

AN ISSUE RECOMMENDER MODEL USING THE DEVELOPER
COLLABORATION NETWORK

by

Bora Çağlayan

B.S., in Mechanical Engineering, Bogazici University, 2006

M.S., in Software Engineering, Bogazici University, 2008

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

Graduate Program in Computer Engineering
Boğaziçi University

2013

AN ISSUE RECOMMENDER MODEL USING THE DEVELOPER
COLLABORATION NETWORK

APPROVED BY:

Prof. Ayşe Başar Bener
(Thesis Co-supervisor)

Prof. Oğuz Tosun
(Thesis Co-supervisor)

Prof. Fikret Gürgen

Prof. Stefan Koch

Prof. Levent Akın

Burak Turhan, Ph.D.

DATE OF APPROVAL: 13.05.2013

ACKNOWLEDGEMENTS

ABSTRACT

AN ISSUE RECOMMENDER MODEL USING THE DEVELOPER COLLABORATION NETWORK

Assignment of new issues to developers is an important part of software maintenance activities. In this research, we build an issue recommendation model that recommends new issues to developers and highlights the defect prone software modules to the developer who owns an issue. Existing solutions address the automated issue assignment problem through text mining. We propose a recommender model that uses the collaboration network of developers on software modules and the structured issue data as its input. We perform an exploratory analysis using the issue data of two large software systems and observe the trends in the issue ownership, issue defect relations and issue timelines. Our base model estimates the developer centrality for each issue category and recommends issues to developers based on their centrality ranking. We test the performance of our recommender using the maintenance data of a globally developed large enterprise software using recommendation accuracy and workload imbalance as the performance metrics. We extend the recommender to address, 1) The problem of developer workload imbalances, 2) The problem of assigning issues to a new group of developers by using stochastic Kronecker networks to model the future structure of the collaboration network. We change a learning based defect predictor's output based on recent history to update the predicted defect-prone software modules on a real-time basis. Our empirical analysis shows that 1) The performance of our recommender model approximates historical trends of issue allocation, 2) Heuristics applied to the model output reduces the issue ownership inequalities and model approximation performance, 3) Kronecker networks can be used to estimate the future collaboration on the issues and the model can be extended by them to propose issues to new developer groups, 4) Real time defect prediction model can significantly improve probability of detection over time while not changing false alarm rates.

ÖZET

YAZILIM GELİŞTİRİCİ ORTAK ÇALIŞMA AĞI KULLANAN HATA RAPORU ÖNERİ MODELİ

Yeni hata raporlarının geliştiricilere atanması yazılım bakım faaliyetleri arasında önemli bir yer kaplamaktadır. Bu araştırma kapsamında, yeni gelen hata raporlarını kişilere atayan ve hataları düzeltecek yazılım geliştiricilere hataya yatkın yazılım kodu modüllerini tahmin eden bir hata raporu öneri modeli geliştirilmiştir. Varolan çözümler bu soruna metin madenciliği metoduyla yaklaşmaktadır. Bu araştırmadaki öneri modelinde ise yazılım modülleri üzerindeki çalışma ağı verisi ve düzenli hata raporu verisi kullanılmaktadır. İki büyük ölçekli yazılım üzerinde hata raporu hata ilişkileri, hata raporu iş yükü ve hata raporu zaman ilişkisi incelenmiştir. Baz modelde her hata raporu kategorisinde geliştirici merkeziliği üzerinden hata raporlarına geliştirici önerilmektedir. Önerilen modelin performansı küresel olarak geliştirilen büyük ölçekli bir kurumsal yazılım üzerinde tahmin doğruluğu ve iş yükü dağılımı dengesizliği başarımlarına göre sınanmıştır. Model, 1)Yazılım geliştirici iş yükü dengesizliği, 2)Yazılım ekiplerine yeni katılan geliştirici gruplarına hata raporu atanması sorunlarına çözüm üretmesi için genişletilmiştir. Öğrenme tabanlı bir hata tahmin modelinin tahminleri yakın geçmişi göz önüne alarak gerçek zamanlı olarak güncellenmiştir. Yaptığımız deneysel analize göre, 1) Önerilen model hata raporu atamalarının tarihsel dağılımına yakınsamaktadır, 2) Modelimize eklediğimiz buluşsal yöntemler ile geliştirici iş yükü dağılımı daha dengeli hale getirilebilmektedir, 3) Kronecker ağırları yardımıyla ortak çalışmanın ileride nasıl bir yapıda olacağı tahmin edilerek azılım ekiplerine yeni katılan geliştirici gruplarına hata raporu atanabilmektedir, 4) Gerçek zamanlı hata tahmin modeliyle zaman içerisinde yanlış alarm oranları arttırılmadan hata yakalama performansı kayda değer miktarda artmaktadır.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	iii
ABSTRACT	iv
ÖZET	v
LIST OF FIGURES	ix
LIST OF TABLES	xii
LIST OF ACRONYMS/ABBREVIATIONS	xiii
1. Introduction	1
1.1. Software Networks	2
1.2. Issue Recommendation System	3
2. Problem Definition	5
2.1. Understanding the Structure of Software Networks	6
2.2. Recommendation System For Issue Management	7
3. Background	9
3.1. Issue Management	9
3.2. Defect Prediction In Software Engineering	11
3.2.1. Metric Sets Used In Defect Prediction Studies	12
3.2.1.1. Static Code Metrics	12
3.2.1.2. Churn (Repository) Metrics	14
3.2.1.3. Social Network Metrics	15
3.2.1.4. Confirmation Bias Metrics	16
3.2.2. Limitations of Defect Prediction Models	16
3.3. Scale Free Networks	17
3.3.1. Properties of Scale-Free Networks	17
3.3.1.1. Static Properties	17
3.3.1.2. Temporal Properties	18
3.4. Recommendation Systems	20
3.5. Task Management	21
3.6. Literature Survey on Software Networks	22
3.6.1. Search Criteria	23

3.6.2. Survey Results	23
4. Proposed Solution: Collaboration Based Issue Recommender	29
5. Methodology	32
5.1. Datasets	32
5.1.1. Enterprise Software Dataset	32
5.1.2. Android Dataset	32
5.2. Software Network	33
5.2.1. Call-Graphs of Software Modules	33
5.2.2. Collaboration Network	33
5.3. Exploratory Data Analysis	34
5.4. Issue Recommender System	35
5.4.1. Issue Recommendation Model	35
5.4.1.1. Proposed Model	35
5.4.1.2. Performance Measures	37
5.4.2. Extensions to The Basic Issue Recommender Model	38
5.4.2.1. Issue Workload Balance Problem	38
5.4.2.2. Issue Dependency Problem	42
5.4.2.3. Usage of Kronecker Graph to Address Developer Group Initiation Problem	42
5.4.3. Real Time Defect Prediction	47
5.4.3.1. Proposed Model	47
5.4.3.2. Performance Measures	50
6. Experiments and Results	52
6.1. Descriptive Statistics	52
6.1.1. Collaboration Network Analysis	55
6.2. Patterns of Issue Timeline and Ownerships and Issues Relation With Defects In Code	58
6.2.1. Hypothesis I: Limited number of developers own majority of issues	58
6.2.2. Hypothesis II: Issue reports of users change significantly after releases.	60
6.2.3. Hypothesis III: Issues map to multiple defects in source code . .	60

6.2.4.	Hypothesis IV: Issue resolve times are dependent on number of issues that a developer own	61
6.2.5.	Discussion of The Hypothesis Findings	61
6.3.	Recommendation System Component Results	70
6.3.1.	Issue Recommendation Results	70
6.3.2.	Base Model	70
6.3.3.	Workload Balance Problem	71
6.3.4.	Dependency Problem	71
6.3.5.	Assignment of Issues to New Developers Using Kronecker Network	72
6.4.	Real Time Prediction Results	73
7.	Threats to Validity	75
8.	CONCLUSIONS AND FUTURE WORK	79
8.1.	Summary of Results	79
8.1.1.	How can we assign issues to developers?	80
8.1.2.	How can developers more efficiently organize their time in solving the issues?	82
8.2.	Contributions	82
8.2.1.	Theoretical and Methodological Contributions	82
8.2.2.	Practical Contributions	84
8.3.	Future Directions	86
APPENDIX A:	Collaboration Network Descriptive Statistics	87
APPENDIX B:	Software Metrics	91
REFERENCES	99

LIST OF FIGURES

2.1	<i>Software network</i> is represented by the combination of its two underlying networks. a) <i>Collaboration network</i> models the interaction among developers, b) <i>Information network</i> models the relations between software components.	6
2.2	The recommendation system and its two components. Component A recommends an issue that is most relevant to a developer. Component B recommends the part of source code that possibly contains a defect related to that issue.	8
3.1	Degree Distribution of the Nodes in The Enterprise Software Dataset Collaboration Network with log-normal scale	18
3.2	Degree Distribution of the Nodes in Enterprise Software Collaboration Network with log-normal scale with top 80 nodes: notice a closer fit	19
3.3	Task Complexity and Context Variability vs Planning Opportunities [1]	22
5.1	Summary of The Issue Recommender System	36
5.2	Lorenz Curves for income and issue ownership distributions	40
5.3	One iteration of the adjecancy matrix with Stochastic Kronecker genarator[2]	44
5.4	KronFit Algorithm [2]	45
5.5	The productivity change during a member change in a software development team. Notice that the productivity actually goes negative in the beginning [3].	46
5.6	In the above generated graph blue nodes represent existing developers and gray nodes represent new developers. By parametrizing the existing developer collaboration using Kronfit algorithm and generating a network based on the number of existing developers plus new developers we can estimate the future state of collaboration	47

5.7	Reweighting method of real time defect prediction. Initially E_h weight is set to 0 and we exclusively use the classifier output. After module or its neighbors in the call-graph changes for a number of times we re-weight E_h as number of changes due to defects over number of all changes in the module or its neighbors.	50
6.1	Issue owner and originator locations for the enterprise software . .	56
6.2	State Diagram of the Issues For The Enterprise Software.	56
6.3	Changes in issue category percentages of the enterprise software over time. Dashed line is the overall probability of different categories of issues. Changing lines are the distribution of issue categories for that quarter. Bars are the number of issues reported in a given quarter. Note the decrease in issue reports by system test teams before release. After the release (2009-07) relative number of field issues increase rapidly. The most active quarter is the 2009-02 in terms of issues that were reported. The product is released just after this quarter.	58
6.4	In this figure we see that for three categories of issue in the enterprise software and Android a similar trend can be observed. The distribution of issue ownerships among developers comply with 80-20 Pareto law. Note that for Android issues can be categorized as field issues since they were reported by customers.	64
6.5	Issue resolve of each developer over time for Enterprise Software. Highlighted lines are time periods in which a developer had at least one issue open. Red and green ticks are issue resolve and ownership times respectively.	65
6.6	Issue resolve of each developer over time for Android. Highlighted lines are time period in which a developer had at least one issue open. Red and green ticks are issue resolve and ownership times respectively.	66
6.7	Box plots of number of issues owned for enterprise software and Android. For the Enterprise Software ownership of issues in three distinct categories are also plotted.	67

6.8	Change in number of issues that are actively worked on during the timespan. Green vertical lines identify release dates of the software. In the enterprise software trends for different categories of bugs were observed separately.	68
6.9	Average issue resolve times over number of issues owned by developers.	69
6.10	pd, pf and f-measure changes for Enterprise Software dataset over time for each month after the model is run using the real time defect prediction model	74
A.1	Evolution of The Enterprise Software Communities. The marker shows the size and relative stability of the community for a given month.	89
A.2	Relation between different defect symptoms in the Enterprise Project. Links are weighted for the number of symptom co-occurrence in same software function. Nodes are combinations of symptoms and defect categories.	90
B.1	Metric Box Plots- 1/3	93
B.2	Metric Box Plots- 2/3	94
B.3	Metric Box Plots- 3/3	95

LIST OF TABLES

3.1	Summary of The Literature Survey	26
5.1	Empirical Setup for the New Developer Assignment Problem Using Kronecker Networks	48
5.2	Confusion Matrix	51
6.1	Symptoms for each defect category in the enterprise project	57
6.2	Spearman rank correlations between the defect categories	57
6.3	Defects Per Issues For Enterprise Software and Android Datasets .	61
6.4	Performance of The Basic Issue Recommender	70
6.5	Performance of The Issue Recommender With Static Load Balanc- ing. GINI index is lowest.	71
6.6	Performance of The Issue Recommender With Dynamic Load Bal- ancing	71
6.7	Performance of The Issue Recommender With Issue Dependencies and Heuristics 1	72
6.8	Performance of The Issue Recommender With Issue Dependencies and Heuristics 2	72
6.9	Issue Recommendation Performance of The Model For The New De- velopers when we estimate future collaboration based on Stochastic Kronecker Networks	73
A.1	Evolution of Enterprise Software Collaboration Network	88
B.1	Correlation of The Software Metrics For The Enterprise Software 1/3	96
B.2	Correlation of The Software Metrics For The Enterprise Software 2/3	97
B.3	Correlation of The Software Metrics For The Enterprise Software 3/3	98

LIST OF ACRONYMS/ABBREVIATIONS

AI	Artificial Intelligence
3P	People, Process, Product
GINI	GINI Index
pd	Probability of detection
pf	Probability of false alarms
P(X)	Probability of recommending the right developer to an issue among the top X recommendations of the issue recommender
LOC	Lines of Code
E_m	Learning based predictor
E_h	History based predictor

1. Introduction

Software is a fundamental part of modern life. We depend on the operation of software whether we are watching a video, traveling on a plane, calling a friend or using electricity power. Human life [4] and monetary losses [5] may occur because of issues that are not addressed in software components. Software quality assurance is the collection of activities that ensure the operations of software are complaint with the requirements [6]. Similar to all economical activities, software quality assurance is constrained by the budget of software projects. Efficient allocation of software quality assurance resources is crucial for the development of more reliable software. Development of a large software is beyond the technical capabilities of a single person or a few people. For this reason, large software is built and tested through the collaboration of many developers. Efficient coordination of the developers during the software quality assurance activities is one of the fundamental problems of software engineering.

Delivering a bug-free software is practically impossible due to the impracticality of performing exhaustive testing in terms of time and cost. Software are released with an unknown amount of inherent issues. It is a common belief in software engineering that as the number of eyeballs increase bugs become easier to notice [7]. Therefore after the release of a software, the most important element of the software quality assurance activities is the publicly available issue management systems. In the issue management process, developers own (or get assigned to) issues reported by testing teams or software users. The key problem of this process is assigning the right developer to the right issue and showing the defect-prone parts of the software to the developers. In organizations twenty five percent of the time is spent while waiting for decisions [1]. We may improve the efficiency of issue management process by building an automated issue recommendation system to reduce the idle waiting time.

1.1. Software Networks

Large software can be classified as complex systems. Artificial and natural complex systems arise from the nature of the relations between many relatively simple entities [8]. Graph is a useful abstraction in computer science used to model complex systems and it is widely used in a range of application areas [9, 10, 11, 12]. In recent years, there has been considerable interest in modeling different complex systems as networks. Various researchers examined the network properties in order to identify similarities among networks from diverse domains [11]. Scale-free network is defined as a network whose degree distribution follows power-law [10]. Scale-free networks are observed in different domains, and they surprisingly share similar properties [10]. Networks such as academical collaboration, epidemics, world wide web and internet routers show some similar properties such as power-law degree distribution and low network diameter[10]. In software domain, researchers have also found that many dependency networks of large software systems are scale-free networks [13].

In software engineering, research on software networks goes back to early 1980s [14]. In the early years, most of the research has been on building the dependency graph for various programming languages and execution environments. Over the years, interest of researchers moved to investigating the relation of the dependency network and software quality. Therefore, various researchers extracted metrics from dependency graphs based on common global and local properties of the graphs [15, 16].

More recently, the edits on same software modules have been used to construct a collaboration network among developers similar to scientific collaboration network. This information has been used in empirical context in research on defect prediction models [15] and in exploratory studies [16]. One reason behind this surge of interest may be the emergence of global software development [17]. Software network has been mostly studied for exploratory research previously. In this research, we used the software network to model the relation of issues with developers. We used co-edits on the same defective software modules related to the same category of issues to form the collaboration links among developers.

1.2. Issue Recommendation System

Recommendation systems are a subclass of information filtering system that predicts the rating or preference of a user for an item [18]. The recommendation system we propose is made of two core components namely bug triager and a real time defect predictor. Bug triage is defined as assigning a developer to the most relevant issue based on past data [19]. Previous research shows that assigning a bug report to an owner takes five minutes per report on average for large software [19]. For popular projects, this task may make up the entire workload of several people. Previously, bug triage problem has been addressed by estimating the text similarity among issues [19]. This approach has various shortcomings such as language and dependence on issue report content [19]. In our proposed recommender system, we categorize the issues based on their content markings. For each category, we rank the developers based on their collaboration activity and recommend new issues based on this ranking.

Real time defect prediction is the prediction of defect-prone modules related to an issue. A defect predictor can be used to present defect-prone modules to the assigned owner of an issue. In most defect prediction models the output of the model is provided at a special snapshot such as the production release or beginning of testing phase [20]. In our proposed model we extend the machine learning based model by using the historical defect proneness data of related software modules. For this goal, we utilize the call-graph of the software to update the predictions at any point during development based on the categories of recent changes.

In order to build such a recommender, we also investigated the mapping between the issues and defects in order to understand their relation with each other.

If we examine the successful applications of AI that are widely used in practice, we observe that most of them need minimum interference by their users [18]. Successful applications such as movie recommenders, machine translators, image recognizers fit easily to the existing work flow of the users. Since the most common element in the work flow of developers regarding quality assurance is issue management system, we believe

that a practical recommendation system should fit easily to the issue management process. In order to make our proposed issue recommender to be easy to integrate into the issue management software, we use data that can be extracted with automated tools from the issue and source code repositories.

2. Problem Definition

Most of the management decisions in software engineering are based on the perceptions of the people about the state of the software and their estimations about its future states. Some of these decisions are; resource allocation, team building, budget estimation and release planning. There are many models and applications in both academia and industry that provide estimations about an aspect of the future state of a software [21], [20]. While useful, most of these models do not recommend actions. Most of the prediction models do not present the causal relationships among the predicted phenomena. The predictions of these models about an artifact can not be used directly to guide the critical decisions of the organizations. For example, a defect prediction model output is usually a list of defect-prone software modules but it contains no information about how to deal with these defects. In other words, these models mostly act as automated analysts but lack actual management support functionality. Two key reasons of this problem are modeling different aspects of software incompletely and providing output with little information content. Therefore, a problem in software engineering is modeling people, process and product (3P) with a more holistic approach.

In our work, we intend to provide a recommendation system whose output can be directly used to take actions. For this goal, we employ a model of the software that includes information about all aspects of 3P. The problem addressed by this dissertation can be divided into two dimensions: 1) Understanding the structure of software networks, 2) Building a recommender system that helps efficient issue management.

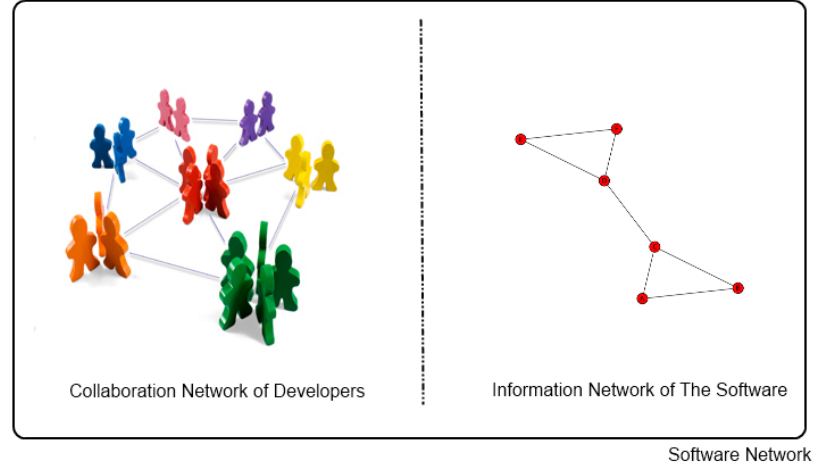


Figure 2.1: *Software network* is represented by the combination of its two underlying networks. a) *Collaboration network* models the interaction among developers, b) *Information network* models the relations between software components.

2.1. Understanding the Structure of Software Networks

In this dissertation work we model software projects as two distinct interlinked networks of developers and information elements (software modules) respectively. These two networks, namely the information network and collaboration network, can be seen in Figure 2.1. The *collaboration network* models the cooperation between developers, whereas the *information network* models the dependency relations among the software modules. These two networks together, provide us an abstraction of the software in the form of a *software network*.

The *information network* is a directed network which is based on the relations among software modules. An arc forms from software module a to software module b if a can call b at one point in its execution. The *collaboration network* is an un-directed network in which nodes are developers. An edge forms between two developers if they have contributed on a piece of software at one point.

The nodes in the collaboration network work on a subset of the information network of software modules at a given time. This information is represented in our model by links between software modules and contributor programmers. In the model, software quality is measured by number of defects found in a software module.

Once these two models have been built we aim to investigate the relation of information networks and collaboration networks in software with software quality. In addition, by modeling how the *software network* model evolves we aim to predict its future states. The *software network* model covers all aspects of 3P. The state of the *information network* provides information about the the product, the state of the *collaboration network* provides information about people and the evolution of these networks provides information about the processes.

Previously these two networks were mined separately to the best of our knowledge. The evolution of these network has not been modelled either.

2.2. Recommendation System For Issue Management

Defect prediction models provide a list of defect prone modules. However, they recommend no actions about how or by whom to handle them. While in theory providing successful results, defect prediction models face challenges when actively used in practice. One probable reason of these challenges is the difficulty of fitting defect prediction to the existing development environment in an organization.

Issue management is the central element of the software maintenance operations. Therefore, we propose a recommendation system that triages issues and recommends files that contain defects related to issues. In Figure 2.2 two main parts of the issue recommendation model can be observed. First part of such a system is the bug triage component. The problem of bug triage has been tackled by various researchers as a text mining problem previously [19] [22]. However, to the best of our knowledge no one used either past development activity of developers or the issue timelines in triage. After assigning a bug to a developer, the first thing the developer will need would

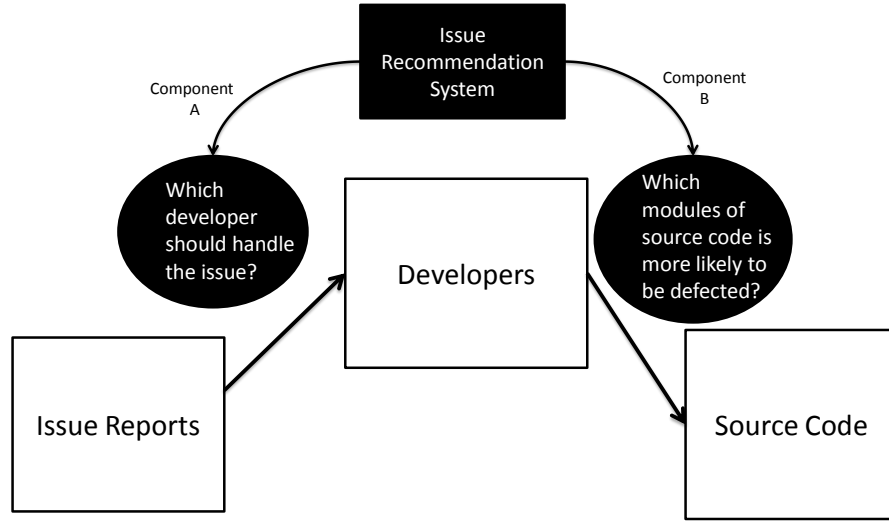


Figure 2.2: The recommendation system and its two components. Component A recommends an issue that is most relevant to a developer. Component B recommends the part of source code that possibly contains a defect related to that issue.

probably be the defective code relevant to the issue. In order to answer this problem we propose a real time defect prediction model that predicts the defect prone modules that are relevant to the developer. We believe that with its two components such a model would provide a major boost to the productivity of developers without requiring any input or change to their processes.

In order to develop such a recommendation system one needs to understand the relations between issues and defects and how the issue reports get generated. Therefore our research questions regarding the recommendation model are as follows:

- (i) How can we assign issues to developers?
- (ii) How can developers more efficiently organize their time in solving the issues?

3. Background

The main goal of this dissertation is aiding the process of issue management through an automated issue assignment recommender. In the following section we discuss the literature related to the topics of issue management and defect prediction models. Following these sections we briefly discuss the general literature on recommender systems. We use the software networks of organizations in the issue recommender. For this reason, lastly, we provide an overview of the literature on software networks through a literature survey.

3.1. Issue Management

Issue management systems of large open source software projects have become publicly available since the early 1990s. Issue management systems of some applications with commercial licenses are publicly available to make their issue handling process transparent for their users. Research on issue repository data has started in parallel with the public availability of past software issue management data [23].

Notable areas of research related to issue management include reopened issues, automated issue triage, factors that change the quality of issue reports, detection of duplicate issues and estimation of the issue fix durations [19, 24, 25].

Issue reopening is defined as the process of activating issues that have been previously categorized as fixed. The reopened issues may be an important problem in issue effort management and software quality. Various researchers analyzed issue reopening cases and its possible causes. Shihab et al. [26] analyzed work habits, bug report, bug fix dimensions for Eclipse Project to find the factors that contributed to bug reopening and built a *reopened bug prediction model* using decision trees. Shihab et al. found that *the comment text, description text, time to resolve the bug and the component the bug was found* were the leading factors that caused bug reopening for Eclipse [26]. On the other hand, Zimmermann et al. [27] analyzed Windows Vista and Windows 7 issue

repositories and conducted a survey on 394 developers in order to find out the important factors that causes bug reopens [27]. Zimmermann et al. built a logistic regression model in order to identify the factors that may cause issue reopening. In their research Zimmermann et al. used organizational and process related factors in addition to factors that can be directly extracted from the issue report. In their logistic regression model nearly all the factors they observed were found to be significant. These factors were related to location, work habits and bug report characteristics. We replicated the work of Shihab et al. and Zimmermann et al. on a large scale commercial software in our previous work [28]. Our results indicated that centrality in the issue proximity network and developer activity are important factors in issue reopening.

One important related research topic about issue repositories is the problem of automated *bug (issue) triage*. Bug triage is the procedure of processing an issue report and assigning the right issue to the right developer. This problem is especially important for large software with millions of users. Anvik et al. found that 300 daily reported issues makes it impossible for developers to triage issues effectively for Mozilla based on an interview with an anonymous developer [19, 29, 25]. Zhang et al. studied the social network of issue reporters and developers to propose a bug triage model [30].

Text mining methods have been used in several studies to find the most relevant developer to handle the bug in the proposed automated bug triage models [19, 22, 31, 32, 33]. Bakir et al. proposed a model that forwarded auto-generated software fault data directly to the relevant developers by mining the patterns in the faults [34]. Measuring the performance of an automated bug triage model is complicated. The benefit is often measured as the percentage of actual owners estimated by the model and the decrease in issue reassignments or *bug tosses*. However, research community is divided about the implications of *bug tosses*. While bug triage studies claim that bug tossing is time consuming [22], Guo et al. observed on Microsoft Vista dataset that the issue reassignment is in some cases not a problem but beneficial to communication [35].

