

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 [1] and monetary losses [2] 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 [3]. 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 [4]. 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 [5]. 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 [6]. Graph is a useful abstraction in computer science used to model complex systems and it is widely used in a range of application areas [7, 8, 9, 10]. 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 [9]. Scale-free network is defined as a network whose degree distribution follows power-law [8]. Scale-free networks are observed in different domains, and they surprisingly share similar properties [8]. 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[8]. In software domain, researchers have also found that many dependency networks of large software systems are scale-free networks [11].

In software engineering, research on software networks goes back to early 1980s [12]. 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 [13, 14].

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 [13] and in exploratory studies [14]. One reason behind this surge of interest may be the emergence of global software development [15]. 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 [16]. The recommendation system we propose is made of two core components namely bug triager and a live defect predictor. Bug triage is defined as assigning a developer to the most relevant issue based on past data [17]. Previous research shows that assigning a bug report to an owner takes five minutes per report on average for large software [17]. 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 [17]. This approach has various shortcomings such as language and dependence on issue report content [17]. 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.

Live 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 [18]. 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 [16]. 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.

1.3. Contributions

In this research we built an issue recommendation system using the collaboration network and the call graph of the software in order to increase the efficiency of resource allocation during the software quality assurance activities. Contributions of this research can be summarized as follows:

- *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:* Developer collaboration network shows the experience and centrality of developers within the organization. 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.
- *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.

- *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 new issues to 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 [19].

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. [20, 21, 22] named Kronocker network to model the future state of the software collaboration network. We chose Kronecker graph generator since it is the only method that produce networks that conform 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.

- *Live defect prediction:* In traditional defect prediction models the defect prone-ness 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 live history of the software modules in the software call-graph to show the defect prone software modules at any time during development.
- *Empirical analysis of the issue recommendation system and the live 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.

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 [23], [18]. 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.

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,

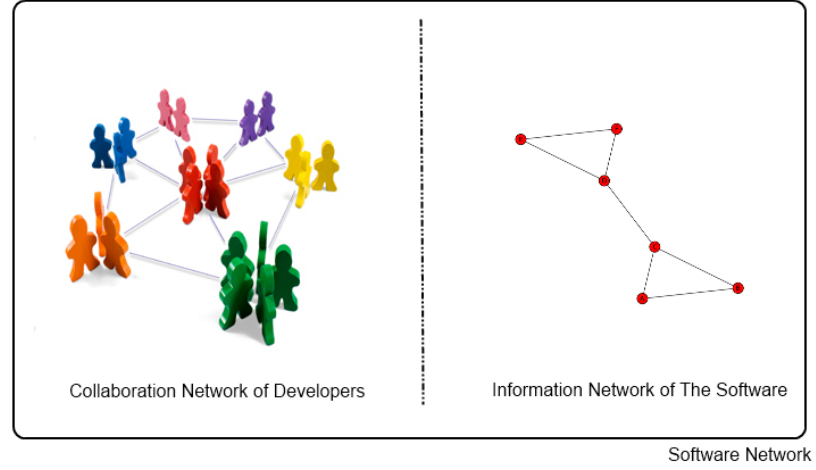


Figure 2.1: *Software network* is represented by the combination of its two underlying networks. a) *Collaboration network* models the interaction among developers, b) *Information network* models the relations between software components.

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 works 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 [17] [24]. 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 probably be the defective code relevant to the issue. In order to answer this problem we propose a live 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.

