APPLYING SOCIAL NETWORK ANALYSIS TO SOFTWARE FAULT-PRONENESS PREDICTION

by

Yihao Li

APPROVED BY SUPERVISORY COMMITTEE:

____

W. Eric Wong, Chair

____

Kamil Sarac

____

Ding-Zhu Du

____

Lawrence Chung

Copyright 2017

Yihao Li

All Rights Reserved

I would like to dedicate this to my loving family – my wife, my father and my mother – each of whom has been instrumental in making this possible.

APPLYING SOCIAL NETWORK ANALYSIS TO SOFTWARE FAULT-PRONENESS PREDICTION

by

YIHAO LI, BS, MS

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

SOFTWARE ENGINEERING

THE UNIVERSITY OF TEXAS AT DALLAS

August 2017

ACKNOWLEDGEMENTS

First and foremost, I want to gratefully acknowledge my PhD supervisor, Professor W. Eric Wong, for his guidance and support over the last six years. It has been a privilege to work with him, and I have learned an immeasurable amount from him. I doubt I would have worked as hard on my research, had Dr. Wong not led by example and worked just as hard, if not harder still. Words cannot convey the respect and admiration I have for him, and the gratitude that I feel for all that he has done for me. Thank you Dr. Wong – for everything.

I would also like to convey my sincerest thanks to my committee members: Professor Kamil Sarac, Professor Ding-Zhu Du, and Professor Lawrence Chung, for being very patient with me, and for their comments and suggestions which have been invaluable towards improving the quality of my research work. I thank them individually, and in different capacities as well. I consider myself very lucky to have such wonderful Professors on my committee.

Special thanks go to Professor Kang Zhang for his tremendous support on my dissertation preparation.

I am very grateful to my colleagues, Ruizhi Gao, Shou-Yu Lee, Xuelin Li, Linghuan Hu, Xiaomin Wei, Dr. Vidroha Debroy for their continuous support and encouragement. It's been a privilege working with them.

August 2017

APPLYING SOCIAL NEWTORK ANALYSIS TO SOFTWARE

FAULT-PRONENESS PREDICTION

Yihao Li, PhD
The University of Texas at Dallas, 2017

Supervising Professor: W. Eric Wong

Due to the rapid pace of software development, end-users now anticipate a seemingly limitless expansion of capabilities from their software. As a result, software systems are becoming increasingly complex and more susceptible to failures. Although software fault localization techniques are becoming more comprehensive, it is still expensive to precisely locate, let alone fix, bugs in a program. Hence, fault-proneness prediction can be applied beforehand to alleviate the cost of program debugging by identifying software modules which are likely to contain faults.

Meanwhile, social network analysis (SNA) has been frequently applied in software engineering to depict relations between (1) modules, (2) developers, or (3) modules and developers. Previous studies have shown that these relations have been used to build social networks to predict fault-prone modules and the results are encouraging.

Although these networks are useful for fault-proneness prediction, they are built either by a single relation or by a pair of relations aforementioned. In addition, these networks appear to

neglect an essential factor: developer quality. After all, it is developers who make mistakes and introduce faults into software.

We therefore, propose Tri-Relation Network (TRN), a weighted social network that integrates all three types of relations. Four network node centrality metrics are correspondingly derived from TRN. Moreover, a calibration mechanism for edge weights on TRN is explored as well. Case studies reveal that TRN holds great promise in the context of fault-proneness prediction and the effectiveness improves further after applying the calibration mechanism on current TRN.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS……………………………………………………………………v

ABSTRACT………………………………………………………………………………...vi

LIST OF FIGURES………………………………………………………………………ix

LIST OF TABLES……………………………………………………………………...x

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 SOFTWARE FAULT-PRONENESS PREDICTION 7

2.1 Methods and Techniques to Build Fault-Proneness Prediction Models 7

2.2 Related Issues of Fault-Proneness Prediction 17

CHAPTER 3 SOCIAL NETWORK ANALYSIS 25

3.1 Social Network Analysis in Software Engineering 25

3.2 Network Code Centrality Metrics 26

3.3 Social Networks for Fault-Proneness Prediction 27

CHAPTER 4 THE PROPOSED TRI-RELATION NEWTORK (TRN).......................................31

4.1 The Construction of TRN 31

4.2 Four Network Centrality Metrics and Other Metrics...33

4.3 Evaluation of TRN...36

CHAPTER 5 EDGE WEIGHT CALIBRATION MECHANISM FOR TRN..............................47

5.1 The Proposed CaTRN 47

5.2 Results for R3 49

CHAPTER 6 CONCLUSION AND FUTURE WORK 54

REFERENCES 58

BIOGRAPHICAL SKETCH 83

CURRICULUM VITAE..84

LIST OF FIGURES

Figure 1. Taxonomy of software metrics 2

Figure 2. A DCN with 2 developers and 3 modules 27

Figure 3. A MDN with 4 modules 28

Figure 4. A STN with 2 developers and 5 modules 29

Figure 5. A DN with 4 developers 30

Figure 6. A TRN with 3 developers and 4 modules 32

Figure 7. CaTRN with 3 developers and 4 modules 49

LIST OF TABLES

Table 1. Ten commonly used software metrics 34

Table 2. Notations relevant to metrics, networks, and prediction models 35

Table 3. Network node centrality metrics derived from TRN, DCN, MDN, STN, and DN 35

Table 4. Summary of subject programs used 37

Table 5. Correlation Analysis Using Spearman Rank Correlation Coefficient for Camel 1.4.0 where correlation is significant at the 0.05 level (2-tailed) 41

Table 6. Correlation Analysis Using Spearman Rank Correlation Coefficient for Flume 1.5.0 where correlation is significant at the 0.05 level (2-tailed) 42

Table 7. Correlation Analysis Using Spearman Rank Correlation Coefficient for Tika 1.6 where correlation is significant at the 0.05 level (2-tailed) 42

Table 8. Confidence that $M_{TRN\text{-}Cen}$ is more correlated with the number of bugs than the corresponding $M_{DCN\text{-}Cen}$, $M_{MDN\text{-}Cen}$, $M_{STN\text{-}Cen}$, and $M_{DN\text{-}Cen}$ 42

Table 9. Prediction effectiveness of $\Phi(M_{TRN})$, $\Phi(M_{DCN})$, $\Phi(M_{MDN})$, $\Phi(M_{STN})$, $\Phi(M_{DN})$, and $\Phi(M_{CO})$ with respect to average Recall and average FPR for Camel 1.4.0 44

Table 10. Prediction effectiveness of $\Phi(M_{TRN})$, $\Phi(M_{DCN})$, $\Phi(M_{MDN})$, $\Phi(M_{STN})$, $\Phi(M_{DN})$, and $\Phi(M_{CO})$ with respect to average Recall and average FPR for Flume 1.5.0 45

Table 11. Prediction effectiveness of $\Phi(M_{TRN})$, $\Phi(M_{DCN})$, $\Phi(M_{MDN})$, $\Phi(M_{STN})$, $\Phi(M_{DN})$, and $\Phi(M_{CO})$ with respect to average Recall and average FPR for Tika 1.6 45

Table 12. Confidence that $\Phi(M_{TRN})$ is more effective than $\Phi(M_{DCN})$, $\Phi(M_{MDN})$, $\Phi(M_{STN})$, $\Phi(M_{DN})$, and $\Phi(M_{CO})$ with respect to Recall and FPR 46

Table 13. Spearman Rank Correlation Coefficient for Camel 1.4.0 where correlation is significant at the 0.05 level (2-tailed) 50

Table 14. Spearman Rank Correlation Coefficient for Flume 1.5.0 where correlation is significant at the 0.05 level (2-tailed) 50

Table 15. Spearman Rank Correlation Coefficient for Tika 1.6 where correlation is significant at the 0.05 level (2-tailed) 50

Table 16. Confidence that $M_{CaX\text{-}Cen}$ is more correlated to the number of bugs than the corresponding $M_{X\text{-}Cen}$ 51

Table 17. Prediction effectiveness of $\Phi(M_{CaTRN})$, $\Phi(M_{CaDCN})$, $\Phi(M_{CaMDN})$, $\Phi(M_{CaSTN})$, and $\Phi(M_{CaDN})$ with respect to average Recall and average FPR for Camel 1.4.0 52

Table 18. Prediction effectiveness of $\Phi(M_{CaTRN})$, $\Phi(M_{CaDCN})$, $\Phi(M_{CaMDN})$, $\Phi(M_{CaSTN})$, and $\Phi(M_{CaDN})$ with respect to average Recall and average FPR for Flume 1.5.0 52

Table 19. Prediction effectiveness of $\Phi(M_{CaTRN})$, $\Phi(M_{CaDCN})$, $\Phi(M_{CaMDN})$, $\Phi(M_{CaSTN})$, and $\Phi(M_{CaDN})$ with respect to average Recall and average FPR for Tika 1.6......................52

Table 20. Confidence that $\Phi(M_{CaX})$ is more effective than the corresponding $\Phi(M_X)$..................53

CHAPTER 1

INTRODUCTION

Due to the rapid pace of software development, end-users now anticipate a seemingly limitless expansion of capabilities from their software. Software quality and reliability are becoming critical issues for software systems[1] and have gained much attention in research. It has been reported that, in an attempt to reduce the number of delivered faults, most companies spend between 50-80% of their software development effort on testing [60]. The project failure rate is a major concern in software project development, and it is increasing as software projects grow in complexity. Galorath [81] reports research by the Standish Group revealing that the success rates of software projects in 1994, 1996, 1998, 2000, 2004, and 2009 were 16%, 27%, 26%, 28%, 29%, and 32% respectively, implying that software project development entails a number of risks which need to be taken into consideration. Although software fault localization techniques are becoming more comprehensive [246], [247], [250], [251], [252], it is still expensive to precisely locate, let alone fix, bugs in a program. As a result, implementing fault-proneness prediction as early as possible has a significant impact on controlling software budgets and software quality [118]. To achieve this, software metrics are collected and used to build fault-proneness prediction models to identify fault-prone modules so that potential problems can be further prevented and a basis for planning and controlling software testing and maintenance can be created.

---

[1] In this dissertation, we use "program", "application" and "software system" interchangeably. We use "method" and "function" interchangeably. We use "bugs", "faults", and "defects" interchangeably. We also use "metric", "feature", and "attribute" interchangeably. A software system consists of various modules. A software module can have different representations depending on how a software system is described on a particular architecture level. For example, it can represent a single function, a single class, or a single file.

The purpose of using software metrics is to obtain knowledge of target software programs and to get insight during software development life cycle. A number of studies on metrics development, investigation, and implementation have been conducted to help better assess software quality as well as identify module fault-proneness over the past twenty years [20], [62], [68], [87], [96], [110], [122], [151], [152], [154], [156], [172], [183], [194], [202], [204], [205], [211], [214], [222], [248], [249], [255], [256], [268], etc.

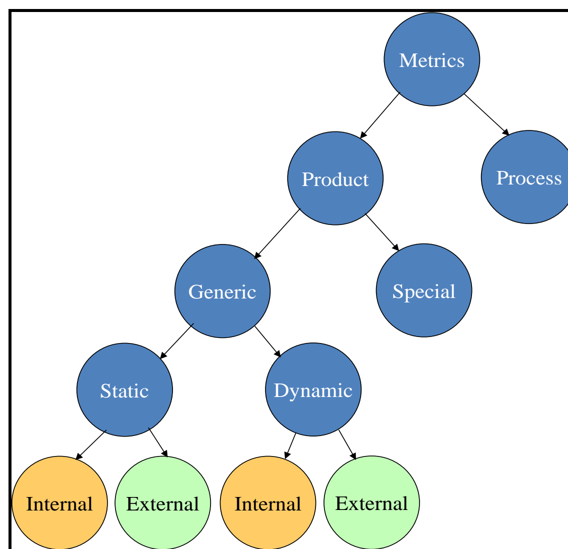

Figure 1. Taxonomy of software metrics

As shown in Figure 1, metrics that can be used for model construction are generally categorized into two types: product metrics and process metrics [262]. Product metrics are used to analyze software from different aspects. For instance, object-oriented metrics can reflect the impact of using object-oriented mechanisms such as inheritance, coupling, cohesion, polymorphism and encapsulation [6]-[13], [26], [40]-[44], [58], and [59]). Product metrics can be divided into generic metrics and special metrics. Generic metrics consist of two types of metrics: static metrics and dynamic metrics. Static metrics are used to measure static attributes of software and can be collected at compile time. Dynamic metrics can only be collected and

evaluated at run-time (e.g., dynamic coupling metrics [17], run-time cohesion metrics [162], and code coverage-based metrics [165] and [254]). Both static metrics and dynamic metrics can be further divided into internal and external metrics. Internal metrics focus the internal structure of a software module (e.g., Lines of Code [181], McCabe's cyclomatic complexity [157], Halstead complexity [98], and Wong's internal design metrics [253]). External metrics emphasize the interactions between a software module and the rest of the system (e.g., information flow [104] and Wong's external design metrics [253]). Special metrics are used to measure the attributes that are unique to the target software (e.g., temporal complexity metrics for concurrent and real-time systems [16]). Process metrics are used to describe how a software system is built and operated such as code churn [174], developer-based metrics [192], [243], [257], organizational metrics [34], [171], [179], etc. Generally speaking, the measurement of product metrics is repeatable, objective, and can be automated while for process metrics different people may have different assessments which are more subjective compared to product metrics. It is also more challenging to automate the collection of process metrics.

However, the effectiveness of these product and process metrics in identifying fault-prone modules is inconsistent or contradictory on different projects. For example, there are many studies investigating the relationship between lines of code and number of faults. Zhang [263] found that 20% of the largest modules were responsible for 51%~63% of the defects on three versions of Eclipse. Ostrand et al. [191] reported that on average 73% and 74% of the faults contained in the 20% of the files that were the largest in terms of the number of lines of code on two large industrial systems. However, a replicated study by Grbac et al. [92] reported that size metrics are not useful as predictors of fault proneness based on the data from five consecutive

releases of a large software system in the telecommunications domain. Fenton and Ohlsson [78] used two versions of a telecommunication software and found that 20% of the modules were responsible for nearly 60% of the faults but contained just 30% of the code. Koru et al. [142], [143] reported that fault proneness increases with size, but at a slower rate. This makes smaller modules proportionally more problematic compared with larger ones. Radjenovic´ et al. [202] observed a strong correlation between the LOC and fault proneness in pre-release and small dataset, while in post-release and large dataset, only a weak association was found. McCabe's cyclomatic complexity was a good indicator of software fault proneness in [11], [108], [161], [162], [177], [188], and [272] but not in [10], [87], and [258]. Radjenovic´ et al. [202] reported that cyclomatic complexity was fairly effective in large post-release environments using OO languages but not in small pre-release environments with procedural languages. One potential explanation for this could be that in large data sets, modules are more complex than in small data sets. Also, the modules used in OO programming languages (usually class) are generally bigger than modules used in procedural languages (usually method) and may, therefore, be more complex. Studies in [125], [160], and [162] indicated that Halstead's metrics were good indicators of fault proneness but not in [174] and [260]. According to [160], Halstead's metrics were not as good as McCabe's cyclomatic complexity or LOC. Studies in [77], [91], [267], and [271] indicated that complexity metrics are useful in identifying fault-prone modules but are not as well as some OO metrics or process metrics. CK metrics are the most frequently used OO metrics. However, not all CK metrics performed equally well. Authors of [5], [20], [43], [44], [71], [95], [114], [185], [195], [211], [222], [262], and [266] found that CBO, WMC, and RFC in the CK metrics suite are good indicators of software fault proneness, while the usefulness of

LCOM is mediocre [20], [95], [195], [222] compared to these three metrics. Surprisingly, DIT and NOC were reported as untrustworthy in [75], [88], [95], [185], [195], [211], [222], [227], and [266]. Process metrics were found to be useful in predicting post-release faults in [19], [65], [79], [91], [99], [113], [136], [149], [156], [172], [177], and [179]. Radjenovic´ et al. [202] suggested that industry practitioners looking for effective and reliable process metrics should consider code churn, the number of changes, the age of a module and the change set size metrics. Researchers looking for poorly investigated areas may consider the number of past faults and the change set size metrics. Once again, there are no decisive conclusions regarding the effectiveness of these metrics in predicting fault-prone software modules.

A potential solution is to explore software metrics that harbor features from the perspectives of both software product and process. In other words, we should consider metrics that depict how software modules and developers are related within/to each as a whole. Therefore, interaction is the key. Meanwhile, social networks and its analysis has been used wildly to understand software engineering activities modeling how people communicate and collaborate among others, which provides critical information for a software development project [28]-[32], [35], [56], [115], [147], [158], [159], [180], [200], [232], [245], [270], [271]. It investigates not only the social organization of the work but also the technical information infrastructures.

In this dissertation, we propose applying social network analysis in fault-proneness prediction by first constructing a network, namely Tri-relation Network (TRN), which depicts the social/ technical relationship between (1) software modules, (2) developers, and (3) modules and developers. Later, four centrality network node metrics are derived from TRN and used to

build our fault-proneness prediction model. Additionally, an edge weight calibration mechanism is proposed and applied on TRN to construct CaTRN to further enhance the effectiveness of fault-proneness prediction.

The remainder of this dissertation is organized as follows. First CHAPTER 2 presents a literature survey on software fault-proneness prediction providing a brief overview of methods and techniques that have been used to conduct fault-proneness prediction activities. CHAPTER 3 generally introduces the use of social network analysis in software engineering, and some related work that has been in fault-proneness prediction. CHAPTER 4 describes our proposed TRN along with an empirical evaluation, followed by CHAPTER 5 which discusses the notion of our edge weight calibration mechanism and the proposed CaTRN based on this mechanism followed by empirical evaluation. Finally, we conclude with a summarization of ongoing work and future work in CHAPTER 6.

CHAPTER 2

SOFTWARE FAULT-PRONENESS PREDICTION

We provide a brief overview of methods and techniques that have been used for predicting software fault-proneness.

2.1 Methods and Techniques to Build Fault-Proneness Prediction Models

We summarize the methods and techniques that have been used to build fault-proneness prediction models and categorize them into different group.

2.1.1 Statistics-based Techniques

Regression analysis is a statistical method used to describe the nature of the relationship between a dependent variable and independent variables [38]. A regression model developed by an estimated regression equation is used to predict the value of the dependent variable given values for the independent variables. For a linear regression model, the dependent variable y, denoted as $\hat{y}$, can be expressed as a linear combination of n independent variables, x_1, x_2, x_3, x_4,…, x_n, in the form of Equation 1.

$$\hat{y} = \alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 + ... + \alpha_n x_n + \varepsilon \quad (1)$$

In this equation, ε is the error term, and parameters α_1, α_2,…, α_n can be estimated from the observation data. R^2, also known as the coefficient of determination, is a commonly used statistic to evaluate the model fit of a regression equation; that is, to show how good the independent variables are at predicting the dependent variable. The value of R^2 ranges from 0 to 1 – the higher the value the better the fit between the regression model and the data. The most

general definition of the coefficient of determination is shown in Equation 2. SS_{tot} is the total sum of squares, and SS_{err} is the residual sum of squares.

$$R^2 = 1 - \frac{SS_{err}}{SS_{tot}} \tag{2}$$

With respect to software fault-proneness prediction, Hassan [99] built linear regression models using data from the second and third years of the source control repository to predict the number of faults in the fourth and fifth years of a software project. Tomaszewski et al. [230] built linear regression models with a number of software metrics as independent variables to predict fault density and the number of faults on a telecommunication system. Basili and Hutchens [21] employed linear regression analysis to examine the relationship between software metrics and program changes on 19 compiler projects. Goel and Singh [88] built linear regression models to predict the number of faults using a public data set, KC1, from NASA Metrics Data Program. In addition, Li and Henry [153] applied linear regression analysis to examine the relationship between software metrics and software maintenance effort (in terms of the number of lines changed per class) on two commercial software systems.

Logistic regression is another frequently used regression modeling method in which the dependent variable takes on one of two different values. By dividing the software module classes into two categories, fault-prone (FP) and non-fault-prone (NFP), logistic regression can be used for building software fault-proneness prediction models. A logistic regression model is shown in Equation 3.

$$\pi(x_1, x_2, ..., x_n) = \frac{e^{\beta_0 + \beta_1 x_1 + ... + \beta_n x_n}}{1 + e^{\beta_0 + \beta_1 x_1 + ... + \beta_n x_n}} \tag{3}$$

In this equation, the x_is are the software metrics included as independent variables, the β_is are the regression coefficients corresponding to x_is, and π is the probability that a module is faulty. Studies [18], [25], [42], [44], [69], [71], [88], [112], [149], [166], [173], [185], [216], [220], [221], [237], [266], and [267] have been conducted to apply logistic regression to predict fault-prone software modules as well as to determine the usefulness of a software metric as an indicator of software module fault-proneness. Rather than logistic regression models, Gao et al in [82] and [132] presented an empirical study on count models (such as a Poisson regression model and a negative binomial regression model) to predicate the number of faults in a file on two Windows-based embedded software applications.

Couto et al. [63] proposed a model by using the Granger causality test to evaluate the causality between source code metrics and the occurrence of defects. The proposed model predicts defects introduced in the source code via triggering alarms whenever a change performed to a class reproduces similar variations that cause defects in the past.