Effective issue reporting is important for reporters as much as it is for the de-

velopers. In an exploratory study, Windows Vista issue repository has been studied extensively by Guo et al. in order to identify the characteristics of bugs that are getting fixed [29]. Bettenburg et al. also analysed the components of the bug report (such as severity information, stack traces, build information and screenshots) that make a bug more likely to be resolved [24]. They investigated the components of a bug report that are most useful to developers through surveys and data analysis, and built an automated recommender to issue reporters that reminds them to complete missing elements.

Duplicate issue reports are the ones with the same content but different wording. Particularly in popular software, duplicate issues may lead to re-work. Wang et al. proposed a novel method to detect these duplicate reports [36] by mining past issue reports. The model compares a new bug with the patterns of existing bugs and checks if the new bug is a duplicate or not.

The estimation of the issue resolve times is another research area. Giger et al. estimated the issue fix times in JBoss project [37]. Weiss et al. estimated the issue fix times on Eclipse, Mozilla and Gnome [25]. It is useful for developers in planning their effort and for users in planning their work related to the issue. The models of Weiss et al. and Giger et al. may be also be valuable for the users as well as software organizations since these models may be used to modify the reported issues to reduce their resolve times.

3.2. Defect Prediction In Software Engineering

Defect prediction in software engineering can be tracked down to the early work done by researchers in the seventies [38, 39]. In one of the earliest studies Akiyama et. al. found that 23 defects per kLOC occurred in Fujitsu software systems. Using simple linear regression Akiyama proposed a simple defect prediction model for software

systems by using LOC as a complexity measure [38]:

$$\text{Defect Count} = 4.86 + 0.018 * LOC \quad (3.1)$$

Over the years more complicated and comprehensive metric sets have been proposed by researchers and various machine learning algorithms have been used to predict defects on software modules [21, 20, 40, 41, 42, 43]. Proponents of defect prediction claim that defect prediction models enable more efficient test resource allocation in real life software systems [21]. Hall et al. gives a structured literature review of the research on defect prediction models [20].

In a benchmark study by Lessmann et al. it was shown that top 12 algorithms have no statistically significant difference in prediction performance [44]. Encouraged by the observations by Lessmann et al., in recent years, researchers have proposed different metric sets for the defect prediction models in order to increase their performance [20], [45], [46], [47]. Performance is only a part of the problem that prevents the wide adoption of defect prediction models by the industry. A prediction model whose output do not highlight actions for the software maintenance activities of an organization may not be worth the trouble of implementation even if its performance is high. Common limitations of the defect prediction models are listed in Section 3.2.2.

3.2.1. Metric Sets Used In Defect Prediction Studies

3.2.1.1. Static Code Metrics. Static code metrics can be defined as the code metrics that are extracted from the source code in a software project. The major metric sets defined in previous works can be listed as follows [39, 48, 49, 50, 15]:

- **Halstead Metrics:** Halstead metrics are proposed by Halstead in his seminal work about empirical software engineering [39]. They are among the first defined metrics for defect prediction and are used in many regression and classification based defect prediction models [39]. Halstead determines the complexity of a

software system by counting the combinations of elements in it. Operands, operators and statements and derived metrics about their relationships make up the Halstead metric set [51]. Basic metrics of Halstead can be listed as follows:

- (i) total number of operators
- (ii) total number of operands
- (iii) unique number of operators
- (iv) unique number of operands

In appendix B full list of Halstead basic metrics have been explained in detail.

- **McCabe Complexity Metrics:** McCabe metrics are proposed by Thomas McCabe for predicting complexity in a software system by traversing the control flow graph [48]. McCabe metrics provide another view of complexity of software systems from the branching structure of the source code modules. The base McCabe metric is cyclomatic complexity. Cyclomatic complexity is computed using the control flow graph of the program. The nodes of the graph correspond to indivisible groups of commands of a program, and a directed edge connects two nodes if the second command might be executed immediately after the first command. Cyclomatic complexity may also be applied to individual functions, modules, methods or classes within a program. Cyclometric complexity can be computed by formula 3.2 where $v(G)$, e and n denotes complexity of the graph, number of edges and number of nodes respectfully.

$$v(G) = e - n + 2 \quad (3.2)$$

Other metrics of McCabe are derived from cyclomatic complexity of source modules. In Appendix B full set of McCabe complexity metrics is given.

- **Chidamber-Kemerer (CK) Metrics :** In the 90s object oriented programming paradigm became the dominant software construction methodology. Chidamber and Kemerer in their seminal work proposed a new metric set based on the class hierarchies and class structures in OOP software [49] In CK metrics, relationships and couplings between objects and classes in an OOP software are counted as a complexity measure. Depth of class hierarchy is measured by calculating depth of the inheritance tree of the classes. In Appendix B full set of basic

CK metrics is given.

3.2.1.2. Churn (Repository) Metrics. Software version repositories hold a wealth of information that has been harnessed in defect prediction work by researchers [50, 52]. Usage of repository information for defect prediction is a relatively new approach in defect prediction studies and was pioneered by Nagappan and Zimmermann as an alternative software model to static code metrics [50, 52]. In this metric set changes in software modules in terms of software modules are quantified. Change as a changed lines of code is used as an important metric called software churn. Relation of developers to software modules in terms of unique committers and inequality of commit effort is also used in the feature selection.

There are studies that focus on other factors affecting an overall software system such as the dependencies, code churn metrics or organizational (repository) metrics related with the development process [50, ?]. Results of these studies show that the ability of process-related factors to identify failures in the system is significantly better than the performance of size and complexity metrics. Zimmermann and Nagappan challenged the limited information content of data in defect prediction [50, 52]. The authors proposed to use network metrics that measure dependencies, i.e. interactions, between binaries of Windows Server 2003. Results of their study show that the network metrics have higher performance measures in finding defective binaries than code complexity metrics. Moser et al. also used repository metrics and they concluded that repository metrics give better prediction results than static code metrics [53]. In Moser et al.'s work only post-release defect data of Eclipse Project defects are extracted. They trained and tested on the same release. They also applied a cost sensitive classification approach to improve the performance of their model. They stated that cost sensitive approach would be hard to implement in real life. It is also difficult to derive clear conclusions from their results such that whether the cost sensitive classification or usage of repository metrics improved the results. In another study the effect of number of committers has been investigated in a large scale commercial project. The outcome of this study revealed that the number of developers did not have any effect on the

defect density [54]. In another recent study decrease in pf by using repository metrics has been realised in open source software [55]

3.2.1.3. Social Network Metrics. Social network metrics consist of the metrics found by modelling development effort of a software in a collaboration graph and calculating common graph metrics like degree, closeness and betweenness [15]. There are just a few works about the usage of social network metrics in defect prediction. Results of these works are not conclusive about the inclusion of these metrics. In their study Meneely et al. formed a network by examining collaboration at file level and within a release period [15]. Metrics proposed on file level collaboration in this study can be listed as following:

- (i) **Closeness (max,sum,min,avg)** : Shortest path between one vertex and another vertex
- (ii) **Degree (max,sum,min,avg)** : Number of edges connected to each developer.
- (iii) **Betweenness (max,sum,min,avg)** : Number of geodesic paths including node v divided by all possible geodesic paths.
- (iv) **Number of hub developers** : Number of distinct hub developers who update this file. Hub developers are selected by calculating degree of developers at project level and finding developers with max 10 % of edges.

When using logistic regression Meneely found that degree and closeness are the most important metrics among several candidates of history and static code metrics.

Weyuker also used developer information and collaborations as metrics on a large scale AT&T project for defect prediction. As a result of the study, using logistic regression no significant improvement on prediction performance of predictors has been realised [56].

Social network metrics also have been used for examining the project governance types in open source software [57, 58]. In this study types of networks that occur in open source projects are associated with the governance types.

3.2.1.4. Confirmation Bias Metrics. Calikli et al. investigated the effects of thought processes of people on the software quality and proposed a set of metrics that quantifies the confirmation bias of people for defect prediction [46]. Confirmation bias is defined as the tendency of people to seek evidence that verifies a hypothesis rather than seeking evidence to falsify a hypothesis [46]. Calikli et al. used the outputs of Wason’s confirmation bias test [59] to extract the confirmation bias metrics of people who touched software modules in the source code repositories. Although these metrics quantify a part of the people aspect in software engineering, they have been shown to increase the performance of defect prediction models [46]. The research by Calikli et al. highlights the importance of using the people element of 3P for the defect prediction problem.

3.2.2. Limitations of Defect Prediction Models

Over more than thirty years researchers have proposed defect prediction models using different metric sets, algorithms and data processing techniques with considerable success [20]. However, all defect prediction models have several common limitations which prevent their more general adoption by the software development organizations. We list the main limitations of defect prediction models as follows:

- (i) **Data Collection Challenges:** Data collection is an important barrier for software organizations. For organizations with limited defect tracking capabilities, Turhan et al. proposed cross-company prediction which eliminated the need for defect tracking by using other training datasets [60]. Even with cross company predictors challenges for metric extraction persists.
- (ii) **Lack of Integrated Tool Support:** There is no integrated tool which provides a solution for the measurement and analysis needs with predictive capabilities in

the same package. Companies have to manually integrate tools which support one part of the problem.

- (iii) **Lack of Information Content of the Output:** In most cases defect prediction models provide only defective/non-defective predictions for software modules. The content of the output recommends no action for the practitioner.
- (iv) **Focus on Management of the Testing Process:** Benefits of defect prediction is explained in terms of the increase of efficiency in testing resource allocation. However, defect prediction output is also valuable for developers who touch a piece of source code during the product lifecycle.

3.3. Scale Free Networks

3.3.1. Properties of Scale-Free Networks

The properties of scale-free networks can be divided to two categories, namely, static and temporal properties.

3.3.1.1. Static Properties. Degree distribution: The degree-distribution of a graph follows power law if the fraction of nodes with degree k is given by $P(c_k) \propto c_k^{-\gamma} (\gamma > 0)$ where γ is called the power-law exponent. For example for the Enterprise Software collaboration network, in Figure 3.1 it can be observed that the 1-degree polyfit has a error rate of mean relative error 0.27. As seen in Figure 3.2, if we take the first 80 most popular (highest degree) nodes, the degree distribution fits 1-degree polyfit more closely with mean relative error rate of 0.09.

Scree plot: This is a plot of the eigenvalues (or singular values) of the adjacency matrix of the graph, versus their rank, using a log-log scale. The scree plot is also often found to approximately obey a power law.

Small diameter: Most real-world graphs exhibit relatively small diameter (the small-world phenomenon): A graph has diameter d if every pair of nodes can be

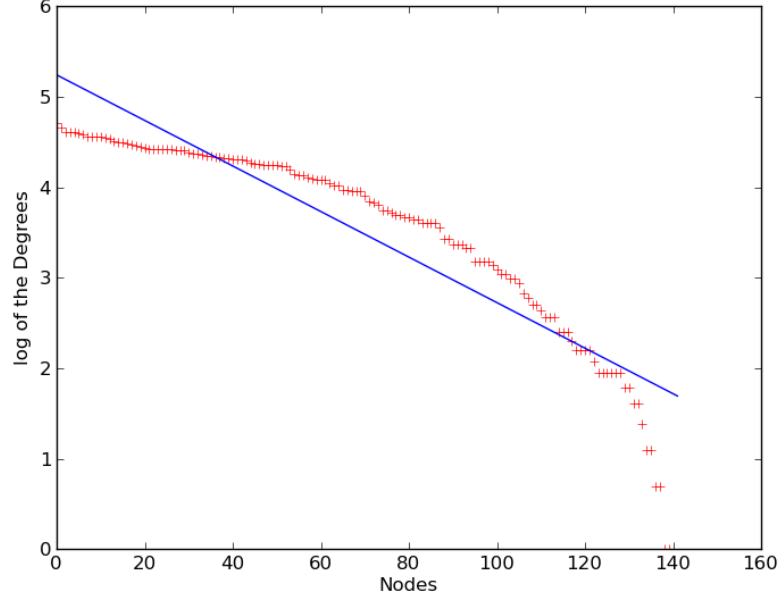


Figure 3.1: Degree Distribution of the Nodes in The Enterprise Software Dataset Collaboration Network with log-normal scale

connected by a path of length at most d . The diameter d is susceptible to outliers. Thus, a more robust measure of the pairwise distances between nodes of a graph is the effective diameter [10]. This is defined as the minimum number of hops in which some fraction (or quantile q , say $q = 90\%$) of all connected pairs of nodes can reach each other.

3.3.1.2. Temporal Properties. There are two temporal properties that are associated with evolution of scale free networks: (a) the effective diameter of graphs tends to shrink or stabilize as the graph grows with time, and (b) the number of edges $E(t)$ and nodes $N(t)$ seems to obey the densification power law [10].

Network evolution is the underlying dynamics of how a graph evolves over time. Researchers have tackled this problem since 1970s [61, 62, 63, 64, 65, 66]. Concepts such as “rich gets richer” have been proposed in order to explain the evolutionary dynamics of the networks [10]. Rich gets richer or preferential attachments explains the degree distribution by a simple probabilistic model. In the model, a node with high

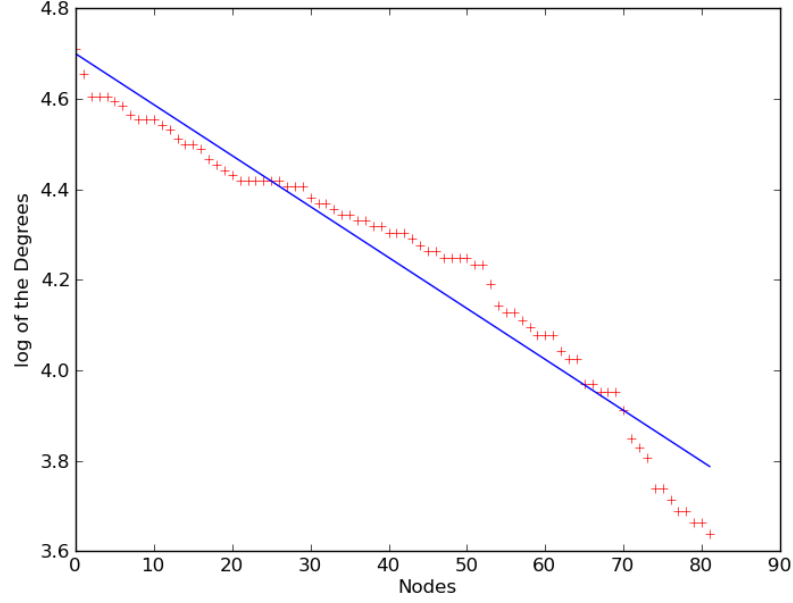


Figure 3.2: Degree Distribution of the Nodes in Enterprise Software Collaboration Network with log-normal scale with top 80 nodes: notice a closer fit

degree is proportionally more likely to get even more connections.

Network densification over time can be used to estimate the speed of node increase by fitting node count over time with a polynomial [67]. Network densification concept can be explained with the following equation:

$$e(t) \propto n(t)^\alpha \quad (3.3)$$

In the equation $e(t)$ and $n(t)$ denote the number of edges and nodes of the graph at time t , and α is an exponent that generally lies strictly between 1 and 2. This equation is also named as growth power law [67, 68]. (Exponent $\alpha = 1$ corresponds to constant average degree over time, while $\alpha = 2$ corresponds to an extremely dense graph where each node has, on average, edges to a constant fraction of all nodes.)

3.4. Recommendation Systems

Recommendation systems gained their current popularity with the popularity of world wide web. One of the common properties of web based recommendation systems is their non-obtrusive nature. Popular sites like stackoverflow, amazon and google employ recommendation systems which guide user decisions with minimal effect on the core functionality [69].

We can group recommendation systems as collaborative, content based and hybrid [70]. The basic idea of collaborative recommendation systems is that if users shared the same interests previously, the previous choices of one user is likely to be chosen by the other *similar* user if he or she has not made the same decision yet. Collaborative recommendation systems are content independent in their nature. All we know is a list of ratings of the users to independent items. Although, such systems provide a robust solution which may eliminate human intervention, pure collaborative recommenders may omit simple clues that can easily be extracted from content. For example, a user who bought science fiction novels in the past may be more likely buy science fiction novels rather than nineteenth century Russian literature. Some problems in collaborative recommender systems include data sparsity, cold start and rating subjectivity. Data sparsity happens when the ratings of users for a lot of items are not defined. Estimating the similarity gets complicated for users who had made a few choices in the system previously. In such cases, matrix factorization have been shown to be successful in inferring the implicit similarities among the users [71]. In addition the success of a system is bounded by the subjectivity of the user ratings and a rating based system can be exploited easily by people with malign intentions.

Content based recommenders extract key attributes of the content and offers similar content, based on user's past selections[70]. For example, for a user who likes James Bond movies, the content based recommendation system will likely recommend movies from the spy film genre. There is more human intervention in content based recommenders compared to collaborative recommenders since certain content modelling should be done in model construction. The most important problem of content based

recommenders is extracting the important attributes from the content items. Another problem is, similar to collaborative recommender systems, making recommendations to people with no history in the system. The classic defect predictor in the software engineering literature may be classified as a content based recommender.

There is a final category of recommender systems which uses a hybrid of collaborative and content based approaches [70]. In these recommenders, content knowledge is mixed with the collaborative recommender to use the strengths of collaborative and content based approaches. The recommender system in our dissertation fits into this category. We used this approach because a content-agnostic approach is not possible in our domain. We do not have the ratings of developers on each issue and every issue is recommended only once by the system. Therefore, there is a issue cold start for every recommendation. In our recommendation we model collaboration implicitly based on code level interaction among people since issue ownership is a non-collaborative activity.

One key problem in most recommendation systems is cold start [72]. Cold start is a special case of the data sparsity problem [72, 73]. It happens when initiating a new user or a new item to the system. In our case since every issue is a new issue we cold start in each issue recommendation. In addition we discuss the problem of new user initiation in our domain for our recommender.

3.5. Task Management

Problem of task assignment has been studied by many researchers in diverse domains such as fault tolerant computing and operations management [74, 75]. In Figure 3.3 the solutions for the different task complexity and context variance combinations can be observed. As the complexity and the context variance of the tasks increase the task assignment problem gets complicated.

Complexity of the tasks and the context variance among the tasks change the applicability of process-aware task management systems [1]. Issue assignment problem

	Low Context Variability	High Context Variability
Low Action Complexity	Realm of Model-based Process Management	Process Pattern Management
High Action Complexity	Task Information Management	Minimal Planning Opportunities

Figure 3.3: Task Complexity and Context Variability vs Planning Opportunities [1]

can be classified as a special case of the task assignment problem where the tasks are knowledge intensive. In a knowledge intensive environment it has been observed that context of the task is highly variable and the complexity of the task is high [1]. Rigidity of the process-aware task management systems restricts their applicability in knowledge intensive environments. The workers in a knowledge intensive environment tend to work-around such task management systems during their daily work due to necessity [1]. Therefore, in knowledge intensive environments simple task management and planning opportunities are minimal. For this reason, in the problem of issue management, application of standard task management solutions is not practical.

3.6. Literature Survey on Software Networks

A structured analysis of existing literature about social networks, dependency networks and their relations to software quality has been conducted. We used the approach proposed by Brereton et al. [76] during the survey.

The main motivation in the literature review was highlighting the open problems related to software networks that have not been addressed by previous research. We focused on two groups of papers in the literature review:

- (i) Papers that analyze the social network of developers and their effect on software quality.
- (ii) Papers that analyze the call-graph of source code and their effect on software quality.

3.6.1. Search Criteria

We queried four well-known online research paper repositories, namely, IEEE Explorer of IEEE, ACM Digital Library by ACM, Springer archive and Elsevier Science Direct. We also manually browsed four major software engineering journals, namely, ACM Transactions on Software Engineering and Methodology (TOSEM), IEEE Transactions on Software Engineering (TSE), Software Quality Journal (SQJ) and Empirical Software Engineering (EMSE).

Since some of the terms in software engineering are used interchangeably we constructed our search queries accordingly. We did the survey on learning from connected instances more informally due to time constraints. We limited the publication years to 2000 and onwards since the seminal works on complex networks were published in late 1990s and early 2000s [12, 77, 78, 79]. Our justification for this short time span is the relative freshness of the topic since the seminal papers that defined the area were published in late nineties. We queried all repositories with similar queries. Our queries in the analysis were as follows:

- (i) ((software engineering) AND ((call graph) OR (dependency)) AND (quality))
- (ii) ((software engineering) AND ((social network) OR (dependency) AND (quality)))

After filtering irrelevant papers, we had 24 papers from the repositories based on our search criteria. In the following sections we will first summarise the papers reached. Afterwards we will analyze some selected papers in depth.

3.6.2. Survey Results

In Table 3.1 an overview of each publication filtered is provided. Properties of the papers are given with the appropriate code values. Overall with 1 exception all the surveyed articles are from 2000s with a higher number of papers from recent years. 7 of the publications were published in scientific journals, 14 are from conference proceedings and 3 are from trade magazines. The distribution among social network and

call-graph analysis papers is even with 11 from each type. None of the papers discussed both networks in the same publication or used both networks in a predictive context. In 6 of the papers complex network research has been used and complex network research has been applied to call-graphs. No complex network paper analyses the social network of developers. 10 of the papers can be categorized as predictive studies. None of the studies used graph based machine learning algorithms. Every paper analysed instances independently and did not consider inter-instance relationships.

We chose 4 papers based on their relevance to the dissertation for a more detailed review. Following is a brief review of the papers. In Table 3.1 we denoted the papers with in-depth review as alpha.

The first paper we overviewed is “Mining edge-weighted call graphs to localise software bugs” by Eichinger et al. [14]. It is the only paper using graph learning techniques in defect localization (prediction) to the best of our knowledge. Eichinger used a dynamical call graph and tried to locate the bugs in software by analysing stack traces. Eichinger weighted directed edges by considering how often they execute during faulty and normal runs. For example, a dependence relation which is only executed in faulty runs is given the highest possible weight. 3 programs that have been developed at Siemens and written in C language were used as the test datasets. The dynamic call graph is formed artificially by giving the system different inputs. One limitation of this work is usage of dynamical call-graph. Gathering dynamical call-graph data can be unrealistic (test runs) or impossible (when customer does not share data) in most scenarios.

The second paper we overviewed is “An empirical study of speed and communication in globally distributed software development” by Herbsleb et al. [17]. Herbsleb et al. investigated the relations between distributed software teams, increased software development costs and reduced productivity. He also examined if works in different sites can be interdependent and how the interdependence may diminish over time. The data from 2 multi-site companies have been used as input. Data was collected by surveys and from change management systems. Majority of surveyers point to difficulty

of communication between sites. The two surveyed companies could not reduce interdependence of tasks between different sites in the projects. As a result they found that teams located at different sites have reduced overall efficiency by examining the software changes. The study can be extended by proposing ways for effective communication and ways to make tasks independent in order to increase the effectiveness of distributed software teams.

The third paper we overviewed is “Power laws in software” by Louridas et al. [80]. Lauridas et al. examined the power-law relation of dependency networks on a variety of open and closed source systems and found that the power-law relation (cx^{-k}) holds for every examined system. Power-law is the most important property of scale-free networks and can be used to model the structure of the dependency networks [10].

The last paper we overviewed is “An Empirical Study of the Factors Relating Field Failures and Dependencies” by Zimmermann et al. [81]. Zimmermann et al. analyzed if there is a correlation of defect proneness between software modules having dependency relations. The results seems to be contradictory in open source project Eclipse and Microsoft Windows. In Microsoft Windows Vista there is no correlation of defect proneness among dependent modules while in Eclipse there is a positive correlation. Investigating the mechanics behind this can be an interesting addition to this work.

After the analysis, we had 2 conclusions about the open problems in the area:

- (i) Lack of integrated models: None of the researchers examined neither the social network nor dependence network as a whole.
- (ii) Instances are assumed to be independent: Relational learning has not been used in software engineering domain.

Table 3.1: Summary of The Literature Survey: Explanation of properties: β -empirical work, λ -social network paper, γ -call graph paper, α major publication analysed in depth, $\sqrt{}$ -estimation, \diamond scale free networks, ϵ evolution of software systems **MAG**-magazine, **JOU**-SCI Journal, **CON**-Referreed conference

	Paper Code	Authors	Paper Name	Publication Year	Published In	Properties
[82]	Applewhite2004	Applewhite, A.	Whose bug is it anyway? The battle over handling software flaws	2004	IEEE Software Vol. 21(2), pp. 94-97	MAG λ
[83]	Barbagallo2008	Barbagallo, D., Francalenei, C., Merlo, F.	The impact of social networking on software design quality and development effort in open source projects	2008	ICIS 2008 proceedings, pp. 201	CON $\lambda \epsilon \beta$
[84]	Benestad2009	Benestad, H.C., Anda, B. Ar- isholm, E.	Understanding software maintenance and evolution by analyzing individual changes: a literature review	2009	Journal of Software Maintenance of Software Maintenance(September), pp. 349-378	JOU ϵ
[24]	Bettenburg2010	Bettenburg, N., Hassan, A.	Studying the Impact of Social Structures on Software Quality	2010	2010 IEEE 18th International Conference on Program Comprehension, pp. 124-133	CON $\beta \lambda \sqrt{}$
[14]	Eichinger2008	Eichinger, F., Bhm, K., Huber, M.	Mining edge-weighted call graphs to localise software bugs	2008	Machine Learning and Knowledge Discovery in Databases, pp. 333-348	CON $\alpha \epsilon \beta \sqrt{}$
[17]	Herbsleb2003	Herbsleb, J., Mockus, A.	An empirical study of speed and communication in globally distributed software development	2003	IEEE Transactions on Software Engineering Vol. 29(6), pp. 481-494	JOU $\alpha \beta \lambda \sqrt{}$
[85]	Ichii2008	Ichii, M., Mat- sushita, M., In- oue, K.	An exploration of power-law in use-relation of java software systems	2008	Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on, pp. 422-431	CON $\diamond \sqrt{} \beta \gamma \gamma$
[86]	Ko2008	Ko, A.J., Myers, B.A.	Debugging Reinvented : Asking and Answering Why and Why Not Questions about Program Behavior	2008	Human-Computer Interaction, pp. 301-310	JOU $\lambda \gamma$
[87]	Krinke2008	Krinke, J.	Mining execution relations for crosscutting concerns	2008	Software, IET Vol. 2(2), pp. 65-78	MAG $\gamma \beta \gamma$
[88]	Law2003	Law, J., Rother- mel, G.	Whole program path-based dynamic impact analysis	2003	25th International Conference on Software Engineering, 2003. Proceedings. Vol. 6, pp. 308-318	CON γ
[80]	Louridas2008	Louridas, P., Spinellis, D., Vlachos, V.	Power laws in software	2008	ACM Transactions on Software Engineering and Methodology Vol. 18(1), pp. 1-26	JOU $\diamond \gamma$

[16]	Madey2002	Madey, G., Freeh, V., Tynan, R.	The open source software development phenomenon: An analysis based on social network theory	2002	Americas Conference on Information, pp. 1806-1813	CON $\beta \diamond \lambda$
[15]	Meneely2008	Meneely, A., Williams, L., Snipes, W., Osborne, J.	Predicting failures with developer networks and social network analysis	2008	Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, pp. 13-23	CON $\beta \lambda \checkmark$
[89]	Palmer1994	Palmer, T., Fields, N.	Computer supported cooperative work	1994	Computer Vol. 27(5), pp. 15-17	MAG λ
[90]	Petersen2010	Petersen, K.	Measuring and Predicting Software Productivity: A Systematic Map and Review	2010	Information and Software Technology	JOU $\diamond \checkmark$
[91]	Turhan2008	Turhan, B., Kocak, G., Bener, A.	Software Defect Prediction Using Call Graph Based Ranking (CGBR) Framework	2008	2008 34th Euromicro Conference Software Engineering and Advanced Applications, pp. 191-198	CON $\beta \gamma \checkmark$
[92]	Wang2009	Wang, L., Wang, Z., Yang, C., Zhang, L., Ye, Q.	Linux kernels as complex networks: A novel method to study evolution	2009	2009 IEEE International Conference on Software Maintenance, pp. 41-50	CON $\diamond \gamma$
[13]	Wen2009	Wen, L., Dromey, R.G., Kirk, D.	Software engineering and scale-free networks.	2009	IEEE transactions on systems, man, and cybernetics. Part B, Cybernetics : a publication of the IEEE Systems, Man, and Cybernetics Society Vol. 39(4), pp. 845-54	JOU \diamond
[93]	Wolf2009	Wolf, T., Schroter, A., Damian, D., Nguyen, T.	Predicting build failures using social network analysis on developer communication	2009	Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, pp. 1-11	CON $\beta \lambda \checkmark$
[94]	Xie2006	Xie, T., Pei, J.	MAPO: mining API usages from open source repositories	2006	Proceedings of the 2006 international workshop on Mining software repositories, pp. 54-57	CON β
[95]	Yang2007	Yang, B., Cheung, W., Liu, J.	Community Mining from Signed Social Networks	2007	IEEE Transactions on Knowledge and Data Engineering Vol. 19(10), pp. 1333-1348	JOU $\beta \lambda \gamma$
[96]	Zimmermann2009	Zimmermann, T., Nagappan, N.	Predicting defects with program dependencies	2009	2009 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 435-438	CON $\beta \gamma \checkmark$

[81]	Zimmermann2010	Zimmermann, T., Nagappan, N., Williams, L., Herzig, K., Premraj, R.	An Empirical Study of the Factors Relating Field Failures and Dependencies	2010	Proceedings of the 4th International Conference on Software Testing, Verification and Validation	CON $\alpha \beta \gamma \checkmark$
[57]	Ohira2005	Ohira, M., N. Ohsugi, T. Ohoka, and K.-i. Matsumoto	Accelerating cross-project knowledge collaboration using collaborative Filtering and social networks	2005	MSR	CON $\beta \lambda$

4. Proposed Solution: Collaboration Based Issue Recommender

In this research, we propose an issue recommender model which uses the software network (call-graph and collaboration network) and software module metrics to help developers in organizing their time more effectively while handling software issues. In addition, our recommender can be extended to propose solutions for workload imbalances, employee turnovers and team size changes during maintenance. Ease of extendibility and ease of deployment were the two design goals during the construction of the recommender.