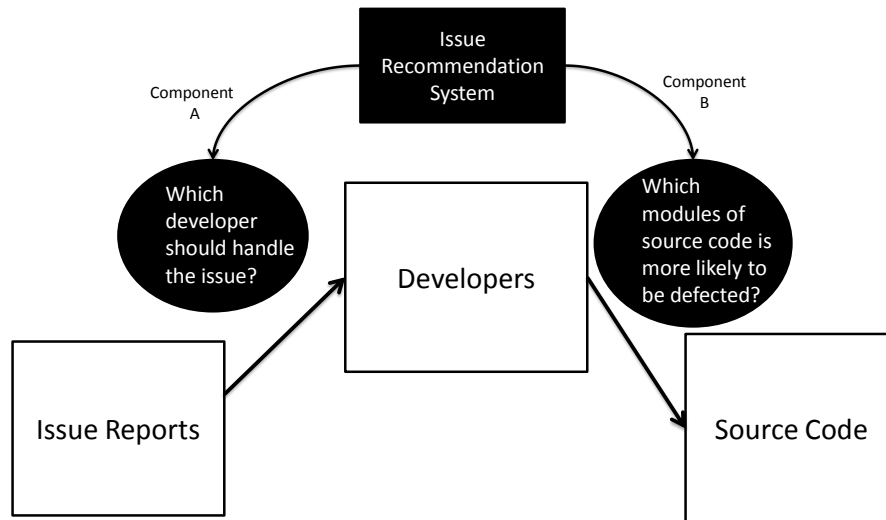


Figure 2.2: The recommendation system and its two components. Component A recommends an issue that is most relevant to a developer. Component B recommends the part of source code that possibly contains a defect related to that issue.

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 their 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 [25].

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 [17, 26, 27].

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. [28] 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 [28]. On the other hand, Zimmermann et al. [29] 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 [29]. 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 [30]. 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[17, 31, 27]. Zhang et al. studied the social network of issue reporters and developers to propose a bug triage model [32].

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 [17, 24, 33, 34, 35]. 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[36]. 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, [24], Guo et al. observed on Microsoft Vista dataset that the issue reassignment is in some cases not a problem but beneficial to communication[37].

Effective issue reporting is important for reporters as much as it is for the developers. 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 [31]. 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 [26]. 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 [38] 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[39]. Weiss et al. estimated the issue fix times on Eclipse, Mozilla and Gnome[27]. 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 [40, 41]. 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[40]:

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

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 [23, 18, 42, 43]. Proponents of defect prediction claim that defect prediction enables better test resource allocation in real life software systems[23]. A critique of existing defect prediction models in software engineering has been done by Fenton et. al. [44, 45]. Hall et al. gives a structured literature review of the research on defect prediction models [18]. In a benchmark study by Lessmann et al. it was shown that top 12 algorithms have no statistically significant difference in prediction performance [46].

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[41, 47, 48, 49, 13]:

- **Halstead Metrics:** Halstead metrics are proposed by Halstead in his seminal work about empirical software engineering [41]. They are among the first defined metrics for defect prediction and are used in many regression and classification based defect prediction models [41]. 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 [50]. 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[47]. 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. Cyclomatic 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 respectively.

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

Other metrics of McCabe are derived from cyclomatic complexity of source modules. In appendix C 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[48] 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 C full set of basic CK metrics are 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[49, 51]. 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[49, 51]. 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 [49, 52]. 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 [49, 51]. 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 have 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 rates by using repository metrics have 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 [13]. 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 [13]. Metrics proposed on file level collaboration in this study can be listed as following: When using logistic regression Meneely found that degree and closeness

are the most important metrics among several candidates of history and static code metrics.

- (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.

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 have 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 [59]. Confirmation bias is defined as the tendency of people to seek evidence that verifies a hypothesis rather than seeking evidence to falsify a hypothesis [59]. Calikli et al. used the outputs of Wason's confirmation bias test [60] 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 [59]. 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 [18]. 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 [61]. 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 supports 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. 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

employs recommendation systems which guide user decisions with minimal effect on the core functionality [62].

We can group recommendation systems as collaborative, content based and hybrid [63]. 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 [64]. 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[63]. 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 [63]. In these recommenders, content knowledge is mixed with the collaborative recommender to use the strengths of collaborative and content based approaches. We believe that 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 [65]. Cold start is a special case of the data sparsity problem. 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.

3.4. 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. [66] 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.

3.4.1. 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 my 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 [10, 67, 68, 69]. 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.

3.4.2. 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 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. [12]. 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. Eichenger 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. [15]. 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 may the interdependence 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. [70]. 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 [8].