2.1.2 Machine Learning-based Techniques

Artificial neural networks (ANNs) are based on learning algorithms inspired by biological neural networks and can be used to build software fault-proneness prediction models. Prior to training, the user must decide on the network topology such as the number of units in the input layer, the number of hidden layers, the number of units in each hidden layer, and the number of units in the output layer. A neural network is then applied to a given data set of training tuples (software modules with known classes represented by a set of software metrics with known values). An iterative learning procedure is later followed to minimize the difference between the predicted value (the training output) and actual target value (the actual output). With respect to software

fault-proneness prediction, Khoshgoftaar and Szabo [137] built a Backpropagation (BP) network, Thwin and Quah [228] built a BP network and a General Regression Neural Network to predict the number of defects in a software module. Bezerra et al. [30] built a Radial Basis Function (RBF) network to predict the probability of defect existed in a software module. Researchers of 0, [29], [57], [89], [106]-[109], [119], [123], [128], [133], [134], [146], [189], [198], and [264] built different neural networks to predict fault-prone software modules. Zhou et al. [269] explored another five neural network-based techniques (Batch Gradient without momentum, Batch Gradient with momentum, Variable Learning Rate without momentum, Variable Learning Rate with momentum, and Resilient Backpropagation) for the modeling of fault severity in a software module. Generally speaking, the accuracy of the resulting trained network relies on the network design, which is a trial-and-error process. If the accuracy is not acceptable, then it is necessary to repeat the training process with a different network topology.

A large software system may consist of hundreds of software modules, each of which possesses quite different characteristics. It is sometimes not the particular value but specific ranges that significantly identify the qualitative difference. In this way, high-defect modules with different characteristics for different partitions can be identified, and different actions can be carried out to correct the problems. Tree-based modeling handles data partitions and relation analysis. The model construction involves recursively partitioning the data set into smaller subsets using split conditions defined on independent variables, resulting in increasing homogeneity of the dependent variable. Each subset of data associated with a tree node is uniquely described by the path of associated split conditions. The results presented in such forms are natural to the decision process and, consequently, are easy to interpret and use. Researchers

of [27], [86], [94], [95], [116], [124], [127], [129], [131], [135], [138], and [266] applied different algorithms to building decision trees [201] in order to identify fault-prone software modules and predict the number of bugs in a software module. Knab et al. [141] predicted defect densities in source code files with different decision tree learners. Shin et al. [218] built decision tree learning models to predict vulnerable files in Mozilla Firefox and Linux kernel. Tian and Troster [229] constructed tree-based models to identify and characterize high-defect software modules in both a legacy and a new software system, while Koru and Tian [144], [145] built their tree-based models to analyze the relationship among high-change software modules, high-defect software modules, and modules with high measurement values. Similar to tree-based models, Briand et al. in [39] introduced a pattern matching technique called optimal set reduction (OSR) to help predict fault-prone software modules. However, patterns defined by OSR are not mutually exclusive and can be used in conjunction to identify risky areas.

The Naïve Bayesian classifier and the Bayesian Belief Network are two classification algorithms based on Bayes' theorem that can also be used for software fault-proneness prediction. For the Naïve Bayesian classifier, let D be a training set of tuples with associated class labels. Each tuple in D and in the testing set is in the form of n-dimensional attribute vector, $X = (x_1, x_2, \ldots, x_n)$, where x_i (i=1, 2,…, n) is a value of an attribute A_i that characterizes a software module. There are two classes, fault-prone (C_1), and non-fault-prone (C_2) in a Naïve Bayesian classifier. Given a tuple X in the testing set, the classifier will predict that the X belongs to a class (e.g., fault-prone) having the higher posterior probability conditioned on X than the other class (e.g., non-fault-prone) [53], [116], [155], [160], [189], [233], [236]. When using the Naïve Bayesian classifier, it is assumed that the value of an attribute of a given class is independent of

the values of other attributes. If dependencies exist between attributes, a Bayesian Belief Network (BBN) is used instead. Bayesian Belief Networks are also known as Belief Networks, Bayesian Networks, or Probabilistic Networks. A belief network consists of two components: a directed acyclic graph and a set of conditional probability tables (CPTs). Both may be given in advance or inferred from the data. Rather than returning a single class label, the trained network can return a probability distribution that gives the probability of each class. Researchers of [14], [67], [76], [77], [86], [187], and [195] built different BBN models to estimate fault content in software modules of different applications.

Authors of [93] and [196] proposed using Dempster-Shafer (D-S) belief networks to predict fault-prone modules. The structure of the D-S network does not represent causal relationship as in the Bayesian network. Instead, it represents implication relationship among the nodes. Each node represents an individual attribute that characterizes a software module. Each arc in the graph signifies the existence of a direct implication rule between two adjacent nodes. The class label of a software module is dependent on the values of all attributes that influence it.

Support vector machine (SVM) is another machine learning-based technique used for data classification. Given a set of training tuples, it uses nonlinear mapping to transform these tuples into a higher dimension. Within this new dimension, it searches for the linear optimal separating hyperplane (the decision boundary), known as the maximum marginal hyperplane (MMH), to separate the new tuples of one class from another (fault-prone from non-fault-prone) with minimum classification error [72], [86], [89], [139], and [258].

Mizuno and his colleagues [102], [168]-[170] proposed fault-prone filtering where a software module is considered as an e-mail message and all software modules are categorized as

either spam emails (fault-prone modules) or regular emails (non-fault-prone modules). After learning of existing FP and NFP modules, a new module will be classified into as either FP or NFP by applying a spam filter.

Discriminant analysis is often applied as another useful statistical analysis technique. With regard to software fault-proneness prediction, it classifies software modules characterized by a set of software metrics into two groups: fault-prone or non-fault-prone [25], [84], [106], [126], [130], [175], [177], [186], [226]. This classification is made by building a discriminant function from the training dataset to assign data points (software modules characterized by a set of software metrics) in the testing set to either the fault-prone or the non-fault-prone group.

2.1.3 Miscellaneous Techniques

Pandey and Goyal [197] built a fuzzy inference system (FIS) to learn the relationship between used software metrics and software module fault-proneness and later was used to predict fault-proneness degree of an unlabeled module (e.g., low, medium, or high fault-prone). Rodriguez et al. [207] proposed a descriptive approach using a subgroup discovery (SD) algorithm for software fault-proneness prediction which allows characterizing defective software modules with rules that can describe thresholds and relationships between software metrics. Jing et al. in [120] introduced a supervised dictionary learning-based technique for software fault classification and prediction. Czibula et al. [64] applied relational association rules to define different types of relationships between the values of used software metrics and fault-proneness of software modules. Such relationships will be used as useful patterns to predict if a new software module is fault-prone or not.

Seliya and Khoshgoftaar [210] proposed a constraint-based semi-supervised clustering scheme that uses the K-Means clustering method to cluster software modules into different clusters based on several modules whose labels (fault-prone or non-fault-prone) are already decided by an expert. Yang et al. [261] reported that using an affinity propagation clustering algorithm can achieve a better performance in clustering software modules into fault-prone or non-fault-prone cluster than using K-Means based on case studies of two real-world software projects. In addition, Catal et al. [51], [52] and Bishnu et al. [37] proposed a clustering and metrics thresholds-based approach for software fault-proneness prediction without previous fault data. First, a clustering technique is applied to cluster software modules. Then, the representative software module of the cluster is checked against a set of predefined software metrics thresholds. A cluster is predicted as fault-prone if at least one software metric of the representative module is higher than the specified threshold value of that software metric. Last but not least, Scanniello et al. [209] applied a graph clustering algorithm, BorderFlow, to group software modules based on their dependencies (e.g., there is a dependency between class c_j and class c_i if there is any class instantiation, method invocation, or field access of c_j in the body of the class c_i or vice versa). The idea behind BorderFlow is to maximize the number of dependencies to the nodes within the same cluster while minimizing the number of dependencies to the nodes in a different cluster. Data from one cluster is used to train the model, which is later used to predict fault-proneness for software modules in other clusters.

He et al. [103] built software fault-proneness prediction models from the selected data of other projects when historical data for the target project is not available and concluded that cross-project software fault-proneness prediction was feasible as long as it involved a careful selection

of training data. Rahman et al. [203] also reported that cross-project prediction performance is no worse than within-project prediction performance using project data from the GIT and JIRA repositories. However, Zimmermann et al. [272] ran 622 cross-project predictions and found that only 3.4% actually worked. Herbold [105] reported that although the quality of cross-project-based software fault-proneness prediction can improve with an appropriate training data selection strategy, it still cannot compete with the quality of within-project software fault-proneness prediction. Cotroneo et al. [61] also found that cross-project software fault-proneness prediction does not achieve acceptable performance in most cases compared to the results of within-project software fault-proneness prediction due to the different characteristics of the analyzed projects. Turhan et al. [235] combined within- and cross-project data from 73 versions of 41 projects to build software fault-proneness prediction models and found that collecting data from other projects is not feasible in terms of practical performance improvement when there is already an established within-project software fault predictor using full project history. However, when there is limited project history (e.g., early phases of development), mixed project predictions (e.g., using only 10% of available within project data) may perform as well as full within-project models. A major issue in cross-project software fault-proneness prediction is how to find the right training data in a software repository. Nagappan et al. in [178] conducted an empirical study of the post-release defect history of five Microsoft software systems. The results indicate that predictors are accurate only when obtained from the same or similar projects. Regarding this issue, Burak et al. [234] and Peters et al. [199] proposed two filters respectively to select appropriate training data. The former selects instances using KNN to measure the similarity (with Euclidean distance) between test instances and training data set instances. While

the latter finds nearest test instances for each training data set instance and then a test instance chooses the closest training data instance as a candidate for the filtered training data set while rejecting all the others. Bell et al. [23] reported that models could not be applied to other systems while Denaro and Pezzè [70] reported that good predictive performance only across homogenous applications. Nagappan et al. [178] also found that models are only accurate when trained on the same or similar systems. However, other studies report more promising transferability. Mizuno and Hirata [167] used 28 versions of 8 projects to conduct experiments of the cross-project and intra-project prediction. They found that intra-project prediction had better precision than cross-project. In contrast, recall is better using cross-project prediction.

Zhong et al. [265] proposed an unsupervised learning method for expert-based software quality estimation. It first clusters hundreds of software modules into a small number of groups. The representative of each group is then inspected by a software quality expert, who labels each cluster as either fault-prone or non-fault-prone based on his/her domain knowledge as well as other data statistics. They found that the proposed method could achieve comparable prediction accuracies with other fault-proneness classifiers. Tomaszewski et al. [231] invited eleven experts involved in the development of two telecommunication systems to perform fault-proneness prediction in the latest releases of both systems. On the contrary, they reported that the prediction accuracy of fault-proneness prediction models using historical fault data outperformed expert estimations. The biggest disadvantage of expert-based fault-proneness prediction is that its performance is largely dependent on the size of the dataset since estimating the fault-proneness of all modules in a system can be both time- and labor-consuming especially when the system is large and complex. As reported in [231], invited experts were confident when it comes to

ranking the first couple of components. Beyond a certain number of components, they admitted they would put remaining components in a random order. In addition, it is quite possible that experts do not agree with each other on determining the fault-proneness of same modules which makes such prediction even more challenging.

Wong et al. [254] proposed a static and a dynamic risk model to identify high-risk code in a program. The static model is constructed by using software metrics related to the static structure of the code, such as the number of c-uses, p-uses, definitions, decisions and function calls. The dynamic model utilizes additional dynamic test coverage of the code such as decision, c-use and p-use coverage to calibrate the metric values used in the model. Users have the choice to select either a summation scheme, which builds a risk model by using the sum of the selected software metrics, or a product scheme, which uses the product of the selected software metrics. Additionally, the two model schemes allow users to assign different weighting factors to the selected metrics based on their understanding of the target software product. These two models are able to identify the fault-proneness at a very fine granularity level (e.g., a basic block), which differs from other studies that identify fault-proneness at a method or class level.

2.2 Related Issues of Fault-Proneness Prediction

In this section, discussions on the related issues of fault-proneness prediction are presented.

2.2.1 Fault Severity in Fault-Proneness Prediction

Although some faults are more important to be identified than others, few studies consider fault severity when into evaluating the fault-proneness prediction performance. Shatnawi and Li [211] reported a model that is able to predict high and medium severity faults (these levels of severity

are based on those reported in Bugzilla by Eclipse developers). Lamkanfi et al. [148], Singh et al. [222], and Zhou and Leung [266] had also investigated fault severity when conducting their fault-proneness prediction studies. This lack of studies that consider severity is probably because, although acknowledged to be important, severity is considered a difficult concept to measure. For example, Menzies et al. [163] said that severity is too vague to reliably investigate. Nikora and Munson [182] said that without a widely agreed definition of severity we could not reason about it, and Ostrand et al. [190] stated that severity levels are highly subjective and can be inaccurate and inconsistent. These problems of how to measure and collect reliable severity data may have limited the usefulness of fault-proneness prediction models.

2.2.2 Challenges on Data Quality

Data quality challenges exist in both literature and applied research when investigating software fault-proneness prediction. In published research, papers may demonstrate inconsistencies that make the conclusions difficult to interpret. Fenton and Martin [77] pointed that in some cases definitions of defects, errors, faults, and failures differs widely between studies; defect rate, defect density, and failure rate are used almost interchangeably. It can also be difficult to tell whether a model is predicting discovered defects or residual defects. In other cases, strong theoretical or practical justification is required in order to remove data points during analysis, but it is sometimes difficult to tell whether any such points have been removed.

Similarly, some papers may acknowledge removed data points but provide no explanation. Moreover, they argued that the use of “averaged” data in analysis rather than the original data prejudices many studies. When using averages instead of raw data, the amount of information available to test the conjecture under study is reduced. However, any corresponding

conclusions may be weakened. In an extreme case, the conclusion reached by using averages can be completely contrary to that of using raw data. For example, the study in [22] used average fault density of grouped data to suggest a trend that was not supported by the raw data.

Additionally, inconsistency regarding the size of the same program reported in different studies posts another challenge on data quality. A typical example is the LOC of project Xerces, an XML parser, reported in [7] and [206]. The former reported that the LOCs of Xerces 1.2 and 1.3 are 60,032 and 64,503 respectively while the LOCs provided by the latter are 159,254 and 167,095 (which means the sizes reported in [206] are twice larger than those reported in [7] even though the subject programs are identical). The discrepancy is even more alarming considering the fact that these two papers were published in the same conference and in the same year. Jiang et al. [118] investigated the impact that the size of the training and test dataset has on the accuracy of fault-proneness predictions. The experiment results indicated that if the study is performed using the data from a small project, or an unreasonable distribution of modules between model training and testing, high variance will indicate that the results are unstable, prompting the need for better software engineering data or improved experimental design. Gray et al. [90] reported that the bulk of fault-proneness prediction experiments based on the NASA Metrics Data Program datasets might have led to erroneous findings due to repeated data points that potentially caused substantial amounts of training and testing data to be identical. Once data collection is complete, it is important to conduct data cleansing to remove noise (e.g., inaccurate/incorrect data points or data points with missing values) and outliers (e.g., data points with faulty class label if all the metrics are within their corresponding thresholds levels) to make the data sets suitable for machine learning.

By and large, it is important to anticipate and address any data quality challenges, as they may compromise the outcome of a given fault-proneness prediction model. To aid in identifying potential obstacles, researchers in [47] and [48] proposed a taxonomy of data quality challenges in empirical software engineering and reviewed techniques for assessing and addressing quality issues. And Hall et al. [97] proposed a set of criteria that can be served as guidance to conduct a reliable study on software fault-proneness prediction.

2.2.3 No Single Set of Software Metrics or Single Model Fits All Projects

Nagappan et al. [178] reported that different subsets of complexity metrics relate to faults in different projects and that no single set of metrics fits all projects. Arisholm et al. [9] found that the choice of fault-proneness modeling technique has limited impact on the resulting classification accuracy or cost-effectiveness. There are large differences between the individual software metric sets and their cost-effectiveness, and the best model is highly dependent on the criteria used to evaluate and compare the models. Similarly, Shepperd et al. [213] concluded that it is not the choice of fault proneness classifier but the people who conduct the research have a significant impact on the final fault-proneness prediction performance. Moreover, Shepperd and Kadoda [215] argued that the accuracy of a specific software fault-proneness prediction model is very much dependent on the attributes of the dataset. The study results observed in [223] also indicate that different learning schemes should be selected for different datasets, since no scheme dominates. Likewise, experiments conducted by Wong et al. [253] suggest that there is no particular metric having a clear advantage over the others in determining which function is fault-laden. As a result, it is better to ask which model is the best in a specified context rather than asking which one is the best in general [215]. More reliable research procedures need to be

developed before confident conclusions can be reached for comparative studies of software fault-proneness prediction models [176].

2.2.4 New and Modified Modules Seem More Fault-Prone

In [99], Hassan built software fault-proneness prediction models using change complexity metrics on six open-source software systems. The results indicate that a complex code change process negatively affects the software system. The more complex changes to a file, the higher the chance the file will contain faults. Tomaszewski et al. [230] also found that the modified code can be an important source of faults. They built fault density prediction models using data from two large telecommunication systems produced by Ericsson and reported that the faults found in the modified code accounted for more than 60% of the total number of faults found in the systems. Kim et al. [140] found that if a method was changed or had been added recently, it tended to introduce faults soon. Illes-Seifert and Paech [113] used nine open source Java projects to investigate the relationship between history characteristics of files and their defect count. They reported that the number of changes, the number of distinct authors performing changes to a file, and the file's age are positively related to a file's defect count. Another study conducted by Basili and Perricone [22] on a software project containing 90,000 lines of code indicate that: (1) both modified and new software modules have a high percentage of interface errors, (2) new software modules have a higher percentage of control errors, (3) modified software modules contain a high percentage of commission errors, and (4) modified software modules appeared to be more vulnerable to errors due to misunderstanding of the specifications, which is consistent with Endres's result [74]. Shin et al. in [217] made three observations: (1) the more method invocations in a file, the more fault-prone the file might be, (2) new files that have less chance to

be tested are more fault-prone than existing files, and (3) new and modified methods are more likely to be faulty than other methods. Bell et al. in [24] found that most faults in the system analyzed occurred in files that had been changed in the prior release. Similarly, Shen et al. [212] also found that new and modified modules are more fault-prone than unchanged modules from a previous release of the same product. In [100], Hassan et al. highlighted that some of the most susceptible subsystems (directories) that may have a fault such as Most Frequently Modified (MFM), Most Recently Modified (MRM), Most Frequently Fixed (MFF), and Most Recently Fixed (MRF).

2.2.5 The Pareto Law Universally Applies

In a telecommunication system produced by Ericsson Telecom AB, Fenton and Ohlsson [77] found that 20% of software modules contain approximately 60% of the total faults found. Similarly, Tomaszewski et al. in [230] conducted their research based on data from two different telecommunication systems developed by Ericsson and found that 60% of faults can be found in 20% of the codes. Kim et al. studied seven open source projects in [140] and found that 10% of the source code files account for 73%-95% of the total faults. Bell et al. [23] performed an empirical study of software fault-proneness prediction for an industrial voice response system and found that 20% of the files contained 75% of the faults. English et al. [75] carried out an empirical study for an open-source software project based on the information retrieved from the CVS repository and the Bugzilla database and found that Pareto's Law holds: 82% of faults occurred in 20% of the classes. Carrozza et al. [50] investigated how to predict the location(s) of Mandelbugs (faults that are triggered by complex conditions, such as interaction with hardware and other software, and timing or ordering of events) on an industrial software system developed

for the military domain and found that, on average, 60%-80% of Mandelbugs could be identified in the top 20% failure-prone components. Researchers of [15], [89], [190], [189], [241]-[244] also found that the faults contained in the 20% of the files roughly contains 60%~94% of the faults.

2.2.6 Findings Related to Code Management

Binkley et al. [33] derived three language-processing-based measures from the comments and identifiers in the source code to do software fault-proneness prediction. The first measure calculates the percentage of natural language used in a program's identifiers. The second measure calculates percentage of identifiers violating a variant of Deissenboeck and Pizka's rules [66] for concise and consistent identifiers. The third, referred to as the QALP score, is named after a project aimed at providing quality assessment using language processing [148]. Aman [7], [8] investigates the impact of comment statements on the fault-proneness of software modules. Their results showed that software modules containing comments are more likely to be faulty than non-commented software modules. Abede et al. in [4] found that Lexicon Bad Smells (LBS), anomalies that reduce the quality of identifier names, contribute to the identification of fault-prone modules. Good quality identifiers contribute to the understanding of the software and hence make it less susceptible to the introduction of faults. In [225], Taba et al. observed that files containing antipatterns have higher bug density than other files through a case study on multiple versions of Eclipse and ArgoUML. Antipatterns are specific design and implementation styles that can identify poor system designs. They are usually introduced in software systems due to the lack of knowledge or experience of developers when solving a particular problem. In [36], Bird et al. point out that ownership can also have a strong relationship to software defects.

Changes made by minor contributors, as well as components with low ownership, should be reviewed more carefully. Minor contributors should communicate with experienced developers or major contributors on the desired changes before taking any further actions.