Most of the maintenance operations during the software development life cycle are done reactively i.e. developers work on quality issues as new issues are reported by the users or the testing teams. We observe that the issue management software is the central element of the maintenance operations. We believe that an artificial intelligence based solution which aims to help maintenance of complex software systems should be easy to integrate with the issue management software. During the life-cycle of an issue, the issue is owned, and then it is resolved or terminated in some other way by a developer. A practical recommender plugged into the issue management software can help the ownership and the resolve activities.

We propose a solution which may be used both to recommend issues to developers and to recommend defect prone modules related to an issue to developers. Our solution can be easily integrated to the issue management software and it is independent of the context of the new issues except category information. Our approach and its key differences with the existing solutions in this domain can be explained in three parts:

Issue Recommender: Previous bug triage systems use the unstructured text data in issue reports extensively [19]. These recommenders match new issues with developers based on the text similarity of the issues with the past issues owned by developers. Instead of unstructured text data which may be unreliable in the cases where multiple languages are used, we focus on the activity of the software modules related to different

issue categories. We use the collaboration data on the software modules related to the issues to link the issue reports with new developer groups. Our model ranks developers based on their centrality in the collaboration network for a group of issues. A key strength of our recommender is that we can highlight all the relevant developers to a newly reported issue. Another key strength of our approach is the relative simplicity of the data extraction step since the model uses structured data instead of free format text as the input.

Extensions of the Basic Model: The proposed issue recommender can be extended to eliminate load balancing problem. In a recent study we observed that the issues owned by developers are distributed non-uniformly particularly during times when many new issues are reported [97]. This trend is not sustainable in the long run since it indicates over-dependence on few individuals in large projects. We propose to handle workload imbalances in our model by limiting the number of active issues that can be assigned to a developer.

Cold start is a common problem in recommender systems. Another extension of the model is recommending issues to a new group of developers. Existing solutions can not handle the cases of user cold start [19]. They can not propose issues to the new developers since they do not have the activity history of the new developers. We use the stochastic Kronecker networks to model the collaboration network and predict the future collaboration network by using the kernel stochastic adjacency matrix [2]. We use the future network to predict the collaboration structure of a new group of developers and recommend new issues to a the new group based on the predicted future collaboration structure.

Real Time Defect Prediction: Existing defect prediction models attempt to provide a list of defect prone software modules for the testing teams to help using their testing resources more efficiently [20]. The target users of these models are software testers and testing managers in the companies. In contrast, our proposed model highlights the defect prone modules related to a new issue to help issue resolve activity. Our target users are the developers who maintain software. Therefore, we update the

original predictions of our defect predictor based on the historical data to highlight the defect prone modules. We use the software call-graph to weight the defect proneness of neighboring software modules.

5. Methodology

5.1. Datasets

5.1.1. Enterprise Software Dataset

We used a commercial large-scale enterprise software product to test the issue recommender system and to analyze the relations between software issues and software defects. The enterprise software product has a 20 year old code base. We examined a 500 kLOC part of the product that constitutes a set of architectural functionality of the project as the dataset. The programming languages of the project are C and C++. The software is developed by an international group of developers in five different countries.

5.1.2. Android Dataset

We used Android dataset to analyze the relations between the software issues and software defects. Android is a Linux based FOSS operating system developed by Google for smartphones and tablets [98]. It was initially released in 2008 and has since been adopted by many mobile phone vendors as their preferred operating system. According to a report by Gartner, Android holds nearly 70 percent of the smartphone operating system market as of 2012 Q4 [99]. Android can be classified as a strictly governed commercial open-source project [100]. The issue management system of Android is hosted by Google and any user can post issues related to the software through the web interface. We used the issue repository data of Android from 2008 to December 2011. During this timespan, Android had 9 releases and 887 issues were owned and resolved by fixing the related code defects.

The two software had nearly hundred developers who owned and resolved at least one issue within the time period we observed (98 for enterprise software, 101 for Chrome). For the Enterprise software most of the issue reports have been reported by

tester groups who tested as a part of their job description. Only 53 issues (less than ten per cent) were reported by customers. In the Android issue repository a different trend is observed. All of the issues were reported by users who had `gmail.com` email domains (Google employees have `google.com` domain names). Therefore, we assumed that all of the Android issues were reported by users.

We extracted the data from the in-house issue management and source code version systems of the company. We used Mining Software Repositories Conference 2012 challenge data as a source for Android issues and defects. We used Python language and its scientific programming libraries Numpy, Matplotlib and Networkx for data extraction and analysis in all phases. We stored issue and defect data in a Sqlite database using Pysqlite library.

5.2. Software Network

In the proposed recommender system, we model the collaboration and the information network of the software extensively as a software network. In our research, the collaboration and the information network are formed as follows:

5.2.1. Call-Graphs of Software Modules

We used the static call-graphs to model the relations among the software functions in the source code files. The static call-graphs can be extracted by various tools easily for a lot of major programming languages. One of their weakness is that by static analysis we can not be sure if a module is called, or how much a module is called during runtime. However, their ease of extraction makes them a convenient way when analyzing complex programs.

5.2.2. Collaboration Network

We constructed the collaboration network from source repository. From the source repository we constructed the collaboration network as follows: In the net-

work $G = \langle V, E \rangle$, $v_i \in V$ are developers and $\langle v_i, v_j \rangle: e_k \in E$ are formed when two developers v_i and v_j collaborates on at least one software method. Other researchers have proposed different names for the collaboration network such as socio-technical network [101]. We chose the collaboration network name in order to avoid confusion with the aforementioned module call-graph.

We have found the collaboration network to be scale-free in our research. For the Enterprise Software collaboration network, in Figure 3.1 it can be observed that the 1-degree polyfit has an error rate of mean relative error 0.27. As seen in Figure 3.2, if we take the first 80 most popular (highest degree) nodes, the degree distribution fits 1-degree polyfit more closely with mean relative error rate of 0.09. Our findings indicate that, there is a scale-free structure in the collaboration network which can be exploited by a model.

5.3. Exploratory Data Analysis

We used Android and Enterprise Software datasets to check four hypotheses about the relations of defects, issues and developers. By examining these relations we aimed to gain insights that we can use in the proposed issue recommendation model. The factors we tested empirically were the following:

- Hypothesis *I*: Limited number of developers own majority of the issues.
- Hypothesis *II*: Issue reports change significantly after releases.
- Hypothesis *III*: Issues map to multiple defects in the source code.
- Hypothesis *IV*: Issue resolve times are dependent on number of issues that a developer owns.

In previous research, various researchers found that in both close-source and open source software the code change distribution of developers are in accordance with power law [102]. We found that same trend continues in our dataset (see Figure 3.1 and 3.2). By answering Hypothesis *I*, we will find if the same trend continues for the issue ownership distribution. In order to check Hypothesis *I* we examined the relationship

between developers and owned issues. By answering Hypothesis *I* we will find if the same trend holds for issue ownership data.

We often hear stories of programmers working overtime after a software release date [103]. Upon analyzing Hypothesis *II* we will see if the number of issues indeed change significantly after a new release. In Hypothesis *III* we try to understand if an issue maps to multiple defects in software on average.

Multitasking is often a stressful or hard to manage work style for human beings. It has been observed by Aral et al. that productivity is greatest for small amounts of multitasking but beyond an optimum, multitasking is associated with declining project completion rates and revenue generation [104]. We test whether multitasking change the average bug resolve times of developers by answering Hypothesis *IV*.

In addition, we checked the structure and the evolution of the collaboration network during the enterprise software development work. We checked the temporal and structural properties of the network.

5.4. Issue Recommender System

5.4.1. Issue Recommendation Model

5.4.1.1. Proposed Model. The proposed issue recommendation algorithm has three parts: 1) Clustering of issues, 2) Identify the developers who were active in that cluster based on their network centralities and rank them according to their centralities in the collaboration network, 3) Suggest developers that are most relevant to the issue based on the ranks computed in Part 2. The steps of the recommender system is also shown in Figure 5.1.

Part 1 has been addressed previously with text mining methods [19] which identify most similar issues. Reliance on free form text data may hinder the reliability of the text mining methods. In our experiments, we used symptom and category data to

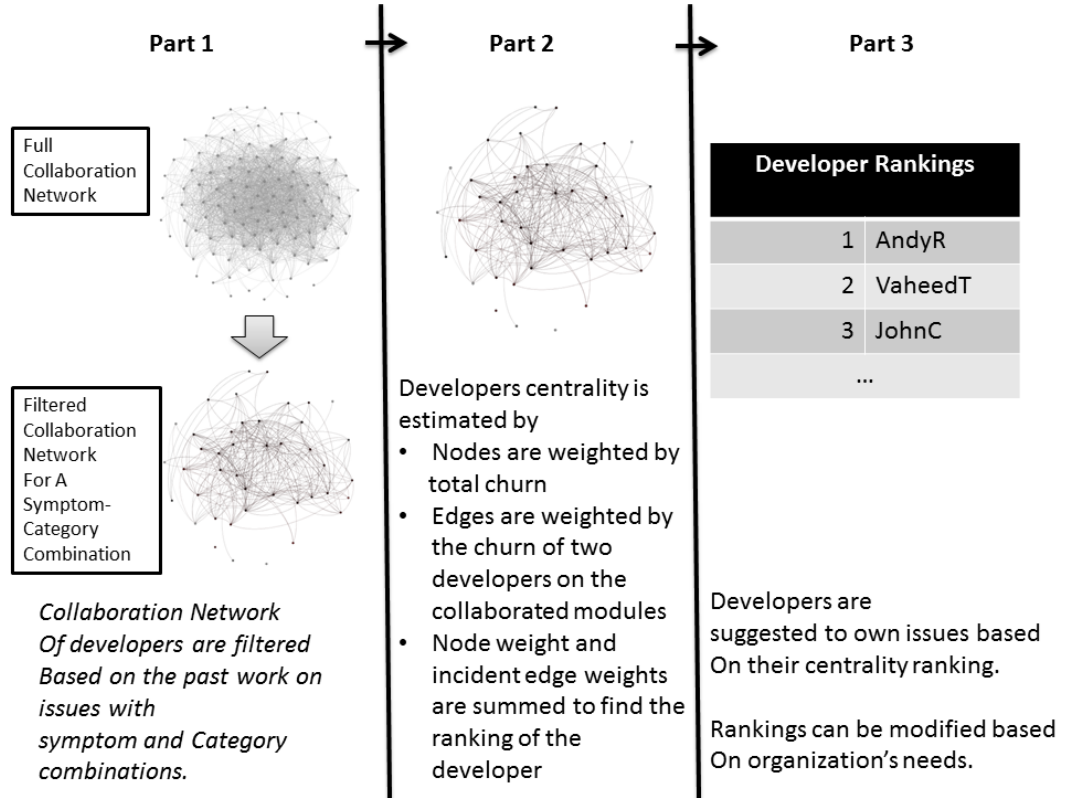


Figure 5.1: Summary of The Issue Recommender System

group issues into clusters. If issues are reported with sufficient detail we think that such a method may lead to better results than text categorization.

In Part 2 we start with a group of issues and the defect fixes associated with them. In this step we build the call-graph of the modules of software that had at least one bug in that cluster previously. It is evident that, in some cases this may be a graph containing small disconnected components. We rank developers based on the extent of code change and collaboration they did on that part of the software. Part 2 is explained step by step as follows:

- (i) Assume *ClusterA* is a cluster of issues identified in part 1.
- (ii) Directed graph, G_{call} is the call graph of software where $\langle nodes \rangle$ of the call graph consist of the software modules that were fixed due to issues in the *ClusterA*. Directed $\langle edges \rangle$ in G_{call} are formed if $\langle nodes \rangle$ have a caller-callee relation.

- (iii) The collaboration network is a weighted undirected graph G_{collab} . Nodes of G_{collab} are developers who contributed on the modules in G_{call} .
- (iv) Nodes of G_{collab} are weighted based on $\log_{10}(totalchurn)$. *totalchurn* is the total of LOC removed and LOC added of a developer among the nodes of G_{call} .
- (v) Edges of G_{collab} is weighted based on the total churn of the two developers *collaborated* on previously. *Co-work* of 2 developers is present in a module if they changed it at any point in time previously.
- (vi) In order to estimate the ranking, the weight of each developer(node) and the weights of the edges it is incident to in G_{collab} is summed.

Ranks are estimated as a combination of development experience in the source code related to a category of issues and the total collaboration activity of the developer. We represent the development experience in terms of node weight and the collaboration activity among two developers as the edge weight. For example, a developer who worked only in a part of code and never collaborated with other developers will have lower ranking than a developer with similar total activity who collaborated with other developers based on the model.

Finally in part 3, a group of developers are suggested to own an issue based on the rankings proposed in part 2. In such a model, several factors can be considered to modify the rankings. For example, a developer who is new in the project team may be given a higher ranking by the model in order to make the model assign issues to inexperienced people. Likewise, the ranking of a developer who has many active assigned issues may be lowered to prevent imbalances in the work load of developers. In the base model we do not change the rankings of the developers. If the developer had no activity on the specified issue category, he will not be placed in the ranking list. In the extensions of the base model, we change the rankings based on several heuristics in order to address issue workload balance and new member addition problems.

5.4.1.2. Performance Measures. Finding the performance of an issue recommender is harder than estimating the performance of a defect predictor. In previous studies,

the percentage of the issues recommended by the model correctly was estimated by comparing the actual assignments with model's suggestions [31], [22]. For the sake of comparison, we use the same measure. This performance measure makes the assumption that final owner of an issue is the *right* owner. In order to flex this measure we use $P(X)$ and check if top X developers suggested by the model have actually owned the issue. We use X values of 1,2,5 and 10 for the performance evaluation.

5.4.2. Extensions to The Basic Issue Recommender Model

The basic issue recommendation model has only one success criterion: To assign issues to developers in a manner that matches the existing issue assignment patterns in a company. However, assigning the issues based on the past patterns may not be the top priority of a company. In this case, issue recommendation model can be extended to handle different success criteria.

Issue workload balance and dependencies among issues may bring additional complexities to the problem of issue assignment. In our extensions to the basic model, we attempted to handle these complexities through heuristics that can be used to change the recommendations proposed by the basic model.

5.4.2.1. Issue Workload Balance Problem. We used GINI index as an inequality measure in this research to calculate the inequality in the number of issues assigned. GINI index was invented for estimating the inequalities of income among people in economics. It was proposed by Corrado Gini in 1910s [105]. It is based on the area between a uniform cumulative distribution function and the actual cumulative distribution function (Lorenz Curve) [106].

A detailed survey of various calculation methods and history of GINI index is provided by Xu [106]. We explain our computation method briefly based on Xu's paper [106]. In this work we used the geometric interpretation of GINI index. $y = [y_1, y_2 \dots y_n]^T$ denotes the vector of the number of issues owned by each active developers where

developers are ranked according to the number of issues they own so $y_1 < y_2 < \dots < y_n$. n is the active issue owner count at a given day. $F(y_k)$ is the cumulative probability up to y_k and can be calculated as, $F(y_k) = \sum_{i=1}^k \frac{i}{n}$. The mean of the number of issues can be found as, $\mu_y = \frac{1}{n} \sum_{i=1}^n y_i$. Cumulative issue shares of developers can be computed by the following equation:

$$L_i = \frac{1}{n\mu_y} \sum_{j=1}^i (y_j) \quad (5.1)$$

In Figure 5.2a Lorenz curve for an income distribution can be observed. Actual distribution's area ($B - A$) can be divided to the area of perfect equality A in order to estimate GINI index $G = (A - B)/A$. Since the data we use is discrete, one can use the following formula to find the GINI index on a given day:

$$GINI = (A - B)/A \quad (5.2)$$

$$GINI = 1 - \sum_{i=0}^{n-1} (F_{i+1} - F_i)(L_{i+1} + L_i) \quad (5.3)$$

In our study, we adapted GINI index to the software engineering research domain in order to estimate the inequalities within the number of issues owned by different developers over time. The GINI index is calculated based on the ownerships of active issues on a given day by active developers. We define an active issue as an issue that is reported and assigned but not yet resolved. We define an active developer as a developer who owns at least one issue on a given day. On a certain day within the development life cycle, the active issues are owned by a set of developers. The GINI index gets a value between 0 and 1 based on the amount of inequality on a given day, 1 representing total inequality and 0 representing total equality in the number of issues owned.

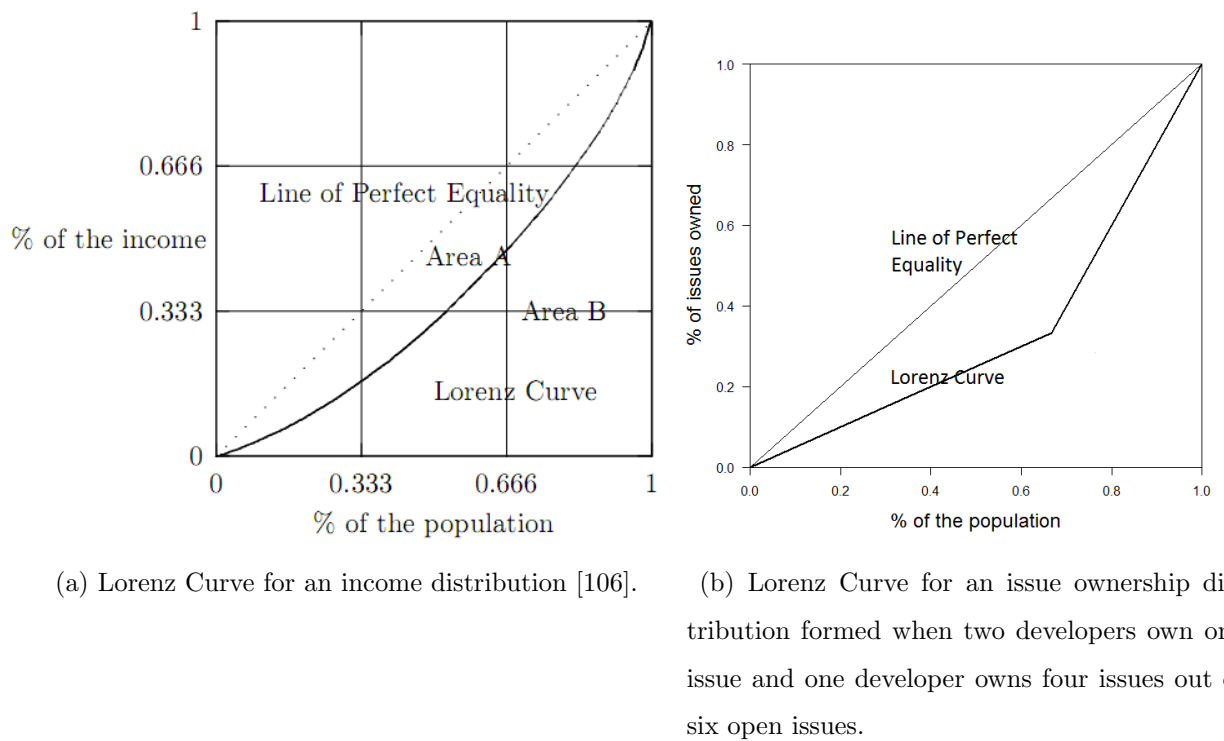


Figure 5.2: Lorenz Curves for income and issue ownership distributions

An Example: Assume that three active developers share the ownership of six issues during the maintenance of a software. If each of the three active developers own two issues there is a total equality in the issue ownership and GINI index is 0. On the other hand, if one of the developers owns four issues while others own one issue each the GINI index increases to 0.42. The Lorenz curves for the the distribution can be seen in Figure 5.2b.

In order to compute GINI index we assume the issues to be homogeneous. We do not use the complexity and context differences among the issues to weight them since the relation of the issue effort and issue complexity is not clear according to the quality assurance team of the enterprise software. It is not possible to find the effort from the issue report since accurate issue effort data is not calculated in the issue management system.

For reducing the inequalities among the developer workloads, we extend the model with two heuristics. The first heuristic is a simple limit to the maximum number of

issues that can be assigned to one developer. The second heuristic uses past commit counts in order to assign maximum number of issues that is correlated with the commit count of a developer.

Heuristic 1 (Static Threshold)

We limit the maximum number of active issues that can be assigned to an individual developer to 5. We can not use a relative number such as number of total active issues over number of developers because the number of active issues will keep fluctuating throughout the span of the project. In the case where everyone has 5 assigned issues the threshold can be incremented in order to work around extreme cases.

The advantage of a simple threshold is its simplicity and lack of severe distribution inequalities. The disadvantage of such a model is that it treats every developer the same. For example, in a project where most of the members work half-time such a heuristic may be problematic.

Heuristic 2 (Dynamic Threshold):

We limit the number of issues that can be assigned based on the effort as a dynamic threshold. In this threshold every developer has a maximum threshold proportional to the log of number of his or her commits divided by all the commits in a project. In order to work around extreme cases every developer has a threshold of at least 1.

An example: Assume that in a toy project dev x,y and z commits 10, 5 and 6 times respectively. In this case if every developer owns a maximum number of issues that is proportional to his or her commit count developer x,y and z will own %40, %28 and %31 of all the issues respectively.

The advantage of setting a dynamic threshold is identifying relatively more active developers and assigning issues based on their commit activity levels. The disadvantage

of such a model is that, it makes the distribution less uniform than heuristic 1. Another disadvantage is the possibility of penalizing productive employees by setting a higher threshold for them and rewarding poorly performing employees by setting a lower threshold for them.

5.4.2.2. Issue Dependency Problem. In our issue assignment scenario we assign issues based on their arrival (reported) times. However, in some cases, a group of issues may be dependent on each other. For example, dependencies may occur if two issues are related to the same critical parts of the software.

In order to test our model in such a scenario, we built a dependency network among issues artificially. We built a random acyclic directed network among all of the issues with a constant occurrence probability of an arc of $P(arc) = 0.05$ that are not connected. We checked the model's performance and assignment inequality (GINI index) in such a scenario afterwards.

Issue dependencies may change the assignment order of issues. Different past issue maintenance activity will change the input of the model. Therefore, in this case the model's recommendations may change.

5.4.2.3. Usage of Kronecker Graph to Address Developer Group Initiation Problem.

Cold start is a common problem in recommender systems and it happens when a new user or item(issue) is added to the system [107]. In our proposed recommendation system, we solved the item(issue) cold start problem with a content-based approach. We categorized the new issue into an existing category and symptom combination and afterwards ran the recommender algorithm. In fact, in our proposed model, each issue is recommended based on the item cold start scenario.

Recommending issues to new developers is a more serious problem for the issue recommender. It is partially the problem of introducing new people to existing software development teams. As the project progresses initiation of new people gets harder.

There is a lot of knowledge that must be transferred to new people. This reduces the productivity of both old and new employees during initiation. Figure 5.5 that is adopted from the book *Peopleware* shows the effect of member change in a software development team in terms of productivity [3]. Fred Brooks claims that the addition of new people may even cause delayed project delivery dates [108].

In the recommender system literature, several strategies are proposed to handle the cold-start problem [107, 109]. In the content-based approach, the system must be capable of matching the characteristics of an item against relevant features in the user's profile. In order to match users with content, a new user may be requested to show their preferences for a number of items. In the collaborative filtering approach, the recommender system would identify users who share the same preferences with the active user, and propose items which the like-minded users favoured. It should be noted that the problem of web based recommenders is very different than the problem of issue assignment. On the other hand, in web based recommenders an item may be preferred by any number of users. In our case, every issue can be owned by one people and preferences of the developers for issues can not be inferred beforehand. Existing solutions are not adequate to address the people cold-start problem of the issue recommender. Therefore, we propose a novel approach using Kronecker networks.

New developer initiation problem has many dimensions like training and social issues. Although we can not propose a solution to all of these problems related to new developer initiation, using Kronecker graphs we can predict how a new developer group will collaborate with the existing teams in the future. We can make recommendations based on the future collaboration predictions to avoid cold-start problems.

Kronecker matrix multiplication was recently proposed for realistic graph generation, and shown to be able to produce graphs that match many of the patterns found in real graphs [110, 2]. The main idea is to create self-similar graphs, recursively. We begin with an initiator graph G_1 , with N nodes, and by recursion we produce successively larger graphs $G_1 \dots G_n$ such that the k th graph G_k is on $N_k = N_1^k$ nodes. Kronecker product is the name of the recursive multiplication operation. We

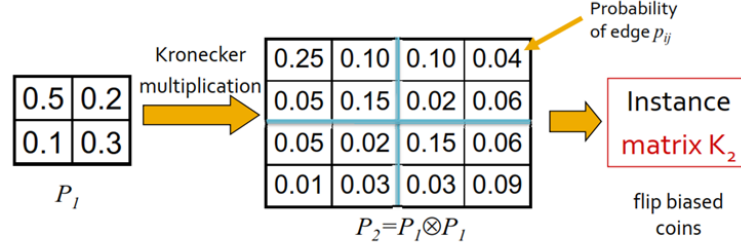


Figure 5.3: One iteration of the adjacency matrix with Stochastic Kronecker generator[2]

use stochastic Kronecker network G_1 as our basis. Stochastic kronecker network can be defined by using any number between 0 and 1 instead of 0s and 1s in the adjacency matrix of the network. In Figure 5.3 the recursive generation example of this network is provided. We use 2×2 initiator matrix for simplicity and easier network structure analysis in our experiments[2].

