Gorges and W. Rosenstiel, “Defect Cost Flow Model: A Bayesian Network for Predicting Defect Correction Effort”, *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, No. 16 in PROMISE ’10, pp. 1–11, ACM, New York, NY, USA, 2010.

57. Wagner, S., “A Bayesian Network Approach to Assess and Predict Software Quality Using Activity-Based Quality Models”, *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, PROMISE ’09, pp. 6:1–6:9, ACM, New York, NY, USA, 2009.

58. Mendes, E. and N. Mosley, “Bayesian Network Models for Web Effort Prediction: A Comparative Study”, *IEEE Transactions on Software Engineering*, Vol. 34, No. 6, pp. 723–737, 2008.

59. Heckerman, D., “Learning in Graphical Models”, chap. A Tutorial on Learning with Bayesian Networks, pp. 301–354, MIT Press, Cambridge, MA, USA, 1999.

60. J. Pai, G. and J. Bechta Dugan, “Empirical Analysis of Software Fault Content and Fault Proneness Using Bayesian Methods”, *IEEE Transactions on Software Engineering*, Vol. 33, No. 10, pp. 675–686, 2007.

61. Wiegerinck, W., B. Kappen and W. Burgers, “Bayesian Networks for Expert Systems: Theory and Practical Applications”, Vol. 281, pp. 547–578, 2010.

62. Fayyad, M. U. and B. K. Irani, “Multi-Interval Discretization of Continuous-Valued Attributes for Classification Learning”, *Proceedings of the International Joint Conference on Uncertainty in AI*, pp. 1022–1029, 1993.

63. Onisko, A., M. Druzdzel and H. Wasyluk, “A Bayesian Network Model for Diagnosis of Liver Disorders”, *Proceedings of the Eleventh Conference on Biocybernetics and Biomedical Engineering*, pp. 842–846, 1999.

64. Antal, P., H. Verrelst, D. Timmerman, S. Van Huffel, B. de Moor and I. Vergote, "Bayesian Networks in Ovarian Cancer Diagnosis: Potentials and Limitations", *Proceedings of the 13th IEEE Symposium on Computer-Based Medical Systems (CBMS'00)*, CBMS '00, p. 103, IEEE Computer Society, Washington, DC, USA, 2000.

65. *Hugin Expert, Advanced Decision Support using Bayesian Networks and Influence Diagrams*, `http://www.hugin.com,` accessed at January 2010.

66. *Netica*, `http://www.norsys.com/netica.html,` accessed at March 2010.

67. Murphy, K., "Software for Graphical Models: A Review", *Software Highlight*, Vol. 14, pp. 1–3, 2007.

68. Ogunyemi, O., J. R. Clarke and B. Webber, "Using Bayesian Networks for Diagnostic Reasoning in Penetrating Injury Assessment", *Proceedings of the 13th IEEE Symposium on Computer-Based Medical Systems (CBMS'00)*, CBMS '00, p. 115, IEEE Computer Society, Washington, DC, USA, 2000.

69. Reiz, B. and L. C., "Tree-Like Bayesian Network Classifiers for Surgery Survival Chance Prediction", *International Journal of Computers, Communications & Control*, Vol. 3, pp. 470–474, 2008.

70. Meloni, A., A. Ripoli, V. Positano and L. Landini, "Mutual Information Preconditioning Improves Structure Learning of Bayesian Networks from Medical Databases", *Transactions on Information Technology in Biomedicine*, Vol. 13, No. 6, pp. 984–989, 2009.

71. Yuan, C. and M. J. Druzdzel, "Importance Sampling for General Hybrid Bayesian Networks", *Journal of Machine Learning Research - Proceedings Track*, Vol. 2, pp. 652–659, 2007.

72. Friedman, N., M. Goldszmidt and T. J. Lee, "Bayesian Network Classification

with Continuous Attributes: Getting the Best of Both Discretization and Parametric Fitting", *Proceedings of the Fifteenth International Conference on Machine Learning*, ICML '98, pp. 179–187, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

73. Davies, S. and A. Moore, *Mix-nets: Factored Mixtures of Gaussians in Bayesian Networks with Mixed Continuous and Discrete Variables*, Research paper, School of Computer Science, Carnegie Mellon University, 2000.