CHAPTER 3

SOCIAL NETWORK ANALYSIS

3.1 Social Network Analysis in Software Engineering

Social network analysis (SNA) investigates social structures using network and graph theory [193]. It characterizes networked structures in terms of nodes and the edges that connect them. In the field of software engineering, SNA has been adopted as both a useful approach to understand software engineering activities. A social network models how people communicate and collaborate among others, which provides critical information for a software development project. It investigates not only the social organization of the work but also the technical information infrastructures. For example, Ghosh [85] reports that many open source projects at SourceForge are organized as social networks. Xu et al. [259] classify people working an open source project at SourceForge into project leader, core developer, co-developer, and active user. Ohira et al. [184] apply social network analysis and collaborative filtering to identify experts across different projects. Howison et al. also [111] use data collected from SourceForge to investigate how the social structures in projects are changing. In addition, Sarma et al. [208] have developed a tool that visualizes many different aspects of development artifacts. Cataldo et al. [54] propose the socio-technical congruence framework which examines the relationship between the structure of technical and work dependencies and the impact of dependencies on software development productivity. Their study indicates that when developers' coordination patterns are congruent with their coordination needs, the resolution time of modification requests is significantly reduced. In addition, logical dependency is a more accurate representation of product dependency affecting the development effort than call and data dependencies. Later, they

report that the logical dependency explains most of the variance in fault-proneness [55]. Many researchers have conducted different studies to analyze the interaction between developer and software module. Bird et al. [36] point out that ownership can have a strong relationship to software defects. Changes made by minor contributors, as well as components with low ownership, should be reviewed more carefully. Minor contributors should communicate with experienced developers or major contributors regarding the desired changes before taking further action. Ell [73]and Simpson [219] use the Failure Index (FI) to determine the failure-inducing possibility of developer pairs in developer social networks. Nagappan et al. [179] propose eight organizational metrics that quantify the complexity of a software development organization (e.g., the absolute number of unique engineers who have touched a binary and are still employed by/have left the company) to identify fault-prone binaries in Windows Vista.

3.2 Network Code Centrality Metrics

Network node centrality metrics stem from social network theory and are used to quantify the location of a node to the rest of the network [45]. There are three types of network node centrality [239]: (1) degree centrality, (2) closeness centrality, and (3) betweenness centrality. Degree centrality metrics are computed based on the number of edges that a node has [49]. The more edges a node has, the more central is the node. Closeness centrality emphasizes the distance of a node to other nodes in the network [80]. Betweenness centrality denotes the extent to which information flows through a node to get from one node to another [101]. The more information flows through a node the higher is its betweenness centrality.

3.3 Social Networks for Fault-Proneness Prediction

This section gives a brief overview of some existing social networks that have been proposed for fault-proneness prediction.

3.3.1 Developer Contribution Network (DCN)

In [200], Pinzger et al. represent developer contributions with a developer-module network that is called a contribution network. Case studies based on data collected from Windows Vista indicate that centrality metrics derived from the contribution network are good indicators of the number of post-release faults.

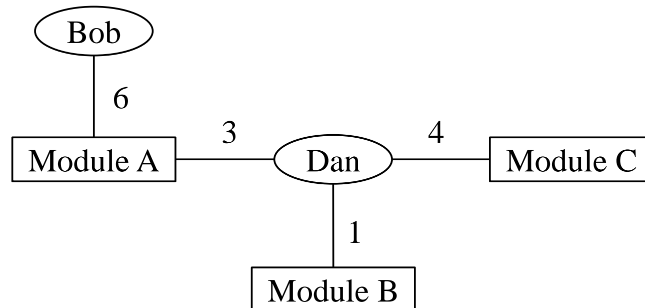

Figure 2. A DCN with 2 developers and 3 modules

A contribution network is an undirected graph G that is formally defined as $G = (D, N, E)$. D and N are the two sets of vertices that represent the two partitions of the graph, and E is a set of edges between vertices $E \subseteq \{(d, n) \mid d \in D \wedge n \in N\}$. D represents the set of developers and N the set of software modules. An edge $e \in E$ denotes a contribution of a developer $d \in D$ to a module $n \in N$. A contribution refers to a commit of a developer to a module. Edges are always between a developer and a module, and there are no self-loops (i.e., neither modules nor developers can contribute to themselves). Edge weights are used to denote the number of commits a developer

has made to a module. Figure 2 depicts a sample developer contribution network. Circles represent developers, rectangles represent software modules, and edges represent developer contributions to modules. For example, developer *Bob* has made 6 commits to module *A*. Developer *Dan* has made 3, 1, and 4 commits to Modules *A*, *B*, and *C*, respectively.

3.3.2 Module Dependency Network (MDN)

Zimmermann and Nagappan [270],[271] construct a network from dependency information for software modules in Windows Server 2003. They also find that social network analysis-based metrics derived from the dependency network are good indicators of the number of post-release faults and module fault-proneness, which is consistent with the results presented in [180], [200], and [232].

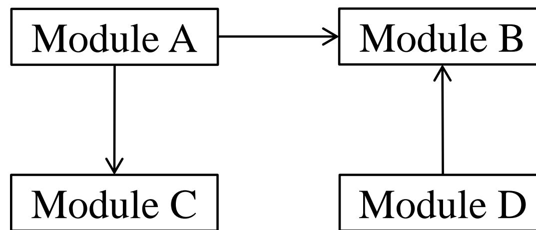

Figure 3. A MDN with 4 modules

Generally, a dependency network models the dependency relationships (e.g., call graphs, class inheritance, class coupling, etc.) between software modules within a software system. It is a directed graph that is formally defined as $G = (N, E)$ where N is the set of software modules and E is the set of directed edges such that $(n_1, n_2) \in E$ if Module n_1 has a dependency on Module n_2. Figure 3 shows a simple dependency network where rectangles represent software modules and directed edges represent module dependency relationships. For example, Module *A* has a

dependency on Modules *B* and *C* respectively. Module *A* and Module *D* are both dependent on Module *B*.

3.3.3 Socio-Technical Network (STN)

In [35], Bird et al. argue that the dependency relations and contribution history should be used together for fault-proneness prediction. They construct a socio-technical network by combining the developer contribution network and the module dependency network.

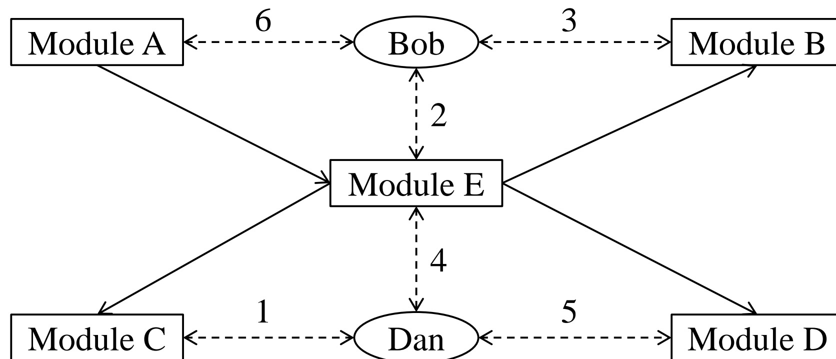

Figure 4. A STN with 2 developers and 5 modules

In the socio-technical network, there is a bidirectional dashed edge (denoted as the contribution edge) between a developer and a software module if the developer has made a commit to the module. The weight on the contribution edge is set as the number of commits from a developer to a module, and the weight of module dependencies is set to 1. Figure 4 shows a sample socio-technical network with 2 developers and 5 modules. For example, developer *Bob* has made 6, 2, and 3 commits to Modules *A*, *E*, and *B*, respectively. Module *E* has dependencies on Modules *B*, *C*, and *D*, respectively.

3.3.4 Developer Collaboration Network (DN)

Meneeley et al. [159] construct a developer collaboration network consisting solely of developers in which edges between developers are based on collaboration on common modules. The authors use social network analysis to assign values of metrics to developers. The value of a metric for a module is based on the values of the developers that contributed to that module (e.g., the sum of a metric for developers for a module).

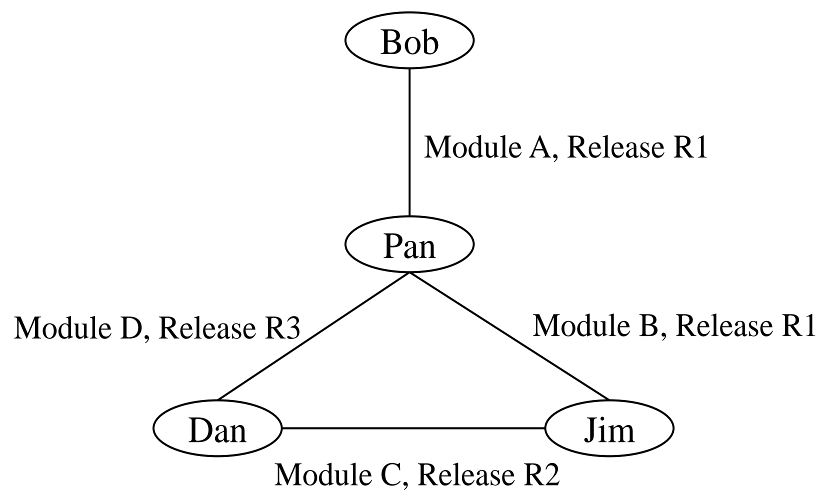

Figure 5. A DN with 4 developers

Figure 5 depicts a sample developer network with 4 developers. Circles represent developers and edges represent common files that two developers have both worked on in a particular release. For example, developers *Bob* and *Pan* have both worked on Module *A* during Release *R*1.

CHAPTER 4

THE PROPOSED TRI-RELATION NEWTORK (TRN)

Having overviewed four exiting social networks used for fault-proneness prediction in general, we are now ready to discuss the proposed TRN in this dissertation. We first present how TRN is constructed and the four network centrality metrics derived come TRN. Later, we move on to evaluate TRN, in terms of the correlation between its centrality metrics and the number of faults as well as the effectiveness of fault-proneness prediction model using these centrality metrics.

4.1 The Construction of TRN

The motivation behind TRN is that a network integrating developer contribution, module dependency, and developer collaboration can provide a more fully comprehensive insight into the interactions between developers and modules than the use of networks based on either a single or a paired relation. This insight is expected to ultimately enhance the effectiveness of software fault-proneness prediction.

In a TRN, there is a directed edge (denoted as the developer contribution) between a developer and a software module if the developer has made a commit to the module between two consecutive releases (e.g., between Release *R* and Release *R*+1). The weight on the contribution edge is set as the normalized[2] number of commits made from a developer to a module between Release *R* and Release *R*+1. Dependencies between modules are represented as directed dash-dot edges with arrows pointing to the modules upon which other modules depend. It is worth noting that we consider two types of dependency: functional dependency (e.g., a file calls another file)

[2] We apply the Min-Max approach for data normalization here.

and logical dependency (e.g., two files are modified in the same commit). We believe the use of both dependency types provides a more accurate representation of module dependencies affecting the development effort. The tool *Understand* from SciTools [238] is used to quantify the normalized dependencies between two modules. The weights for logical dependency between two modules are computed as the normalized number of times that these two modules are modified in the same commit. The resulting module dependency is computed as the sum of normalized functional dependency and normalized logical dependency.

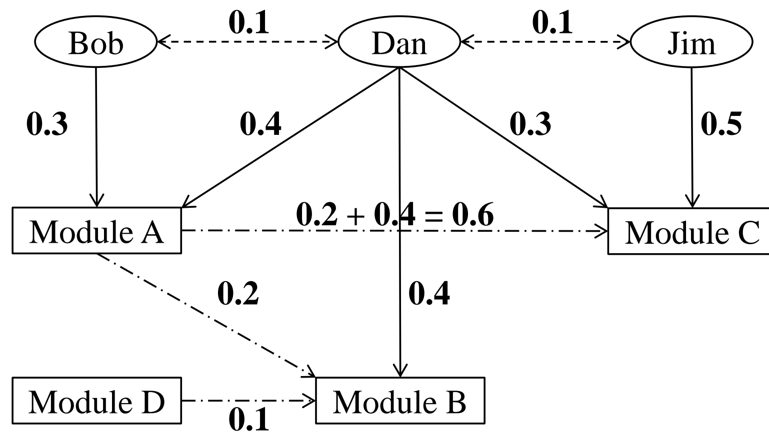

Figure 6. A TRN with 3 developers and 4 modules

In addition, there is a bidirectional dotted edge between one developer and another if these two developers have made at least one commit on the same module between release *R* and Release *R*+1. The weight on collaboration edge is computed as the normalized number of modules two developers have worked on together between Release *R* and Release *R*+1. Figure 6 presents a TRN with 3 developers and 4 modules. For example, the weight on the developer contribution edge *Bob*-to-Module *A* is 0.3. The module dependency edge Module *A*-to-Module *C* is 0.2 (normalized functional dependency) + 0.4 (normalized logical dependency) = 0.6. The developer collaboration edge *Bob*-to-*Dan* is 0.1.

4.2 Four Network Centrality Metrics and Other Metrics

Four centrality metrics, Freeman degree centrality (denoted as M_{FDC}), Bonacich's Power (denoted as M_{BP}), eigenvector of geodesic distances (denoted as M_{EGD}), Freeman node betweenness (denoted as M_{FNB}), which have been widely used in both SNA and fault-proneness prediction [35], [158], [179], [199], [231], [270], [271] are collected in this study. M_{FDC} is calculated as the number of direct edges a node has to its neighbors. M_{BP} is based on the adjacencies. It takes into account the connections of one's connections, in addition to one's own connections. M_{FDC} and M_{BP} focus on the number of developers and other modules it directly connects to, the impact of direct interactions on the module. More people working on the module, higher probability of introducing faults due to inconsistent coding style especially when these people have never worked/communicated with each other before. Due to the direct dependency relationship, more changes made on its neighbor modules higher probability that appropriate changes should be made on the module accordingly, thus more difficult to maintain the module. Closeness centrality emphasizes the distance of a node to other nodes in the network. In this paper, we use one such node distance measure: eigenvector of geodesic distances (denoted as M_{EGD}). M_{EGD} find the most central nodes (i.e. those with the smallest farness from others) in terms of the "global" or "overall" structure of the network, and to pay less attention to patterns that are more "local". Specifically, M_{EGD} applies factor analysis to identify "dimensions" of the distances among nodes. The location of each node with respect to each dimension is called an "eigenvalue", and the collection of such values is called the "eigenvector". Usually, the first dimension captures the "global" aspects of distances among nodes; second and further dimensions capture more specific and local sub-structures.

Betweenness centrality denotes the extent to which information flows through a node to get from one node to another. The more information flows through a node the higher is its betweenness centrality. For betweenness centrality, we use one such metric, Freeman node betweenness (denoted as M_{FNB}). It counts up how frequently each node falls in the geodesic paths between all pairs of nodes. M_{EGD} and M_{FNB} focus on the connection strength of the module to all modules and developers that it either directly or indirectly connects to. The closer the module to other modules and developers, the stronger the connection, the more likely the module can be affected by other modules and developers in a way. We use a tool, *Ucinet* [46], to compute and collect the values of these four centrality metrics.

Table 1. Ten commonly used software metrics

Metric	Description
M_{LOC}	Lines of code
M_{McCabe}	McCabe cyclomatic complexity
M_{LCOM}	Lack of cohesion in methods
M_{DIT}	Depth of inheritance tree
M_{CBO}	Coupling between object classes
M_{NOC}	Number of children
M_{RFC}	Response for a class
M_{WMC}	Weighted methods per class
M_{NC}	Number of commits
M_{ND}	Number of developers

Additionally, we introduce another ten software metrics, as shown in Table 1, that are commonly used for predicting software fault-proneness, including Lines of Code [181], McCabe Complexity [157], all six CK metrics [59], number of commits [177], and number of developers [192], all of which will be used later in our case studies. We use a tool, *Understand* [238], to compute and collect the values of these ten metrics.

Table 2. Notations relevant to metrics, networks, and prediction models

Notation	Description
X	a network
M	a generic software metric
Φ	a generic software fault-proneness prediction model
M_{Cen}	a generic network code centrality metric
$M_{X\text{-}Cen}$	a network code centrality metric derived from X
M_X	a set of four network code centrality metrics derived from X
M_{CO}	a set of ten software metrics that are commonly used for software fault-proneness prediction
$\Phi(M_X)$	a software fault-proneness prediction model using M_X
$\Phi(M_{CO})$	a software fault-proneness prediction model using M_{CO}

Table 3. Network node centrality metrics derived from TRN, DCN, MDN, STN, and DN

$M_{TRN\text{-}FDC}$	$M_{DCN\text{-}FDC}$	$M_{MDN\text{-}FDC}$	$M_{STN\text{-}FDC}$	$M_{DN\text{-}FDC}$
$M_{TRN\text{-}BP}$	$M_{DCN\text{-}BP}$	$M_{MDN\text{-}BP}$	$M_{STN\text{-}BP}$	$M_{DN\text{-}BP}$
$M_{TRN\text{-}EGD}$	$M_{DCN\text{-}EGD}$	$M_{MDN\text{-}EGD}$	$M_{STN\text{-}EGD}$	$M_{DN\text{-}EGD}$
$M_{TRN\text{-}FNB}$	$M_{DCN\text{-}FNB}$	$M_{MDN\text{-}FNB}$	$M_{STN\text{-}FNB}$	$M_{DN\text{-}FNB}$

For the sake of both simplicity and consistency, we use the notations provided in Table 2. For example, X represents a generic weighted network such as TRN, DCN, MDN, STN, or DN. M_{Cen} represents a generic network code centrality metric (e.g., M_{FDC}, M_{BP}, M_{EGD}, or M_{FNB}). $M_{X\text{-}Cen}$ represents a M_{Cen} derived from X. For example, $M_{TRN\text{-}FDC}$ represents the FDC network node centrality metric derived from a TRN. Meanwhile, all four network node centrality metrics derived from a TRN (i.e., $M_{TRN\text{-}FDC}$, $M_{TRN\text{-}BP}$, $M_{TRN\text{-}EGD}$, and $M_{TRN\text{-}FNB}$) can now be simplified to M_{TRN}. We use M_{CO} to denote a metric set that contains the ten software metrics described in Table 2. In addition, we use Φ (M_X) and Φ (M_{CO}) to denote a software fault-proneness prediction model using all four network code centrality metrics derived from X and a prediction model using the ten commonly used metrics, respectively. Since for each network we have four centrality metrics, accordingly, a total of 20 metrics can be derived as shown in Table 3.

4.3 Evaluation of TRN

We first examine the three research questions related to our study followed by a discussion of the software programs and the data analysis techniques used in our case studies. Results are presented at the end of this section.

4.3.1 Three Research Questions

In our study, we answer the following two research questions, which are thoroughly discussed later in this chapter:

R1: Are TRN-based centrality metrics important indicators for the number of post-released bugs in a file?

R2: Do TRN-based centrality metrics effectively improve software fault-proneness prediction models?

R3: Is there any potential enhancement that can be applied to TRN and other networks in order to further improve the effectiveness of fault-proneness prediction?

Answers to these three questions can help determine whether TRN-based centrality metrics are more powerful in building fault-proneness prediction models than not only DCN-, MDN-, STN-, or DN-based centrality metrics, but also software metrics that are commonly used for fault-proneness prediction. Moreover, valuable insights will be gained regarding the contributing factors that can be used to refine current networks in order to further enhance the prediction effectiveness.

4.3.2 Subject Program Studied

Our experiments use three Java programs, Camel 0, Flume [2], and Tika [3]. Camel is an open-source integration framework to define routing and mediation rules in a variety of domain-specific languages. Flume is a distributed service for collecting, aggregating, and moving log data from different sources to a centralized data store. Tika detects and extracts metadata and text from different file types such as PPT, XLS, and PDF.

Table 4. Summary of subject programs used

Project	Release	LOC	Files
Camel	1.3.0	110,113	1,245
Camel	1.4.0	114,621	1,488
Flume	1.4.0	92,437	507
Flume	1.5.0	99,127	547
Tika	1.5	83,502	472
Tika	1.6	87,180	507

Table 4 summarizes the information for these three programs used in our case studies. The columns, starting from the left, give project name, release version, lines of code (including blanks and comments), number of Java files, number of faulty Java files, and the number of faults. Each program contains two consecutive releases. The values of all metrics are collected at file level.

4.3.3 Experimental Methodology

In order to answer R1, the Spearman rank correlation coefficient [224] is used to measure the correlation between each metric (described in Table 5 through Table 7) and the number of post-released bugs for Camel 1.4.0, Flume 1.5.0, and Tika 1.6, respectively. The coefficient is

between +1 and −1, inclusive, in which +1 is total positive correlation, 0 is no correlation, and −1 is total negative correlation.