We use KronFit algorithm to estimate the parameters of G_1 [2]. Details of the algorithm can be seen in Figure 5.4. The actual code of the implementation is 2000+ LOC so it is not practical to attach to this document ¹. Kronfit implemented in C++ without multicore optimization runs in 12 hours for our three datasets on a 6 core PC (Core i7) with 12 GB RAM. The runtime of the naive implementation of KronFit is much higher. For this reason, we employ greedy methods to overcome intractable parts of the problem.

A sample visualization of Kronecker Network estimation of the future graph state is provided in Figure 5.6. Our approach has several limitations since Kronecker graph generator produces no label information. We can only predict how a group of new developers will fit the existing group in the collaboration network for an issue and symptom combination topologically. In addition since Kronecker network has 2^n nodes we needed to apply heuristics to remove nonexistent nodes from the graph. For example if we predict the network topology for 100 nodes, we need to remove the $128 - 100 = 28$ nodes from the generated Kronecker graph with 2×2 Kernel matrix. Removing random

¹The source code of the implementation can be found in the following link: <http://dl.dropbox.com/u/695188/kronecker.cpp>

```

input : size of parameter matrix  $N_1$ , graph  $G$  on  $N = N_1^k$  nodes, and learning rate  $\lambda$ 
output: MLE parameters  $\hat{\Theta}$  ( $N_1 \times N_1$  probability matrix)
1 initialize  $\hat{\Theta}_1$ 
2 while not converged do
3   evaluate gradient:  $\frac{\partial}{\partial \Theta_t} l(\hat{\Theta}_t)$ 
4   update parameter estimates:  $\hat{\Theta}_{t+1} = \hat{\Theta}_t + \lambda \frac{\partial}{\partial \Theta_t} l(\hat{\Theta}_t)$ 
5 end
6 return  $\hat{\Theta} = \hat{\Theta}_t$ 

```

Algorithm 1: KRONFIT algorithm.

```

input : Parameter matrix  $\Theta$ , and graph  $G$ 
output: Log-likelihood  $l(\Theta)$ , and gradient  $\frac{\partial}{\partial \Theta} l(\Theta)$ 
1 for  $t := 1$  to  $T$  do
2    $\sigma_t := \text{SamplePermutation}(G, \Theta)$ 
3    $l_t = \log P(G|\sigma^{(t)}, \Theta)$ 
4    $\text{grad}_t := \frac{\partial}{\partial \Theta} \log P(G|\sigma^{(t)}, \Theta)$ 
5 end
6 return  $l(\Theta) = \frac{1}{T} \sum_t l_t$ , and  $\frac{\partial}{\partial \Theta} l(\Theta) = \frac{1}{T} \sum_t \text{grad}_t$ 

```

Algorithm 2: Calculating log-likelihood and gradient

Figure 5.4: KronFit Algorithm [2]

nodes may dramatically change the structure of the network. Therefore, we remove random nodes iteratively if the removal operation does not affect graph connectivity.

The edge and node weights for the extended collaboration network can not be estimated based on the Kronecker parameters. Therefore when using Kronecker networks, we treated the collaboration network as non-weighted instead of weighted contrary to the base model.

We believe that our approach has three benefits: 1) We propose a novel solution for the cold start problem in our recommender. 2) We predict how the team collaboration evolves over time. 3) We parametrize the collaboration structure which may provide a basis for comparing existing team building strategies.

We designed the empirical setup to test the merits of Kronecker networks in assigning issues to a group of new developers. We could not check the model performance on individual new developers because of the limitations of the Kronecker network model. We extracted the Kronecker parameters from the training state and tested the efficiency of our method for assigning issues to the new developers who are appended to the network afterwards. We applied the extension operation for a new group of developers and for the filtered network in every category-symptom combina-

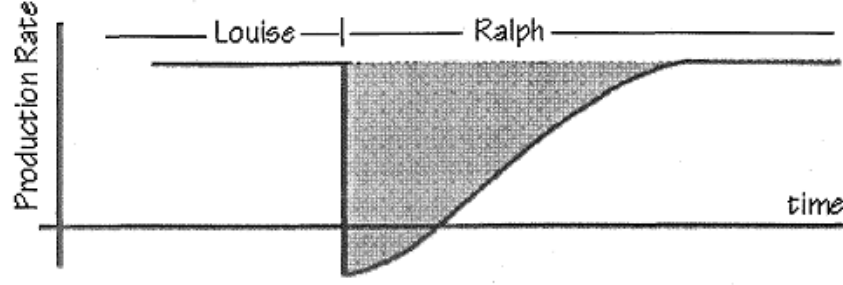


Figure 5.5: The productivity change during a member change in a software development team. Notice that the productivity actually goes negative in the beginning [3].

tion. The points we used for training and testing is provided in Table 5.1. We have January 2009, April 2009 and September 2009 as our training periods and April 2009, September 2009 and December 2009 as the test periods. We used the mean of 1000 random runs in order to reduce the effect of random chances related with Kronecker networks.

We checked the performance of our model in assigning new issues to new developers by the top 5 ranked recommendations of the model. The two performance parameters in our experiment were as follows:

$$Precision = \frac{\text{Number of Issues Correctly Assigned To The New Developer Group}}{\text{Number of Issues Assigned to The New Developer Group by The Model}} \quad (5.4)$$

$$Recall = \frac{\text{Number of Issues Correctly Assigned To The New Developer Group}}{\text{Number of Issues Assigned to The New Developer Group}} \quad (5.5)$$

Similar to the recommender model, these performance measures have certain limitations related to their assumptions of correct assignment. Since, we could not test the merits of our approach in a deployed recommendation system, we had to test our approach on the data related to historical issue assignment. The historical assignments may not be optimal.

5.4.3. Real Time Defect Prediction

5.4.3.1. Proposed Model. Current defect prediction models are mostly tested by cross validating on the same dataset or training on one dataset and testing the model on

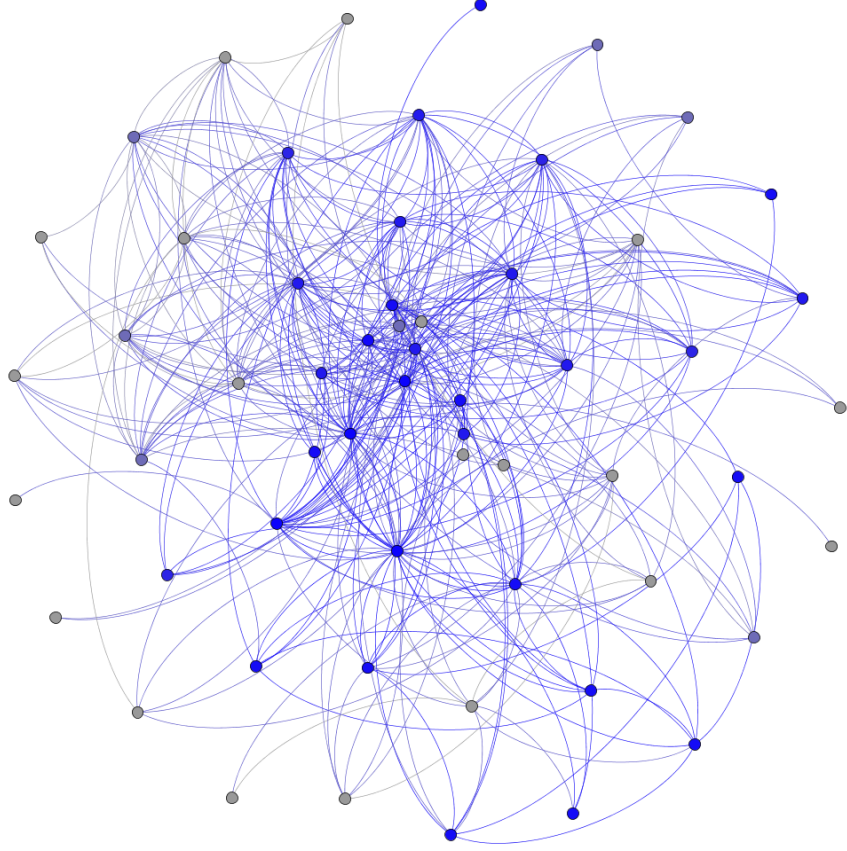


Figure 5.6: In the above generated graph blue nodes represent existing developers and gray nodes represent new developers. By parametrizing the existing developer collaboration using Kronfit algorithm and generating a network based on the number of existing developers plus new developers we can estimate the future state of collaboration

another dataset. The model output is used by testing teams to increase the efficiency of their testing process. However, during software development, defect prone module information may be needed real-time. If a module has been labeled as *not defect prone* by the model it always predicts that module to be not defect prone over time. Our model mixes direct past knowledge with the outcome of a defect predictor in order to make it learn from its mistakes even after the training phase is complete. We call this model real time since the model output changes for every change in the source code.

In this case our model uses a mixture of two experts to estimate defect prone modules. The first expert which we will name as E_m uses the outcome of a traditional

Table 5.1: Empirical Setup for the New Developer Assignment Problem Using Kronecker Networks

Date	# of Developers	Increase	Training	Test
Jan-09	32	-	✓	-
Feb-09	38	6	-	-
Mar-09	51	13	-	-
Apr-09	63	12	✓	✓
May-09	81	18	-	-
Jun-09	82	1	-	-
Jul-09	83	1	-	-
Aug-09	93	10	-	-
Sep-09	96	3	-	✓
Oct-09	101	5	-	-
Nov-09	107	6	-	-
Dec-09	107	0	-	✓

defect prediction model as its data source. If it encounters a defective label there, it suggests that a module has that label at any future time based on the predicted probability. The second expert is a dummy one that we will name as E_h . It only suggests modules which have defects or have defects among their neighbors in the call-graph in the revision history to be defect prone. If we mix the outcomes of these experts a conflict may occur among their outcomes. In this case, we decide the outcome by weighting the experts based on the past defect counts. Therefore in the core of the model there are two parts: 1) Re-weighting method for the weights of the two experts over time 2) The heuristics with which E_h labels the modules as defect prone.

For the machine learning based predictor E_m , we used the Naïve Bayes algorithm with log-filter pre-processing. Naïve Bayes is one of the best performing algorithms for the defect prediction problem based on the benchmarks by Lessmann et al. [21] [44]. Naïve Bayes assumes all of the input metrics to be independent and is optimal for a problem with independent attributes [111]. One may assume that independence

assumption may hinder performance since there is a high correlation among the static code metrics [111]. However, Domingos and Pazzani showed that the independence assumption is a problem only in a small number of cases [111], [112]. These findings may explain the past success of the Naïve Bayes algorithm in defect prediction models [111]. We used the churn and static code metrics as the metric set and used Prest tool which has been developed in Softlab to extract the static code metrics [113]. Full list of metrics used and their basic statistical analysis can be found in the Appendix B.

E_h labels modules as defect prone if it had at least one defect previously. In the call graph of software if the module A has N neighbors (connected with edges pointing either direction) and K of the neighbors are labeled as defective we label the node A as defective with a probability K/N as defect prone. E_h searches up to its immediate neighbors in the call graph. As a summary, E_h looks around the module in the call graph locally and if it finds nearby defect prone modules it labels the module as defect prone with a probability.

Second part of the model is re-weighting part. Weights of E_h and E_m starts at 0 and 1 respectively. In other terms, we use the learning model exclusively in the case of a conflict. After that, at any given future time we re-weight the experts for each change in the module state. Module state changes when the module or any of its neighbors in the call-graph are changed. State change is associated with a defect if the module or majority of its neighbors is changed due to a defect. A visual representation is presented in Figure 5.7: A short description of the steps of the model is provided below:

- (i) E_h labels a node as defective if it had at least 1 defect or based on the fraction of its neighbours in the call-graph having defect.
- (ii) E_h and E_m weights are initialized to 0 and 1 respectively.
- (iii) Weights are re-evaluated at each turn.
- (iv) At any future state of the module E_h weight is equal to number of changes on the module or majority of its neighbors due to defects over number of all changes to the module or its neighbors in the call-graph.

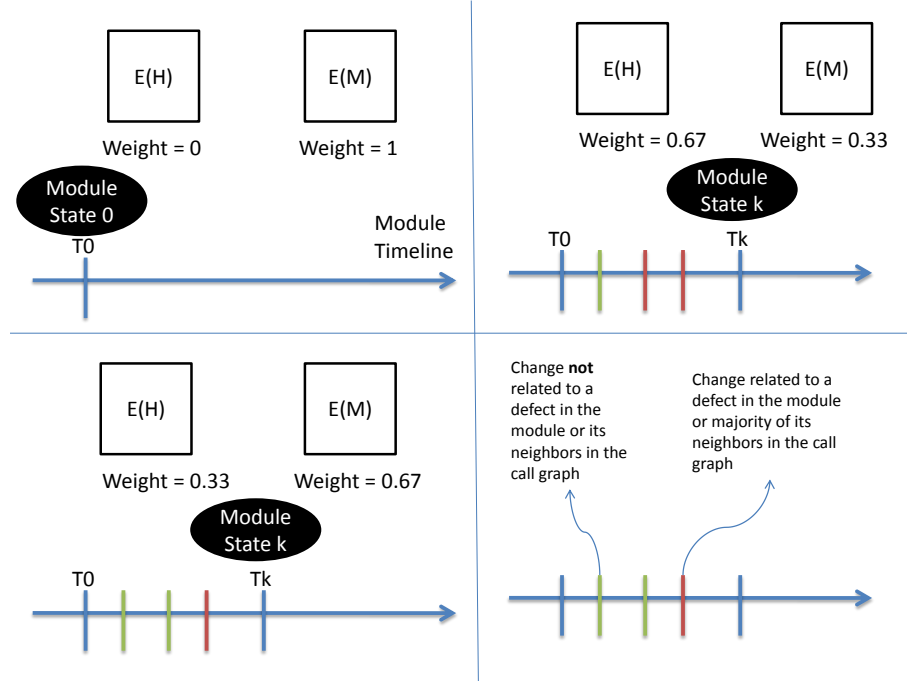


Figure 5.7: Reweighting method of real time defect prediction. Initially E_h weight is set to 0 and we exclusively use the classifier output. After module or its neighbors in the call-graph changes for a number of times we re-weight E_h as number of changes due to defects over number of all changes in the module or its neighbors.

5.4.3.2. Performance Measures. In order to assess the performance of our defect predictor on various metric sets, we have calculated well-known performance measures: probability of detection (pd), probability of false alarms (pf) rates and f-measure [114]. Pd, which is also defined as recall, measures how good our predictor is in finding actual defective modules. Pf, on the other hand, measures the false alarms of the predictor, when it classifies defect-free modules as defective. In the ideal case, we expect from a predictor to catch all defective modules ($pd = 1$). Moreover, it should not give any false alarms by misclassifying actual defect-free modules as defective ($pf = 0$). The ideal case is very rare, since the predictor is activated more often in order to get higher probability of detection rates [114]. This, in turn, leads to higher false alarm rates. Thus, we need to achieve a prediction performance which is as near to (1,0) in terms of (pd,pf) rates as possible. These parameters are found from the confusion matrix shown in table 5.2. The optimum pf,pd rate combinations can vary from project to project. On a safety critical project pd rate can be more important while on a budget

Table 5.2: Confusion Matrix

	actual	
	defective	defect free
predicted		
defective	A	B
defect free	C	D

constrained project low pf rate can be more important for resource allocation.

$$pd = \frac{A}{A + C} \quad (5.6)$$

$$pf = \frac{B}{B + D} \quad (5.7)$$

$$Precision = \frac{A}{A + C} \quad (5.8)$$

$$Recall = pd = \frac{A}{A + B} \quad (5.9)$$

$$f_measure = 2 * \frac{Precision * Recall}{(Precision + Recall)} \quad (5.10)$$

6. Experiments and Results

6.1. Descriptive Statistics

We observed the issue ownership activity for one major release of the enterprise software. In total, 1412 issues were reported and resolved within the span of the observed period between January 2008 and July 2010. 1409 of the issues were associated with code defects. We limited the dataset to the issues which have been opened and resolved within the release period.

Based on data, we think that there is a well defined issue management process in the organization. There are two main responsibilities in the issue management process, namely the issue originator and the issue owner. Issue originator is an employee or a key beta user who reports an issue about the enterprise software. In the issue report, the originator needs to define a category and a symptom for the issue by choosing from a drop down menu. In addition, the originator reports the phase found, classification and the problematic module in the issue management software. Issue owner changes the status of an issue based on predefined rules. We observed all the activity related to the issues within this period. The state diagram of all possible states of an issue can be seen in Figure 6.2 for the enterprise software. Based on the evidence from the data, we believe that issue owners and the originators adhered to all policies and procedures.

The enterprise software can be classified as a globally developed software [17]. Herbsleb et al. showed previously that coordination costs increase for globally developed software [17]. For such software, an automated issue recommender may be helpful in reducing the coordination cost. Enterprise software is developed in five distinct locations and the majority of the development is done in Canada. The geographical dispersion of the issue owners and the originators can be seen in Figure 6.1. Around seventy five percent of issue reporters worked from Canadian branch of the company and eighty three percent of the issues were reported from Canada. Similarly, forty one percent of the issues were owned by developers located in Canada and seventy two per-

cent of the issue owners are located in Canada. Following Canada, USA and China are the second and third most active countries respectively and there is a marginal amount of issue activity in Ireland and India. In addition to the five countries mentioned, there is a small amount of issue ownership activity in Australia, Brazil, France, Germany and UK. According to the quality assurance team, issues owned in these locations are owned by employees who work remotely with the teams in Canada.

Three categories of issues are defined within the organization:

- FT Issues: Issues found during function test [115].
- ST Issues: Issues found during system test [115].
- Field Issues: Issues found in the field (by the customers).

Of all software methods that have defects during functional verification testing, 20% also have defects during system verification testing, whereas 33% also have defects discovered in the field. On the other hand, among all software methods that have defects during system testing, 27% of them also have defects discovered in the field. Of the software methods that have both functional and system test defects, 55% also have defects discovered in the field. This ratio is very high; however, these software methods constitute 8% of all field defects, i.e., 37 out of 445 field defects.

We conducted Spearman rank correlation tests to check whether defect types are correlated with each other. Table 6.2 presents correlation coefficients between the number of defects regardless of their type (ALL), the number of defects found during function test (FT), the number of defects found during system test (ST), and the number of defects found in the field (Field). It is seen that FT and Field defects are correlated with ALL defects with correlation coefficients around 60%, since their percentages among all defects are the highest. However, FT, ST, and Field issue types are not correlated with each other. Hence, they should be coming from different distributions. Mann-Whitney U-tests also show that the three defect types have significantly different medians ($p < 0.05$).

Symptoms are anomalies which cause testers to report the defect, e.g., “incorrect io” or “core dump”. In Table 6.1 symptom frequencies for each defect category can be seen. Every defect maps to a symptom in the project. The frequencies of different symptom observations are dramatically different for different defect categories. The most frequent symptoms in FT and ST defects are “function needed”, “program defect” and “test failed” if we order them in terms of their frequency of occurrence. In ST defects “test failed” symptom type is much more concentrated than in FT defects. The difference in symptom distributions in different categories of defects suggest that the defect categories have different underlying mechanics. This gives a possible explanation of the reason behind the different distributions for different metric categories. The most frequent field defect symptoms are “core dump”, “program defect” and “program suspended”. One reason for this is perhaps the methods that identify the symptoms of field defects are much more end-user oriented since testing in customer IT environment is inherently different than testing in-house. For example “core dump” is usually stored in the customer computers and may be sent to the company for identifying the reason of the malfunction. These findings suggest that the different categories employed by the company are associated with different problems in the source code.

We investigated the relationship among the defect categories to identify the likelihood of occurrence of different defect categories in the large-scale enterprise software dataset. Of all software methods that have defects during functional verification testing, 20% also have defects during system verification testing, whereas 33% also have defects discovered in the field. On the other hand, among all software methods that have defects during system testing, 27% of them also have defects discovered in the field. Of the software methods that have both functional and system test defects, 55% also have defects discovered in the field. This ratio is very high; however, these software methods constitute 8% of all field defects, i.e., 37 out of 445 field defects. Therefore, field defects are difficult to predict using defects found during functional and system verification tests.

6.1.1. Collaboration Network Analysis

The structure and evolution of the enterprise software project over one year time range show the general properties of the scale-free networks. The basic network statistics, their evolution and related Kronecker coefficients for the fitted network for each month can be seen in Table A in the Appendix.

The network consists of one giant component that is composed of all the nodes (developers) in the network in all months. In other words, there is no disconnected periphery network. The network diameter is small related to a random network with similar size. This is a common property in scale free networks [10].

The most common trend during the evolution of the collaboration network is densification over time. The number of nodes in the network increases ~ 3 times in the 13 month period while the number of edges increase ~ 17 times during the same period. The network diameter stays constant even though the network size increases significantly because of the densification effect. Other parameters such as mean clique size and mean clustering coefficient increases significantly with the densification. The densification effect may have several problems for the recommendation system in the long term. If the densification continues and the network becomes a complete network collaboration based recommendation approach may not be useful in the end. In this case, filtering edges may be helpful.

We estimated how a new group of developers will enter the collaboration network in a given month with the Kronecker parameters. Kronecker parameters that fit the network in a given month gives clues about the networks structure. The A parameter is the link probability in the core component. B and C are the link probability between the core and the peripheral parts of the network. We observe that the common $A > B \cong C > D$ trend is seen for all months in Table A in the appendix [67]. The B, C and D parameters increase over time. This is consistent with the observed densification over time.

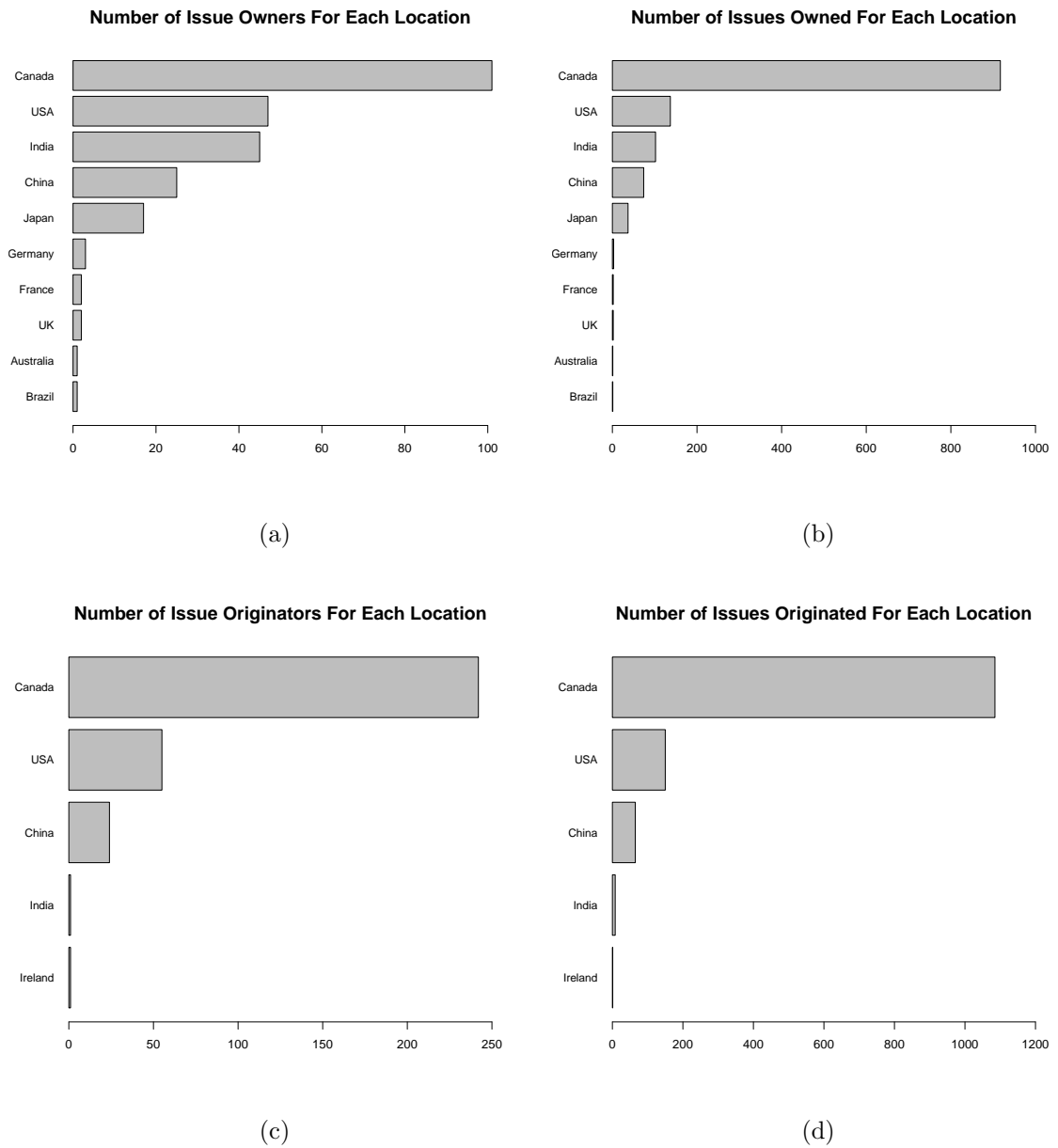


Figure 6.1: Issue owner and originator locations for the enterprise software

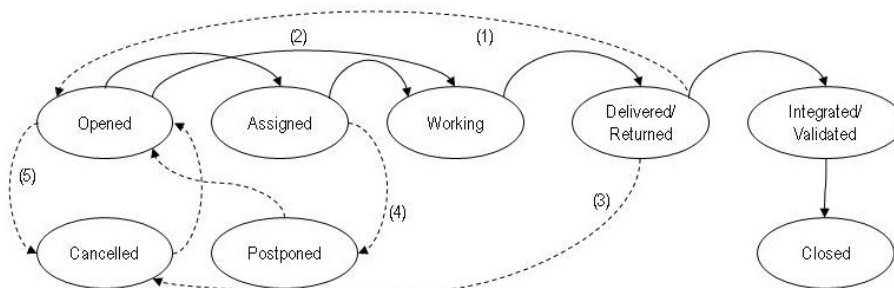


Figure 6.2: State Diagram of the Issues For The Enterprise Software.