74. Larraaga, P., C. M. H. Kuijpers, R. H. Murga and Y. Yurramendi, "Learning Bayesian Network Structures by Searching For the Best Ordering With Genetic Algorithms", *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 26, pp. 487–493, 1996.

75. Margaritis, D., "Distribution-Free Learning of Bayesian Network Structure in Continuous Domains", *Proceedings of the 20th national conference on Artificial intelligence - Volume 2*, AAAI'05, pp. 825–830, AAAI Press, 2005.

76. Lagerstrm, R., P. Johnson, D. Hk and J. Knig, "Software Change Project Cost Estimation - A Bayesian Network and A Method for Expert Elicitation", *Proceedings of International Workshop on Software Quality and Maintainability Proceedings*, 2009.

77. Xiaoxu, W., W. Chaoying and M. Lin, "Software Project Schedule Variance Prediction Using Bayesian Network", *Advanced Management Science (ICAMS), 2010 IEEE International Conference on*, Vol. 2, pp. 26 –30, 2010.

78. Torkar, R., M. Awan, A. Alvi and W. Afzal, "Predicting Software Test Effort in Iterative Development Using a Dynamic Bayesian Network", *Proceedings of 21st IEEE International Symposium on Software Reliability Engineering*, 2010.

79. Neil, M., N. Fenton and L. Nielson, "Building Large-Scale Bayesian Networks", *Knowledge Engineering Review*, Vol. 15, No. 3, pp. 257–284, 2000.

80. Ciolkowski, M., O. Laitenberger, S. Vegas and S. Biffl, “Practical Experiences in the Design and Conduct of Surveys in Empirical Software Engineering”, *Empirical Methods and Studies in Software Engineering*, pp. 104–128, 2003.

81. Boehm, B. W., *Software Engineering Economics*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edn., 1981.

82. Ropponen, J. and K. Lyytinen, “Components of Software Development Risk: How to Address Them? A Project Manager Survey”, *IEEE Transactions on Software Engineering*, Vol. 26, No. 2, pp. 98–112, 2000.

83. Zimmermann, T., “Changes and Bugs - Mining and Predicting Development Activities”, *Proceedings of 25th IEEE International Conference on Software Maintenance*, pp. 443–446, 2009.

84. Ostrand, T. and E. Weyuker, “The Distribution of Faults in a Large Industrial Software System”, *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA ’02, pp. 55–64, ACM, New York, NY, USA, 2002.

85. Seifert, T. I. and B. Paech, “Exploring the Relationship of a File History and Its Fault-Proneness: An Empirical Method and Its Application to Open Source Programs”, *Information & Software Technology*, Vol. 52, No. 5, pp. 539–558, 2010.

86. Kan, S. H., J. Parrish and D. Manlove, “In-process Metrics for Software Testing”, *IBM Systems Journal*, Vol. 40, No. 1, pp. 220–241, 2001.

87. Talby, D., O. Hazzan, Y. Dubinsky and A. Keren, “Agile Software Testing in a Large-Scale Project”, *IEEE Software*, Vol. 23, No. 4, pp. 30–37, 2006.

88. Little, R. and D. Rubin, *Statistical Analysis with Missing Data*, John Wiley & Sons, New Jersey, 2002.

89. Cemgil, A. T., *Bayesian Statistics and Machine Learning, Graphical Models Lecture 2*, `http://www.cmpe.boun.edu.tr/courses/cmpe58K/fall2011/`, accessed at January 2012.

90. Jensen, F., *An Introduction to Bayesian Networks*, University College London Press, UK, 1996.

91. *Agena: Bayesian Network and Simulation Software for Risk Analysis and Decision Support*, `http://www.agenarisk.com`, accessed at December 2011.

92. *Bayes Net Toolbox for Matlab*, `http://code.google.com/p/bnt/`, accessed at December 2011.

93. Cemgil, A. T., *Monte Carlo Methods: The Gibbs Sampler, Applications, Lecture 4*, `http://www.cmpe.boun.edu.tr/courses/cmpe58n/spring2011/`, accessed at January 2012.

94. Szekely, B. and M. Rizzo, "Brownian Distance Covariance", *The Annals of Applied Statistics*, Vol. 3, No. 4, pp. 1236–1265, 2009.