In order to answer R2, a data mining tool, Weka [240], is used to construct different fault-proneness prediction models. For each program, a total of six datasets are formed based on TRN, DCN, MDN, STN, and DN, as well as a dataset consisting of ten commonly used metrics (denoted as CO-based dataset) from two consecutive software releases (say Release 1 and Release 2). We use all data points collected from Release 1 as the training set and all data points collected from Release 2 as the validation set. Because of the class imbalance issue in the training set we apply SMOTE to oversample the minority class (i.e., classified as fault-prone) so that the size of fault-prone samples is equal to the size of samples that are non-fault-prone. We use BayesNet as our training algorithm. For example, in a TRN-based dataset, each data point is a labeled (i.e., fault-prone or non-fault-prone) file characterized by M_{TRN} (i.e., $M_{TRN\text{-}FDC}$, $M_{TRN\text{-}BP}$, $M_{TRN\text{-}EGD}$, and $M_{TRN\text{-}FNB}$). In a CO-based dataset, each data point is a labeled file characterized by M_{CO}. The same train-predict process is repeated 30 times. For example, we use $\Phi(M_{TRN})$ to represent a fault-proneness prediction model using M_{TRN}. As a result, for each program we have constructed 180 (i.e., 30×6) fault-proneness prediction models including 30 $\Phi(M_{TRN})$, 30 $\Phi(M_{DCN})$, 30 $\Phi(M_{MDN})$, 30 $\Phi(M_{STN})$, 30 $\Phi(M_{DN})$, and 30 $\Phi(M_{CO})$.

Two measures, recall (defined in Equation (4)) and false positive rate (defined in Equation (5)) are used to evaluate the fault-proneness prediction effectiveness of each model. In these two equations, TP (true positive) is the number of fault-prone modules that are correctly predicted, TN (true negative) is the number of non-fault-prone modules that are correctly predicted, FP (false positive) is the number of non-fault-prone modules that are predicted as

fault-prone, and FN (false negative) is the number of fault-prone modules that are incorrectly predicted as non-fault-prone.

$$\text{Recall} = \frac{\text{TP}}{\text{TP+FN}} \tag{4}$$

$$\text{False Positive Rate (FPR)} = \frac{\text{FP}}{\text{FP+TN}} \tag{5}$$

For discussion purposes, let us assume there are 100 software modules in which 30 are faulty and 70 are non-faulty. Assume also that a fault-proneness prediction model predicts 35 modules as fault-prone, 10 of which are actually non-faulty. This also implies that among the 65 modules predicted as non-fault-prone, 5 are actually fault-prone. Therefore, TP = 25 (25 faulty modules are correctly predicted as fault-prone), TN = 60 (60 non-faulty modules are correctly predicted as non-fault-prone), FP = 10 (ten non-faulty modules are incorrectly predicted as fault-prone), and FN = 5 (five faulty modules are incorrectly predicted as non-fault-prone). Based on Equations (1) and (2), the recall is 25/(25+5) ≈ 83.33% and the FPR is 10/(10+60) ≈ 14.29%. For two different fault-proneness prediction models Φ_1 and Φ_2, if Φ_1 has a higher recall than Φ_2, then it can be said that Φ_1 is more effective than Φ_2 with respect to recall. If Φ_1 has a lower FPR than Φ_2, then it can be said that Φ_1 is more effective than Φ_2 with respect to FPR.

For each program, we compute and compare the respective average recall and FPR of 30 $\Phi(M_{TRN})$, 30 $\Phi(M_{DCN})$, 30 $\Phi(M_{MDN})$, 30 $\Phi(M_{STN})$, 30 $\Phi(M_{DN})$, and 30 $\Phi(M_{CO})$. For example, regarding R2, if $\Phi(M_{TRN})$ has a higher average recall than $\Phi(M_{DCN})$, then $\Phi(M_{TRN})$ is more effective than $\Phi(M_{DCN})$ with respect to average recall.

In addition, we employ the paired Wilcoxon signed-rank test [38] to investigate R1 and R2. For example, regarding R1, we can make the following null hypothesis with respect to the computed Spearman rank correlation coefficient using $M_{TRN\text{-}Cen}$ and the same coefficient using $M_{DCN\text{-}Cen}$:

H_0: the computed Spearman rank correlation coefficient using $M_{TRN\text{-}Cen}$ is equal to or smaller than the computed Spearman rank correlation coefficient using $M_{DCN\text{-}Cen}$

If H_0 is rejected (i.e., the alternative hypothesis is accepted), then it implies that $M_{TRN\text{-}Cen}$ is more correlated with the number of post-released bugs than $M_{DCN\text{-}Cen}$.

Regarding R2, we can make the following null hypothesis with respect to the recall of $\Phi(M_{TRN})$ and $\Phi(M_{DCN})$:

H_0: $\Phi(M_{TRN})$ has equal or lower recall than $\Phi(M_{DCN})$.

If H_0 is rejected (i.e., the alternative hypothesis is accepted), then it implies that $\Phi(M_{TRN})$ will correctly predict more fault-prone files than $\Phi(M_{DCN})$. This also implies that $\Phi(M_{TRN})$ is more effective than $\Phi(M_{DCN})$ with respect to recall.

To answer R3, we propose CaTRN. The motivation behind the construction of the CaTRN is to investigate whether integrating additional factors that describe the development effort in the current TRN will better present the interactions between developers and modules and therefore further improve the fault-proneness prediction using the metrics derived from CaTRN. Consequently, in order to construct a CaTRN, for each type of relation in a TRN, a particular mechanism is applied to further calibrate the corresponding relation strength (i.e., the weight on the corresponding edges). More details regarding CaTRN will be presented the next chapter.

4.3.4 Results for R1

To answer R1, we use the Spearman rank correlation coefficient to measure the correlation between each metric and the number of post-released bugs in a file.[3] The results are shown in Table 5 through Table 7. Each entry in the tables gives the coefficient between a metric and the number of bugs. For example, let us look at the first row of Table 5. The correlation between metric $M_{TRN\text{-}FDC}$ and the number of bugs is 0.79, and the correlation between $M_{DCN\text{-}FDC}$ and the number of bugs is 0.73. The corresponding correlations between metrics $M_{MDN\text{-}FDC}$, $M_{STN\text{-}FDC}$, and $M_{DN\text{-}FDC}$ and the number of bugs are 0.62, 0.71, and 0.63, respectively. Therefore, the centrality metric, M_{FDC}, derived from TRN (i.e., $M_{TRN\text{-}FDC}$) has the strongest correlation with the number of bugs compared to the corresponding M_{FDC} derived from DCN (i.e., $M_{DCN\text{-}FDC}$), MDN (i.e., $M_{MDN\text{-}FDC}$), STN (i.e., $M_{STN\text{-}FDC}$), and DN (i.e., $M_{DN\text{-}FDC}$). Let us now look at the second column of the same table. The correlation between $M_{TRN\text{-}BP}$ and the number of bugs is 0.55, the correlation between $M_{TRN\text{-}EGD}$ and the number of bugs is 0.54, and the correlation between $M_{TRN\text{-}FNB}$ and the number of bugs is 0.44. Therefore, the M_{FDC} derived from TRN (i.e., $M_{TRN\text{-}FDC}$) has the strongest correlation with the number of bugs compared to $M_{TRN\text{-}BP}$, $M_{TRN\text{-}EGD}$, and $M_{TRN\text{-}FNB}$ which are derived from the same TRN.

Table 5. Correlation Analysis Using Spearman Rank Correlation Coefficient for Camel 1.4.0 where correlation is significant at the 0.05 level (2-tailed)

$M_{TRN\text{-}FDC}$	0.79	$M_{DCN\text{-}FDC}$	0.73	$M_{MDN\text{-}FDC}$	0.62	$M_{STN\text{-}FDC}$	0.71	$M_{DN\text{-}FDC}$	0.63	M_{LOC}	0.33	M_{CBO}	0.23	M_{NC}	0.38
$M_{TRN\text{-}BP}$	0.55	$M_{DCN\text{-}BP}$	0.50	$M_{MDN\text{-}BP}$	0.47	$M_{STN\text{-}BP}$	0.50	$M_{DN\text{-}BP}$	0.45	M_{McCabe}	0.31	M_{NOC}	0.03	M_{ND}	0.34
$M_{TRN\text{-}EGD}$	0.54	$M_{DCN\text{-}EGD}$	0.46	$M_{MDN\text{-}EGD}$	0.39	$M_{STN\text{-}EGD}$	0.46	$M_{DN\text{-}EGD}$	0.36	M_{LCOM}	0.11	M_{RFC}	0.10		
$M_{TRN\text{-}FNB}$	0.44	$M_{DCN\text{-}FNB}$	0.37	$M_{MDN\text{-}FNB}$	0.26	$M_{STN\text{-}FNB}$	0.36	$M_{DN\text{-}FNB}$	0.33	M_{DIT}	0.05	M_{WMC}	0.25		

[3] The term "bugs" here are used to refer specifically to post-released bugs.

Table 6. Correlation Analysis Using Spearman Rank Correlation Coefficient for Flume 1.5.0 where correlation is significant at the 0.05 level (2-tailed)

$M_{TRN-FDC}$	0.76	$M_{DCN-FDC}$	0.71	$M_{MDN-FDC}$	0.61	$M_{STN-FDC}$	0.65	M_{DN-FDC}	0.62	M_{LOC}	0.21	M_{CBO}	0.05	M_{NC}	0.58
M_{TRN-BP}	0.69	M_{DCN-BP}	0.60	M_{MDN-BP}	0.49	M_{STN-BP}	0.63	M_{DN-BP}	0.52	M_{McCabe}	0.06	M_{NOC}	0.01	M_{ND}	0.43
$M_{TRN-EGD}$	0.60	$M_{DCN-EGD}$	0.50	$M_{MDN-EGD}$	0.40	$M_{STN-EGD}$	0.50	M_{DN-EGD}	0.40	M_{LCOM}	0.12	M_{RFC}	0.01		
$M_{TRN-FNB}$	0.56	$M_{DCN-FNB}$	0.45	$M_{MDN-FNB}$	0.37	$M_{STN-FNB}$	0.45	M_{DN-FNB}	0.40	M_{DIT}	0.02	M_{WMC}	0.01		

Table 7. Correlation Analysis Using Spearman Rank Correlation Coefficient for Tika 1.6 where correlation is significant at the 0.05 level (2-tailed)

$M_{TRN-FDC}$	0.65	$M_{DCN-FDC}$	0.53	$M_{MDN-FDC}$	0.46	$M_{STN-FDC}$	0.52	M_{DN-FDC}	0.41	M_{LOC}	0.29	M_{CBO}	0.19	M_{NC}	0.62
M_{TRN-BP}	0.52	M_{DCN-BP}	0.42	M_{MDN-BP}	0.39	M_{STN-BP}	0.45	M_{DN-BP}	0.30	M_{McCabe}	0.12	M_{NOC}	0.03	M_{ND}	0.42
$M_{TRN-EGD}$	0.56	$M_{DCN-EGD}$	0.47	$M_{MDN-EGD}$	0.36	$M_{STN-EGD}$	0.46	M_{DN-EGD}	0.39	M_{LCOM}	0.32	M_{RFC}	0.06		
$M_{TRN-FNB}$	0.40	$M_{DCN-FNB}$	0.33	$M_{MDN-FNB}$	0.25	$M_{STN-FNB}$	0.34	M_{DN-FNB}	0.21	M_{DIT}	0.09	M_{WMC}	0.14		

In general, from Table 5 to Table 7, we observe that: (1) $M_{TRN-FDC}$ has the strongest correlation (i.e., 0.79) with the number of bugs among all metrics; (2) for any M_{Cen} derived from TRN, $M_{TRN-Cen}$, it has the strongest correlation with the number of bugs compared to the corresponding M_{Cen} derived from DCN (i.e., $M_{DCN-Cen}$), MDN (i.e., $M_{MDN-Cen}$), STN (i.e., $M_{STN-Cen}$), and DN (i.e., M_{DN-Cen}).

Table 8. Confidence that $M_{TRN-Cen}$ is more correlated with the number of bugs than the corresponding $M_{DCN-Cen}$, $M_{MDN-Cen}$, $M_{STN-Cen}$, and M_{DN-Cen}

	$M_{DCN-Cen}$	$M_{MDN-Cen}$	$M_{STN-Cen}$	M_{DN-Cen}
Camel 1.4.0	98.98%	99.98%	99.98%	99.98%
Flume 1.5.0	97.97%	99.98%	98.97%	99.97%
Tika 1.6	98.98%	99.97%	99.98%	99.98%

In addition, we use the paired Wilcoxon signed-rank test to investigate R1 from a statistical point of view. Table 8 presents the results of a Wilcoxon signed-rank test showing the confidence with which it can be claimed that $M_{TRN-Cen}$ is more correlated with the number of bugs than the corresponding $M_{DCN-Cen}$, $M_{MDN-Cen}$, $M_{STN-Cen}$, and M_{DN-Cen}. Each entry in the table strengthens the conviction that the alternative hypothesis stands. Furthermore, for each program in Table 8, at the 0.05 level, the correlation coefficient distributions are significantly different

between (1) $M_{TRN\text{-}Cen}$ and $M_{DCN\text{-}Cen}$, (2) $M_{TRN\text{-}Cen}$ and $M_{MDN\text{-}Cen}$, (3) $M_{TRN\text{-}Cen}$ and $M_{STN\text{-}Cen}$, and (4) $M_{TRN\text{-}Cen}$ and $M_{DN\text{-}Cen}$.

For example, for Camel 1.4.0, it can be said with 98.98% confidence that $M_{TRN\text{-}Cen}$ is more correlated with the number of bugs than the corresponding $M_{DCN\text{-}Cen}$. Let us look at the third row of Table 8; for Flume 1.5.0, it can be said with 97.97%, 99.98%, 98.97%, and 99.97% confidence that $M_{TRN\text{-}Cen}$ is more correlated with the number of bugs than the corresponding $M_{DCN\text{-}Cen}$, $M_{MDN\text{-}Cen}$, $M_{STN\text{-}Cen}$, and $M_{DN\text{-}Cen}$. In general, from Table 8 it can be claimed with high confidence (at least 97%) that $M_{TRN\text{-}Cen}$ is more correlated with the number of bugs than the corresponding $M_{DCN\text{-}Cen}$, $M_{MDN\text{-}Cen}$, $M_{STN\text{-}Cen}$, and $M_{DN\text{-}Cen}$ for all three programs. If we change our alternative hypothesis to "$M_{TRN\text{-}Cen}$ is equally/more correlated with the number of bugs as/than the corresponding $M_{DCN\text{-}Cen}$, $M_{MDN\text{-}Cen}$, $M_{STN\text{-}Cen}$, and $M_{DN\text{-}Cen}$," then the confidence is 100% for almost very scenario.

***Summary with respect to R1*:**

- Metrics derived from the proposed TRN are significant indicators for the number of bugs in a file
- Metrics derived from the proposed TRN are generally more correlated to the number of bugs than corresponding metrics derived from DCN, MDN, STN, and DN
- The FDC metric derived from TRN, $M_{TRN\text{-}FDC}$, has the strongest correlation with the number of bugs among all metrics used in our case studies. This also indicates that for a software module (a file in our case), (1) the number of direct interactions with its contributing software developers, (2) the contribution frequency of these developers, (3) the number of modules it has a direct dependency (both functional and logical) relationship with, and (4)

their mutual dependence intensity, jointly have a significant impact on the quality of the module itself

4.3.5 Results for R2

To answer R2, for each program, we compute and compare the average recall and FPR of $\Phi(M_{TRN})$, $\Phi(M_{DCN})$, $\Phi(M_{MDN})$, $\Phi(M_{STN})$, $\Phi(M_{DN})$, and $\Phi(M_{CO})$. The results are shown in Table 9 through Table 11. For example, in Table 9, the average recall and FPR of $\Phi(M_{TRN})$ are 69.79% and 4.60%, respectively. In the same table, the average recall and FPR of $\Phi(M_{DCN})$ are 62.95% and 4.86%, respectively. Therefore, $\Phi(M_{TRN})$ has a larger average recall and a lower average FRP compared to $\Phi(M_{DCN})$.

Table 9. Prediction effectiveness of $\Phi(M_{TRN})$, $\Phi(M_{DCN})$, $\Phi(M_{MDN})$, $\Phi(M_{STN})$, $\Phi(M_{DN})$, and $\Phi(M_{CO})$ with respect to average Recall and average FPR for Camel 1.4.0

Prediction Model	Average Recall	Average FPR
$\Phi(M_{TRN})$	69.79%	4.60%
$\Phi(M_{DCN})$	62.95%	4.86%
$\Phi(M_{MDN})$	26.49%	6.23%
$\Phi(M_{STN})$	44.09%	5.66%
$\Phi(M_{DN})$	38.47%	6.63%
$\Phi(M_{CO})$	54.15%	4.90%

From Table 9, we observe that $\Phi(M_{TRN})$ has the highest average recall (i.e., 69.79%) and the lowest average FPR (i.e., 4.60%) among all fault-proneness prediction models in the table. The same also applies to Table 10 and Table 11 where $\Phi(M_{TRN})$ has the highest average recall (i.e., 87.01% and 64.60%) and the lowest average FPR (i.e., 4.51% and 2.10%) among all fault-proneness prediction models in these two tables.

Table 10. Prediction effectiveness of $\Phi(M_{TRN})$, $\Phi(M_{DCN})$, $\Phi(M_{MDN})$, $\Phi(M_{STN})$, $\Phi(M_{DN})$, and $\Phi(M_{CO})$ with respect to average Recall and average FPR for Flume 1.5.0

Prediction Model	Average Recall	Average FPR
$\Phi(M_{TRN})$	87.01%	4.51%
$\Phi(M_{DCN})$	76.91%	6.17%
$\Phi(M_{MDN})$	44.71%	9.96%
$\Phi(M_{STN})$	64.79%	7.05%
$\Phi(M_{DN})$	53.22%	9.58%
$\Phi(M_{CO})$	68.14%	6.39%

Table 11. Prediction effectiveness of $\Phi(M_{TRN})$, $\Phi(M_{DCN})$, $\Phi(M_{MDN})$, $\Phi(M_{STN})$, $\Phi(M_{DN})$, and $\Phi(M_{CO})$ with respect to average Recall and average FPR for Tika 1.6

Prediction Model	Average Recall	Average FPR
$\Phi(M_{TRN})$	64.60%	2.10%
$\Phi(M_{DCN})$	63.78%	2.38%
$\Phi(M_{MDN})$	40.67%	3.79%
$\Phi(M_{STN})$	52.78%	2.78%
$\Phi(M_{DN})$	45.06%	2.97%
$\Phi(M_{CO})$	53.97%	2.52%

Once again, from a statistical point of view, we employ the paired Wilcoxon signed-rank test to compare the recall and FPR of $\Phi(M_{TRN})$ against $\Phi(M_{DCN})$, $\Phi(M_{MDN})$, $\Phi(M_{STN})$, $\Phi(M_{DN})$, and $\Phi(M_{CO})$. Table 12 presents the results of a Wilcoxon signed-rank test showing the confidence with which it can be claimed that $\Phi(M_{TRN})$ is more effective (in terms of recall or FPR) than $\Phi(M_{DCN})$, $\Phi(M_{MDN})$, $\Phi(M_{STN})$, $\Phi(M_{DN})$, and $\Phi(M_{CO})$. Each entry in the table gives the assurance with which the alternative hypothesis stands. Furthermore, for each program in Table 12, at the 0.05 level, the recall/FPR distributions are significantly different between (1) $\Phi(M_{TRN})$ and $\Phi(M_{DCN})$, (2) $\Phi(M_{TRN})$ and $\Phi(M_{MDN})$, (3) $\Phi(M_{TRN})$ and $\Phi(M_{STN})$, (4) $\Phi(M_{TRN})$and $\Phi(M_{DN})$, and (5) $\Phi(M_{TRN})$ and $\Phi(M_{CO})$, respectively. To take an example from Table 12, it can be said with 99.98% and 99.99% confidence that $\Phi(M_{TRN})$ has a higher recall and lower FPR, respectively, than $\Phi(M_{DCN})$ for Camel 1.4.0. It also implies that $\Phi(M_{TRN})$ is more effective than

$\Phi(M_{DCN})$ in terms of recall and FPR, respectively. In general, from Table 12 we observe that it can be said with at least 99.90% confidence that $\Phi(M_{TRN})$ has a higher recall and lower FPR than the corresponding $\Phi(M_{DCN})$, $\Phi(M_{MDN})$, $\Phi(M_{STN})$, $\Phi(M_{DN})$, and $\Phi(M_{CO})$ for all three programs. If we change our alternative hypothesis to consider equalities, then the confidence is 100% for almost very scenario.

Table 12. Confidence that $\Phi(M_{TRN})$ is more effective than $\Phi(M_{DCN})$, $\Phi(M_{MDN})$, $\Phi(M_{STN})$, $\Phi(M_{DN})$, and $\Phi(M_{CO})$ with respect to Recall and FPR

		$\Phi(M_{DCN})$	$\Phi(M_{MDN})$	$\Phi(M_{STN})$	$\Phi(M_{DN})$	$\Phi(M_{CO})$
Camel 1.4.0	Recall	99.98%	99.99%	99.99%	99.98%	99.98%
	FPR	99.99%	99.99%	99.99%	99.98%	99.97%
Flume 1.5.0	Recall	99.98%	99.99%	99.98%	99.99%	99.98%
	FPR	99.98%	99.99%	99.99%	99.98%	99.99%
Tika 1.6	Recall	99.96%	99.99%	99.97%	99.98%	99.98%
	FPR	99.96%	99.99%	99.97%	99.98%	99.99%

***Summary with respect to R2*:**

Fault-proneness prediction models using network node centrality metrics derived from the proposed TRN are more effective than prediction models using the same metrics derived from DCN, MDN, STN, and DN as well as prediction models using the ten common metrics, in terms of recall and FPR.