The last paper we overviewed is An Empirical Study of the Factors Relating Field Failures and Dependencies by Zimmermann et al. [71]. Zimmermann et al. analysed 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, \surd -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
[72]	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
[73]	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 $\lambda \epsilon \beta$
[74]	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
[26]	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 $\beta \lambda \surd$
[12]	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 $\alpha \epsilon \beta \surd$
[15]	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 $\alpha \beta \lambda \surd$
[75]	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 $\diamond \surd \beta \gamma \gamma$
[76]	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 $\lambda \gamma$
[77]	Krinke2008	Krinke, J.	Mining execution relations for crosscutting concerns	2008	Software, IET Vol. 2(2), pp. 65-78	MAG $\gamma \beta \gamma$
[78]	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
[70]	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 $\diamond \gamma$

[14]	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 $\beta \diamond \lambda$
[13]	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 $\beta \lambda \checkmark$
[79]	Palmer1994	Palmer, T., Fields, N.	Computer supported cooperative work	1994	Computer Vol. 27(5), pp. 15-17	MAG λ
[80]	Petersen2010	Petersen, K.	Measuring and Predicting Software Productivity: A Systematic Map and Review	2010	Information and Software Technology	JOU $\diamond \checkmark$
[81]	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 $\beta \gamma \checkmark$
[82]	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 $\diamond \gamma$
[11]	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
[83]	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 $\beta \lambda \checkmark$
[84]	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
[85]	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 $\beta \lambda \gamma$
[86]	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 $\beta \gamma \checkmark$

[71]	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, it is owned and resolved or terminated in some other way by a developer. A practical recommender plugged to the issue management software can help the ownership and the resolve activities.

We propose a solution which can 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 [17]. 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 on 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 ranks in the collaboration network. 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 [87]. This trend is not sustainable in the long run since it indicates over-dependence on a 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 [17]. 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 [22]. 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.

Live 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 [18]. 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. For this aim, 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 nearby 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 [88]. 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 53 percent of the smartphone operating system market as of 2011 Q3 [89]. Android can be classified as a strictly governed commercial open-source project [90]. 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.

5.2. Issue Recommender System

5.2.1. Relations of Software Issues and Defects

5.2.1.1. 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 our issue recommendation model. The factors we tested empirically were the following:

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

For investigating *I* and *IV*, we checked the relationship between developers and issues. In previous research, various researchers found that in both close-source and open source software the code change distribution of developers are distributed in accordance with power law [91]. When we examined the degrees of collaboration in the dissertation work we found that same trend continues. By answering *I* we will find if the same trend holds for issue ownership data. Multi-tasking is often a stressful or hard to manage work style for human beings. We will test if multi-tasking change the average of bug resolve times of developers by answering *IV*.

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

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 and their relation to the Kronecker parameters.

5.2.2. Issue Recommendation Model

5.2.2.1. Model. The proposed issue recommendation algorithm has three parts: 1) Issue clustering, 2) Identify the developers who were active in that cluster, 3) Suggest developers that are most relevant to the issue based on the ranks in the cluster.

Part 1 has been solved previously with text mining methods [17] that identifies most similar issues. In our experiments using the enterprise software dataset we used symptom and category data to 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 number of modules the two developers *co-worked* on. *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.

Finally in part 3 a group of developers are suggested to own an issue. In such a model, several factors can be considered to reconsider ranks. For example a developer who is new may be given a higher ranking than proposed. Likewise, the ranking of a developer who has many actively 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.

5.2.2.2. 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 [33], [24]. For the sake of comparability 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.

5.2.3. Extensions to The Basic Issue Recommender Model

The basic issue recommendation model has only one success criteria: 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 possible 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.

5.2.3.1. 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 [92]. It is based on the area between a uniform cumulative distribution function and the actual cumulative distribution function (Lorenz Curve) [93].