Table 6.1: Symptoms for each defect category in the enterprise project

Symptom Count	in FT Defects	in ST Defects	in Field Defects
incorrect io	158	45	5
plans incorrect	22	3	0
build failed	2	0	0
non standard	7	0	0
function needed	332	29	6
core dump	96	18	34
prog suspended	30	0	25
program defect	198	29	30
Build failed	43	0	0
corrupt dbase	2	0	1
performance	108	7	1
install configuration	1	1	0
install add remove files	1	0	0
mixed code releases	1	0	0
obsolete code	9	0	0
docs incorrect	1	0	0
intgr problem	3	0	0
program loop	6	1	0
not to spec	60	21	1
reliability	50	10	7
test failed	398	311	25
usability	28	1	0

Table 6.2: Spearman rank correlations between the defect categories

Defect Type	FT	ST	Field
ALL	0.604	0.466	0.617
FT	1	0.247	0.477
ST		1	0.466
Field			1

6.2. Patterns of Issue Timeline and Ownerships and Issues Relation With Defects In Code

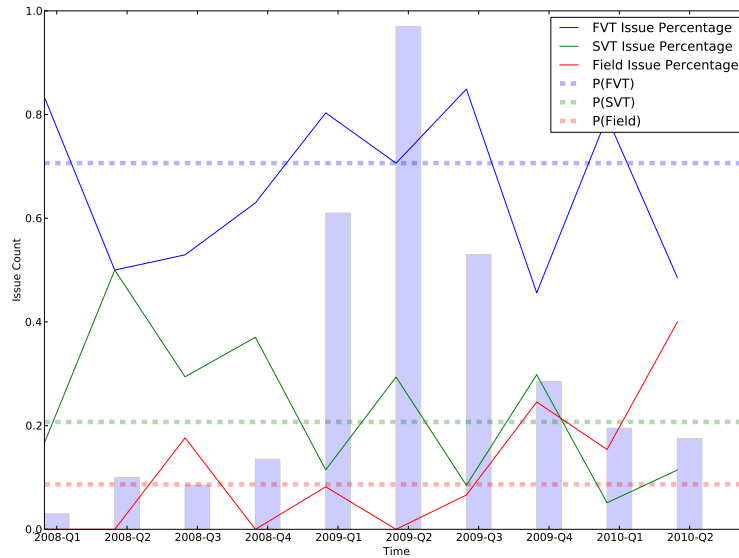


Figure 6.3: Changes in issue category percentages of the enterprise software over time. Dashed line is the overall probability of different categories of issues. Changing lines are the distribution of issue categories for that quarter. Bars are the number of issues reported in a given quarter. Note the decrease in issue reports by system test teams before release. After the release (2009-07) relative number of field issues increase rapidly. The most active quarter is the 2009-02 in terms of issues that were reported. The product is released just after this quarter.

6.2.1. Hypothesis I: Limited number of developers own majority of issues

In order to test this hypothesis we checked the histogram of ownership counts by developers for Android and the enterprise software. We see that for three categories of defects in the enterprise software and Android issues the same trend holds. 20% of the developers own more than 80% of the issues. Some reasons behind this trend may be differences in experience, differences on time allocation and differences in issue hardness or complexity. A more detailed analysis of ownership over time can be found in Figure 6.4. In these figures we notice that *activity ranges* of developer are significantly different.

The number of open and owned issues show interesting trends both for enterprise software and Android. The complete timeline of active issues can be seen in Figure 6.8a. The number of active issues peak at 260 three months prior to the release for the enterprise software. For Android the maximum number of active issues is 120 and it was observed on July 2010. Within the Enterprise Software dataset, the number of active issues reported by in-house testing teams decreases as the product release date gets closer. However, in Android dataset the similar trend is not clear. This may be due to the different release cycles of the two software projects. While the enterprise software has a multi-year release cycle, Android had more than one release within one year period.

The GINI index and the number of active issues are correlated with $\rho = 0.85$ and $\rho = 0.81$ for the enterprise software dataset and the Android dataset respectively. The GINI index range was between 0 and 0.50 for the Enterprise Software and between 0 and 0.58 for the Android dataset. The changes in active issues and GINI index can be seen in Figure 6.8a. The positive correlation among the two variables is statistically significant with $p < 0.001$ for both datasets. Therefore we accept the alternate hypothesis H_1 : The number of issues that developers work on gets more unequal as the number of active issues increases.

The concentration of responsibility on a few developers may be harmful for the project in the long term. Being dependent on a few people is not a sustainable policy for a large software development project. Key employees may leave the project sometime during software development life cycle taking several decades of their unique experiences with them. Software projects should not be vulnerable to people turn over. In addition, especially after releases, managers should have contingency plans to handle unexpected changes in the active issues. Planning ahead on distributing certain categories of issues may reduce the dependency on a few developers. More uniform allocation of issues may also reduce the issue resolve durations and improve customer/user satisfaction.

In Figure 6.7 the positive outliers in the issue ownership distribution can be seen.

The distribution of issue ownerships has a long tailed distribution. The top issue owner in the Enterprise Software dataset owns more than 10 times the number of issues of the median. The average issue resolve duration is 129 days for enterprise software and 113 days for Android. The issue resolve time difference is not statistically different (Mann Whitney-U $P < 0.05$) for both Android and Enterprise Software datasets.

6.2.2. Hypothesis II: Issue reports of users change significantly after releases.

In Figure 6.8 the changes in open issues for Android and Enterprise Software over time can be observed. As evident from the figure, it is hard to make confident conclusions. We estimate that a significant amount of open issue increase would be a 20 per cent increase of open issues, 2 weeks after a release. When we use this metric to check if there is an abnormal increase in issue counts we see that for only 2 releases out of 9 for Android this pattern holds. For the enterprise software the pattern is not evident for one major release we investigated. We can not claim that HII holds for the dataset with 95 percent confidence based on Mann-Whitney U test. Based on the evidence, we conclude that abnormal increases in open issues is not associated with software releases for the two software.

6.2.3. Hypothesis III: Issues map to multiple defects in source code

In Table 6.3 some key statistics for defect per issue in both Android and Enterprise Software are provided. Although mean number of defects per issue is more than 2 for all issue types and datasets, one can observe that the median is 1 meaning more than half of the issues contain only one defect. If we remove the outliers in the distribution we observe that the mean value also reduces significantly. When we remove the outliers beyond $mean + (upperquartile - lowerquartile) * 3$ the mean defects per issue reduce to 1.46 and 1.37 for Android and the Enterprise Software respectively. Based on the evidence we can conclude that most of the issues map to a single defect in the software module.

Dataset	Min	Max	Mean	Upper Quart.	Median	Lower Quart.	Std
Enterprise S.	1	46	2.28	1	1	1	2.41
Enterprise S. (FVT)	1	24	2.32	1	1	1	2.32
Enterprise S. (SVT)	1	10	1.87	1	1	1	1.87
Enterprise S. (Field)	1	46	4.12	2	1	1	4.12
Android (Field)	1	82	3.86	1	1	1	6.17

Table 6.3: Defects Per Issues For Enterprise Software and Android Datasets

6.2.4. Hypothesis IV: Issue resolve times are dependent on number of issues that a developer own

In order to visualize this information we plotted the issue average resolve times versus total issues owned by the developer. In Figure 6.9 the relevant plots and correlation coefficient values can be seen. We found no significant correlation between the issue resolve times and the number of issues owned. This result indicates that the experience of a developer do not affect the issue resolve times. From the data we can conclude that developers who solve the highest number of issues do not resolve issues significantly earlier or later for both the enterprise software and Android. Therefore, this hypothesis do not hold.

6.2.5. Discussion of The Hypothesis Findings

Our results show that the majority of issues are owned by a small group of developers irrespective of the size of the software product [97]. This fact may cause several problems: 1) Too much dependence on individual performance, 2) More workload on certain programmers, 3) Chaos and panic in unexpected situations. These may also make teams abandon their well-defined processes and return to ad-hoc development practices. We believe that dealing with these problems efficiently is possible with a long-term strategy. An automated recommender which uses ownership equality as a performance measure may help to deal with the issue inequality. Related to other process changes, we make the following recommendations for building strategies on issue management in the long term:

Develop a long-term plan for the distribution of the workload among active developers

A significant increase in developer experience is not possible in the short-term. Collective code ownership of developers may increase the interchangeability of developers of the project in the long-term. Different methods may be used to increase the collective code ownership of developers as advised by agile software development community [116]. These methods include using pair programming in development, changing developer responsibilities in software periodically and encouraging communication among developers.

Assigning an issue to a junior and a senior developer together may also facilitate transfer of experience between developers. A junior developer can learn new defect detection and correction methods by working with an experienced developer. This may reduce the excessive workload of experienced developers over time.

Get prepared for emergency scenarios in advance

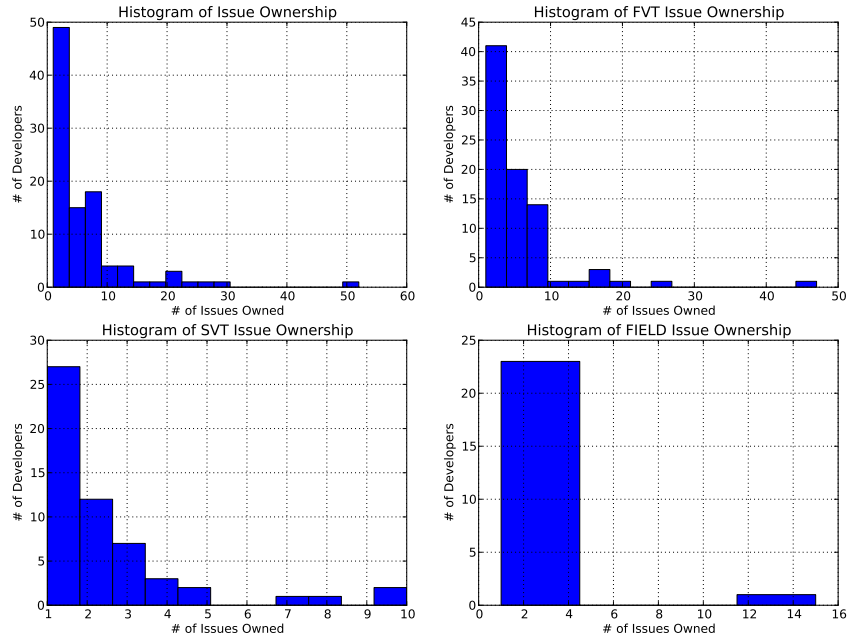
In our context, an emergency situation means the sudden increase in the amount of active issues. Other emergencies in issue management may occur when a very critical issue is reported that should be resolved as soon as possible. Various pre-defined action plans should be developed to cope with these situations in order to avoid confusion. Analysis of past data may aid in defining new strategies. If there is a sufficient number of developers who have experience in dealing with different types of issues, planning for emergency situations would be easier.

Track issue ownership distributions over time and try to find the cause(s) of inequalities

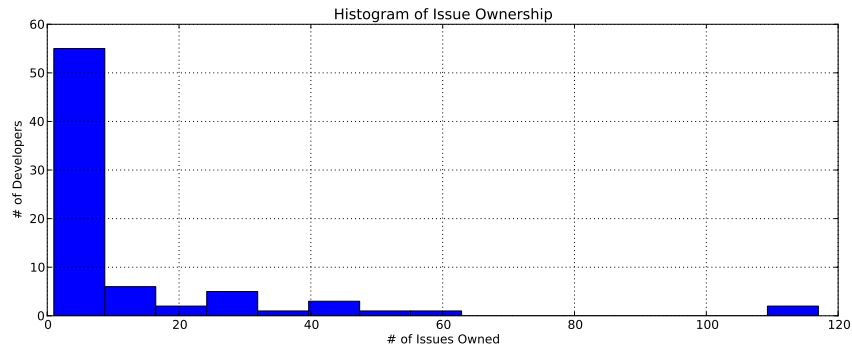
The factors that cause inequalities in issue ownership may change from project to project and in some cases be outside the scope of issue management. These factors may include poor quality of a software component developed and maintained by a

single developer, over-commitment of certain developers and competitions within the development team. Issue ownership inequalities observed in a software product may be a symptom about underlying problems in the development team and software quality. Actively tracking the inequalities in issue ownership and its change over time may help development teams in identifying these potential problems.

Our results do not indicate that the developers with more issues owned are more productive. The number of issues owned or issue ownership patterns over the project life-cycle are misleading in estimating the productivity of developers since such a measure would ignore many external factors. Factors such as the other development and design assignments of developers may have created ownership inequality. Simple measures such as ratios may lead one to erroneously consider them to estimate the productivity of developers. In a systematic review of the literature on software productivity, Petersen concluded that ratio measures are misleading in general for productivity models [90].



(a) Enterprise Software



(b) Android

Figure 6.4: In this figure we see that for three categories of issue in the enterprise software and Android a similar trend can be observed. The distribution of issue ownerships among developers comply with 80-20 Pareto law. Note that for Android issues can be categorized as field issues since they were reported by customers.

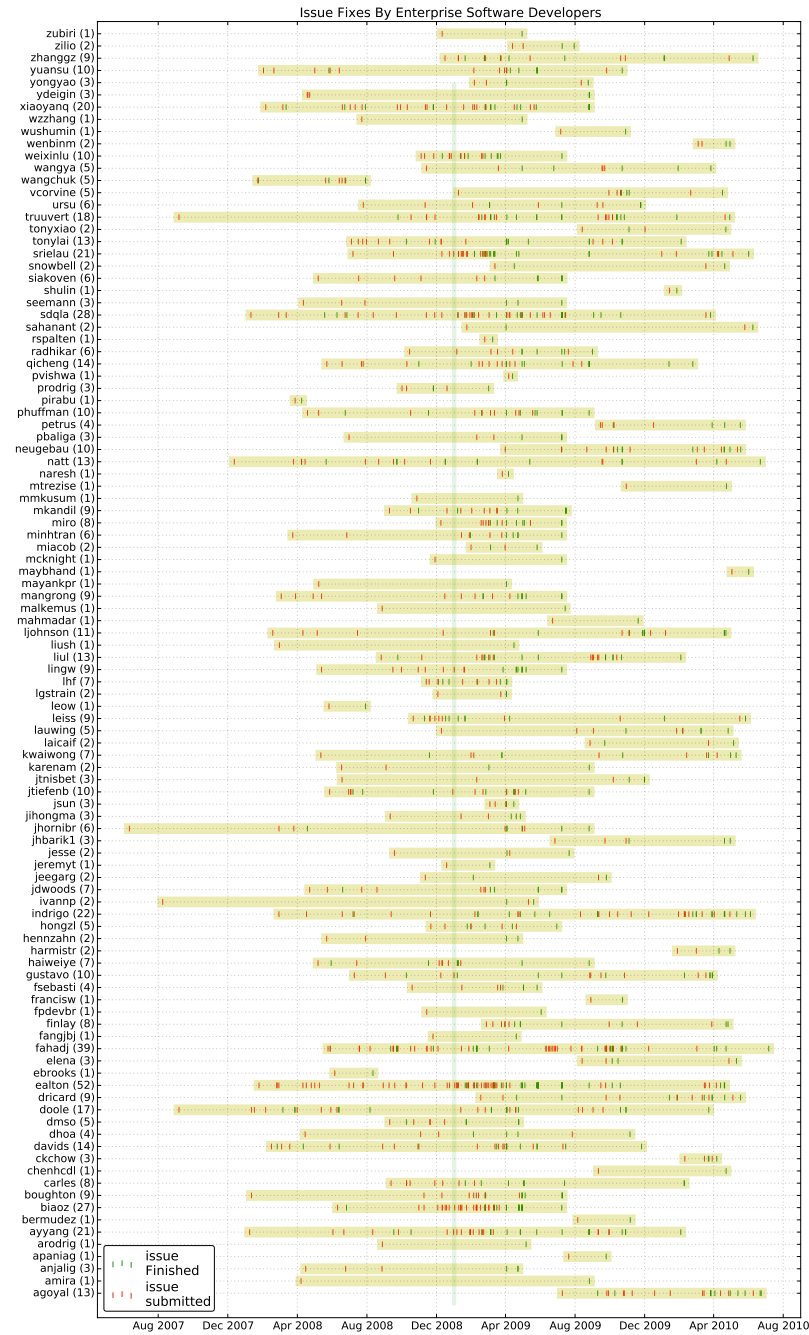


Figure 6.5: Issue resolve of each developer over time for Enterprise Software. Highlighted lines are time periods in which a developer had at least one issue open. Red and green ticks are issue resolve and ownership times respectively.

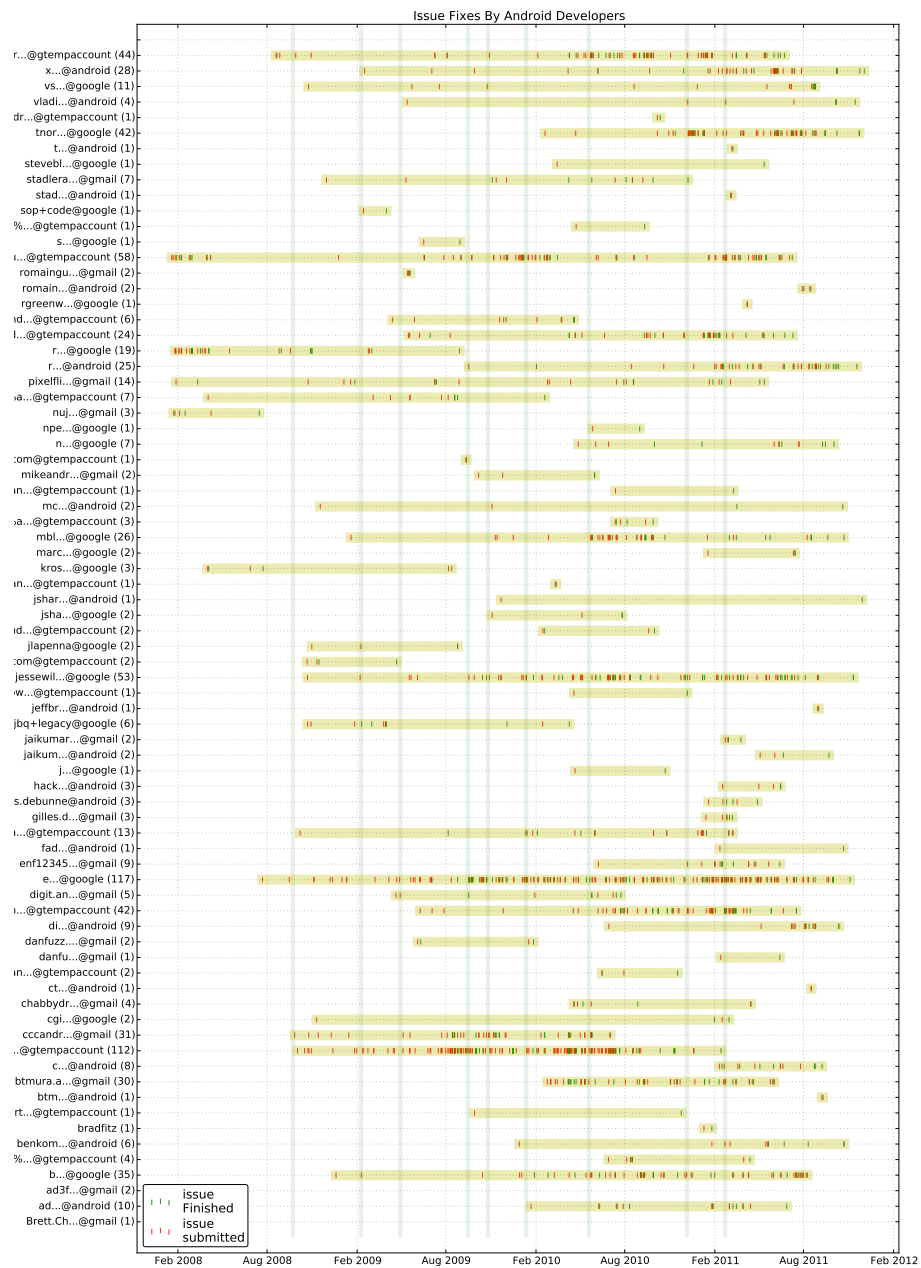
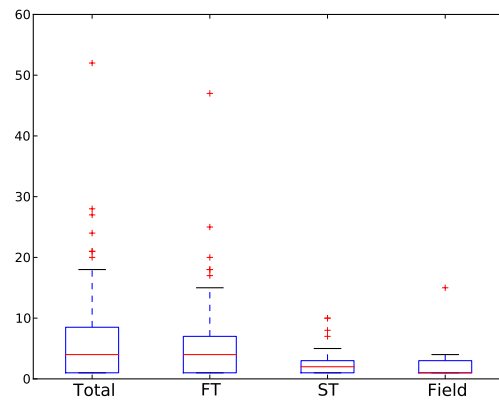
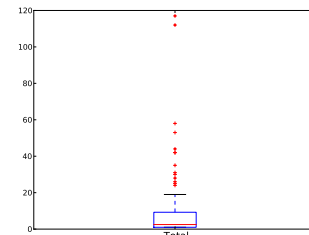


Figure 6.6: Issue resolve of each developer over time for Android. Highlighted lines are time period in which a developer had at least one issue open. Red and green ticks are issue resolve and ownership times respectively.

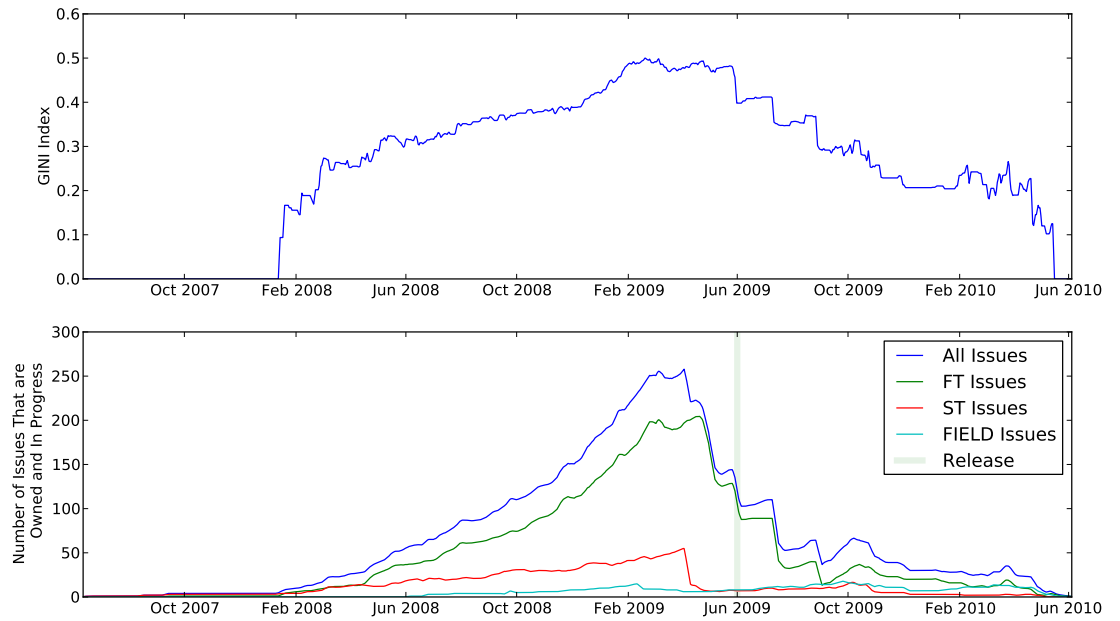


(a) Enterprise Software

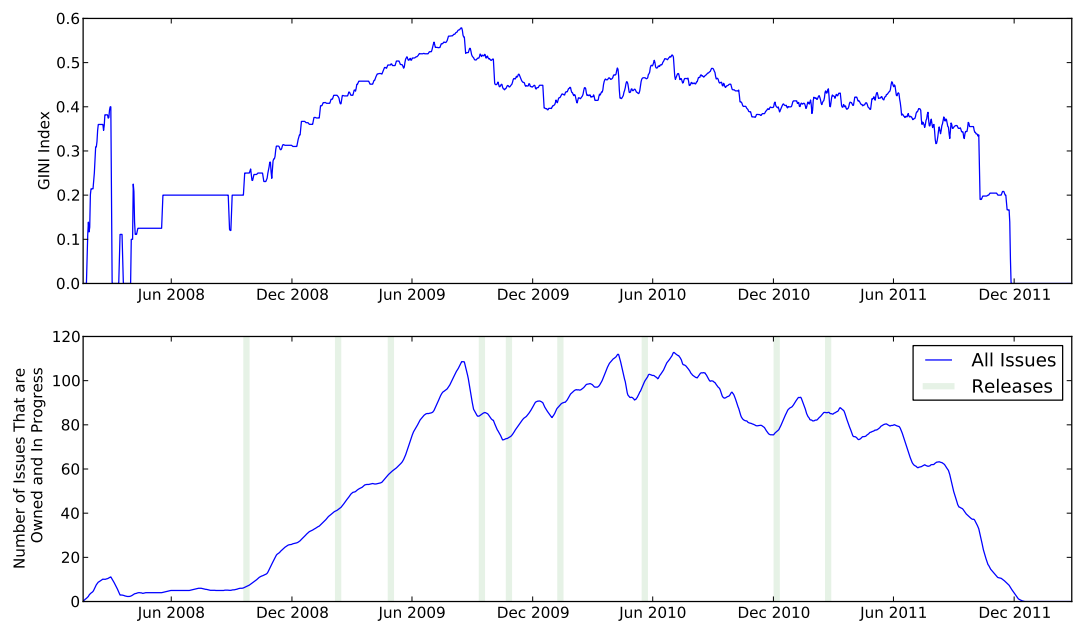


(b) Android

Figure 6.7: Box plots of number of issues owned for enterprise software and Android. For the Enterprise Software ownership of issues in three distinct categories are also plotted.

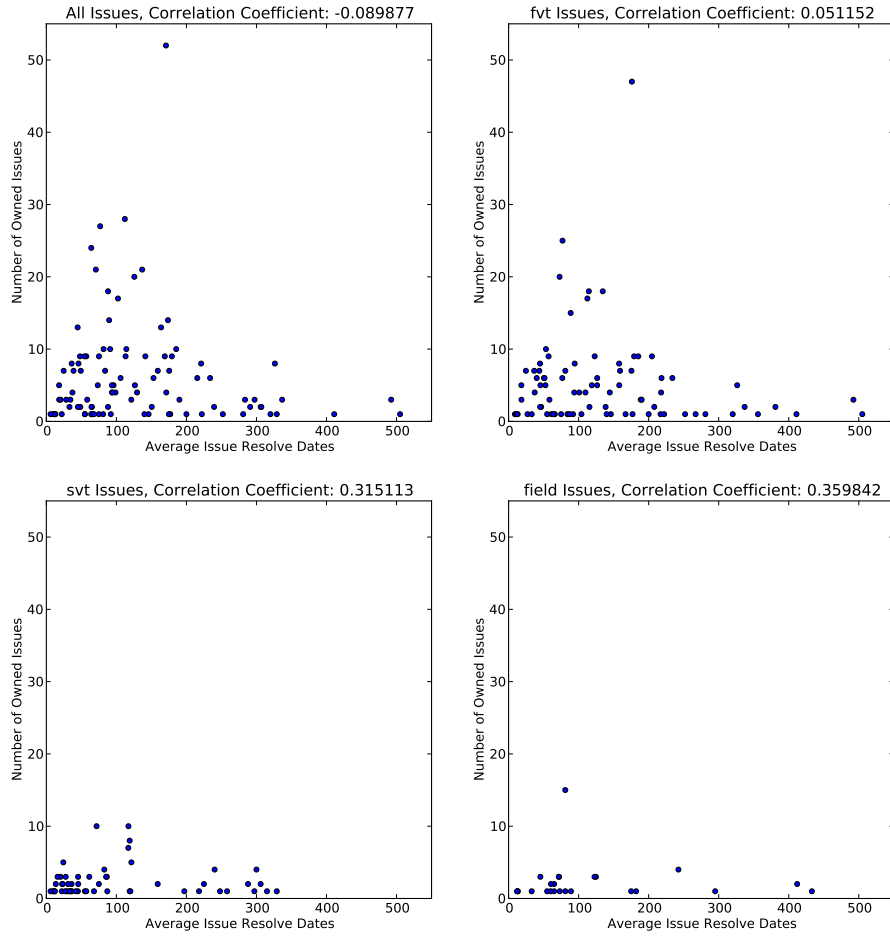


(a) Enterprise Software

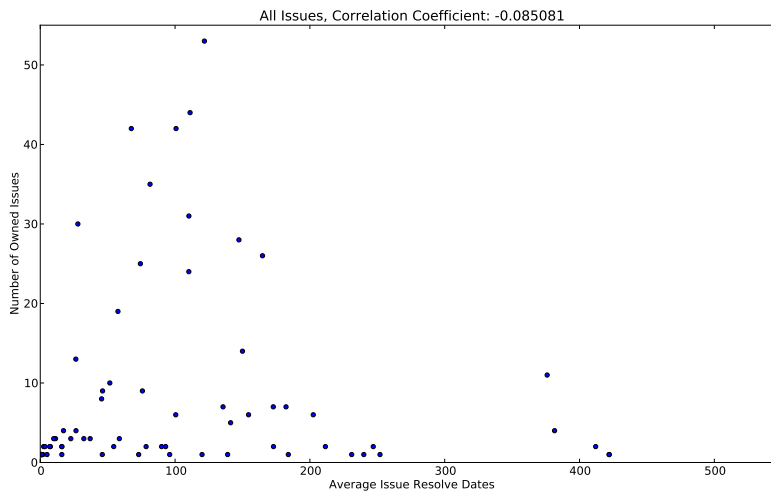


(b) Android

Figure 6.8: Change in number of issues that are actively worked on during the timespan. Green vertical lines identify release dates of the software. In the enterprise software trends for different categories of bugs were observed separately.