CHAPTER 5

EDGE WEIGHT CALIBRATION MECHANISM FOR TRN

5.1 The Proposed CaTRN

To answer R3, we propose CaTRN. The motivation behind the construction of the CaTRN is to investigate whether integrating additional factors that describe the development effort in the current TRN will better present the interactions between developers and modules and therefore further improve the fault-proneness prediction using the metrics derived from CaTRN. Consequently, in order to construct a CaTRN, for each type of relation in a TRN, a particular mechanism is applied to further calibrate the corresponding relation strength (i.e., the weight on the corresponding edges).

This section contains material from the following source [150]: S. Lee and Y. Li, "DRS: A Developer Risk Metric for Better Predicting Software Fault-Proneness," in *Proceedings of the 2nd International Conference on Trustworthy Systems and Their Applications*, Hualien, Taiwan, July 2015, pp. 120-127. Specifically, we introduce developer risk score (DRS) [150], which computes the risk of a developer working on the modules, and use it for further edge weight calibration. DRS is based on two heuristics: (1) with respect to a given program, the more frequently a developer has introduced bugs in past releases, and the greater the severity of those bugs, the higher the risk that this program will contain a bug if this same developer makes a commit on the current release; and (2) the greater the complexity of a program, the greater the difficulty a developer has in working on this program and the higher the risk that the developer will introduce a bug into the program. For a given software system, assume that m_j is the jth module in the system, and c_k is the kth bug-introducing commit made by developer d in the jth

module. We retrieve the bug severity of each bug-introducing commit (e.g., *critical*, *major*, *minor*, or *trivial*) from JIRA [121] and use the function *SeverityScore*(f_j, c_k, d, R−1) to map it to one of the following scores: 4 (*critical)*, 3 (*major)*, 2 (*minor)*, and 1 (*trivial*). A score of 4 is assigned to the variable *MaxSeverityScore.* The bug severity ratio of the kth bug-introducing commit made by developer d in the jth module in release R−1 is defined as:

$$SeverityRatio(m_j, c_k, d, R-1) = \frac{SeverityScore(m_j, c_k, d, R-1)}{MaxSeverityScore} \tag{6}$$

The overall complexity value of the jth module in release R−1 is computed by *Complexity*(m_j, R−1) as the sum of normalized LOC [181], McCabe Complexity [157], and all six CK metrics [59]. *TotalCommits*(d, R−1) gives the total number of commits made by developer d in release R−1. *DRS*(d, R), the developer risk score of developer d at release R, is defined as:

$$DRS(d, R) = \frac{\sum \frac{SeverityRatio(m_j, c_k, d, R-1)}{Complexity(m_j, R-1)}}{TotalCommits(d, R-1)} \tag{7}$$

To calibrate the weight of a developer contribution edge, we multiply the original weight (i.e., the number of commits made by a developer) by the DRS value[4] of this developer. To calibrate the weight of a module dependency edge, we multiply the original weight (i.e., the quantified dependency value for a pair of modules) by the sum of DRS values of distinct developers who have worked on the two modules. To calibrate the weight of a developer collaboration edge, we multiply the original weight (i.e., the number of modules on which two

[4] For a newly joined developer, its DRS value is set to the median of the DRS value set which is currently available.

developers have both worked) by the sum of DRS values of these two developers. Let us assume the DRS values for developers *Bob*, *Dan*, and *Jim* are 0.5, 1.2, and 3, respectively. The CaTRN is shown in Figure 7. Compared to TRN, CaTRN contains additional information by considering developer risk, program complexity, and bug severity, thus describing the development effort from a more comprehensive perspective. The same calibrating strategy can also apply to DCN, MDN, STN, and DN. As a result, a total of seven modified networks are obtained (i.e., CaTRN, CaDCN, CaMDN, CaSTN, and CaDN). Once we have these modified networks, the corresponding network centrality metrics from each modified network are derived, respectively.

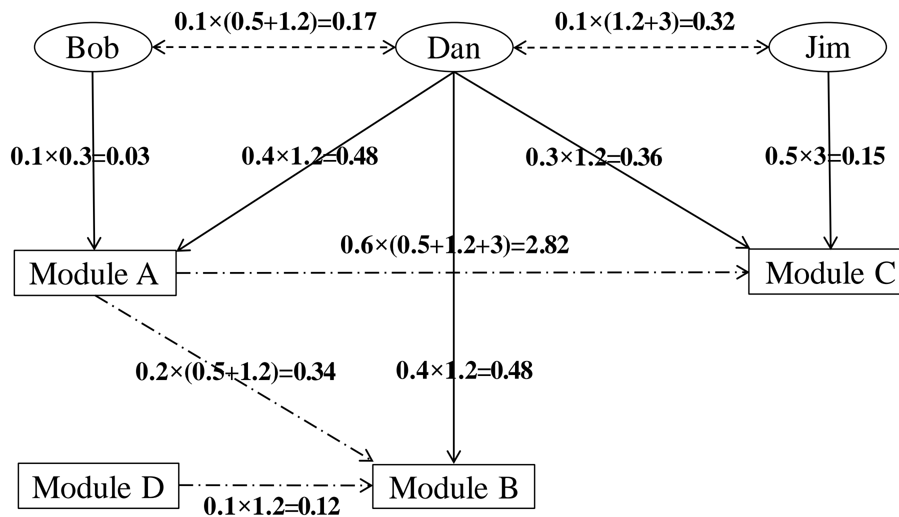

Figure 7. CaTRN with 3 developers and 4 modules

5.2 Results for R3

We answer R3 by re-conducting similar data analysis which has been used to investigate R1 and R2. Table 13 through Table 15 present the Spearman rank correlation coefficient used to measure the correlation between each metric derived from the corresponding modified networks and the number of bugs in a file. For example, in the first row of Table 13, the correlation

between the FDC metric derived from CaTRN (i.e., $M_{CaTRN\text{-}FDC}$) and the number of bugs is 0.87. As you may recall, the same FDC metric derived from TRN (i.e., $M_{TRN\text{-}FDC}$) in Table 5 is 0.79. This indicates that the FDC metric derived from the modified TRN (i.e., $M_{CaTRN\text{-}FDC}$) which consider calibrated edge weight has a stronger correlation with the number of bugs than the same FDC metric derived from the original TRN which does not.

Table 13. Spearman Rank Correlation Coefficient for Camel 1.4.0 where correlation is significant at the 0.05 level (2-tailed)

$M_{CaTRN\text{-}FDC}$	0.87	$M_{CaDCN\text{-}FDC}$	0.80	$M_{CaMDN\text{-}FDC}$	0.67	$M_{CaSTN\text{-}FDC}$	0.75	$M_{CaDN\text{-}FDC}$	0.69
$M_{CaTRN\text{-}BP}$	0.60	$M_{CaDCN\text{-}BP}$	0.51	$M_{CaMDN\text{-}BP}$	0.50	$M_{CaSTN\text{-}BP}$	0.55	$M_{CaDN\text{-}BP}$	0.47
$M_{CaTRN\text{-}EGD}$	0.57	$M_{CaDCN\text{-}EGD}$	0.53	$M_{CaMDN\text{-}EGD}$	0.40	$M_{CaSTN\text{-}EGD}$	0.52	$M_{CaDN\text{-}EGD}$	0.40
$M_{CaTRN\text{-}FNB}$	0.52	$M_{CaDCN\text{-}FNB}$	0.42	$M_{CaMDN\text{-}FNB}$	0.31	$M_{CaSTN\text{-}FNB}$	0.37	$M_{CaDN\text{-}FNB}$	0.38

Table 14. Spearman Rank Correlation Coefficient for Flume 1.5.0 where correlation is significant at the 0.05 level (2-tailed)

$M_{CaTRN\text{-}FDC}$	0.83	$M_{CaDCN\text{-}FDC}$	0.79	$M_{CaMDN\text{-}FDC}$	0.64	$M_{CaSTN\text{-}FDC}$	0.71	$M_{CaDN\text{-}FDC}$	0.69
$M_{CaTRN\text{-}BP}$	0.71	$M_{CaDCN\text{-}BP}$	0.64	$M_{CaMDN\text{-}BP}$	0.56	$M_{CaSTN\text{-}BP}$	0.69	$M_{CaDN\text{-}BP}$	0.55
$M_{CaTRN\text{-}EGD}$	0.64	$M_{CaDCN\text{-}EGD}$	0.58	$M_{CaMDN\text{-}EGD}$	0.48	$M_{CaSTN\text{-}EGD}$	0.57	$M_{CaDN\text{-}EGD}$	0.44
$M_{CaTRN\text{-}FNB}$	0.61	$M_{CaDCN\text{-}FNB}$	0.50	$M_{CaMDN\text{-}FNB}$	0.40	$M_{CaSTN\text{-}FNB}$	0.51	$M_{CaDN\text{-}FNB}$	0.43

Table 15. Spearman Rank Correlation Coefficient for Tika 1.6 where correlation is significant at the 0.05 level (2-tailed)

$M_{CaTRN\text{-}FDC}$	0.67	$M_{CaDCN\text{-}FDC}$	0.57	$M_{CaMDN\text{-}FDC}$	0.52	$M_{CaSTN\text{-}FDC}$	0.54	$M_{CaDN\text{-}FDC}$	0.45
$M_{CaTRN\text{-}BP}$	0.55	$M_{CaDCN\text{-}BP}$	0.49	$M_{CaMDN\text{-}BP}$	0.41	$M_{CaSTN\text{-}BP}$	0.47	$M_{CaDN\text{-}BP}$	0.33
$M_{CaTRN\text{-}EGD}$	0.61	$M_{CaDCN\text{-}EGD}$	0.51	$M_{CaMDN\text{-}EGD}$	0.42	$M_{CaSTN\text{-}EGD}$	0.51	$M_{CaDN\text{-}EGD}$	0.40
$M_{CaTRN\text{-}FNB}$	0.47	$M_{CaDCN\text{-}FNB}$	0.37	$M_{CaMDN\text{-}FNB}$	0.32	$M_{CaSTN\text{-}FNB}$	0.39	$M_{CaDN\text{-}FNB}$	0.25

In general, from Table 13 to Table 15, we observe that: (1) the metrics derived from modified networks (i.e., $M_{CaX\text{-}Cen}$) have stronger correlation with the number of bugs than the corresponding metrics derived from the original networks (i.e., $M_{X\text{-}Cen}$) as shown from Table 5 to Table 7; (2) $M_{CaTRN\text{-}FDC}$ has the strongest correlation with the number of bugs among all metrics derived from modified networks; and (3) for any M_{Cen} derived from CaTRN, $M_{CaTRN\text{-}Cen}$, it has

the strongest correlation with the number of bugs compared to the corresponding M_{Cen} derived from CaDCN (i.e., $M_{CaDCN-Cen}$), CaMDN (i.e., $M_{CaMDN-Cen}$), CaSTN (i.e., $M_{CaSTN-Cen}$), and CaDN (i.e., $M_{CaDN-Cen}$).

In addition, the results of the paired Wilcoxon signed-rank test in Table 16 indicate that with high confidence (at least 98%), $M_{CaX-Cen}$ is more correlated with the number of bugs than M_{X-Cen}. The confidence increases to 100% for almost every scenario when considering equalities. In general, metrics derived from the modified networks which consider calibrated edge weights are more correlated with the number of bugs than the same metrics derived from the corresponding networks which do not.

Table 16. Confidence that $M_{CaX-Cen}$ is more correlated to the number of bugs than the corresponding M_{X-Cen}

	$M_{CaX-Cen} > M_{X-Cen}$
Camel 1.4.0	98.98%
Flume 1.5.0	99.96%
Tika 1.6	99.97%

Similarly, we also compute the average recall and the average FPR of $\Phi(M_{CaTRN})$, $\Phi(M_{CaDCN})$, $\Phi(M_{CaMDN})$, $\Phi(M_{CaSTN})$, and $\Phi(M_{CaDN})$. The results are shown in Table 17 through Table 19. For example, in Table 17, the average recall and FPR of $\Phi(M_{CaTRN})$ are 71.50% and 4.59%, respectively. As stated previously, the average recall and FRP of the corresponding $\Phi(M_{TRN})$ in Table 9 are 69.79% and 4.60%, respectively. This indicates that the fault-proneness prediction models based on modified TRN which consider calibrated edge weights (i.e., $\Phi(M_{CaTRN})$) are more effective than the models based on the corresponding TRN which do not (i.e., $\Phi(M_{TRN})$) in terms of average recall and FPR.

In general, from Table 17 to Table 19, we observe that: (1) fault-proneness prediction models based on modified networks which consider calibrated edge weights (i.e., $\Phi(M_{CaX})$) are more effective than the prediction models based on the corresponding networks which do not (i.e., $\Phi(M_X)$) as shown from Table 9 to Table 11 in terms of average recall and FPR; and (2) $\Phi(M_{CaTRN})$ has the highest average recall and FPR among all modified networks.

Table 17. Prediction effectiveness of $\Phi(M_{CaTRN})$, $\Phi(M_{CaDCN})$, $\Phi(M_{CaMDN})$, $\Phi(M_{CaSTN})$, and $\Phi(M_{CaDN})$ with respect to average Recall and average FPR for Camel 1.4.0

Prediction Model	Average Recall	Average FPR
$\Phi(M_{CaTRN})$	71.50%	3.59%
$\Phi(M_{CaDCN})$	66.59%	4.69%
$\Phi(M_{CaMDN})$	27.77%	5.48%
$\Phi(M_{CaSTN})$	45.65%	5.53%
$\Phi(M_{CaDN})$	38.86%	6.47%

Table 18. Prediction effectiveness of $\Phi(M_{CaTRN})$, $\Phi(M_{CaDCN})$, $\Phi(M_{CaMDN})$, $\Phi(M_{CaSTN})$, and $\Phi(M_{CaDN})$ with respect to average Recall and average FPR for Flume 1.5.0

Prediction Model	Average Recall	Average FPR
$\Phi(M_{CaTRN})$	88.80%	3.93%
$\Phi(M_{CaDCN})$	80.73%	5.18%
$\Phi(M_{CaMDN})$	46.92%	9.10%
$\Phi(M_{CaSTN})$	65.75%	6.43%
$\Phi(M_{CaDN})$	54.61%	8.94%

Table 19. Prediction effectiveness of $\Phi(M_{CaTRN})$, $\Phi(M_{CaDCN})$, $\Phi(M_{CaMDN})$, $\Phi(M_{CaSTN})$, and $\Phi(M_{CaDN})$ with respect to average Recall and average FPR for Tika 1.6

Prediction Model	Average Recall	Average FPR
$\Phi(M_{CaTRN})$	65.73%	1.73%
$\Phi(M_{CaDCN})$	65.01%	2.25%
$\Phi(M_{CaMDN})$	41.66%	3.00%
$\Phi(M_{CaSTN})$	56.77%	2.24%
$\Phi(M_{CaDN})$	45.45%	1.75%

Moreover, the results of the paired Wilcoxon signed-rank test in Table 20 indicate that with high confidence (at least 96%), $\Phi(M_{CaX})$ is more effective than $\Phi(M_X)$ in terms of recall and FPR. If we change our alternative hypothesis to consider equalities, then the confidence is 100% for almost very scenario.

Table 20. Confidence that $\Phi(M_{CaX})$ is more effective than the corresponding $\Phi(M_X)$

		$\Phi(M_{CaX}) > \Phi(M_X)$
Camel 1.4.0	Recall	98.99%
	FPR	99.68%
Flume 1.5.0	Recall	96.88%
	FPR	99.99%
Tika 1.6	Recall	97.93%
	FPR	97.98%

***Summary with respect to R3*:**

- The developer risk score (DSR) which takes bug severity, program complexity, and development difficulty into account contributes to the network refinement and improves the effectiveness of software-fault proneness prediction
- Metrics derived from the modified network which considers calibrated edge weight using DSR are generally more correlated to the number of bugs than the same metrics derived from the corresponding networks which do not consider calibrated edge weight
- Fault-proneness prediction models using metrics derived from the modified networks which consider calibrated edge weight are more effective than prediction models using the same metrics derived from the corresponding networks which do not

CHAPTER 6

CONCLUSION AND FUTURE WORK

The more complex a software system is, the more likely programmers will make mistakes, introducing faults that can lead to execution failures. A risk in a software system can be viewed as a potential problem, and a problem is a risk that has manifested. In order to reduce the risk of software operations, software modules that have the potential to cause problems have to be identified so that necessary actions can be taken to prevent any such problems from occurring. This demand, in turn, has fueled the research and development of software fault-proneness prediction.

Aside from various techniques used for building/training a software fault-proneness prediction model, the key to conducting effective fault-proneness prediction is the software metrics used in the model. Although a vast amount of product and process metrics have been proposed in the past decades, there are still no decisive conclusions regarding their effectiveness in predicting fault-prone modules. A potential solution is to explore metrics that combine the strength of both product and process metrics. In other words, metrics that focus on the relations within/between software developers and modules should be taken into account. As a result, social network analysis can fulfill the need as it investigates not only the social organization of the work but also the technical information infrastructures.

Previous studies have shown that the developer contribution relation, module dependency relation, and developer collaboration relation have been used to build different networks for software fault-proneness prediction. However, these networks either use a single relation or a pair of relations. In addition, these networks appear to neglect an essential factor: developer

quality. After all, it is developers who make mistakes and introduce faults into software. Therefore, we propose TRN which integrates all three relations with additional information in a comprehensive network for effective fault-proneness prediction.

In a TRN, a developer contribution edge connects a developer with a software module if the developer has made a commit to the module between two consecutive releases. The weight on the contribution edge is set as the normalized number of commits made from the developer to the module. A module dependency edge in TRN connects a module to another if dependency exists between these two modules. Different from DN, two types of dependency: functional dependency (e.g., a file calls another file) and logical dependency (e.g., two files are modified in the same commit) are considered instead of just functional dependency. This provides a more accurate representation of module dependencies affecting the development effort. The weights for logical dependency between two modules are computed as the normalized number of times that these two modules are modified in the same commit. The resulting module dependency is computed as the sum of normalized functional dependency and normalized logical dependency. For the developer collaboration edge in TRN, it connects two developers if they have made at least one commit on the same module. The weight on collaboration edge is computed as the normalized number of modules two developers have worked on together.

In addition, we use DRS which computes the risk of a developer working on the modules to further calibrate the edge weights on TRN. The calibrated TRN is called CaTRN. DRS is based on two heuristics: (1) with respect to a given program, the more frequently a developer has introduced bugs in past releases, and the greater the severity of those bugs, the higher the risk that this program will contain a bug if this same developer makes a commit on the current

release; and (2) the greater the complexity of a program, the greater the difficulty a developer has in working on this program and the higher the risk that the developer will introduce a bug into the program. Specifically, we multiply the number of commits made by a developer by his/her DRS value to calibrate the weight on a developer contribution edge. We multiply the quantified dependency value for a pair of modules by the sum of DRS values of distinct developers who have worked on the two modules to calibrate the weight on a module dependency edge. And we multiply the number of modules on which two developers have both worked by the sum of DRS values of these two developers to calibrate the weight on a developer collaboration edge.

Four network node centrality metrics (i.e., M_{FDC}, M_{BP}, M_{EDG}, and M_{FNB}) are derived from the corresponding networks to predict the fault-proneness of a given file on three Java programs. M_{FDC} and M_{BP} focus on the number of developers and other modules it directly connects to, the impact of direct interactions on the module. More people working on the module, higher probability of introducing faults due to inconsistent coding style especially when these people have never worked/communicated with each other before. Due to the direct dependency relationship, more changes made on its neighbor modules higher probability that appropriate changes should be made on the module accordingly, thus more difficult to maintain the module. M_{EGD} and M_{FNB} focus on the connection strength of the module to all modules and developers that it either directly or indirectly connects to. The closer the module to other modules and developers, the stronger the connection, the more likely the module can be affected by other modules and developers in a way. The results of our study indicate that (1) TRN-based centrality metrics are more correlated with the number of bugs than the corresponding DCN-, MDN-, STN-, and DN-based centrality metrics as well as the ten software metrics that are commonly used for

software fault-proneness prediction; (2) fault-proneness prediction models using TRN-based centrality metrics outperform the models using DCN-, MDN-, STN-, and DN-based centrality metrics as well as the models based on ten commonly used software metrics; (3) centrality metrics derived from a modified network which consider calibrated edge weight using developer risk score are more correlated to the number of bugs than those derived from the same network which does not; and (4) fault-proneness prediction models using centrality metrics derived from a modified network outperform the models using centrality metrics derived from the same network which does not. In the future, we plan to repeat our study on a wider variety of programs and include additional software metrics for comparison to further validate the effectiveness of our TRN-based metrics. We also intend to search for potential intelligent algorithms to better train our prediction models. In addition, it is interesting to investigate whether our TRN-based centrality metrics can be used for cross-project software fault-proneness prediction when the historical information is limited or unavailable. Last but not least, we also plan to seek other superior mechanisms to further refine current networks.