95. Fisher, N. I. and P. Switzer, "Chi-Plots for Assessing Dependence", *Biometrika*, Vol. 72, No. 2, pp. 253–265, 1985.

96. Kitchenham, B., L. Pickard, S. G. MacDonell and M. J. Shepperd, "What Accuracy Statistics Really Measure", *IEEE Proceedings - Software*, Vol. 148, No. 3, pp. 81–85, 2001.

97. Chulani, S., B. Boehm and B. Steece, "Bayesian Analysis of Empirical Software Engineering Cost Models", *IEEE Transactions on Software Engineering*, Vol. 25, No. 4, pp. 573 –583, 1999.

98. Port, D. and M. Korte, "Comparative Studies of the Model Evaluation Criterions MMRE and PRED in Software Cost Estimation Research", *Proceedings of ESEM*,

pp. 51–61, 2008.

99. Spiegelhalter, D. J., N. G. Best, B. P. Carlin and A. van der Linde, “Bayesian Measures of Model Complexity and Fit (with Discussion)”, *Journal of the Royal Statistical Society, Series B*, Vol. 64, pp. 583–639(57), 2002.

100. Turhan, B., “On the Dataset Shift Problem in Software Engineering Prediction Models”, *Empirical Software Engineering*, Vol. 17, No. 1-2, pp. 62–74, 2012.

101. Bird, C., N. Nagappan, B. Murphy, H. Gall and P. Devanbu, “Don’t Touch My Code!: Examining the Effects of Ownership on Software Quality”, *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE ’11, pp. 4–14, ACM, New York, NY, USA, 2011.

102. Lilliefors, H. W., “On the Kolmogorov-Smirnov Test for the Exponential Distribution with Mean Unknown”, *Journal of the American Statistical Association*, Vol. 64, No. 325, p. 387389, 1969.

103. Genest, C. and A. Favre, “Everything You Always Wanted to Know about Copula Modeling, but Were Afraid to Ask”, *Journal of Hydrologic Engineering*, Vol. 12, No. 4, pp. 347–368, 2007.

104. Genest, C. and J. C. Boies, “Detecting Dependence with Kendall Plots”, *The American Statistician*, Vol. 57, No. 4, 2003.

105. Nelsen, R., *An Introduction to Copulas*, Springer Series in Statistics, Springer, New York, 2006.

106. Fisher, N. I. and P. Switzer, “Graphical Assessment of Dependence: Is a Picture Worth 100 Tests?”, *The American Statistician*, Vol. 55, No. 3, pp. 233–239, 2001.

107. Chambers, J., *Graphical Methods for Data Analysis*, Chapman & Hall statistics

series, Wadsworth International Group, 1983.

108. Lunn, D. J., A. Thomas, N. Best and D. Spiegelhalter, "WinBUGS: A Bayesian Modelling Framework: Concepts, Structure, and Extensibility", *Statistics and Computing*, Vol. 10, No. 4, pp. 325–337, 2000.

109. Peter, M. L., *Bayesian Statistics: An Introduction*, New York, Willey, 2004.

110. Kriegel, H., P. Krger and A. Zimek, "Outlier Detection Techniques", *Tutorial at 16th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD 2010)*, 2010.

111. Vanselst, M. and P. Jolicoeur, "A Solution to the Effect of Sample-Size on Outlier Elimination", *Quarterly Journal of Experimental Psychology Section AHuman Experimental Psychology*, Vol. 47, No. 3, pp. 631–650, 1994.

112. Chandola, V., A. Banerjee and V. Kumar, "Anomaly Detection: A Survey", *ACM Computing Surveys*, Vol. 41, No. 3, p. 158, 2009.

113. Box, P. E. G. and R. D. Cox, "An Analysis of Transformations", *Journal of the Royal Statistical Society. Series B (Methodological)*, Vol. 26, No. 2, pp. 211–252, 1964.

114. Cook, T. and D. Campbell, *Quasi-Experimentation: Design & Analysis Issues for Field Settings*, Rand McNally College, 1979.

115. Campbell, D. and J. Stanley, *Experimental and Quasi-Experimental Designs for Research*, Rand McNally, 1973.

116. Shull, F., J. Singer and D. I. Sjøberg, *Guide to Advanced Empirical Software Engineering*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.