(a) Enterprise Software



(b) Android

Figure 6.9: Average issue resolve times over number of issues owned by developers.

6.3. Recommendation System Component Results

6.3.1. Issue Recommendation Results

6.3.2. Base Model

We used the Enterprise Software dataset to test the issue recommender. We trained the model until a fraction of T where T is the difference between the development end and start times in the dataset. In the case of Enterprise Software the time span is between October 2007 and June 2010 or approximately 1k days.

For the basic performance measure $P(1)$ base model provides significantly better results than previous models that were tested on other datasets [19] (Mann-Whitney U Test $P < 0.05$). Comparing results on two different datasets is not right as a benchmark, however this comparison highlights the potential of our approach. On the other hand, model accuracy is greater than 90% for top five ($P(5)$) and top ten ($P(10)$) recommendations.

Issue ownership inequality is an important drawback of the base model. By approximating the actual issue assignments, our basic model provides a high GINI index. In other words issue workload distribution is highly inequal. The GINI index range between 0.76 and 0.83 which is even greater than the actual GINI index of 0.67 among the actual issue owners.

Dataset	Train Period	GINI Index	P(1)	P(2)	P(5)	P(10)
Enterprise Software	0-T/4	0.83	0.62	0.93	0.95	0.97
Enterprise Software	0-2T/4	0.76	0.73	0.86	0.91	0.95
Enterprise Software	0-3T/4	0.81	0.81	0.97	0.97	1.00

Table 6.4: Performance of The Basic Issue Recommender

Dataset	Train Period	GINI Index	P(1)	P(2)	P(5)	P(10)
Enterprise Software	0-T/4	0.17	0.23	0.37	0.38	0.39
Enterprise Software	0-2T/4	0.26	0.18	0.28	0.31	0.37
Enterprise Software	0-3T/4	0.32	0.19	0.26	0.29	0.35

Table 6.5: Performance of The Issue Recommender With Static Load Balancing. GINI index is lowest.

Dataset	Train Period	GINI Index	P(1)	P(2)	P(5)	P(10)
Enterprise Software	0-T/4	0.38	0.41	0.53	0.57	0.61
Enterprise Software	0-2T/4	0.46	0.37	0.57	0.62	0.65
Enterprise Software	0-3T/4	0.31	0.36	0.59	0.64	0.63

Table 6.6: Performance of The Issue Recommender With Dynamic Load Balancing

6.3.3. Workload Balance Problem

We applied two heuristics that we mentioned previously to build the basic issue recommender in order to reduce the inequalities in issue workload distribution in the base models. In Tables 6.5 and 6.6 results of the two approaches can be seen. If we use the static load balancing (limiting maximum issue numbers to 5) as the heuristics, the performance of the recommender (in terms of mapping the actual issue assignments) is significantly lower. However, maximum workload inequality is also very low among the developers. On the other hand, if we use dynamic load balancing, the performance of the model is better than static load balancing although the workload inequality is higher than static load balancing. Based on our results, we observe that there is a clear trade-off between the model accuracy and the issue ownership distribution inequality.

6.3.4. Dependency Problem

In this experiment we created a dependency relation among issues by creating a random network among them. The dependency relation among issues change the assignments times of issues and may change the recommendations of the model. In Tables 6.7 and 6.8, the mean of the results of creating dependency network randomly

Dataset	Train Period	GINI Index	P(1)	P(2)	P(5)	P(10)
Enterprise Software	0-T/4	0.37	0.07	0.25	0.29	0.31
Enterprise Software	0-2T/4	0.41	0.09	0.21	0.24	0.26
Enterprise Software	0-3T/4	0.32	0.14	0.19	0.27	0.29

Table 6.7: Performance of The Issue Recommender With Issue Dependencies and Heuristics 1

Dataset	Train Period	GINI Index	P(1)	P(2)	P(5)	P(10)
Enterprise Software	0-T/4	0.53	0.16	0.23	0.38	0.41
Enterprise Software	0-2T/4	0.48	0.11	0.22	0.28	0.39
Enterprise Software	0-3T/4	0.51	0.13	0.19	0.27	0.42

Table 6.8: Performance of The Issue Recommender With Issue Dependencies and Heuristics 2

for 100 times are given. From the results we can observe that inequality among the issue owners increase while the approximation to reality for the model decrease. This result, indicates that artificial dependency relations among the issues may reduce the model performance dramatically.

6.3.5. Assignment of Issues to New Developers Using Kronecker Network

The mean precision and recall values for 1000 random runs are provided in Table 6.9. The mean precision over all training testing combinations are 0.755 precision and 0.63 recall on average for P(5). The precision rates are more stable than the recall rates in the model. Our results show that we can assign issues to the new developer group with considerable success. If we take into account that we do not know the attributes of the new developer group apriori, this result is rather surprising. We may speculate that the way a new developer group will be appended to an established collaboration is dictated by the past collaboration.

We think that the changes in the model performance for different month combinations may have several reasons: 1) Number of new developers percentage is different

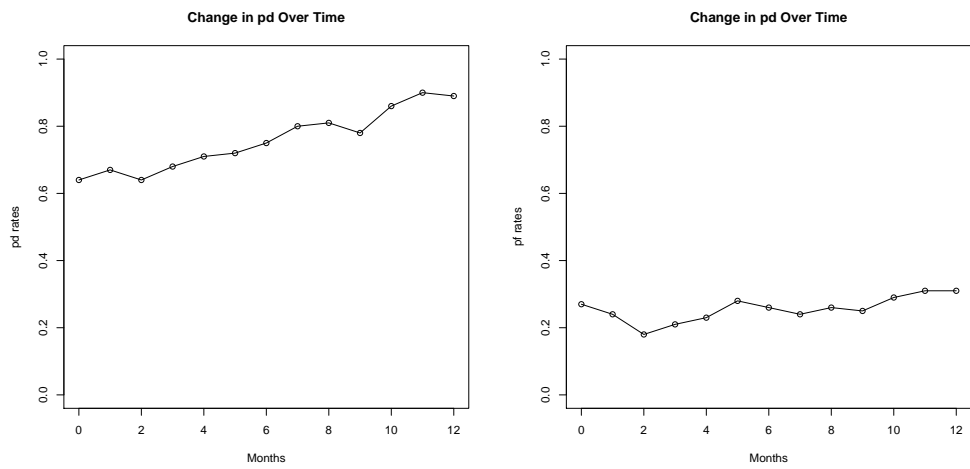
Table 6.9: Issue Recommendation Performance of The Model For The New Developers when we estimate future collaboration based on Stochastic Kronecker Networks

Train	Test	Precision	Recall
Jan-09	Apr-09	0.74	0.66
Jan-09	Sep-09	0.76	0.61
Jan-09	Dec-09	0.74	0.58
Apr-09	Sep-09	0.77	0.65
Apr-09	Dec-09	0.77	0.63

for each combination, 2) The network densification over time changes the Kronecker parameters as seen in Table A, 3) The new collaborations predicted may not have occurred yet at the test snapshot date.

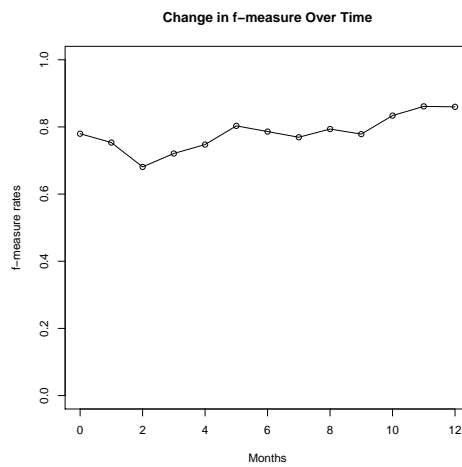
6.4. Real Time Prediction Results

In the real time defect prediction model we update the output of the Naïve Bayes predictor based on the recent defect history of the source code functions. In the experiment, we trained the model for the snapshot of June 2008 and tracked results monthly for 12 months. We found that pd of our model increased consistently over time and never fell below our original pd while not changing pf values significantly (Mann-Whitney U with $p < 0.05$). In order to reduce the effect of randomness we took the mean of the pd and pf values after 100 random trials for each month. The pd, pf and the f-measure changes over time are provided in Figure 6.10.



(a) Enterprise Software

(b) Enterprise Software



(c) Enterprise Software

Figure 6.10: pd, pf and f-measure changes for Enterprise Software dataset over time for each month after the model is run using the real time defect prediction model

7. Threats to Validity

A fundamental question concerning results from an experiment is how valid the results are. An empirical work is conducted to test the causality relation between the candidate cause and effect constructs. In an empirical work, a theory is tested by modeling theoretical cause constructs as independent variables (*treatment*) and effect constructs as dependent variable(s) (*outcome*) [117]. Threats to validity may occur in the theory modeling, the experiment or the theoretical cause effect relation parts of an experiment.

It is nearly impossible to avoid all of the threats in an empirical work. Therefore, in this section we highlight the possible threats the activities we conducted to minimize them. Various researchers categorized threats to validity according to different schemes [118]. Campbell and Stanley define two types namely threats to external and internal validity [119]. On the other hand, Cook and Campbell extend the list to four type of threats: Conclusion, internal, external and construct validity. Wohlin et al. recommends the usage of four categories proposed by Cook and Campbell since they map easily to the steps involved when conducting experiments [118]. In this dissertation, we have highlighted the aforementioned four categories of threats based on the checklist provided by Cook and Campbell [120] [121], and the guideline of Wohlin et al. [118].

In order to ensure the internal validity, we must make sure that the relation between the treatment and outcome is a causal relationship, and not that of a factor of which we have no control [118]. We have done the experimental work on a single group and we could not conduct a controlled experiment. In the literature, controlled experiments related to maintenance are often conducted on mock-up and/or student projects. However, we believe that the insights gained from mock-up and/or undergraduate student projects are not usually transferable to industrial projects. On the other hand, in an industrial setting and large industrial projects, controlled experiments are not usually possible. Therefore we chose to use a large industrial project in order to obtain

transferable results. Another threat to the internal validity is instrumentation [120]. Instrumentation is the effect caused by the artifacts on the outcome of an experiment. In our case, the artifacts we used were the data stored within the issue management software regarding the issue activity and the source code repository. We gathered the data postmortem, therefore we could not check the behavior of the people using these software. However, we checked the results of our analyses with the quality assurance team of the organization in order to verify their correctness.

Construct validity concerns with the relation between theory and observation. Mono-method bias is using a single performance measure in experiments is a threat to construct validity. In order to overcome mono-method bias we used multiple performance measures (pd, pf and f-measure for defect prediction) for all of the evaluations. In addition, we showed the interactions of different treatments (independent variables) explicitly.

Social threats to construct validity are the issues related to the subjects of the experiment. It occurs when subjects of an experiment change their responses since they are aware that they are experimented on. In our experiments, we handled postmortem maintenance data. We believe that the social threats are minimal since at the time of measurement, the subjects were not aware that we would do experiments on their issue maintenance data. However, the increased pressure during certain periods may have caused the inclusion of noise to the issue handling activity. We cross-checked the suspicious activity within the repository by showing the full activity log to the quality assurance team of the organization. We filtered the trivial code changes of the methods within the source code such as adding new blank lines or indentations in order to avoid classifying them as defective.

Conclusion validity concerns with the relation between the treatment and outcome. We tested the statistical significance of findings using rank based tests with no normality assumptions. Reliability of measures is also important for conclusion validity. We used approximation to the organizational issue allocation strategy as the performance measure as used by the previous studies [19, 33]. We used GINI index to

approximate the inequality among the issues owned. In our scenario, we assumed the current issue allocation mechanism of the organization to be ideal in the testing of the issue recommender and tested our recommender against actual allocation of the issues. Therefore, we believe that our results provide a worst case performance for the model.

An important threat to conclusion validity is the non-random heterogeneity of the subjects. It occurs when the selected subset of population is significantly different than the whole population. In our empirical work, we used the whole issue repository of the enterprise project and Android to avoid this threat.

When dealing with the threats to validity conflicts may occur among the different classes of threats to validity [118]. The prioritization of the threats may change based on the type of the experiments. Software engineering research can be classified as applied research [120]. In applied research, external validity has the highest priority since software engineering researchers are concerned with showing the validity of their results for at least a part of the domain [118]. We described the specifics of the organization such as process maturity, development culture and the employed issue maintenance operations in order to show the type of organization our results may be valid for.

Threats to external validity are the conditions that limit to generalize the results of our experiments to industry practices [118]. Interaction of selection and treatment with is a major external validity threat. It occurs when the selected sub-population is not representative of the general population. As discussed earlier, enterprise software project is developed by an international group of developers with mostly PhD level degrees. The case is a project with mature maintenance processes, large user-base and programmers with a more than average technical capability. Similar to any other empirical study in software engineering, further experiments in other projects is necessary for minimizing the threats to external validity.

Interaction of the history with the treatments is another possible threat to external validity. This threat may occur when a selected history portion is significantly

different than the rest of the history of the subjects. In our experiments, we used one major release of the enterprise software within a timespan of 5 years as data source. We chose the release since it has the best data quality according to the quality assurance team. In addition, since the release is relatively new we could interview and discuss our results with people who actually worked in the release. These people provided valuable feedback about the data and helped us in cleaning the data.

8. CONCLUSIONS AND FUTURE WORK

8.1. Summary of Results

In this research, we analyzed the issue maintenance process and developer collaboration in order to design an issue recommendation model. In order to build such a recommender, instead of the unstructured issue text data, we used the structured information within the issue report namely the category and symptoms and the developer collaboration related to them. We analyzed the trends in the collaboration network in order to uncover patterns which may be useful in such an issue recommender. We have observed that collaboration data and structured issue content is valuable in an issue recommendation model based on our experiments.

We analyzed the issue management process for two large software, the enterprise software and Android. Enterprise software is a set of architectural functionality in a large relational database management solution. Stakeholders from ten countries have contributed to the issue repository of the enterprise software. Android is a popular open-source mobile OS financially backed by Google. These two projects follow two completely different release management, development culture and licensing strategies. Despite their differences, we observed similar trends in their issue management data. In our exploratory analysis, the summary of the findings are as follows:

- (i) Developer collaboration during software development forms a *scale-free* network. Scale-free networks are a class of real-life networks which have been observed previously in other domains.
- (ii) We used GINI index as an inequality estimation measure among the issue ownership distribution. Developer issue workloads during maintenance are unequal with GINI index values of > 0.5 on any point during maintenance. The issue ownership inequality shows that large software which may be maintained for several decades critically depends on the issue maintenance experience of a few individuals.

- (iii) As number of active issues increases the inequality in the issue ownership increases. Issue workload imbalance is particularly high during times when issue influx is high. We believe that this trend may explain the burnout of a few developers during crisis periods.
- (iv) Number of active issues does not change significantly after release of a software.
- (v) Issue resolve times are not dependent on the number of issues that a developer owns. Experienced developers and developers who own many issues simultaneously resolve issues in similar time compared to inexperienced developers. This may be caused by the high workload of experienced developers.
- (vi) The collaboration network densifies over time. In other words, number of edges / number of nodes gets higher over time. This trend shows that, during maintenance people from different teams with different responsibilities tend to collaborate between themselves. In other words, collective code ownership is established over time among the experienced developers of large projects.

Our exploratory analysis results indicate that the issue management processes in organizations have many problems. These problems may occur even in software organizations with mature issue maintenance processes. We believe that one of the most important of these problems is the inequality among issue ownership counts among full-time developers.

In our work, we answered our two research questions by building an issue recommender. We present the summary of the results with their relations to the research questions of this dissertation in the following sections:

8.1.1. How can we assign issues to developers?

Issue management is a complex allocation problem for large software. Assigning the issues to the right developer may take up to 5 minutes per issue [33]. Similarly, it has been observed that more than 25% of the time of people in an organization is spent on waiting for decisions [1]. By automating the issue assignment process, we can reduce the wasted time of developers spent while waiting for decisions and reduce the

effort of issue allocation (bug triage).

We propose a novel issue recommender model which uses the collaboration of developers and the structured issue content as its input to solve the issue assignment problem. Our model approximates the centrality of developers in the collaboration network related to a category and symptom of issues and ranks the developers based on their approximated centralities. We propose a base model, and two extensions to the base model. The first extension uses heuristics to reduce assignment inequalities. The second extension addresses the people cold start in the recommender i.e. the problem of assigning issues to new developers in the projects.

We defined GINI index and approximation to real issue assignment as the two performance measures for the model. We had promising results based on our empirical work on the issue recommender using the enterprise software dataset. The base model finds the actual owner of an issue among its 2 highest ranked developers with a probability > 0.9 . We extended the model two heuristics in order to deal with the problem of issue workload inequalities during software maintenance. By using the first heuristic, we limited the number of issues that a developer may own to 5. The model resulted in lower inequality with this heuristics with a significant decrease in model's approximation performance in terms of mapping the actual issue assignment pattern. In the second heuristic, we limited the number of issues that a developer may own based on their commit counts. By using the second heuristic, we increased the approximation performance of the model while keeping the GINI index lower than the base model.

We generated an artificial dependency relation among issues. By generating such an artificial dependency relation we highlighted the possible problems for the model when the issue assignment times are changed. In this experiment, we showed that, if we change the ordering relation among the issues the model's performance significantly decreases to $P(1) < 0.1$.

Finally, we used stochastic Kronecker Network to predict the future state of the collaboration network and assign issues to new developer groups. We showed that

past collaboration affects the assignment patterns of the new developers. Our model predicted the assignments of the new developer groups with 0.76 precision and 0.63 recall rates on the average.

8.1.2. How can developers more efficiently organize their time in solving the issues?

Existing defect prediction models focus on testing resource allocation. We believe that the defect prediction models would be useful for developers during their maintenance operations. We used the defect history of the software modules to change the predictions of a learning based predictor to change the output of the model. We showed the benefit of our model by testing it on the enterprise software dataset. Our model increased the pd rates significantly ($P < 0.05$ with Mann-Whitney U Test) while keeping pf rates same ($P < 0.05$ with Mann-Whitney U Test) over time compared to the based Naive Bayes model.

8.2. Contributions

In the following sections we summarize the contributions of this research. We categorized our contributions into theoretical and methodological and practical contributions.

8.2.1. Theoretical and Methodological Contributions

- *Model the software as the combination of developer collaboration network and the call graph:* We modeled software as the combination of developer collaboration and software call-graph networks. Collaboration is defined as the activity on the similar software modules by two developers during a release. Call graph is formed by the caller-callee relations of software modules. We showed the static and temporal properties of the software collaboration network for a large scale software and their relation with software issues.
- *Build an issue recommendation system using developer collaboration network:* De-

veloper collaboration network shows the experience and centrality of developers within the organization. We developed a method for estimating the centralities of developers in the collaboration network. We used the symptom and category information of the issues to categorize the past issues and built the collaboration network for each cluster. Our model ranked the developers based on their centrality within the collaboration network and recommended the new issue reports based on the ranking.

- *Usage of Kronecker networks to estimate the future collaborations of new developers:* Cold start is a common problem in recommender systems and it happens when a new user or item (issue) is added to the system. In our recommender, we solved the item (issue) cold start problem with a content based approach. We categorized the new issue into an existing category and afterwards ran the recommender algorithm.

Recommending issues to new developers is a more serious problem for the issue recommender. It is part of the problem of introducing new people to existing software development teams. As the project progresses initiation of new people gets harder. There is a lot of knowledge to be transferred to the new people reducing the productivity of both old and new employees during initiation. Fred Brooks claims that the addition of new people may even cause delayed project delivery dates [108].

We propose to use the Kronecker network to address the new people cold start problem in the issue recommendation system. We used a parametric modelling method proposed by Leskovec et al. [110, 67, 2] named Kronecker network to model the future state of the software collaboration network. We chose Kronecker graph generator since it is the only method that produces networks that conform to all the properties of scale free networks. Kronecker networks are formed by fitting Kronecker parameters to the actual network and recursively replicating stochastic Kronecker kernel adjacency matrix by a matrix operation called Kronecker product.

The Kronecker network approach can be used to recommend issues only to a new group of developers since the node labels of the generated network can not be inferred. The state of the future collaboration network can be estimated using

the Kronecker parameters and issues can be recommended to the new group of developers.

- *Address developer workload balance problem using the issue recommendation model:* We extended the issue recommendation model to address the issue workload balancing problem. Issue workload balancing is a critical problem in industry. We initially showed that in two large software, issues are distributed non-uniformly among maintenance team members especially during the times when active issue count is the highest. One problem of the base recommendation model was the lack of workload balance among developers that may occur when the recommendations of the model were used for issue assignment. We used an approach based on heuristics in order to overcome the problem of issue workload imbalances. Model that used workload balancing heuristics performed worse than the model with no heuristics but distributed issues more uniformly than the original model. Imbalances of the actual issue assignments may be the possible reason for the reduction of performance.
- *Replication of experiments:* We have defined the characteristics of the dataset and performed exploratory analysis on the issue management data and the collaboration networks to help other researchers when replicating our experiments.

8.2.2. Practical Contributions

- *Empirical analysis of the issue recommendation system and the real time defect prediction model:* In our research, we collected the development data of one industrial large-scale software developed at five international sites as a dataset. The software is produced with the collaboration of more than two hundred developers. We extracted all the issue, collaboration activity and all the code change information over a time span of five years which includes two major releases of the product. We tested the issue recommendation model and its extensions on the extracted datasets and compared its performance with actual issue assignments. We showed that our model performance is better than the state of the art text similarity based issue recommendation systems. We showed the performance of Kronecker networks in addressing the people cold-start problem and

the performance of the workload-balancing recommendation system extensions.

- *Empirical analysis of the issue recommendation system and the real time defect prediction model:* In our research, we collected the development data of one industrial large-scale software developed at five international sites as a dataset. The software is produced with the collaboration of more than two hundred developers. We extracted all the issue, collaboration activity and all the code change information over a time span of five years which includes two major releases of the product. We tested the issue recommendation model and its extensions on the extracted datasets and compared its performance with actual issue assignments. We showed that our model performance is better than the state of the art text similarity based issue recommendation systems. We showed the performance of Kronecker networks in addressing the people cold-start problem and the performance of the workload-balancing recommendation system extensions.
- *Real time defect prediction:* In traditional defect prediction models the defect proneness prediction for software modules is provided for specific snapshots of the software. The predictions of the models help managers in the effective allocation of the testing resources. However, data that is comprised of defect prone modules is also valuable for developers at any time during maintenance. We updated the predictions of the machine learning based prediction model based on the recent history of the software modules in the software call-graph to show the defect prone software modules at any time during development.
- *Development of An Integrated Measurement and Decision Support Solution For Software Engineers:* Data extraction and difficulty of model output interpretation are two major problems preventing wide adoption of recommendation models by the industry. In order to make models developed by researchers more beneficial to the industry, integrated solutions should be developed with data extraction and analysis functionality. For this aim, we are developing a software named Dione with data extraction, measurement, analysis and decision support functionality [122].

8.3. Future Directions

Dependence on the expertise of a few experts for allocation decisions is not sustainable for large projects in the long term. Therefore, we believe that the importance of automated recommender solutions will increase in the future to cope with the ever increasing complexity of developing large software.

We think that the future progress in the area of recommender systems in software engineering will target automating task management during development. Understanding how humans cooperate during their development activity and the people factor in software engineering in general will be important to build these recommenders. The issue management system proposed by this research can be considered as a step towards this goal.

By integrating decision support systems to Dione, we believe that the gap between software engineering research and practice will be decreased in the future. We will continue the development of Dione to achieve this goal.