REFERENCES

[1] Apache Camel, http://camel.apache.org/

[2] Apache Flume, http://flume.apache.org/

[3] Apache Tika, http://tika.apache.org/

[4] S. L. Abede, V. Arnaoudova, P. Tonella, G. Antoniol, and Y. Gueheneuc, "Can Lexicon Bad Smells Improve Fault Prediction?," in *Proceedings of the 19th Working Conference on Reverse Engineering*, Kingston, Canada, October 2012, pp. 235-244

[5] K. Aggarwal, Y. Singh, A. Kaur, and R. Malhotra, "Empirical Analysis for Investigating the Effect of Object-Oriented Metrics on Fault Proneness: A Replicated Case Study," *Software Process: Improvement and Practice*, vol. 14, no. 1, pp. 39-62, January 2009

[6] S. N. Ahsan, and F. Wotawa, "Fault Prediction Capability of Program File's Logical-Coupling Metrics," in *Proceedings of 2011 Joint Conference of the 21st International Workshop on Software Measurement and 6th International Conference on Software Process and Product Measurement*, Nara, Japan, November 2011, pp. 257-262

[7] H. Aman, "An Empirical Analysis of the Impact of Comment Statements on Fault-Proneness of Small-Size Module," in *Proceedings of the 19th Asia-Pacific Software Engineering Conference*, Hong Kong, China, December 2012, pp. 659-666

[8] Hirohisa Aman, "An Empirical Analysis on Fault-Proneness of Well-Commented Modules," in *Proceedings of the 2012 Fourth International Workshop on Empirical Software Engineering in Practice*, Osaka, Japan, October 2012, pp. 3-9

[9] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A Systematic and Comprehensive Investigation of Methods to Build and Evaluate Fault Prediction Models," *Journal of Systems and Software*, vol. 83, no. 1, pp. 2-17, January 2010

[10] A. Binkley and S. Schach, "Prediction of Run-Time Failures using Static Product Quality Metrics," *Software Quality Journal*, vol. 7, no. 2, pp. 141-147, July 1998

[11] I. Chowdhury and M. Zulkernine, "Using Complexity, Coupling, and Cohesion Metrics as Early Indicators of Vulnerabilities," *Journal of Systems Architecture*, vol. 57, no. 3, pp. 294-313, March 2011

[12] J. Al Dallal, "Fault Prediction and the Discriminative Powers of Connectivity-based Object-Oriented Class Cohesion Metrics," *Information and Software Technology*, vol. 54, no. 4, pp. 396-416, April 2012

[13] J. Al Dallal, "The Impact of Accounting for Special Methods in the Measurement of Object-Oriented Class Cohesion on Refactoring and Fault Prediction Activities," *Journal of Systems and Software*, vol. 85, no. 5, pp. 1042-1057, May 2012

[14] S. Amasaki, Y. Takagi, O. Mizuno, and T. Kikuno, "A Bayesian Belief Network for Assessing the Likelihood of Fault Content," in *Proceedings of the 14th International Symposium on Software Reliability Engineering*, Denver, Colorado, USA, November 2003, pp. 215-226

[15] C. Andersson and P. Runeson, "A Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems," *IEEE Transactions on Software Engineering*, vol. 33, no. 5, pp. 273-286, May 2007

[16] F. D. Anger, J. C. Munson, and R. V. Rodriguez, "Temporal Complexity and Software Faults," in *Proceedings of the 5th IEEE International Symposium on Software Reliability Engineering*, Monterey, CA, USA, November 1994, pp. 115-125

[17] E. Arisholm, "Dynamic Coupling Measures for Objected-Oriented Software," in *Proceedings of the 8th IEEE International Symposium on Software Metrics*, Ottawa, Canada, June 2002, pp. 33-42

[18] E. Arisholm and L. C. Briand, "Predicting Fault-Prone Components in A Java Legacy System," in *Proceedings of the 5th ACM/IEEE International Symposium on Empirical Software Engineering*, Rio de Janeiro, Brazil, September 2006, pp. 8-17

[19] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A Systematic and Comprehensive Investigation of Methods to Build and Evaluate Fault Prediction Models," *Journal of Systems and Software*, vol. 83, no. 1, pp. 2-17, January 2010

[20] V. R. Basili, L. C. Briand, and W. L. Melo, "A Validation of Object-oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751-761, May 1996

[21] V. R. Basili and D. H. Hutchens, "An Empirical Study of a Syntactic Complexity Family," *IEEE Transactions on Software Engineering*, vol. 9, no. 6, pp. 664-672, November 1983

[22] V. R. Basili and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM*, vol. 27, no. 1, pp. 42-52, January 1984

[23] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "Looking for Bugs in All the Right Places," in *Proceedings of the 29th ACM International Symposium on Software Testing and Analysis*, Portland, Maine, July 2006, pp. 61-72

[24] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "Does Measuring Code Change Improve Fault Prediction?," in *Proceedings of the 7th ACM International Conference on Predictive Models in Software Engineering*, Banff, AB, Canada, September 2011

[25] P. Bellini, I. Bruno, P. Nesi, and D. Rogai, "Comparing Fault-Proneness Estimation Models," in *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, Shanghai, China, June 2005, pp. 205-214

[26] S. Benlarbi and W. L. Melo, "Polymorphism Measures for Early Risk Prediction," in *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, California, May 1999, pp. 334-344

[27] A. Bernstein, J. Ekanayake, and M. Pinzger, "Improving Defect Prediction Using Temporal Features and Non Linear Models," in *Proceedings of the 9th International Workshop on Principles of Software Evolution*, Dubrovnik, Croatia, September 2007, pp. 11-18

[28] N. Bettenburg and A. Hassan, "Studying the Impact of Social Interactions on Software Quality," *Empirical Software Engineering*, vol. 18, no. 2, pp. 375-431, April 2013

[29] M. Bezerra, A. Oliveira, P. Adeodato, and S. Meira, "Enhancing RBF-DDA Algorithm's Robustness: Neural Networks Applied to Prediction of Fault-Prone Software Modules," in *Proceedings of the 20th IFIP World Computer Congress*, Milano, Italy, September 2008, pp. 119-128

[30] M. Bezerra, A. Oliveira, and S. Meira, "A Constructive RBF Neural Network for Estimating the Probability of Defects in Software Modules," in *Proceedings of the 19th International Joint Conference on Neural Networks*, Orlando, Florida, USA, August 2007, pp. 2869-2874

[31] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based Analysis and Prediction for Software Evolution," in *Proceedings of the 34th International Conference on Software Engineering*, Zurich, Switzerland, June 2012, pp. 419-429

[32] S. Biçer, A. B. Bener, and B. Çağlayan, "Defect Prediction Using Social Network Analysis on Issue Repositories," in *Proceedings of the 7th International Conference on Software and Systems Process*, Honolulu, Hawaii, USA, May 2011, pp. 63-71

[33] D. Binkley, H. Feild, D. Lawrie, and M. Pighin, "Increasing Diversity: Natural Language Measures for Software Fault Prediction," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1793-1803, November 2009

[34] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy, "Does Distributed Development Affect Software Quality? An Empirical Case Study of Windows Vista," in

Proceedings of the 31st International Conference on Software Engineering, Vancouver, Canada, May 2009, pp. 518-528

[35] C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu, "Putting It All Together: Using Socio-Technical Networks to Predict Failures," in *Proceedings of the 20th International Symposium on Software Reliability Engineering*, Mysuru, India, November 2009, pp. 109-119

[36] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't Touch My Code!: Examining the Effects of Ownership on Software Quality," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, Szeged, Hungary, September 2011, pp. 4-14

[37] P. S. Bishnu and V. Bhattacherjee, "Software Fault Prediction Using Quad Tree-based K-Means Clustering Algorithm," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 6, pp. 1146-1150, June 2012

[38] A. G. Bluman, *Elementary Statistics, A Step by Step Approach*, 6th ed. New York, NY: McGraw-Hill, 2007

[39] L. C. Briand, V. R. Basili, and C. J. Hetmanski, "Developing Interpretable Models with Optimized Set Reduction for Identifying High-Risk Software Components," *IEEE Transactions on Software Engineering*, vol. 19, no. 11, pp. 1020-1044, November 1993

[40] L. C. Briand, J. W. Daly, and J. Wüst, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *Empirical Software Engineering*, vol. 3, no .1, pp. 65-117, 1998

[41] L. C. Briand, J. W. Daly, and J. Wüst, "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Transactions on Software Engineering*, vol. 25, no. 1, pp. 91-121, February 1999

[42] L. C. Briand, S. Morasca, and V. R. Basili, "Defining and Validating Measures for Object-based High-level Design," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 722-743, October 1999

[43] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter, "Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems," *Journal of Systems and Software*, vol. 51, no. 3, pp. 245-273, May 2000

[44] L. C. Briand, J. Wüst, and H. Lounis, "Replicated Case Studies for Investigating Quality Factors in Object-Oriented Designs," *Empirical Software Engineering*, vol. 6, no. 1, pp. 11-58, 2001

[45] P. Bonacich, "Power and Centrality: A Family of Measures," *American Journal of Sociology*, no. 92, pp. 1170-1182, 1987

[46] S. P. Borgatti, M. G. Everett, and L. C. Freeman, *Ucinet for Windows: Software for Social Network Analysis*, Analytic Technologies, Harvard, MA, 2002

[47] M. F. Bosu and S. G. MacDonell, "A Taxonomy of Data Quality Challenges in Empirical Software Engineering," in *Proceedings of the 22nd IEEE Australian Conference on Software Engineering*, Melbourne, Australia, June 2013, pp. 97-106

[48] M. F. Bosu and S. G. MacDonell, "Data Quality in Empirical Software Engineering: A Targeted Review," in *Proceedings of the 17th ACM International Conference on Evaluation and Assessment*, Melbourne, Australia, Porto de Galinhas, Brazil, April 2013, pp. 171-176

[49] U. Brandes and T. Erlebach, *Network Analysis: Methodological Foundations*, Springer, March 2005

[50] G. Carrozza, D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "Analysis and Prediction of Mandelbugs in an Industrial Software System," in *Proceedings of the 6th International Conference on Software Testing, Verification and Validation*, Luxembourg, March 2013, pp. 262-271

[51] C. Catal, U. Sevim, and B. Diri, "Clustering and Metrics Thresholds Based Software Fault Prediction of Unlabeled Program Modules," in *Proceedings of the 6th International Conference on Information Technology: New Generations*, Las Vegas, USA, April 2009, pp. 199-204

[52] C. Catal, U. Sevim, and B. Diri, "Metrics-driven Software Quality Prediction without Prior Fault Data," *Electronic Engineering and Computing Technology*, Chapter 17, pp. 189-199. Springer Netherlands, 2010

[53] C. Catal, U. Sevim, and B. Diri, "Practical Development of An Eclipse-based Software Fault Prediction Tool using Naive Bayes Algorithm," *Expert Systems with Applications*, vol. 38, no. 3, pp. 2347-2353, March 2011

[54] M. Cataldo, J. D. Herbsleb, and K. M. Carley, "Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity," in *Proceedings of the 2nd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, Kaiserslautern, Germany, October 2008, pp. 2-11

[55] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb, "Software Dependencies, Work Dependencies, and Their Impact Failures," *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 864-878, November 2009

[56] M. Cataldo and J. Herbsleb, "Coordination Breakdowns and Their Impact on Development Productivity and Software Failures," *IEEE Transactions on Software Engineering*, vol. 39, no. 3, pp. 343-360, March 2013

[57] G. Chandrashekar, F. Sahin, E. Cinar, A. Radomski, and D. Sarosky, "In-Vivo Fault Analysis and Real-Time Fault Prediction for RF Generators Using State-of-the-Art Classifiers," in *Proceedings of the 26th IEEE International Conference on Systems, Man, and Cybernetics*, Manchester, UK, October 2013, pp. 1634-1639

[58] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer, "Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis," *IEEE Transactions on Software Engineering*, vol. 24, no. 8, pp. 629-639, August 1998

[59] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476-493, June 1994

[60] J. S. Collofello and S. N. Woodfield, "Evaluating the Effectiveness of Reliability-Assurance Techniques," *Journal of Systems and Software*, vol. 9, no. 3, pp. 191-195, March 1989

[61] D. Cotroneo, R. Natella, and R. Pietrantuono, "Predicting Aging-related Bugs Using Software Complexity Metrics," *Performance Evaluation*, vol. 70, no. 3, pp. 163-178, March 2013

[62] S. D. Conte, H. E. Dunsmore, and Y. E. Shen, *Software Engineering Metrics and Models*, 1st ed. San Francisco, California: Benjamin-Cummings Publishing Co., Inc., 1986

[63] C. Couto, P. Pires, M. T. Valente, and R. S. Bigonha, "Predicting Software Defects with Causality Tests," *Journal of Systems and Software*, vol. 93, pp. 24-41, July 2014

[64] G. Czibula, Z. Marian, and I. G. Czibula, "Software Defect Prediction Using Relational Association Rule Mining," *Information Science*, vol. 264, pp. 260-278, April 2014

[65] M. D'Ambros, M. Lanza, and R. Robbes, "On the Relationship between Chang Coupling and Software Defects," in *Proceedings of the 16th Working Conference on Reverse Engineering*, Lille, France, October 2009, pp. 135-144

[66] F. Deissenboeck and M. Pizka, "Concise and Consistent Naming," *Software Quality Journal*, vol. 14, no. 3, pp. 261-282, September 2006

[67] K. Dejaeger, T. Verbraken, and B. Baesens, “Toward Comprehensible Software Fault Prediction Models Using Bayesian Network Classifiers,” *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 237-257, February 2013

[68] T. DeMarco, *Controlling Software Projects: Management, Measurement & Estimation*, 1st ed. Englewood Cliffs, New Jersey: Prentice Hall, June 1986

[69] G. Denaro, S. Morasca, and M. Pezzè, “Deriving Models of Software Fault-Proneness,” in *Proceedings of the 14th ACM International Conference on Software Engineering and Knowledge Engineering*, Ischia, Italy, July 2002, pp. 361-368

[70] G. Denaro and M. Pezzè, “An Empirical Evaluation of Fault-Proneness Models,” in *Proceedings of the 24th International Conference on Software Engineering*, Limerick, Ireland, May 2002, pp. 241-251

[71] K. El Emam, W. Melo, and J. C. Machado, “The Prediction of Faulty Classes Using Object-Oriented Design Metrics,” *Journal of Systems and Software*, vol. 56, no. 1, pp. 63-75, February 2001

[72] K. O. Elish and M. O. Elish, “Predicting Defect-Prone Software Modules Using Support Vector Machines,” *Journal of Systems and Software*, vol. 81, no. 5, pp. 649-660, 2008

[73] J. Ell, “Identifying Failure Inducing Developer Pairs within Developer Networks,” in *Proceedings of the 34th International Conference on Software Engineering*, San Francisco, CA, USA, May 2013, pp. 1471-1473

[74] A. Endres, “An Analysis of Errors and Their Causes in System Programs,” *IEEE Transactions on Software Engineering*, vol. 1, no. 2, pp. 140-149, June 1975

[75] M. English, C. Exton, I. Rigon, and B. Cleary, “Fault Detection and Prediction in an Open-Source Software Project,” in *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, Vancouver, British Columbia, May 2009

[76] N. E. Fenton, P. Krause, and M. Neil, “Software Measurement: Uncertainty and Causal Modeling,” *IEEE Software*, vol. 19, no. 4, pp. 116-122, August 2002

[77] N. E. Fenton and M. Neil, “A Critique of Software Defect Prediction Models,” *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 675-689, October 1999

[78] N. E. Fenton and N. Ohlsson, “Quantiative Analysis of Faults and Failures in a Complex Software System,” *IEEE Transactions on Software Engineering*, vol. 26, no. 8, pp. 797-814, August 2000

[79] J. Ferzund, S. Ahsan, and F. Wotawa, “Empirical Evaluation of Hunk Metrics as Bug Predictors,” *Software Process and Product Measurement*, Lecture Notes in Computer Science, vol. 5891, pp. 242-254, 2009

[80] L. C. Freeman, “Centrality in Social Networks: Conceptual Clarification,” *Social Networks*, vol. 1, no. 3, pp. 215-239, 1979

[81] D. Galorath, “Software Project Failure Costs Billions. Better Estimation & Planning Can Help”, June 7, 2012 http://www.galorath.com/wp/software-project-failure-costs-billions-better-estimation-planning-can-help.php

[82] K. Gao and T. M. Khoshgoftaar, “A Comprehensive Empirical Study of Count Models for Software Fault Prediction,” *IEEE Transactions on Reliability*, vol. 56, no. 2, pp. 223-236, June 2007

[83] D. Gary, D. Bowes, N. Davey, Y. Sun, and B. Christianson, “The Misuse of the NASA Metrics Data Program Data Sets for Automated Software Defect Prediction,” in *Proceedings of the 15th Annual Conference on Evaluation & Assessment in Software Engineering*, Durham, United Kingdom, April 2011, pp. 96-103

[84] M. Gegick and L. Williams, “Toward the Use of Automated Static Analysis Alerts for Early Identification of Vulnerability- and Attack-Prone Components,” in *Proceedings of the 2nd International Conference on Internet Monitoring and Protection*, San Jose, CA, USA, July 2007

[85] R. A. Ghosh, “Clustering and Dependencies in Free/Open Source Software Development: Methodology and Tools, *First Monday*, vol. 8, no. 4, April 2003

[86] E. Giger, M. D’Ambros, M. Pinzger, and H. C. Gall, “Method-Level Bug Prediction,” in *Proceedings of the 6th IEEE/ACM International Symposium on Empirical Software Engineering and Measurement*, Lund, Sweden, pp. 171-180, September 2012

[87] B. M. Goel and P. K. Bhatia, “Investigating of High and Low Impact Faults in Object-Oriented Projects,” *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 6, pp. 1-6, November 2013

[88] B. Goel and Y. Singh, “Empirical Investigation of Metrics for Fault Prediction on Object-Oriented Software,” *Computer and Information Science*, pp. 255-265. Springer Berlin Heidelberg, 2008

[89] I. Gondra, “Applying Machine Learning to Software Fault-Proneness Prediction,” *Journal of Systems and Software*, vol. 81, vol. 2, pp. 186-195, 2008

[90] D. Gary, D. Bowes, N. Davey, Y. Sun, and B. Christianson, “The Misuse of the NASA Metrics Data Program Data Sets for Automated Software Defect Prediction,” in *Proceedings of the 15th Annual Conference on Evaluation & Assessment in Software Engineering*, Durham, United Kingdom, April 2011, pp. 96-103

[91] T. Graves, A. Karr, J. Marron, and H. Siy, “Predicting Fault Incidence Using Software Change History,” *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653-661, July 2000

[92] T. Grbac, P. Runeson, and D. Huljenić, “A Second Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems,” *IEEE Transactions on Software Engineering*, vol. 39, no. 4, pp. 462-476, April 2013

[93] L. Guo, B. Cukic, and H. Singh, “Predicting Fault Prone Modules by the Dempster-Shafer Belief Networks,” in *Proceedings of the 18th International Conference on Automated Software Engineering*, Montreal, Canada, October 2003, pp. 249-252

[94] L. Guo, Y. Ma, B. Cukic, and H. Singh, “Robust Prediction of Fault-Proneness by Random Forests,” in *Proceedings of the 15th International Symposium on Software Reliability Engineering*, Saint-Malo, France, November 2004, pp. 417-428

[95] T. Gyimothy, R. Ferenc, and I. Siket, “Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction,” *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897-910, October 2005

[96] A. Halim, “Predict Fault-Prone Classes using the Complexity of UML Class Diagram,” in *Proceedings of the 1st IEEE International Conference on Computer, Control, Informatics and Its Applications*, Jakarta, Indonesia, November 2013, pp. 289-294

[97] T. Hall, 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*, vol. 38, no. 6, pp. 1276-1304, November 2012

[98] M. H. Halstead, *Elements of Software Science*, 1st ed. Amsterdam, the Netherlands: Elsevier, 1977

[99] A. E. Hassan, “Predicting Faults Using the Complexity of Code Changes,” in *Proceedings of the 31st IEEE International Conference on Software Engineering*, Vancouver, Canada, May 2009, pp. 78-88

[100] A. E. Hassan and R. C. Holt, “The Top Ten List: Dynamic Fault Prediction,” in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, Budapest, Hungary, September 2005, pp. 263-272

[101] R. A. Hanneman and M. Riddle, *Introduction to Social Network Methods*, University of California, Riverside, California, 2005

[102] H. Hata, O. Mizuno, and T. Kikuno, "Fault-Prone Module Detection Using Large-Scale Text Features based on Spam Filtering," *Empirical Software Engineering*, vol. 15, no. 2, pp. 147-165, 2010

[103] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang, "An Investigation on the Feasibility of Cross-Project Defect Prediction," *Automated Software Engineering*, vol. 19, no. 2, pp. 167-199, June 2012

[104] S. M. Henry and D. Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Transactions on Software Engineering*, vol. 7, no. 5, pp. 510-518, September 1981