A detailed survey of various calculation methods and history of GINI index is provided by Xu [93]. We explain our computation method briefly based on Xu's paper [93]. 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 found by the following equation:

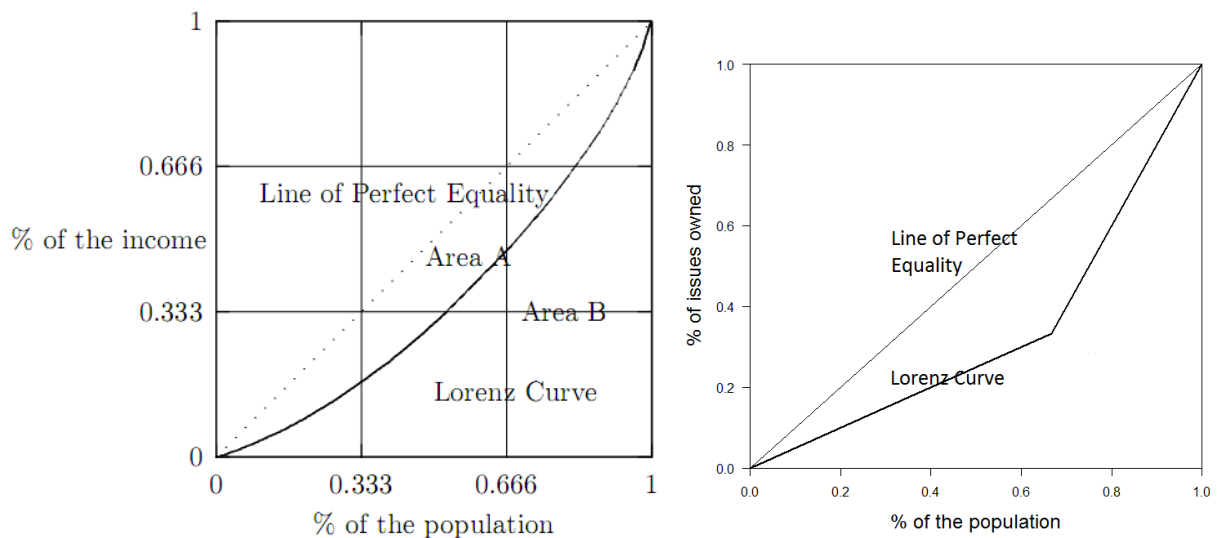
$$L_i = \frac{1}{n\mu_y} \sum_{j=1}^i (y_j) \quad (5.1)$$

In Figure 5.1a 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.



(a) Lorenz Curve for an income distribution [93].

(b) Lorenz Curve for an issue ownership distribution formed when two developers own one issue and one developer owns four issues out of six open issues.

Figure 5.1: 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 5.1b.

In order to reduce 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 maximum number of assignable issues 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. Assume 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 assign issues based on their commit activity level. The disadvantage of such a model is that, it makes the distribution less uniform than heuristic 1.

5.2.3.2. 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 network among with a occurrence probability of an arc of $P(arc) = 0.05$ among all of the issues. 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. In this case, the model's recommendations will change if the past collaboration on the time snapshot where the recommendation is also different.

5.2.3.3. Usage of Kronecker Graph to Address Cold Start Problem. Cold start is a common problem in recommender systems and happens when a new user or item (issue) is added to the system. In our recommender, we solve the item(issue) cold start with a content based recommender approach. We categorize the new issue into an existing category and afterwards run the recommender algorithm. In fact, for our recommender each issue is recommended based on the item cold start scenario.

Recommending new issues to developers is a more serious problem for the issue recommender. It is a 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 that must be transferred new people which reduces the productivity of both old and new employees during initiation. Figure 5.2 adapted from the book *Peopleware* shows the effect of member change in a software development

team in terms of productivity [94]. Fred Brooks claims that the addition of new people may even cause delayed project delivery dates [19].

New developer initiation problem has many dimensions like training, social issues etc. 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.

The usage example of Kronecker Network to estimate the future graph state is provided in Figure 5.3. 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 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 from the $128 - 100 = 28$ nodes from the generated Kronecker graph with 2×2 Kernel matrix. We found that removing random nodes which does not effect the graph connectivity as much as possible gives the best results in this case.