APPENDIX A: Collaboration Network Descriptive Statistics

Table A.1: Evolution of The Software Collaboration Network

Key Network Statistics	Month 1	Month 2	Month 3	Month 4	Month 5	Month 6	Month 7	Month 8	Month 9	Month 10	Month 11	Month 12	Month 13
Node Count	34	40	58	68	90	91	93	102	106	109	114	120	121
Edge Count	148	208	378	559	917	1034	1041	1653	1919	2044	2262	2446	2455
Diameter	4	3	4	4	4	4	4	4	4	4	4	4	4
Mean Clustering Coefficient	0.650469	0.730237	0.672606	0.705336	0.664292	0.690793	0.678065	0.736464	0.751094	0.75301	0.754242	0.765123	0.76537
Mean Transitivity	0.561219	0.525645	0.535743	0.58701	0.56618	0.600575	0.598542	0.652298	0.679767	0.669911	0.679357	0.672996	0.671355
Mean Clique Size	5.387097	6.204545	6.833333	8.717949	9.992933	11.54902	11.504178	16.861998	18.5196	18.648556	20.696069	21.674272	21.668607
Max Clique Size	9	10	13	17	19	22	22	27	32	32	34	36	36
Weakly Connected Component Count	1	1	1	1	1	1	1	1	1	1	1	1	1
Community Count	4	3	4	3	4	4	5	4	4	4	4	4	4
Kronecker Parameters	Month 1	Month 2	Month 3	Month 4	Month 5	Month 6	Month 7	Month 8	Month 9	Month 10	Month 11	Month 12	Month 13
A	0.9999	0.9999	0.9999	0.9999	0.9999	0.9999	0.9999	0.9999	0.9999	0.9999	0.9999	0.9999	0.9999
B	0.557846	0.603398	0.690119	0.590918	0.657154	0.683369	0.675707	0.76492	0.788132	0.798565	0.805294	0.82042	0.823852
C	0.569516	0.644119	0.743232	0.617732	0.709691	0.726637	0.718565	0.775071	0.793947	0.801809	0.818152	0.844589	0.841303
D	0.133147	0.161161	0.266497	0.222737	0.247919	0.273347	0.270912	0.318101	0.344146	0.355361	0.372855	0.398231	0.398853

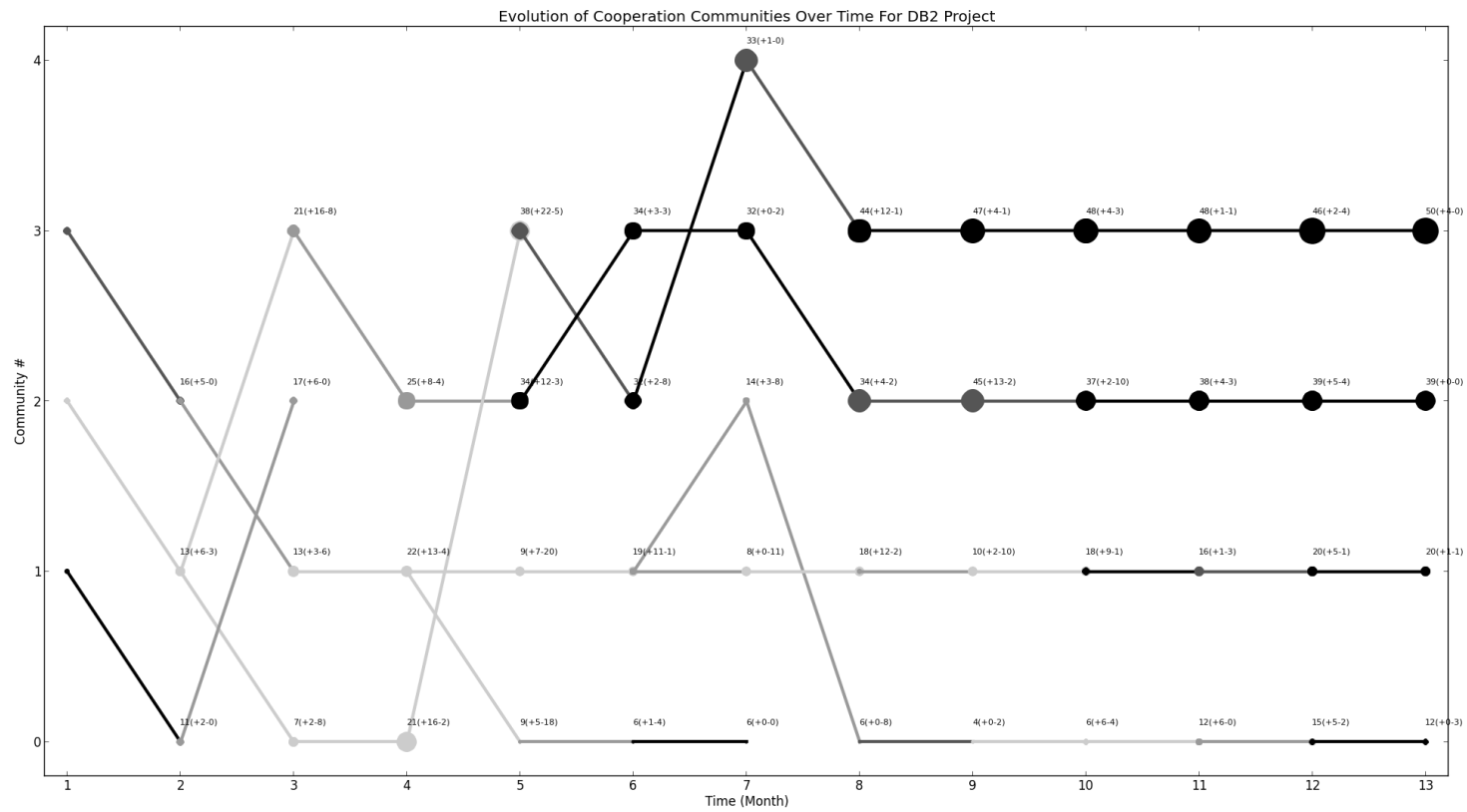


Figure A.1: Evolution of The Enterprise Software Communities. The marker shows the size and relative stability of the community for a given month.

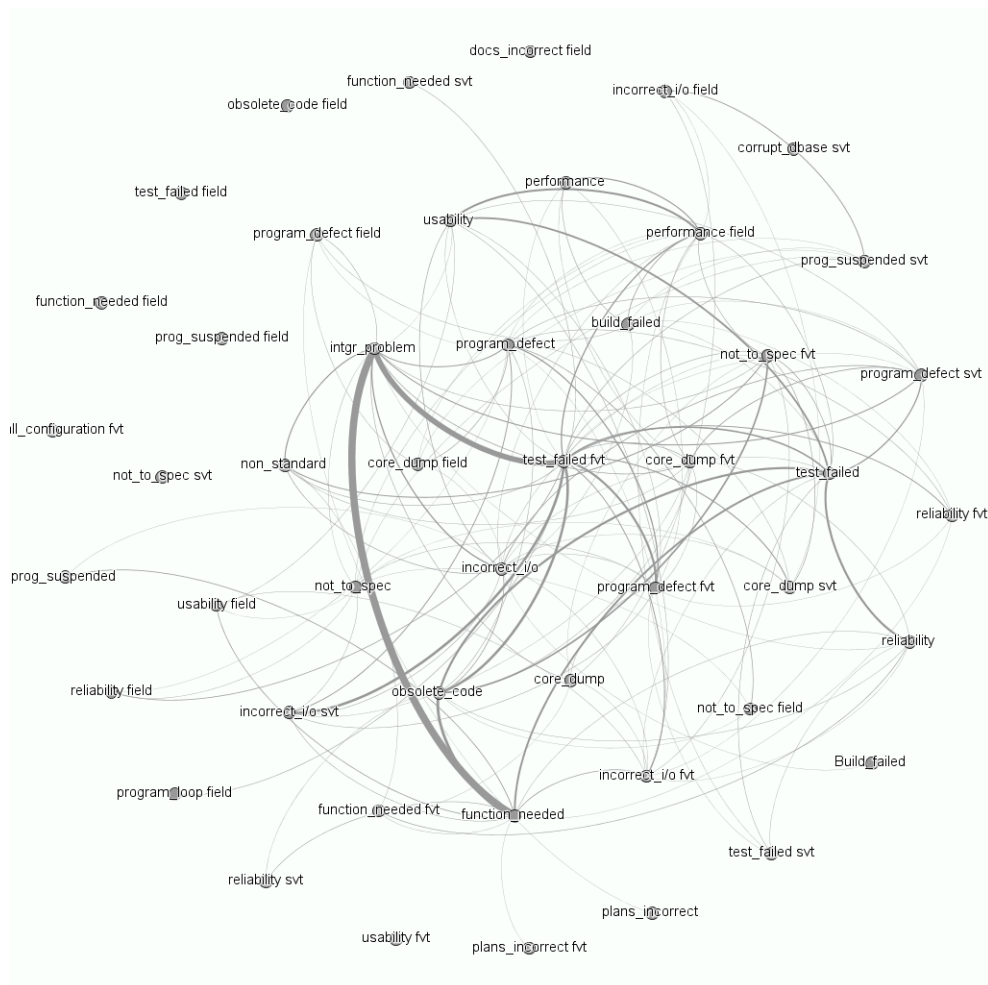


Figure A.2: Relation between different defect symptoms in the Enterprise Project. Links are weighted for the number of symptom co-occurrence in same software function. Nodes are combinations of symptoms and defect categories.

APPENDIX B: Software Metrics

Attribute	Description	Attribute	Description
Static Code Metrics			
<i>Cyclomatic density, $vd(G)$</i>	the ratio of module's cyclomatic complexity to its length	<i>Essential complexity, $ev(G)$</i>	the degree to which a module contains unstructured constructs
<i>Design density, $dd(G)$</i>	condition/ decision	<i>Cyclomatic complexity, $v(G)$</i>	# linearly independent paths
<i>Essential density, $ed(G)$</i>	$(ev(G)-1)/(v(G)-1)$	<i>Maintenance severity</i>	$ev(G)/v(G)$
<i>Unique operands</i>	n1	<i>Executable LOC</i>	Source lines of code that contain only code and white space
<i>Branch count</i>	# of branches	<i>Total operators</i>	N1
<i>Decision count</i>	# of decision points	<i>Total operands</i>	N2
<i>Condition count</i>	# of conditionals	<i>Unique operators</i>	n2
<i>Difficulty (D)</i>	1/L	<i>Length (N)</i>	N1 + N2
<i>Level (L)</i>	$(2/n1)*(n2/N2)$	<i>Programming effort (E)</i>	D*V
<i>Volume (V)</i>	$N*\log(n)$	<i>Programming time (T)</i>	E/18
<i>Commented LOC</i>	# number of lines which have comments		
Churn metrics			
<i>Number of edits</i>	number of edits done on a file	<i>Lines Added</i>	total number of lines added
<i>Number of unique committers</i>	number of unique committers edited a file	<i>Lines Removed</i>	total number of lines removed

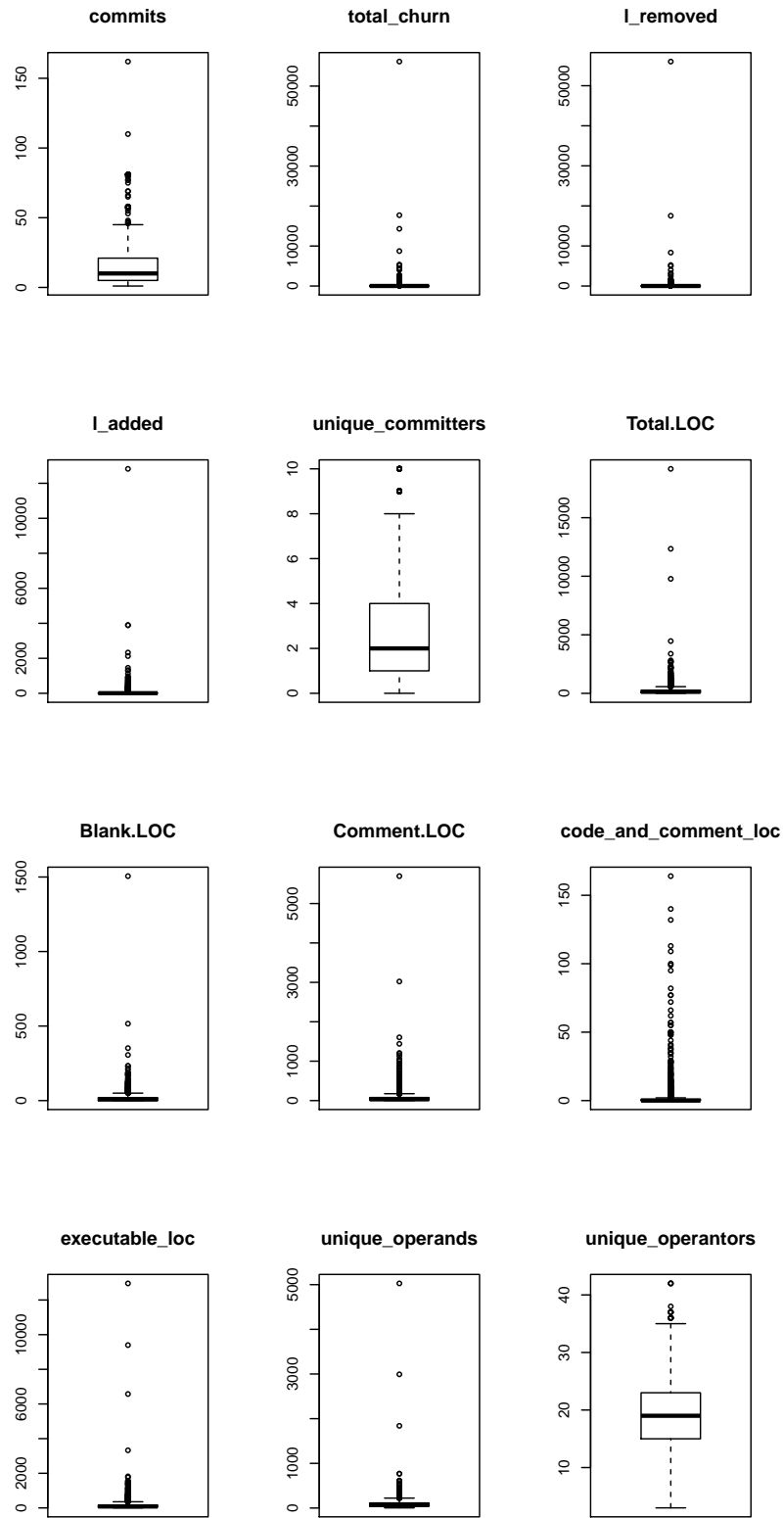


Figure B.1: Metric Box Plots- 1/3

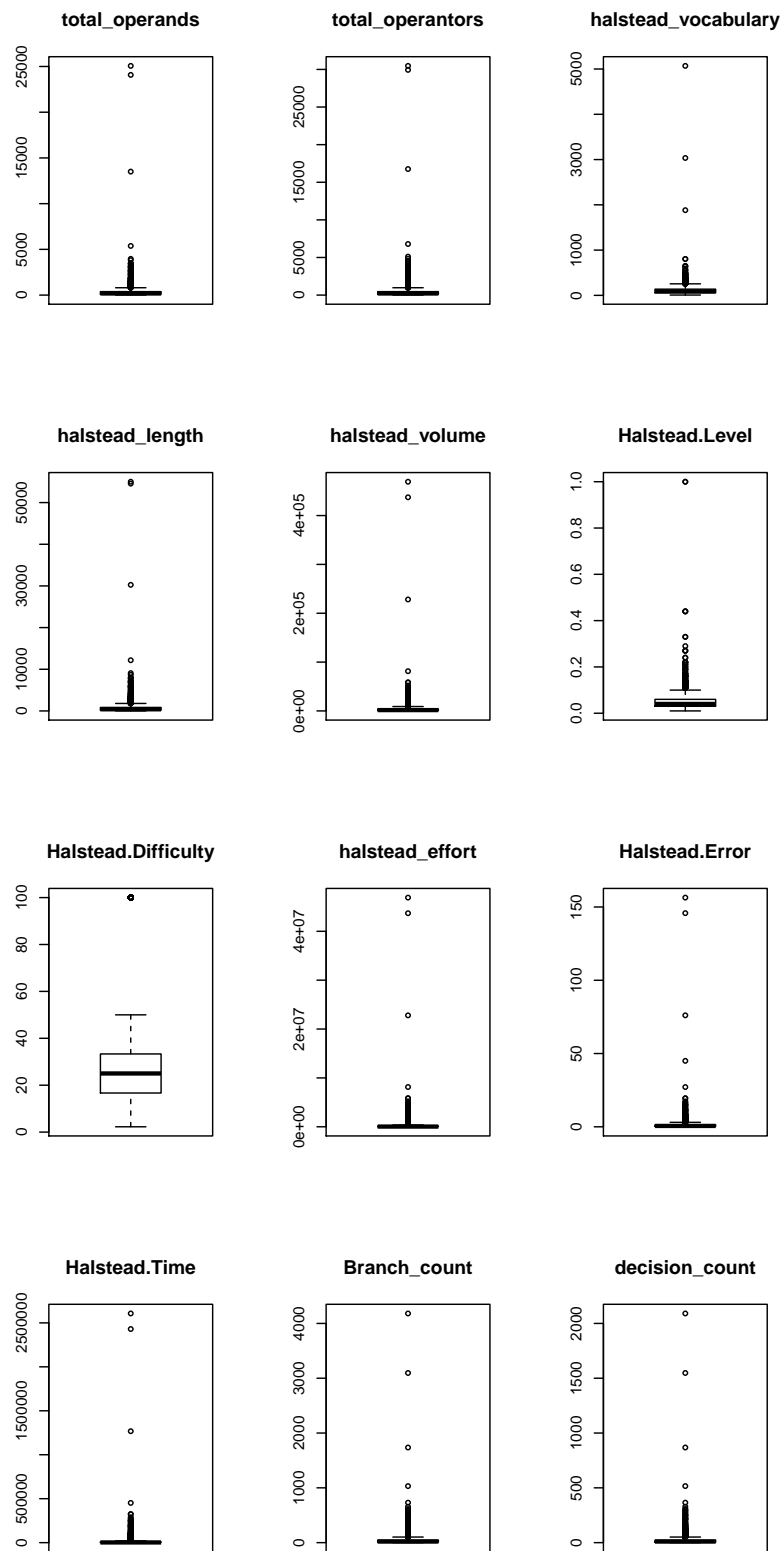


Figure B.2: Metric Box Plots- 2/3

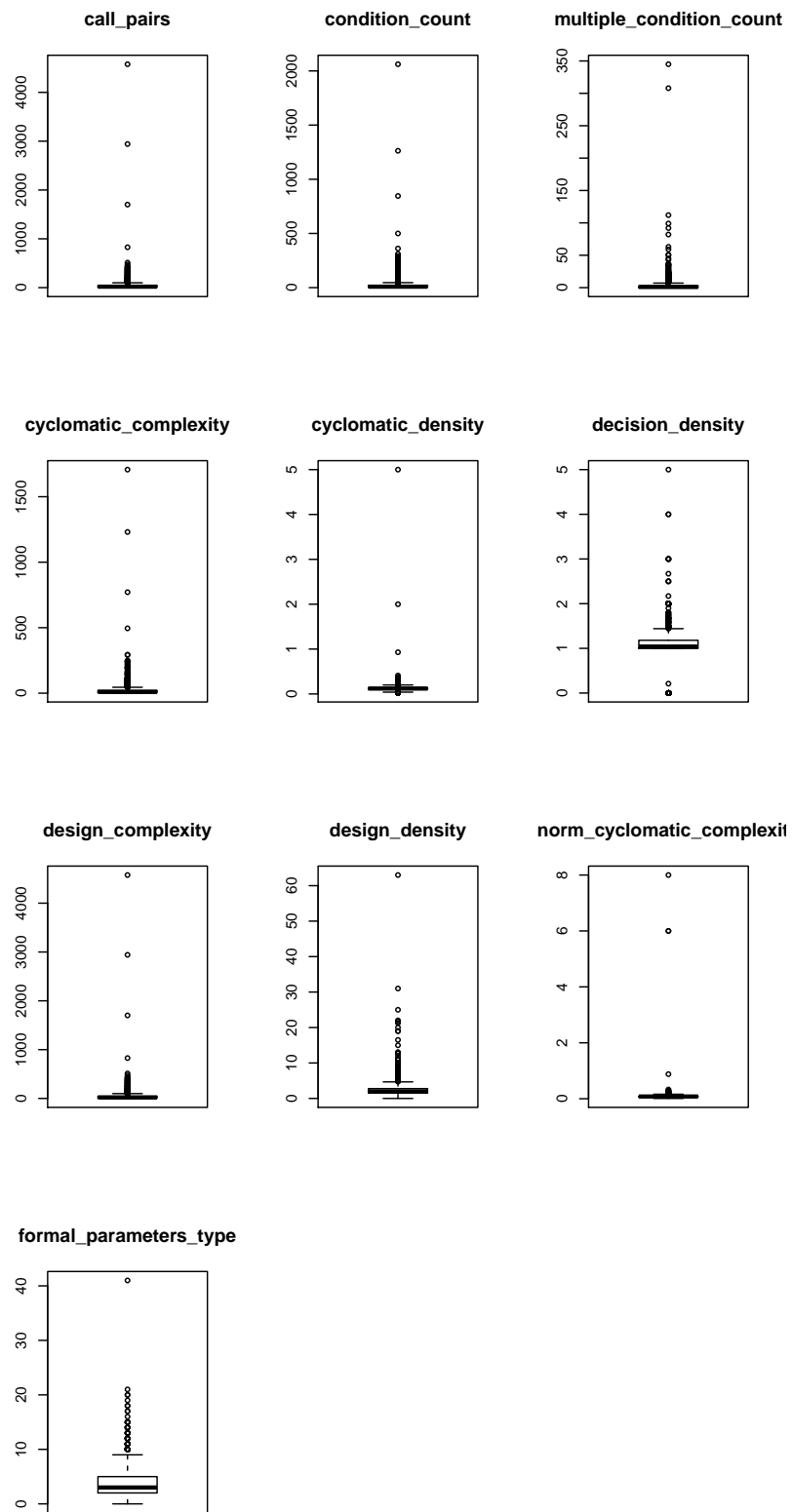


Figure B.3: Metric Box Plots- 3/3

Table B.1: Correlation of The Software Metrics For The Enterprise Software 1/3

	commits	total_churn	l_removed	l_added	unique_committers	Total.LOC	Blank.LOC	Comment.LOC	code_and_comment_loc	executable_loc	unique_operands	unique_operantors
commits												
total_churn	0.19											
l_removed	0.13	0.98***										
l_added	0.41*	0.41*	0.24									
unique_committers	0.77***	0.02	-0.01	0.16								
Total.LOC	-0.41*	-0.39*	-0.34*	-0.39*	-0.23							
Blank.LOC	-0.40*	-0.40*	-0.36*	-0.38*	-0.25	0.96***						
Comment.LOC	-0.42*	-0.40*	-0.35*	-0.39*	-0.23	0.99***	0.93***					
code_and_comment_loc	-0.39*	-0.33	-0.28	-0.33	-0.17	0.50**	0.47**	0.54***				
executable_loc	-0.40*	-0.39*	-0.34*	-0.38*	-0.23	1.00***	0.97***	0.99***	0.48**			
unique_operands	-0.39*	-0.40*	-0.35*	-0.39*	-0.24	0.99***	0.99***	0.97***	0.49**	0.99***		
unique_operantors	-0.37*	-0.35*	-0.31	-0.31	-0.21	0.70***	0.68***	0.73***	0.57***	0.68***	0.70***	
total_operands	-0.40*	-0.40*	-0.35*	-0.39*	-0.24	1.00***	0.98***	0.98***	0.49**	1.00***	1.00***	0.68***
total_operantors	-0.40*	-0.40*	-0.35*	-0.39*	-0.24	1.00***	0.98***	0.98***	0.49**	1.00***	1.00***	0.68***
halstead_vocabulary	-0.40*	-0.40*	-0.35*	-0.39*	-0.24	0.99***	0.99***	0.97***	0.49**	0.99***	1.00***	0.72***
halstead_length	-0.40*	-0.40*	-0.35*	-0.39*	-0.24	1.00***	0.98***	0.98***	0.49**	1.00***	1.00***	0.68***
halstead_volume	-0.39*	-0.39*	-0.34*	-0.38*	-0.23	0.99***	0.98***	0.97***	0.46**	1.00***	0.99***	0.65***
Halstead.Level	0.14	0.16	0.15	0.11	0.04	-0.41*	-0.40*	-0.45**	-0.45**	-0.40*	-0.43*	-0.87***
Halstead.Difficulty	-0.43*	-0.37*	-0.32	-0.34*	-0.26	0.74***	0.72***	0.78***	0.63***	0.73***	0.74***	0.98***
halstead_effort	-0.39*	-0.39*	-0.34*	-0.38*	-0.23	0.99***	0.98***	0.97***	0.46**	1.00***	0.99***	0.63***
Halstead.Error	-0.39*	-0.39*	-0.34*	-0.38*	-0.23	0.99***	0.98***	0.97***	0.46**	1.00***	0.99***	0.65***
Halstead.Time	-0.39*	-0.39*	-0.34*	-0.38*	-0.23	0.99***	0.98***	0.97***	0.46**	1.00***	0.99***	0.63***
Branch_count	-0.40*	-0.39*	-0.34*	-0.39*	-0.23	1.00***	0.97***	0.99***	0.49**	1.00***	0.99***	0.68***
decision_count	-0.40*	-0.39*	-0.34*	-0.39*	-0.23	1.00***	0.97***	0.99***	0.49**	1.00***	0.99***	0.68***
call_pairs	-0.39*	-0.40*	-0.35*	-0.39*	-0.24	0.99***	0.99***	0.96***	0.48**	0.99***	1.00***	0.66***
condition_count	-0.40*	-0.39*	-0.34*	-0.39*	-0.23	1.00***	0.96***	0.99***	0.48**	1.00***	0.99***	0.68***
multiple_condition_count	-0.39*	-0.40*	-0.35*	-0.39*	-0.23	1.00***	0.97***	0.98***	0.46**	1.00***	0.99***	0.66***
cyclomatic_complexity	-0.41*	-0.39*	-0.34*	-0.39*	-0.23	1.00***	0.97***	0.99***	0.49**	1.00***	0.99***	0.69***
cyclomatic_density	-0.17	-0.10	-0.08	-0.11	-0.13	-0.16	-0.16	-0.16	-0.05	-0.16	-0.17	-0.12
decision_density	-0.10	-0.11	-0.10	-0.07	-0.05	-0.03	-0.03	-0.01	0.13	-0.04	-0.02	0.45**
design_complexity	-0.39*	-0.40*	-0.35*	-0.39*	-0.23	0.99***	0.99***	0.96***	0.48**	0.99***	1.00***	0.66***
design_density	0.04	-0.04	-0.04	-0.01	0.00	-0.21	-0.16	-0.23	-0.06	-0.21	-0.18	-0.25
norm_cyclomatic_complexity	-0.06	-0.05	-0.04	-0.05	-0.09	-0.24	-0.23	-0.24	-0.21	-0.23	-0.24	-0.23
formal_parameters_type	-0.01	-0.16	-0.16	-0.05	-0.03	0.05	0.02	0.08	0.07	0.04	0.06	0.31

Table B.2: Correlation of The Software Metrics For The Enterprise Software 2/3

	total_operands	total_operantors	halstead_vocabulary	halstead_length	halstead_volume	Halstead.Level	Halstead.Difficulty	halstead_effort	Halstead.Error	Halstead.Time	Branch_count
commits											
total_churn											
l_removed											
l_added											
unique_committers											
Total.LOC											
Blank.LOC											
Comment.LOC											
code_and_comment_loc											
executable_loc											
unique_operands											
unique_operantors											
total_operands											
total_operantors	1.00***										
halstead_vocabulary		1.00***									
halstead_length	1.00***	1.00***	1.00***								
halstead_volume	1.00***	1.00***	0.99***	1.00***							
Halstead.Level	-0.40*	-0.40*	-0.44**	-0.40*	-0.36*						
Halstead.Difficulty	0.73***	0.73***	0.75***	0.73***	0.69***	-0.80***					
halstead_effort	1.00***	1.00***	0.99***	1.00***	1.00***	-0.35*	0.68***				
Halstead.Error	1.00***	1.00***	0.99***	1.00***	1.00***	-0.36*	0.69***	1.00***			
Halstead.Time	1.00***	1.00***	0.99***	1.00***	1.00***	-0.35*	0.68***	1.00***	1.00***		
Branch_count	1.00***	1.00***	0.99***	1.00***	1.00***	-0.40*	0.73***	1.00***	1.00***	1.00***	
decision_count	1.00***	1.00***	0.99***	1.00***	1.00***	-0.40*	0.73***	0.99***	1.00***	0.99***	1.00***
call_pairs	1.00***	1.00***	1.00***	1.00***	1.00***	-0.38*	0.70***	1.00***	1.00***	1.00***	0.99***
condition_count	1.00***	1.00***	0.99***	1.00***	0.99***	-0.39*	0.73***	0.99***	0.99***	0.99***	1.00***
multiple_condition_count	1.00***	1.00***	0.99***	1.00***	1.00***	-0.38*	0.71***	1.00***	1.00***	1.00***	1.00***
cyclomatic_complexity	1.00***	1.00***	0.99***	1.00***	1.00***	-0.40*	0.74***	0.99***	1.00***	0.99***	1.00***
cyclomatic_density	-0.17	-0.17	-0.17	-0.17	-0.17	0.07	-0.10	-0.17	-0.17	-0.17	-0.15
decision_density	-0.04	-0.04	-0.01	-0.04	-0.06	-0.65***	0.34	-0.07	-0.06	-0.07	-0.04
design_complexity	1.00***	1.00***	1.00***	1.00***	1.00***	-0.38*	0.70***	1.00***	1.00***	1.00***	0.99***
design_density	-0.19	-0.20	-0.18	-0.20	-0.19	0.11	-0.23	-0.19	-0.19	-0.19	-0.22
norm_cyclomatic_complexity	-0.24	-0.24	-0.25	-0.24	-0.23	0.15	-0.23	-0.23	-0.23	-0.23	-0.23
formal_parameters_type	0.04	0.04	0.07	0.04	0.03	-0.39*	0.26	0.02	0.03	0.02	0.04

Table B.3: Correlation of The Software Metrics For The Enterprise Software 3/3

	decision_count	call_pairs	condition_count	multiple_condition_count	cyclomatic_complexity	cyclomatic_density	decision_density	design_complexity	design_density	norm_cyclomatic_complexity
commits										
total_churn										
l_removed										
l_added										
unique_committers										
Total.LOC										
Blank.LOC										
Comment.LOC										
code_and_comment_loc										
executable_loc										
unique_operands										
unique_operantors										
total_operands										
total_operantors										
halstead_vocabulary										
halstead_length										
halstead_volume										
Halstead.Level										
Halstead.Difficulty										
halstead_effort										
Halstead.Error										
Halstead.Time										
Branch_count										
decision_count										
call_pairs	0.99***									
condition_count	1.00***	0.99***								
multiple_condition_count	1.00***	0.99***	1.00***							
cyclomatic_complexity	1.00***	0.99***	1.00***	1.00***						
cyclomatic_density	-0.15	-0.17	-0.15	-0.16	-0.15					
decision_density	-0.04	-0.05	-0.05	-0.06	-0.04	0.03				
design_complexity	0.99***	1.00***	0.99***	0.99***	0.99***	-0.17	-0.05			
design_density	-0.22	-0.18	-0.22	-0.21	-0.22	-0.18	-0.25	-0.18		
norm_cyclomatic_complexity	-0.23	-0.24	-0.23	-0.23	-0.23	0.04	-0.06	-0.24	-0.13	
formal_parameters_type	0.04	0.03	0.04	0.04	0.04	-0.15	0.18	0.03	-0.07	-0.18

REFERENCES

1. U. Riss and A. Rickayzen, “Challenges for business process and task management,” *Journal of Universal Knowledge*, vol. 0, no. 2, pp. 77–100, 2005. [Online]. Available: http://www.jucs.org/jukm_0_2/riss/jukm_0_2_77_100_riss.html
2. J. Leskovec, “Kronecker Graphs : An Approach to Modeling Networks,” *Journal of Machine Learning Research*, vol. 11, pp. 985–1042, 2010.
3. T. DeMarco and T. Lister, *Peopleware: Productive Projects and Teams*, 2nd ed. Dorset House Publishing Company, 1999.
4. W. Everett and S. Honiden, “Reliability and safety of real-time systems,” *Software, IEEE*, vol. 12, no. 3, pp. 13–16, May.
5. B. Nuseibeh, “Ariane 5: Who dunnit?” *Software, IEEE*, vol. 14, no. 3, pp. 15–16, May-June 1997.
6. A. Puntambekar, *Software Engineering And Quality Assurance*. Technical Publications, 2010.
7. E. S. Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O’reilly and Associates, 1999.
8. J. H. Miller, *Complex Adaptive Systems: An Introduction to Computational Models of Social Life*. Princeton University Press, 2007.
9. S. Brin and L. Page, “The anatomy of a large-scale hypertextual Web search engine,” *Computer Networks and ISDN Systems*, vol. 30, no. 1-7, pp. 107–117, Apr. 1998.
10. D. Easley and J. M. Kleinberg, *Networks , Crowds , and Markets : Reasoning*

about a Highly Connected World, draft ed. Cambridge University Press, 2010.