[105] S. Herbold, "Training Data Selection for Cross-Project Defect Prediction," in *Proceedings of the 9th ACM International Conference on Predictive Models in Software Engineering*, Baltimore, USA, October 2013, pp. 6-15

[106] R. Hochman, T. M. Khoshgoftaar, E. B. Allen, and J. P. Hudepohl, "Using the Genetic Algorithm to Build Optimal Neural Networks for Fault-Prone Module Detection," in *Proceedings of the 7th International Symposium on Software Reliability Engineering*, White Plains, NY, USA, 1996, pp. 152-162

[107] R. Hochman, T. M. Khoshgoftaar, E. B. Allen, and J. P. Hudepohl, "Evolutionary Neural Networks: A Robust Approach to Software Reliability Problems," in *Proceedings of the 8th International Symposium on Software Reliability Engineering*, Albuquerque, NM, USA, 1997, pp. 13-26

[108] T. Holschuh, T. Zimmermann, M. Pauser, R. Premraj, K. Herzig, A. Zeller, S. Q. S. Ag, and P. Markus, "Predicting Defects in SAP Java Code: An Experience Report," in *Proceedings of the 31st International Conference on Software Engineering*, Vancouver, Canada, May 2009, pp. 172-181

[109] S. Hong and K. Kim, "Identifying Fault-Prone Function Blocks Using the Neural Networks an Empirical Study," in *Proceedings of the 10th IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, BC, Canada, August 1997, pp. 790-793

[110] E. Hong and C. Wu, "Criticality Prediction Models using SDL Metrics Set," in *Proceedings of the 4th International Computer Science Conference and the 4th Asia-Pacific Software Engineering Conference*, 1997, pp. 23-30

[111] J. Howison, K. Inoue, and K. Crowston, "Social Dynamics of Free and Open Source Team Communication," in *Proceedings of the 2006 International Conference on Open Source Software*, pp. 319- 330

[112] W. Hu and K. Wong, "Using Citation Influence to Predict Software Defects," in *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories*, San Francisco, CA, May 2013, pp. 419-428

[113] T. Illes-Seifert and B. Paech, "Exploring the Relationship of a File's History and its Fault-Proneness: An Empirical Method and Its Application to Open Source Programs," *Information and Software Technology*, vol. 52, no. 5, pp. 539-558, May 2010

[114] B. Isong and E. Obeten, "A Systematic Review of the Empirical Validation of Object-Oriented Metrics towards Fault-Proneness Prediction," *International Journal of Software Engineering and Knowledge Engineering*, vol. 23, no. 20, pp. 1513-1540, December 2013

[115] F. Jaafar, Y. Gueheneuc, S. Hamel, and F. Khomh, "Mining the Relationship between Anti-patterns Dependencies and Fault-Proneness," in *Proceedings of the 20th IEEE Working Conference on Reverse Engineering*, Koblenz, Germany, October 2013, pp. 351-360

[116] Y. Jiang, B. Cukic, and T. Menzies, "Fault Prediction using Early Lifecycle Data," in *Proceedings of the 18th IEEE International Symposium on Software Reliability*, Trollhattan, Swden, November 2007, pp. 237-246

[117] Y. Jiang, J. Lin, B. Cukic, and T. Menzies, "Variance Analysis in Software Fault Prediction Models," in *Proceedings of the 20th International Symposium on Software Reliability Engineering*, Mysuru, India, November 2009, pp. 99-108

[118] Y. Jiang, J. Lin, B. Cukic, S. Lin, and Z. Hu, "Replacing Code Metrics in Software Fault Prediction with Early Life Cycle Metrics," in *Proceedings of the 3rd International Conference on Information Science and Technology*, Yangzhou, China, March 2013, pp. 516-523

[119] C. Jin, S. Jin, J. Ye, and Q. Zhang, "Quality Prediction Model of Object-Oriented Software System using Computational Intelligence," in *Proceedings of the 2nd International Conference on Power Electronics and Intelligent Transportation System*, Shenzhen, China, December 2009, pp. 120-123

[120] X. Jing, S. Ying, Z. Zhang, S. Wu, and J. Liu, "Dictionary Learning Based Software Defect Prediction," in *Proceedings of the 36th International Conference on Software Engineering*, Hyderabad, India, pp. 414-423, June 2014

[121] JIRA, https://www.atlassian.com/software/jira

[122] T. Kamiya, S. Kusumoto, and K. Inoue, "Prediction of Fault-Proneness at Early Phase in Object-Oriented Development," in *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Saint-Malo, France, May 1999, pp. 253-258

[123] S. Kanmani, V. R. Uthariaraj, V. Sankaranarayanan, and P. Thambidurai, "Object-Oriented Software Fault Prediction Using Neural Networks," *Information and Software Technology*, vol. 49, no. 5, pp. 483-492, May 2007

[124] A. Kaur and R. Malhotra, "Application of Random Forest in Predicting Fault-Prone Classes," in *Proceedings of the 1st International Conference on Advanced Computer Theory and Engineering, Phuket Island*, Thailand, December 2008, pp. 37-43

[125] T. M. Khoshgoftaar and E. B. Allen, "Empirical Assessment of a Software Metric: The Information Content of Operators," *Software Quality Journal*, vol. 9, no. 2, pp. 99-112, 2001

[126] T. M. Khoshgoftaar and E. B. Allen, "Classification of Fault-Prone Software Modules: Prior Probabilities, Costs, and Model Evaluation," *Empirical Software Engineering*, vol. 3, no. 3, pp. 275-298, 1998

[127] T. M. Khoshgoftaar and E. B. Allen, "Controlling Overfitting in Classification-Tree Models of Software Quality," *Empirical Software Engineering*, vol. 6, no. 1, pp. 59-79, 2001

[128] T. M. Khoshgoftaar, E. B. Allen, J. P. Hudepohl, and S. J. Aud, "Application of Neural Networks to Software Quality Modeling of a Very Large Telecommunications System," *IEEE Transactions on Neural Networks*, vol. 8, no. 4, pp. 902-909, July 1997

[129] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl, "Classification-Tree Models of Software-Quality Over Multiple Releases," *IEEE Transactions on Reliability*, vol. 49, no. 1, pp. 4-11, March 2000

[130] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, and N. Goel, "Early Quality Prediction: A Case Study in Telecommunications," *IEEE Software*, vol. 13, no. 1, pp. 65-71, January 1996

[131] T. M. Khoshgoftaar, E. B. Allen, A. Naik, W. D. Jones, and J. P. Huderpohl, "Using Classification Trees for Software Quality Models: Lessons Learned," *International Journal of Software Engineering and Knowledge Engineering*, vol. 9, no. 2, pp. 217-231, April 1999

[132] T. M. Khoshgoftaar and K. Gao, "Count Models for Software Quality Estimation," *IEEE Transactions on Reliability*, vol. 56, no. 2, pp. 212-222, June 2007

[133] T. M. Khoshgoftaar, D. L. Lanning, and A. S. Pandya, "A Comparative Study of Pattern Recognition Techniques for Quality Evaluation of Telecommunications Software," *IEEE Journal on Selected Areas in Communications*, vol. 12, no. 2, pp. 279-291, February 1994

[134] T. M. Khoshgoftaar, A. S. Pandya, and D. L. Lanning, "Application of Neural Networks for Predicting Program Faults," *Annals of Software Engineering*, vol. 1, no. 1, pp. 141-154, 1995

[135] T. M. Khoshgoftaar and N. Seliya, "Comparative Assessment of Software Quality Classification Techniques: An Empirical Study," *Empirical Software Engineering*, 9, pp. 229-257, September 2004

[136] T. M. Khoshgoftaar, R. Shan, and E. B. Allen, "Using Product, Process, and Execution Metrics to Predict Fault-Prone Software Modules with Classification Trees," in *Proceedings of the 5th International Symposium on High Assurance Systems Engineering*, Albuquerque, New Mexico, November 2000, pp. 301-310

[137] T. M. Khoshgoftaar and R. M. Szabo, "Using Neural Networks to Predict Software Faults During Testing," *IEEE Transactions on Reliability*, vol. 45, no. 3, pp. 456-462, September 1996

[138] T. M. Khoshgoftaar, X. Yuan, E. B. Allen, W. D. Jones, and J. P. Hudepohl, "Uncertain Classification of Fault-Prone Software Modules," *Empirical Software Engineering*, vol. 7, no. 4, pp. 297-318, 2002

[139] S. Kim, E. J. Whitehead Jr., and Y. Zhang, "Classifying Software Changes: Clean or Buggy?," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181-196, March 2008

[140] S. Kim, T. Zimmermann, and E. J. Whitehead Jr., "Predicting Faults from Cached History," in *Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, May 2007, pp. 489-498

[141] P. Knab, M. Pinzger, and A. Bernstein, "Predicting Defect Densities in Source Code Files with Decision Tree Learners," in *Proceedings of the 4th International Workshop on Mining Software Repositories*, Shanghai, China, May 2006, pp. 119-125

[142] A. G. Koru, K. El Emam, D. Zhang, H. Liu, and D. Mathew, "Theory of Relative Defect Proneness," *Empirical Software Engineering*, vol. 13, no. 5, pp. 473-498, October 2008

[143] A. G. Koru, D. Zhang, K. El Emam, and H. Liu, "An Investigation into the Functional Form of the Size-Defect Relationship for Software Modules," *IEEE Transactions on Software Engineering*, vol. 35, no. 2, pp. 290-304, April 2009

[144] A. G. Koru and J. Tian, "An Empirical Comparison and Characterization of High Defect and High Complexity Modules," *Journal of System and Software*, vol. 67, no. 3, pp. 153-163, September 2003

[145] A. G. Koru and J. Tian, "Comparing High-Change Modules and Modules with the Highest Measurement Values in Two Large-Scale Open-Source Products," *IEEE Transactions on Software Engineering*, vol. 31, no. 8, pp. 625-642, August 2005

[146] R. Kumar, S. Rai, and J. L. Trahan, "Neural-Network Techniques for Software Quality Evaluation," in *Proceedings of the 1998 Annual Reliability and Maintainability Symposium,* January 1998, pp. 155-161

[147] I. Kwan, A. Schroter, and D. Damian, "Does Socio-Technical Congruence Have an Effect on Software Build Success? A Study of Coordination in a Software Project," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 307-324, May 2011

[148] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the Severity of a Reported Bug," in *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, Cape Town, South Africa, May 2010, pp. 1-10

[149] L. Layman, G. Kudrjavets, and N. Nagappan, "Iterative Identification of Fault-Prone Binaries Using In-Process Metrics," in *Proceedings of the 2nd ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, Kaiserslautern, Germany, October 2008, pp. 206-212

[150] S. Lee and Y. Li, "DRS: A Developer Risk Metric for Better Predicting Software Fault-Proneness," in *Proceedings of the 2nd International Conference on Trustworthy Systems and Their Applications*, Hualien ,Taiwan, July 2015, pp. 120-127

[151] B. Lennselius, C. Wohlin, and C. Vrana, "Software Metrics: Fault Content Estimation and Software Process Control," *Microprocessors and Microsystems*, vol. 7, no. 35, September 1987, pp. 365-375

[152] H. Li and W. K. Cheung, "An Empirical Study of Software Metrics," *IEEE Transactions on Software Engineering*, vol. 13, no. 6, pp. 697-708, June 1987

[153] W. Li and S. Henry, "Object-Oriented Metrics that Predict Maintainability," *Journal of Systems and Software*, vol. 23, no. 2, pp. 111-122, November 1993

[154] M. Lipow, "Number of Faults per Line of Code," *IEEE Transactions on Software Engineering*, vol. 8, no. 4, pp. 437-439, July 1982

[155] H. Lu and B. Cukic, "An Adaptive Approach with Active Learning in Software Fault Prediction," in *Proceedings of the 8th International Conference on Predictive Models in Software Engineering*, Lund, Sweden, September 2012, pp. 79-88

[156] S. Matsumoto, Y. Kamei, A. Monden, K. Matsumoto, and M. Nakamura, "An Analysis of Developer Metrics for Fault Prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, Timisoara, Romania, September 2010

[157] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308-320, March 1976

[158] A. Meneely and L. Williams, "Socio-Technical Developer Networks: Should We Trust Our Measurements?," in *Proceedings of the 33rd International Conference on Software Engineering*, Honolulu, HI, USA, May 2011, pp. 281-290

[159] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting Failures with Developer Networks and Social Network Analysis," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Atlanta, GA, USA, November 2008, pp. 13-23

[160] T. Menzies, 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, January 2007

[161] T. Menzies and J. Di Stefano, "How Good is Your Blind Spot Sampling Policy?," in *Proceedings of the 8th International Symposium on High Assurance Systems Engineering*, Tampa, FL, USA, March 2004, pp. 129-138

[162] T. Menzies, J. Di Stefano, M. Chapman, and K. McGill, "Metrics that Matter," in *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop*, Greenbelt, Maryland, USA, December 2002, pp. 51-57

[163] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect Prediction from Static Code Features: Current Results, Limitations, New Approaches," *Automatic Software Engineering*, vol. 17, no. 4, pp. 375-407, December 2010

[164] A. Mitchell and J. F. Power, "Run-time Cohesion Metrics: An Empirical Investigation," in *Proceedings of the 2nd International Conference on Software Engineering Research and Practice*, Oslo, Norway, June 2004, pp. 532-537

[165] A. Mitchell and J. F. Power, "An Approach to Quantifying the Runtime Behaviour of Java GUI Applications," in *Proceedings of the 2004 Winter International Symposium on Information and Communication Technologies*, Cancun, Mexico, January 2004, pp.1-6

[166] P. Mittal, S. Singh, and K. S. Kahlon, "Identification of Error Prone Classes for Fault Prediction Using Object Oriented Metrics," *Advances in Computing and Communications*, pp. 58-68. Springer Berlin Heidelberg, 2011

[167] O. Mizuno and Y. Hirata, "A Cross-Project Evaluation of Text-based Fault-Prone Module Prediction," in *Proceedings of the 6th International Workshop on Empirical Software Engineering in Practice*, Osaka, Japan, November 2014, pp. 43-48

[168] O. Mizuno, S. Ikami, S. Nakaichi, and T. Kikuno, "Fault-Prone Filtering: Detection of Fault-Prone Modules Using Spam Filtering Technique," in *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement*, Madrid, Spain, September 2007, pp. 374-383

[169] O. Mizuno, S. Ikami, S. Nakaichi, and T. Kikuno, "Spam Filter Based Approach for Finding Fault-Prone Software Modules," in *Proceedings of the 4th International Workshop on Mining Software Repositories*, Minneapolis, MN, USA, May 2007

[170] O. Mizuno and T. Kikuno, "Training on Errors Experiment to Detect Fault-Prone Software Modules by Spam Filter," in *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia, September 2007, pp. 405-414

[171] A. Mockus, "Organizational Volatility and Its Effects on Software Defects," in *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Santa Fe, NM, USA, November 2010, pp. 117-126

[172] R. Moser, W. Pedrycz, and G. Succi, "A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction," in *Proceedings of the 30th International Conference on Software Engineering*, Leipzig, Germany, May 2008, pp. 181-190

[173] S. Morasca and G. Ruhe, "A Hybrid Approach to Analyze Empirical Software Engineering Data and Its Application to Predict Module Fault-Proneness in Maintenance," *Journal of Systems and Software*, vol. 53, no. 3, pp. 225-237, September 2000

[174] J. C. Munson and S. Elbaum, "Code Churn: A Measure for Estimating the Impact of Code Change," in *Proceedings of the 13th International Conference on Software Maintenance*, Bethesda, MD, USA, November 1998, pp. 24-31

[175] J. C. Munson and T. M. Khoshgoftaar, “The Detection of Fault-Prone Programs,” *IEEE Transactions on Software Engineering*, vol. 18, no. 5, pp. 423-433, May 1992

[176] I. Myrtveit, E. Stensrud, and M. Shepperd, “Reliability and Validity in Comparative Studies of Software Prediction Models,” *IEEE Transactions on Software Engineering*, vol. 31, no. 5, pp. 380-391, May 2005

[177] N. Nagappan and T. Ball, “Use of Relative Code Churn Measures to Predict System Defect Density,” in *Proceedings of the 27th International Conference on Software Engineering*, Shanghai, China, May 2005, pp. 284-292

[178] N. Nagappan, T. Ball, and A. Zeller, “Mining Metrics to Predict Component Failures,” in *Proceedings of the 28th International Conference on Software Engineering*, Shanghai, China, May 2006, pp. 452-461

[179] N. Nagappan, B. Murphy, and V. Basili, “The Influence of Organizational Structure on Software Quality: An Empirical Case Study,” in *Proceedings of the 30th International Conference on Software Engineering*, Leipzig, Germany, May 2008, pp. 521-530

[180] T. H. D. Nguyen, B. Adams, and A. E. Hassan, “Studying the Impact of Dependency Network Measures on Software Quality,” in *Proceedings of the 26th International Conference on Software Maintenance*, Timișoara, Romania, September 2010, pp. 1-10

[181] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Boehm, “A SLOC Counting Standard,” in *Proceedings of the 22nd International Forum on COCOMO and Systems/Software Cost Modeling*, Los Angeles, CA, USA, October 2007

[182] A. Nikora and J. Munson, “The Effects of Fault Counting Methods on Fault Model Quality,” in *Proceedings of the 28th Annual International Computer Software and Applications Conference*, Hong Kong, China, September 2004, pp. 192-201

[183] A. Nugroho, M. Chaudron, and E. Arisholm, “Assessing UML Design Metrics for Predicting Fault-Prone Classes in A Java System,” in *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, Cape Town, South Africa, May 2010, pp. 21-30

[184] M. Ohira, N. Ohsugi, T. Ohoka, and K. Matsumoto, “Accelerating Crossproject Knowledge Collaboration Using Collaborative Filtering and Social Networks,” in *Proceedings of the 2nd International Workshop on Mining Software Repositories*, Saint Louis, MO, USA, May 2005, pp. 1-5

[185] H. M. Olague, L. H. Etzkon, S. Gholston, and S. Quattlebaum, “Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes

Developed Using Highly Iterative or Agile Software Development Processes," *IEEE Transactions on Software Engineering*, vol. 33, no. 6, pp. 402-419, June 2007

[186] N. Ohlsson, M. Zhao, and M. Helander, "Application of Multivariate Analysis for Software Fault Prediction," *Software Quality Journal*, vol. 7, no.1, pp. 51-66, 1998

[187] A. Okutan and O. T. Yildiz, "Software Defect Prediction using Bayesian Networks," *Empirical Software Engineering*, vol.19, no. 1, pp. 154-181, February 2014

[188] H. M. Olague, L. H. Etzkon, S. Gholston, and S. Quattlebaum, "Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes," *IEEE Transactions on Software Engineering*, vol. 33, no. 6, pp. 402-419, June 2007

[189] A. D. Oral and A. B. Bener, "Defect Prediction for Embedded Software," in *Proceedings of the 22nd International Symposium on Computer and Information Sciences*, Ankara, Turkey, November 2007, pp. 1-6

[190] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the Bugs Are," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Boston, Massachusetts, USA, July 2004, pp. 86-96

[191] T. J. Ostrand, 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, April 2005

[192] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Programmer-based Fault Prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, Timisoara, Romania, no. 9, September 2010

[193] E. Otte and R. Rousseau, "Social network analysis: a powerful strategy, also for the information sciences," *Journal of Information Science*, vol. 28, no. 6, pp. 441–453, December 2002

[194] T. D. Oyetoyan, D. S. Cruzes, and R. Conradi, "A Study of Cyclic Dependencies on Defect Profile of Software Components," *Journal of Systems and Software*, vol. 86, no. 12, July 2013, pp. 3162-3182

[195] G. J. Pai and J. B. 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, October 2007

[196] A. Paksoya and M. Göktürka, "Information Fusion with Dempster-Shafer Evidence Theory for Software Defect Prediction," *Procedia Computer Science*, vol. 3, pp. 600-605, March 2011

[197] A. K. Pandey and N. K. Goyal, "Predicting Fault-prone Software Module Using Data Mining Technique and Fuzzy Logic," *International Journal of Computer and Communication Technology*, vol. 2, no. 2, pp. 56-63, December 2010

[198] P. Pendharkar, "Exhaustive and Heuristic Search Approaches for Learning a Software Defect Prediction Model," *Engineering Applications of Artificial Intelligence*, vol. 23, no. 1, pp. 34-40, February 2010

[199] F. Peters, T. Menzies, and A. Marcus, "Better Cross Company Defect Prediction," in *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories*, San Francisco, CA, May 2013, pp. 409-418

[200] M. Pinzger, N. Nagappan, and B. Murphy, "Can Developer-Module Networks Predict Failures?," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Atlanta, GA, USA, November 2008, pp. 2-12

[201] J. R. Quinlan, "Induction of Decision Tress," *Machine Learning*, vol. 1, no. 1, pp. 81-106, March 1986

[202] D. Radjenovic, M. Hericko, R. Torkar, and A. Zivkovic, "Software Fault Prediction Metrics: A Systematic Literature Review," *Information and Software Technology*, vol. 55, no. 8, pp. 1397-1418, August 2013

[203] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the Imprecision of Cross-Project Defect Prediction," in *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Cary, NC, USA, November 2012