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. The points we used for training and testing is provided in Figure 5.1. We have January 2009, April 2009 and September 2009 as

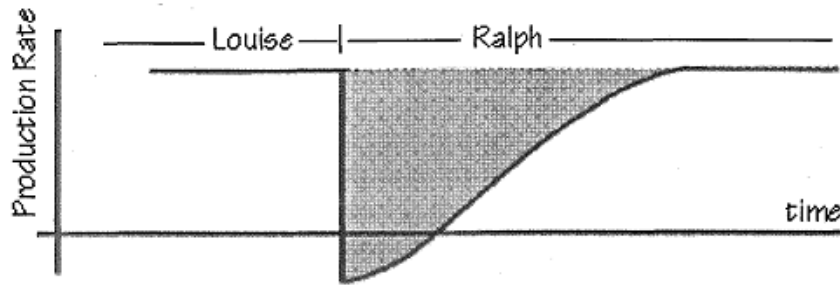


Figure 5.2: The productivity change during a member change in a software development team. Notice that the productivity actually goes negative in the beginning [94].

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 based on our approach. 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.

5.2.4. Live Defect Prediction

5.2.4.1. 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 another dataset. 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

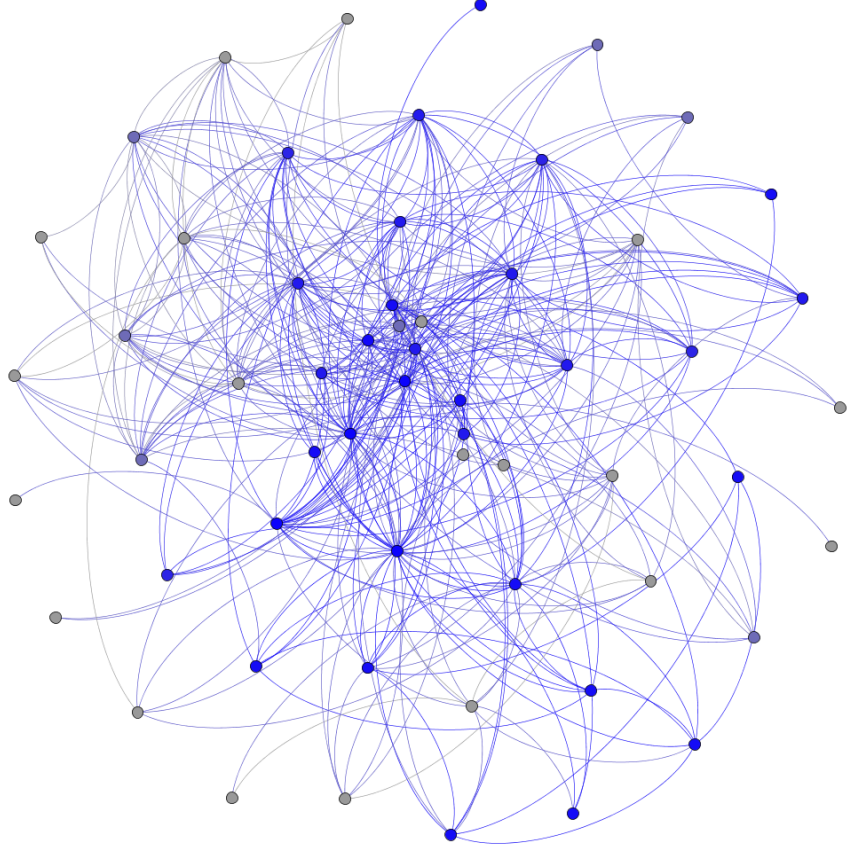


Figure 5.3: In the above generated graph blue nodes represent existing developers and gray nodes represent new developers. By parametrizing the existing developer collaboration using Kronfit algorithm and generating a network based on the number of existing developers plus new developers we can estimate the future state of collaboration

it always predicts that module to be not defect prone over time. For this reason 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.

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 traditional defect prediction model as its data source. If it sees a label there, it suggests that a module has that label at any future time. The second expert is a dumb one that we will name as E_h . It only suggests modules which has defects or defects among their neighbors in the call graph to be defect prone. If we mix the outcomes of these experts

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	-	✓

only one conflict may occur: It is the case where E_h predicts a module as defect prone while E_m predicts it as defect free. In this case, we decide the outcome probabilistically weighted by the weights of the experts. 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.

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 probabilistically labels the module as defect prone.