11. J. M. Kleinberg, “Authoritative sources in a hyperlinked environment,” *Journal of the ACM*, vol. 46, no. 5, pp. 604–632, Sept. 1999. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=324133.324140>
12. A.-L. Barabási and E. Bonabeau, “Scale-free networks,” pp. 50–59, 2003. [Online]. Available: <http://elibrary.ru/item.asp?id=7821500>
13. L. Wen, R. G. Dromey, and D. Kirk, “Software engineering and scale-free networks.” *IEEE transactions on systems, man, and cybernetics. Part B, Cybernetics : a publication of the IEEE Systems, Man, and Cybernetics Society*, vol. 39, no. 4, pp. 845–854, Aug. 2009. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pubmed/19380275>
14. F. Eichinger, K. Böhm, and M. Huber, “Mining edge-weighted call graphs to localise software bugs,” *Machine Learning and Knowledge Discovery in Databases*, pp. 333–348, 2008. [Online]. Available: <http://www.springerlink.com/index/26808jx13817xr73.pdf>
15. 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*. ACM, 2008, pp. 13–23. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1453106>
16. G. Madey, V. Freeh, and R. Tynan, “The open source software development phenomenon: An analysis based on social network theory,” *Americas Conference on Information*, pp. 1806–1813, 2002. [Online]. Available: <http://ais.bepress.com/cgi/viewcontent.cgi?article=1606&context=amcis2002>
17. J. Herbsleb and A. Mockus, “An empirical study of speed and communication in globally distributed software development,” *IEEE Transactions on Software*

- Engineering*, vol. 29, no. 6, pp. 481–494, June 2003. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1205177>
18. D. Jannach, M. Zanker, A. Felfernig, and G. Friedrich, *Recommender Systems: An Introduction*, 1st ed. Cambridge University Press, 2010.
 19. J. Anvik, “Determining implementation expertise from bug reports,” *MSR ’07 Proceedings of the Fourth International Workshop on Mining Software Repositories*, 2007. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1134285.1134457><http://dl.acm.org/citation.cfm?id=1808933><http://www.cs.ucdavis.edu/~barr/publications/fixation.pdf>http://ieeexplore.ieee.org/xpls/abs/_all.jsp?arnumber=4228639
 20. 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*, pp. 1–31, 2010.
 21. T. Menzies, J. Greenwald, and A. Frank, “Data mining static code attributes to learn defect predictors,” *IEEE Transactions on Software Engineering*, vol. 33(1), pp. 2–13, 2007.
 22. A. Tamrawi, T. Nguyen, and J. Al-Kofahi, “Fuzzy set-based automatic bug triaging: NIER track,” *Proceedings of the 33rd International Conference on Software Engineering*, pp. 884–887, 2011. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs/_all.jsp?arnumber=6032543
 23. O. Baysal, R. Holmes, and M. W. Godfrey, “Revisiting bug triage and resolution practices,” *2012 First International Workshop on User Evaluation for Software Engineering Researchers (USER)*, pp. 29–30, June 2012. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6226578>
 24. N. Bettenburg and A. Hassan, “Studying the Impact of Social Structures on Software Quality,” in *2010 IEEE 18th International Conference on*

- Program Comprehension*. IEEE, 2010, pp. 124–133. [Online]. Available: <http://www.computer.org/portal/web/csdl/doi/10.1109/ICPC.2010.46>
25. C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, “How long will it take to fix this Bug?” in *Fourth International Workshop on Mining Software Repositories, 2007. ICSE Workshops MSR’07*, no. 2, 2007. [Online]. Available: <http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:How+long+will+it+take+to+fix+this+Bug?#0>
 26. E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto, “Predicting Re-opened Bugs: A Case Study on the Eclipse Project,” *2010 17th Working Conference on Reverse Engineering*, pp. 249–258, Oct. 2010. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5645566>
 27. T. Zimmermann, N. Nagappan, P. Guo, and B. Murphy, “Characterizing and predicting which bugs get reopened,” in *Proceedings of the 34th International Conference on Software Engineering [ACCEPTED]*, 2012. [Online]. Available: <http://research.microsoft.com/pubs/159352/zimmermann-icse-2012.pdf>
 28. B. Caglayan, A. Misirli, and A. Miransky, “Factors characterizing reopened issues: a case study,” in *Promise 2012*, 2012, pp. 1–10. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2365327>
 29. P. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, “Characterizing and predicting which bugs get fixed: An empirical study of Microsoft Windows,” in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 1. IEEE, 2010, pp. 495–504. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs/_all.jsp?arnumber=6062117
 30. T. Zhang and B. Lee, “An Automated Bug Triage Approach: A Concept Profile,” *LNCS 7389*, pp. 505–512, 2012.

31. J. Anvik, L. Hiew, and G. C. Murphy, “Who should fix this bug?” in *Proceedings of the International Conference on Software Engineering*, Shanghai, China, 2006, pp. 361–370. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1134285.1134336>
32. D. Cubranic and G. Murphy, “Automatic bug triage using text categorization,” in *Proceedings of the Sixteenth International Conference on Software Engineering Knowledge Engineering*. Citeseer, 2004, pp. 1–6. [Online]. Available: <http://www.mendeley.com/research/automatic-bug-triage-using-text-categorization/>
33. J. Anvik and G. C. Murphy, “Reducing the effort of bug report triage: Recommenders for development-oriented decisions,” *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 10:1–10:35, Aug. 2011. [Online]. Available: <http://doi.acm.org/10.1145/2000791.2000794>
34. A. Bakir, E. Kocaguneli, A. Tosun, A. Bener, and B. Turhan, “Xiruxe: An Intelligent Fault Tracking Tool,” *AIPR09, Orlando*, 2009. [Online]. Available: <http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Xiruxe++An+Intelligent+Fault+Tracking+Tool\#0>
35. P. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, “Not my bug! and other reasons for software bug report reassignments,” in *Proceedings of the ACM 2011 conference on Computer supported cooperative work*. ACM, 2011, pp. 395–404. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1958887>
36. X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, “An Approach to Detecting Duplicate Bug Reports using Natural Language and Execution Information,” in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 461–470.
37. E. Giger, M. Pinzger, and H. Gall, “Predicting the Fix Time of Bugs,” in *RSSE ’10 Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, 2010, pp. 52–56.

38. F. Akiyama, “An example of software system debugging,” *Information Processing*, vol. 71, pp. 353–379, 1971.
39. M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. New York, NY, USA: Elsevier Science Inc., 1977.
40. T. J. Ostrand, E. J. Weyuker, and R. M. Bell, “Predicting the location and number of faults in large software systems,” *Software Engineering, IEEE Transactions on*, vol. 31, no. 4, pp. 340–355, 2005.
41. —, “Automating algorithms for the identification of fault-prone files,” in *Proceedings of the 2007 international symposium on Software testing and analysis*, ser. ISSTA '07. New York, NY, USA: ACM, 2007, pp. 219–227. [Online]. Available: <http://doi.acm.org/10.1145/1273463.1273493>
42. N. E. Fenton and M. Neil, “A critique of software defect prediction models,” *Software Engineering, IEEE Transactions on*, vol. 25, no. 5, pp. 675–689, 1999.
43. N. E. Fenton and N. Ohlsson, “Quantitative analysis of faults and failures in a complex software system,” *Software Engineering, IEEE Transactions on*, vol. 26, no. 8, pp. 797–814, 2000.
44. S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, “Benchmarking classification models for software defect prediction: A proposed framework and novel findings,” *Software Engineering, IEEE Transactions on*, vol. 34, pp. 485–496, 2008.
45. B. Caglayan, A. Bener, and S. Koch, “Merits of using repository metrics in defect prediction for open source projects,” in *2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*. Ieee, May 2009, pp. 31–36. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5071357>
46. G. Çalikhand A. B. Bener, “Influence of confirmation biases of developers

- on software quality: an empirical study,” *Software Quality Journal*, vol. 21, no. 2, pp. 377–416, July 2012. [Online]. Available: <http://link.springer.com/10.1007/s11219-012-9180-0>
47. N. Nagappan, B. Murphy, and V. R. Basili, “The influence of organizational structure on software quality,” in *Proceedings of the 13th international conference on Software engineering - ICSE '08*, ser. ICSE '08. New York, New York, USA: ACM Press, 2008, p. 521. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=1368088.1368160>
 48. T. J. McCabe, “A complexity measure,” in *ICSE '76: Proceedings of the 2nd international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, p. 407.
 49. S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” *Software Engineering, IEEE Transactions on*, vol. 20, no. 6, pp. 476–493–, 1994.
 50. N. Nagappan and T. Ball, “Using software dependencies and churn metrics to predict field failures: An empirical case study,” pp. –, 2007.
 51. C. T. Bailey and W. L. Dingee, “A software study using halstead metrics,” in *Proceedings of the 1981 ACM workshop/symposium on Measurement and evaluation of software quality*. New York, NY, USA: ACM, 1981, pp. 189–197.
 52. T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, “Mining version histories to guide software changes,” *Software Engineering, IEEE Transactions on*, vol. 31, no. 6, pp. 429–445–, 2005.
 53. R. Moser, W. Pedrycz, and G. Succi, “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction,” pp. 181–190, 2008.
 54. E. Weyuker, T. Ostrand, and R. Bell, “Do too many cooks spoil the broth?”

- using the number of developers to enhance defect prediction models,” *Empirical Software Engineering*, vol. 13, no. 5, pp. 539–559–, 2008. [Online]. Available: <http://dx.doi.org/10.1007/s10664-008-9082-8>
55. B. Caglayan, A. Bener, and K. S., “Merits of using repository metrics in defect prediction for open source projects,” in *Proceedings of FLOSS Workshop in 31st International Conference on Software Engineering*, 2009.
 56. E. J. Weyuker, T. J. Ostrand, and R. M. Bell, “Using developer information as a factor for fault prediction,” in *PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, p. 8.
 57. M. Ohira, N. Ohsugi, T. Ohoka, and K.-i. Matsumoto, “Accelerating cross-project knowledge collaboration using collaborative filtering and social networks,” in *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, vol. 30, no. 4. St. Louis, Missouri: ACM, 2005, pp. 1–5.
 58. L. Lopez-Fernandez, G. Robles, J. M. Gonzalez-barahona, and J. Carlos, “Applying social network analysis to the information in cvs repositories,” in *Proceedings of the Mining Software Repositories Workshop. 26th International Conference on Software Engineering*, 2004.
 59. P. C. Wason, “On the failure to eliminate hypotheses in a conceptual task,” *Quarterly Journal of Experimental Psychology*, vol. 12, no. 3, pp. 129–140, 1960.
 60. 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 Journal*, 2009, in print. DOI 10.1007/s10664-008-9103-7.
 61. J. L. Payne and M. J. Eppstein, “Evolutionary Dynamics on Scale-Free Interaction Networks,” *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 4, pp. 895–912, Aug. 2009. [Online]. Available: <http://dx.doi.org/10.1109/TEVC.2009.6288888>

//ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5175362

62. M. A. Nowak and K. Sigmund, “Evolution of indirect reciprocity.” *Nature*, vol. 437, no. 7063, pp. 1291–8, Oct. 2005. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pubmed/16251955>
63. H. Ohtsuki, C. Hauert, E. Lieberman, and M. A. Nowak, “A simple rule for the evolution of cooperation on graphs and social networks.” *Nature*, vol. 441, no. 7092, pp. 502–5, May 2006. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pubmed/16724065>
64. E. Lieberman, C. Hauert, and M. Nowak, “Evolutionary dynamics on graphs,” *Nature*, vol. 433, no. January, pp. 312–316, 2005. [Online]. Available: <http://www.nature.com/nature/journal/v433/n7023/abs/nature03204.html>
65. A.-L. Barabási, “The origin of bursts and heavy tails in human dynamics,” *Nature*, vol. 435, no. May, 2005. [Online]. Available: <http://www.nature.com/nature/journal/v435/n7039/abs/nature03459.html>
66. S. Goyal and F. Vega-Redondo, “Network Formation and Social Coordination,” *SSRN Electronic Journal*, no. 970131, pp. 1–35, 2003. [Online]. Available: <http://www.ssrn.com/abstract=369460>
67. J. Leskovec, J. Kleinberg, and C. Faloutsos, “Graph evolution,” *ACM Transactions on Knowledge Discovery from Data*, vol. 1, no. 1, pp. 2–41, Mar. 2007. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=1217299.1217301>
68. S. Arbesman, J. Kleinberg, and S. Strogatz, “Superlinear scaling for innovation in cities,” *Physical Review E*, vol. 79, no. 1, p. 16115, 2009. [Online]. Available: <http://link.aps.org/doi/10.1103/PhysRevE.79.016115>
69. A. Spector, P. Norvig, and S. Petrov, “Google’s hybrid approach to research,”

- Commun. ACM*, vol. 55, no. 7, pp. 34–37, July 2012. [Online]. Available: <http://doi.acm.org/10.1145/2209249.2209262>
70. D. Jannach, M. Zanker, A. Felfernig, and G. Friedrich, *Recommender Systems: An Introduction*, 1st ed. Cambridge University Press, 2010.
 71. Y. Koren, R. Bell, and C. Volinsky, “Matrix factorization techniques for recommender systems,” *Computer*, pp. 42–49, 2009. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs/_all.jsp?arnumber=5197422
 72. Z. Huang, H. Chen, and D. Zeng, “Applying associative retrieval techniques to alleviate the sparsity problem in collaborative filtering,” *ACM Transactions on Information Systems*, vol. 22, no. 1, pp. 116–142, Jan. 2004. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=963770.963775>
 73. M. Robillard, R. Walker, and T. Zimmermann, “Recommendation systems for software engineering,” *Software, IEEE*, pp. 80–86, 2010. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs/_all.jsp?arnumber=5235134
 74. I. Ahmad and A. Ghafoor, “Fault-tolerant task management and load redistribution on massively parallel hypercube systems,” in *Supercomputing '92., Proceedings*, 1992, pp. 750–759.
 75. F. S. Hillier, *Intro To Operations Research 8E (Iae)*. Tata McGraw-Hill Education, 1990.
 76. P. Brereton, B. Kitchenham, D. Budgen, M. Turner, and M. Khalil, “Lessons from applying the systematic literature review process within the software engineering domain,” *Journal of Systems and Software*, vol. 80, no. 4, pp. 571–583, Apr. 2007. [Online]. Available: <http://linkinghub.elsevier.com/retrieve/pii/S016412120600197X>
 77. J. M. Kleinberg, “The Small-World Phenomenon : An Algorithmic Perspective,”

Cornell University, Tech. Rep., 1999.

78. ———, “Navigation in a small world,” *Nature*, vol. 406, no. 6798, p. 845, Aug. 2000. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pubmed/16907213>
79. D. J. Watts and S. H. Strogatz, “Collective dynamics of ‘small-world’ networks,” *Nature*, vol. 393, no. 6684, pp. 440–2, June 1998. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pubmed/9623998>
80. P. Louridas, D. Spinellis, and V. Vlachos, “Power laws in software,” *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 1, pp. 1–26, Sept. 2008. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=1391984.1391986>
81. T. Zimmermann, N. Nagappan, L. Williams, K. Herzig, and R. Premraj, “An Empirical Study of the Factors Relating Field Failures and Dependencies,” *st.cs.uni-saarland.de*, 2010. [Online]. Available: <http://www.st.cs.uni-saarland.de/~{}mileva/failuresAndDependencies.pdf>
82. A. Applewhite, “Whose bug is it anyway? The battle over handling software flaws,” *IEEE Software*, vol. 21, no. 2, pp. 94–97, Mar. 2004. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1270771>
83. D. Barbagallo, C. Francalenei, and F. Merlo, “The impact of social networking on software design quality and development effort in open source projects,” *ICIS 2008 proceedings*, p. 201, 2008. [Online]. Available: <http://aisel.aisnet.org/cgi/viewcontent.cgi?article=1016&context=icis2008>
84. H. C. Benestad, B. Anda, and E. Arisholm, “Understanding software maintenance and evolution by analyzing individual changes: a literature review,” *Journal of Software Maintenance of Software Maintenance*, no. September, pp. 349–378, 2009. [Online]. Available: <http://www3.interscience.wiley.com/journal/122587594/abstract>

85. M. Ichii, M. Matsushita, and K. Inoue, "An exploration of power-law in use-relation of java software systems," in *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*. IEEE, 2008, pp. 422–431. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs/_all.jsp?arnumber=4483231
86. A. J. Ko and B. A. Myers, "Debugging Reinvented : Asking and Answering Why and Why Not Questions about Program Behavior," *Human-Computer Interaction*, pp. 301–310, 2008.
87. J. Krinke, "Mining execution relations for crosscutting concerns," *Software, IET*, vol. 2, no. 2, pp. 65–78, 2008. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs/_all.jsp?arnumber=4483545
88. J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," *25th International Conference on Software Engineering, 2003. Proceedings.*, vol. 6, pp. 308–318, 2003. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1201210>
89. T. Palmer and N. Fields, "Computer supported cooperative work," *Computer*, vol. 27, no. 5, pp. 15–17, May 1994. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=291295>
90. K. Petersen, "Measuring and Predicting Software Productivity: A Systematic Map and Review," *Information and Software Technology*, no. December, Dec. 2010. [Online]. Available: <http://linkinghub.elsevier.com/retrieve/pii/S0950584910002156>
91. B. Turhan, G. Kocak, and A. Bener, "Software Defect Prediction Using Call Graph Based Ranking (CGBR) Framework," *2008 34th Euromicro Conference Software Engineering and Advanced Applications*, pp. 191–198, Sept. 2008. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4725722>

92. L. Wang, Z. Wang, C. Yang, L. Zhang, and Q. Ye, "Linux kernels as complex networks: A novel method to study evolution," *2009 IEEE International Conference on Software Maintenance*, pp. 41–50, Sept. 2009. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5306348>
93. T. Wolf, A. Schroter, D. Damian, and T. Nguyen, "Predicting build failures using social network analysis on developer communication," in *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. IEEE Computer Society, May 2009, pp. 1–11. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1555017>
94. T. Xie and J. Pei, "MAPO: mining API usages from open source repositories," in *Proceedings of the 2006 international workshop on Mining software repositories*. ACM, 2006, pp. 54–57. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1137983.1137997>
95. B. Yang, W. Cheung, and J. Liu, "Community Mining from Signed Social Networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 10, pp. 1333–1348, Oct. 2007. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4302742>
96. T. Zimmermann and N. Nagappan, "Predicting defects with program dependencies," *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pp. 435–438, Oct. 2009. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5316024>
97. B. Caglayan and A. Bener, "Issue ownership activity in two large software projects," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, p. 1, Nov. 2012. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=2382756.2382786>
98. "Android Web Page, <http://www.android.com/>." [Online]. Available: <http://www.android.com/>

99. “Gartner Report on Third Quarter of Market Share: Mobile Communication Devices by Region and Country, 4Q12,” 2012. [Online]. Available: <http://www.gartner.com/newsroom/id/2335616>
100. E. Capra, C. Francalanci, and F. Merlo, “An empirical study on the relationship between software design quality, development effort and governance in open source projects,” *IEEE Transactions on Software Engineering*, vol. 34, no. 6, pp. 765–782, 2008. [Online]. Available: <http://www.computer.org/portal/web/csdl/doi/10.1109/TSE.2008.68>
101. M. Cataldo and S. Nambiar, “On the relationship between process maturity and geographic distribution: an empirical analysis of their impact on software quality,” in *FSE*, 2009, pp. 101–110. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1595714>
102. S. Koch, “Effort modeling and programmer participation in open source software projects,” *Information Economics and Policy*, vol. 20, no. 4, pp. 345–355, Dec. 2008. [Online]. Available: <http://linkinghub.elsevier.com/retrieve/pii/S0167624508000334>
103. J. Richardson, J. William A. Gwaltney, and P. P. (Firm), *Ship It!: A Practical Guide To Successful Software Projects*, ser. Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2005. [Online]. Available: <http://books.google.com.tr/books?id=ka1QAAAAMAAJ>
104. S. Aral, E. Brynjolfsson, and M. V. Alstyne, “Information, technology and information worker productivity: Task level evidence,” National Bureau of Economic Research, Working Paper 13172, June 2007. [Online]. Available: <http://www.nber.org/papers/w13172>
105. J. L. Gastwirth, “The Estimation of the Lorenz Curve and Gini Index,” *The Review of Economics and Statistics*, vol. 54, no. 3, pp. 306–316, 1972.

106. K. Xu, "How has the literature on GINI's Index evolved in the past 80 years," *China Economic Quarterly*, pp. 1–41, 2003.
107. A. I. Schein, A. Popescul, L. H. Ungar, and D. M. Pennock, "Methods and metrics for cold-start recommendations," in *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, ser. SIGIR '02. New York, NY, USA: ACM, 2002, pp. 253–260. [Online]. Available: <http://doi.acm.org/10.1145/564376.564421>
108. F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2nd Edition)*. Addison-Wesley Professional, 1995.
109. G. Adomavicius and a. Tuzhilin, "Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 6, pp. 734–749, June 2005. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1423975>
110. J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos, "Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication," in *Knowledge Discovery in Databases: PKDD 2005*. Springer, 2005, pp. 133–145. [Online]. Available: <http://www.springerlink.com/index/ph80u3764t433020.pdf>
111. T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," *Automated Software Engineering*, vol. 17, no. 4, pp. 375–407, May 2010. [Online]. Available: <http://www.springerlink.com/index/10.1007/s10515-010-0069-5>
112. P. Domingos and M. Pazzani, "On the optimality of the simple Bayesian classifier under zero-one loss," *Machine learning*, vol. 130, pp. 103–130, 1997. [Online]. Available: <http://link.springer.com/article/10.1023/A:1007413511361>

113. E. Kocaguneli, A. Tosun, A. Bener, B. Turhan, and B. Caglayan, “Prest: An intelligent software metrics extraction, analysis and defect prediction tool,” in *SEKE Proceedings*, 2009.
114. E. Alpaydin, *Introduction to Machine Learning*. MIT Press, 2004.
115. G. J. Myers, T. Badgett, T. Thomas, and C. Sandler, *The Art of Software Testing*. 2nd ed. John Wiley & Sons, 2004.
116. L. Williams, R. Kessler, W. Cunningham, and R. Jeffries, “Strengthening the case for pair programming,” *Software, IEEE*, vol. 17, no. 4, pp. 19–25, 2000.
117. W. Trochim, *Advances in quasi-experimental design and analysis*, ser. New directions for program evaluation. Jossey-Bass, 1986. [Online]. Available: <http://books.google.com.tr/books?id=AbIOAQAAMAAJ>
118. C. Wohlin, *Experimentation in Software Engineering: An Introduction*, ser. The Kluwer International Series in Software Engineering. Kluwer Academic, 2000. [Online]. Available: <http://books.google.com.tr/books?id=nG2USHV0wAEC>
119. D. Campbell and J. Stanley, *Experimental and Quasi-experimental Designs for Research*. R. McNally College Publishing Company, 1973. [Online]. Available: <http://books.google.com.tr/books?id=2zQfAQAAIAAJ>
120. T. Cook and D. Campbell, *Quasi-experimentation: design & analysis issues for field settings*. Rand McNally College, 1979. [Online]. Available: <http://books.google.com.tr/books?id=OKdEAAAAIAAJ>
121. W. Shadish, T. Cook, and D. Campbell, *Experimental and quasi-experimental designs for generalized causal inference*. Houghton Mifflin, 2002. [Online]. Available: <http://books.google.com.tr/books?id=o7jaAAAAMAAJ>
122. B. Caglayan, A. Misirli, G. Calikli, A. B. Bener, T. Aytac, and B. Turhan, “Dione: an integrated measurement and defect prediction solution,” in

Proceedings of the SIGSOFT'12/FSE-20, Cary, North Carolina, USA, 2012, pp. 1–4. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2393619>