[204] F. Rahman and P. Devanbu, "How, and Why, Process Metrics Are Better," in *Proceedings of the 35th International Conference on Software Engineering*, San Francisco, May 2013, pp. 432-441

[205] S. S. Rathore and A. Gupta, "Investigating Object-Oriented Design Metrics to Predict Fault-Proneness of Software Modules," in *Proceedings of the CSI Sixth International Conference on Software Engineering*, Madhay Pradesh, India, September 2012, pp. 1-10

[206] S. S. Rathore and A. Gupta, "Validating the Effectiveness of Object-Oriented Metrics over Multiple Releases for Predicting Fault Proneness," in *Proceedings of the 19th Asia-Pacific Software Engineering Conference*, Hong Kong, China, December 2012, pp. 350-355

[207] D. Rodriguez, R. Ruiz, J. C. Riquelme, and R. Harrison, “A Study of Subgroup Discovery Approaches for Defect Prediction,” *Information and Software Technology*, vol. 55, no. 10, pp. 1810-1822, May 2013

[208] A. Sarma, L. Maccherone, P. Wagstrom, and J. Herbsleb, “Tesseract: Interactive Visual Exploration of Socio-Technical Relationships in Software Development,” in *Proceedings of the 31st IEEE International Conference on Software Engineering*, Vancouver, Canada, May 2009, pp. 22-33

[209] G. Scanniello, C. Gravino, A. Marcus, and T. Menzies, “Class Level Fault Prediction using Software Clustering,” in *Proceedings of the 28th International Conference on Automated Software Engineering*, Silicon Valley, CA, November 2013, pp. 640-645

[210] N. Seliya and T. M. Khoshgoftaar, “Software Quality Analysis of Unlabeled Program Modules with Semi-Supervised clustering,” *IEEE Transactions On Systems, Man, and Cybernetics-Part A: Systems And Humans*, vol. 37, no. 2, pp. 201-211, March 2007

[211] R. Shatnawi and W. Li, “The Effectiveness of Software Metrics in Identifying Error-Prone Classes in Post-Release Software Evolution Process,” *Journal of Systems and Software*, vol. 81, no. 11, pp. 1868-1882, November 2008

[212] V. Y. Shen, T. Yu, S. M. Thebaut, and L. R. Paulsen, “Identifying Error-Prone Software-An Empirical Study,” *IEEE Transactions on Software Engineering*, vol. 11, no. 4, pp. 317-324, April 1985

[213] M. Shepperd, D. Bowes, and Tracy Hall, “Research Bias: The Use of Machine Learning in Software Defect Prediction,” *IEEE Transactions on Software Engineering*, vol. 40, no. 6, pp. 603-616, June 2014

[214] M. Shepperd and D. Ince, *Derivation and Validation of Software Metrics*. Oxford, United Kingdom: Clarendon Press, 1993

[215] M. Shepperd and G. Kadoda, “Comparing Software Prediction Techniques Using Simulation,” *IEEE Transactions on Software Engineering*, vol. 27, no. 11, pp. 1014-1022, November 2001

[216] E. Shihab, Z. Jiang, W. M. Ibrahim, B. Adams, and A. E. Hassan, “Understanding the Impact of Code and Process Metrics on Post-Release Defects: A Case Study on the Eclipse Project,” in *Proceedings of the 4th IEEE/ACM International Symposium on Empirical Software Engineering and Measurement*, Bozen, Italy, September 2010

[217] Y. Shin, R. M. Bell, T. J. Ostrand, and E. J. Weyuker, “On the Use of Calling Structure Information to Improve Fault Prediction,” *Empirical Software Engineering*, vol. 17, no. 4, pp. 390-423, August 2012

[218] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, “Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities,” *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772-787, December 2011

[219] A. Simpson, “Changeset based Developer Communication to Detect Software Failures,” in *Proceedings of the 35th IEEE International Conference on Software Engineering*, San Francisco, CA, USA, May 2013, pp. 1468-1470

[220] S. Singh and K. S. Kahlon, “Effectiveness of Encapsulation and Object-Oriented Metrics to Refactor Code and Identify Error Prone Classes Using Bad Smells,” *ACM SIGSOFT Software Engineering Notes*, vol. 36, no. 5, pp. 1-10, September 2011

[221] S. Singh and K. S. Kahlon, “Effectiveness of Refactoring Metrics Model to Identify Smelly and Error Prone Classes in Open Source Software,” *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 2, pp. 1-11, March 2012

[222] Y. Singh, A. Kaur, and R. Malhotra, “Empirical Validation of Object-Oriented Metrics for Predicting Fault Proneness Models,” *Software Quality Journal*, vol. 18, no. 1, pp. 3-35, March 2010

[223] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu, “A General Software Defect-Proneness Prediction Framework,” *IEEE Transactions Software Engineering*, vol. 37, no. 3, pp. 356-370, June 2011

[224] S. Spearman, “The Proof and Measurement of Association between Two Things,” *American Journal of Psychology*, vol. 15, no.1, pp. 72-101, January 1904

[225] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan, “Predicting Bugs Using Antipatterns,” in *Proceedings of the 29th IEEE International Conference on Software Maintenance*, Eindhoven, The Netherlands, pp. 270-279, September 2013

[226] R. Takahashi, “Software Quality Classification Model based on McCabe’s Complexity Measure,” *Journal of Systems and Software*, vol. 38, no. 1, pp. 61-69, July 1997

[227] M. Tang, M. Kao, and M. Chen, “An Empirical Study on Object-Oriented Metrics,” in *Proceedings of the 6th International Software Metrics Symposium*, Boca Raton, FL. USA, November 1999, pp. 242-249

[228] M. M. T. Thwin and T. Quah, “Application of Neural Networks for Software Quality Prediction Using Object-Oriented Metrics,” *Journal of Systems and Software*, vol. 76, no. 2, pp. 147-156, May 2005

[229] J. Tian and J. Troster, "A Comparison of Measurement and Defect Characteristics of New and Legacy Software Systems," *Journal of Systems and Software*, vol. 44, no. 2, pp. 135-146, December 1998

[230] P. Tomaszewski, L. Lundberg, and H. Grahn, "Improving Fault Detection in Modified Code: A Study from the Telecommunication Industry," *Journal of Computer Science and Technology*, vol. 22, no. 3, pp. 397-409, May 2007

[231] P. Tomaszewski, J. Hakansson, H. Grahn, and L. Lundberg, "Statistical Models vs. Expert Estimation for Fault Prediction in Modified Code−An Industrial Case Study," *Journal of Systems and Software*, vol. 80, no. 8, pp. 1227–1238, August 2007

[232] A. Tosun, B. Turhan, and A. Bener, "Validation of Network Measures as Indicators of Defective Modules in Software Systems," in *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, Vancouver, Canada, May 2009

[233] A. Tosun, A. Bener, B. Turhan, and T. Menzies, "Practical Considerations in Deploying Statistical Methods for Defect Prediction: A Case Study within the Turkish Telecommunications Industry," *Information and Software Technology*, vol. 52, no. 1, pp. 1242-1257, November 2010

[234] B. Turhan, T. Menzies, A. Bener, and J. Distefano, "On the Relative Value of Cross-Company and Within-Company Data for Defect Prediction," *Empirical Software Engineering*, vol. 14, no. 5, pp. 540-578, January 2009

[235] B. Turhan, A. T. Mısırlı, and A. Bener, "Empirical Evaluation of the Effects of Mixed Project Data on Learning Defect Predictors," *Information and Software Technology*, vol. 55, no. 6, pp. 1101-1118, June 2013

[236] B. Twala, "Predicting Software Faults in Large Space Systems Using Machine Learning Techniques," *Defence Science Journal*, vol. 61, no. 4, pp. 306-316, July 2011

[237] B. Újházi, R. Ferenc, D. Poshyvanyk, and T. Gyimóthy, "New Conceptual Coupling and Cohesion Metrics for Object-Oriented Systems," in *Proceedings of the 10th IEEE International Working Conference on Source Code Analysis and Manipulation*, Timişoara, Romania, pp. 33-42, September 2010

[238] Understand, https://scitools.com/

[239] S. Wasserman and K. Faust, *Social Network Analysis: Methods and Applications*, Cambridge University Press, New York, 1994

[240] WEKA, http://www.cs.waikato.ac.nz/ml/weka/

[241] E. J. Weyuker and T. J. Ostrand, "What Can Fault Prediction Do For You?," *Tests and Proofs*, Springer Berlin Heidelberg, pp. 18-29, 2008

[242] E. J. Weyuker, T. J. Ostrand, and R. M. Bell, "Using Developer Information as A Factor for Fault Prediction," in *Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering*, Minneapolis, MN, USA, May 2007

[243] E. J. Weyuker, T. J. Ostrand, and R. M. 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, October 2008

[244] E. J. Weyuker, T. J. Ostrand, and R. M. Bell, "Comparing the Effectiveness of Several Modeling Methods for Fault Prediction," *Empirical Software Engineering*, vol. 15, no. 3, pp. 277-295, June 2010

[245] T. Wolf, A. Schroter, D. Damian, and T. Nguyen, "Predicting Build Failures Using Social Network Analysis on Developer Communication," in *Proceedings of the 31st International Conference on Software Engineering*, Vancouver, Canada, May 2009, pp. 1-11

[246] W. E. Wong, V. Debroy, and B. Choi, "A Family of Code Coverage-based Heuristics for Effective Fault Localization," *Journal of Systems and Software*, vol. 83, no. 2, pp. 188-208, February 2010

[247] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The DStar Method for Effective Software Fault Localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290-308, March 2014

[248] W. E. Wong and S. Gokhale, "Static and Dynamic Distance Metrics for Feature-Based Code Analysis," *Journal of Systems and Software*, vol. 74, no. 3, pp. 283-295, February 2005

[249] W. E. Wong, S. Gokhale, J. Horgan, "Metrics for Quantifying the Disparity, Concentration, and Dedication between Program Components and Features," *IEEE METRICS*, 1999

[250] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham, "Effective Software Fault Localization using an RBF Neural Network," *IEEE Transactions on Reliability*, vol. 61, no. 1, pp. 149-169, March 2012

[251] W. E. Wong, V. Debroy, and D, Xu, "Towards Better Fault Localization: A Crosstab-based Statistical Approach," *IEEE Transactions on Systems, Man, and Cybernetics − Part C: Applications & Reviews*, vol. 42, no. 3, pp. 378-396, May 2012

[252] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A Survey on Software Fault Localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707-740, August 2016

[253] W. E. Wong, J. R. Horgan, M. Syring, W. M. Zage, and D. M. Zage, "Applying Design Metrics to Predict Fault-Proneness: A Case Study on Large-Scale Software System," *Software: Practice and Experience*, vol. 30, no .14, pp. 1587-1608, November 2000

[254] W. E. Wong, Y. Qi, and K. Cooper, "Source Code-Based Software Risk Assessing," in *Proceedings of the 20th ACM Symposium on Applied Computing*, Santa Fe, NM, USA, March 2005, pp. 1485-1490

[255] W. E. Wong, J. Zhao, V. K. Y. Chan, "Applying Statistical Methodology to Optimize and Simplify Software Metric Models with Missing Data," in *Proceedings of the 21st Annual ACM Symposium on Applied Computing*, Dijon, France, April 2006, pp. 1728-1733

[256] F. Wu, "Empirical Validation of Object-Oriented Metrics on NASA for Fault Prediction," *Advances in Information Technology and Education*, pp. 168-175. Springer Berlin Heidelberg, 2011

[257] Y. Wu, Y. Yang, Y. Zhao, H. Liu, Y. Zhou, and B. Xu, "The Influence of Developer Quality on Software Fault-Proneness Prediction," in *Proceedings of the 8th International Conference on Software Security and Reliability*, San Francisco, CA, June 2014, pp. 11-19

[258] F. Xing, P. Guo, and M. R. Lyu, "A Novel Method for Early Software Quality Prediction Based on Support Vector Machine," in *Proceedings of the 16th International Symposium on Software Reliability Engineering*, Chicago, Illinois, November 2005, pp. 213-222

[259] J. Xu, Y. Gao, S. Christley, and G. Madey, "A Topological Analysis of the Open Source Software Development Community," in *Proceedings of the 38th IEEE Annual Hawaii International Conference on System Sciences*, Hawaii, USA, January 2005, pp. 198-208

[260] Z. Xu, X. Zheng, and P. Guo, "Empirically Validating Software Metrics for Risk Prediction based on Intelligent Methods," *Journal of Digital Information Management*, vol. 5, no. 3, pp. 99-106, June 2007

[261] B. Yang, Q. Yin, S. Xu, and P. Guo, "Software Quality Prediction Using Affinity Propagation Algorithm," in *Proceedings of the 5th International Joint Conference on Neural Networks*, Hong Kong, China, June 2008, pp. 1891-1896

[262] P. Yu, H. Muller, and T. Systa, "Predicting Fault-Proneness Using OO Metrics: An Industrial Case Study," in *Proceedings of the 6th European Conference on Software Maintenance and Reengineering*, Budapest, Hungary, March 2002, pp. 99-107

[263] H. Y. Zhang, "An Investigation of the Relationships between Lines of Code and Defects," in *Proceedings of the 25th International Conference on Software Maintenance*, Alberta, Canada, September 2009, pp. 274-283

[264] J. Zheng, "Cost-sensitive Boosting Neural Networks for Software Defect Prediction," *Expert Systems with Applications*, vol. 37, no. 6, pp. 4537-4543, June 2010

[265] S. Zhong, T. M. Khoshgoftaar, and N. Seliya, "Unsupervised Learning for Expert-Based Software Quality Estimation," in *Proceedings of the 8th IEEE International Symposium on High-Assurance Systems Engineering*, Tampa, FL, USA, March 2004, pp. 149-155

[266] Y. Zhou and H. Leung, "Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults," *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 771-789, October 2006

[267] Y. Zhou, B. Xu, and H. Leung, "On the Ability of Complexity Metrics to Predict Fault-Prone Classes in Object-Oriented Systems," *Journal of Systems and Software*, vol. 83, no. 4, pp. 660-674, April 2010

[268] Y. Zhou, B. Xu, H. Leung, and L. Chen, "An In-Depth Study of the Potentially Confounding Effect of Class Size in Fault Prediction," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 1, article 10, February 2014

[269] J. Zhou, P. S. Sandhu, and S. Rani, "A Neural Network Based Approach for Modeling of Severity of Defects in Function Based Software Systems," in *Proceedings of 1st International Conference on Electronics and Information Engineering*, Kyoto, Japan, August 2010, pp. 568-575

[270] T. Zimmermann and N. Nagappan, "Predicting Subsystem Failures using Dependency Graph Complexities," in *Proceedings of the 18th International Symposium on Software Reliability Engineering*, Trollhättan, Sweden, November 2007, pp. 227-236

[271] T. Zimmermann and N. Nagappan, "Predicting Defects using Network Analysis on Dependency Graphs," in *Proceedings of the 30th International Conference on Software Engineering*, Leipzig, Germany, May 2008, pp. 531-540

[272] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-Project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, Amsterdam, The Netherlands, August 2009, pp. 91-100

BIOGRAPHICAL SKETCH

Yihao Li was born in Nanchang, Jiangxi, China. After completing his schoolwork at No.1 Railway Middle School in Nanchang in 2005, Yihao entered East China University of Technology (ECUT) in Nanchang. He received a Bachelor of Science with a major in software engineering from ECUT in 2009. During the following one and a half years, he studied at Southeastern Louisiana University in Hammond, Louisiana and received a Master of Science with a major in computer science. In August 2011, Yihao entered the Graduate School of The University of Texas at Dallas.

CURRICULUM VITAE

YIHAO LI

EDUCATION

Ph.D. in Computer Science, University of Texas at Dallas, USA *Aug. 2011 – Aug. 2017*

- Dissertation title: Applying Social Network Analysis to Software Fault-Proneness Prediction
- Supervisor: W. Eric Wong

Master of Science, Southeastern Louisiana, USA *Aug. 2009 – May. 2011*

- Thesis title: The K-MM Clustering Algorithm Based on K-Means and K-Medoids in Data Mining
- Supervisor: Theresa Beaubouef

B.S. in Software Engineering, East China University of Technology, China *Sep. 2005 – Jun. 2009*

RESEARCH INTERESTS

Software Fault-Proneness Prediction, Program Debugging, Quality Assurance, Metrics, Risk Analysis

SKILLS

Programming Language: Java, C#
Quality Assurance: JUnit, xSuds Toolsuite, HP LoadRunner, Understand, Ucinet,
Web Framework: .NET, Joomla
Database: MySQL, SQL Server

PROFESSIONAL EXPERIENCE

- **Academic Service**

– **External Reviewer**
- IEEE Transactions on Software Engineering (TSE)
- ACM Transactions on Software Engineering and Methodology (TOSEM)
- IEEE Transactions on Reliability (TR)
- Elsevier Journal of Systems and Software (JSS)
- Empirical Software Engineering (EMSE)
- Wiley Software Testing, Verification and Reliability (STVR)
- Springer Software Quality Journal (SQJ)
- IEEE International Symposium on Software Reliability Engineering (ISSRE)
- IEEE Computer Society International Conference on Computers, Software & Applications (COMPSAC)
- International Conference on Software Engineering & Knowledge Engineering (SEKE)

– **Conference Organizer**
- 2015, 2016, 2017 International Conference on Software Quality, Reliability, and Security (QRS)
- 2016 IEEE Conference on Software Engineering Education and Training (CSEE&T)
- 2014 IEEE International Conference on Quality Software (QISC)
- 2012, 2013, 2014 International Conference on Software Security and Reliability (SERE)
- 2012 IEEE International Symposium on Software Reliability Engineering (ISSRE)

– **Web Master**
- 2015, 2016, 2017 International Conference on Software Quality, Reliability, and Security (QRS)
- 2015 International Conference on Trustworthy Systems and Their Applications (TSA)
- 2012, 2013 International Conference on Software Security and Reliability (SERE)
- 2012 IEEE International Symposium on Software Reliability Engineering (ISSRE)

HONORS AND AWARDS

- IEEE Reliability Society Service Award
  2016, sponsored by IEEE Reliability Society Dallas Section
- Certificate of Appreciation for Computer Educators in Texas
  2016, sponsored by Association for Computer Educator in Texas
- Best Paper Award
  2012, sponsored by IEEE International Conference on Software Security and Reliability
- Best Student Paper Award
  2010, sponsored by Consortium for Computing Sciences in Colleges South Central Conference
- Best Undergraduate
  2009, sponsored by East China University of Technology
- Best Announcer
  2008, sponsored by East China University of Technology

ACADEMIC APPOINTMENTS

Research Assistant
Advanced Research Center for Software Testing and Quality Assurance, UTD

Teaching Assistant
CS6367: Software Testing
CS6362: Advanced Software Architecture
CS6388: Software Project Planning and Management
CS4485: Software Engineering Senior Project

PUBLICATIONS

Book

Chapter Author for "Handbook on Software Debugging− Foundations and Advances", by *Wiley-IEEE Press* (accepted for publication)

Journal Paper

- **Yihao Li**, Shou-Yu Lee, and W. Eric Wong, "Tri-Relation Network for Effective Software Fault-Proneness Prediction," *International Journal of Software Engineering and Knowledge Engineering* (under revision)
- W. Eric Wong, Ruizhi Gao, **Yihao Li**, Rui Abreu, and Franz Wotawa, "A Survey on Software Fault Localization," *IEEE Transactions on Software Engineering*, Volume 42, Issue 8, pp. 707-740, August 2016
- W. Eric. Wong, Vidroha Debroy, Ruizhi Gao, and **Yihao Li**, "The DStar Method for Effective Software Fault Localization," *IEEE Transactions on Reliability*, Volume 63, Issue 1, pp. 290-308, March 2014

Conference Paper

- **Yihao Li**, Dong Li, Fuqun Huang, Shou-Yu Lee, and Jun Ai, Chunhui Yang, and Dong Li, "An Exploratory Analysis on Software Developers' Bug-Introducing Tendency over Time," in *Proceedings of the 2016 Annual Conference on Software Analysis, Testing and Evolution*, pp.12-17, Kunming, China, November 2016
- Shou-Yu Lee, Dong Li, and **Yihao Li**, "An Investigation of Essential Topics on Software Fault-Proneness Prediction," in *Proceedings of the 2016 IEEE International Symposium on System and Software Reliability*, pp.37-46, Shanghai, China, October 2016

- Shou-Yu Lee and **Yihao Li**, "DRS: A Developer Risk Metric for Better Predicting Software Fault-Proneness," in *Proceedings of the 2015 International Conference on Trustworthy Systems and Their Applications*, pp.120-127, Hualien, Taiwan, July 2015
- W. Eric Wong, Vidroha Debroy, **Yihao Li**, and Ruizhi Gao, "Software Fault Localization Using DStar (D*)", in *Proceedings of International Conference on Software Security and Safety*, pp.21-30, Washington DC, USA, 2014 **(Best Paper Award)**
- **Yihao Li** and Theresa Beaubouef, "Data Mining: Concepts, Background, and Methods of Integrating Uncertainty in Data," in *Proceedings of the 2010 Consortium for Computing Sciences in Colleges (South Central Region)*, Austin, TX, April 2010 **(Best Student Paper Award)**