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 thrust our learning model completely in the case of a conflict. After that, at any given future time we reweight the experts. A visual representation is presented in Figure 5.4: A short description of the steps of the model is provided below:

- (i) E_h labels a module as defective if it had at least 1 past defect.
- (ii) E_h and E_m weights are initialized to 0 and 1 respectively.
- (iii) 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.
- (iv) Weights are re-evaluated at each turn.
- (v) At any future time E_h weight is equal to number of changes on the module due to defects over number of changes to the module.
- (vi) If at any future time changes due to defects over changes due to any reason in a module is ≥ 0.5 change E_m weight to 0.
- (vii) If E_h predicts the module to be non-defect prone its weight is set to 0.

5.2.4.2. 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), and probability of false alarms (pf) rates [95]. 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 [95]. 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 constrained project low pf rate can be more important for resource allocation.

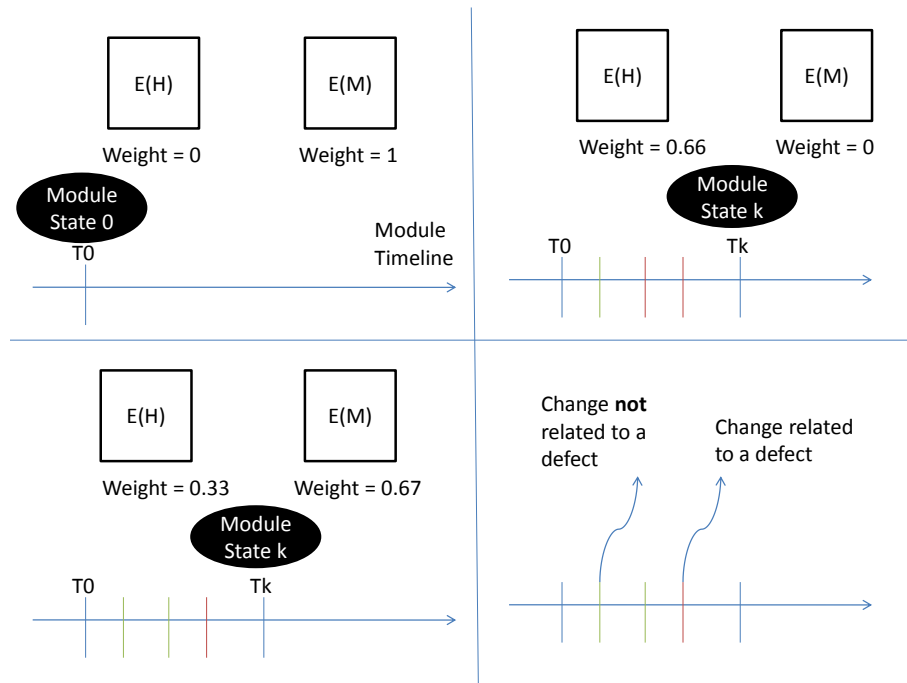


Figure 5.4: Reweighting method of live defect prediction. Initially E_h weight is set to 0 and we exclusively use the classifier output. After module changed for a number of times we re-weight E_h as number of changes due to defects over number of all changes in a module. If weight of E_h is ≥ 0.5 we exclusively use the output of E_h .

Table 5.2: Confusion Matrix

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

$$pd = \frac{A}{A + C} \tag{5.6}$$

$$pf = \frac{B}{B + D} \tag{5.7}$$

6. Software Networks

6.1. Software Network

6.1.1. Call-Graphs of Software Modules

We used the static call-graphs of software methods/functions. 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 statical analysis we can not be sure if a module is called, or how much a module is called at runtime. However, their ease of extraction make them a convenient way in analysing complex programs.

6.1.2. Collaboration Network

We constructed the collaboration graph from source repository. From the source repository we constructed the collaboration network like the following: In the network $G = \langle V, E \rangle$, $v_i \in V$ 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.

6.2. Properties of Scale-Free Networks

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

6.2.1. 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. In previous progress report, we have shown that collaboration network conforms this rule. For example for the Enterprise Software collaboration network, in Figure 6.1 it can be observed that the 1-degree polyfit has a error rate of

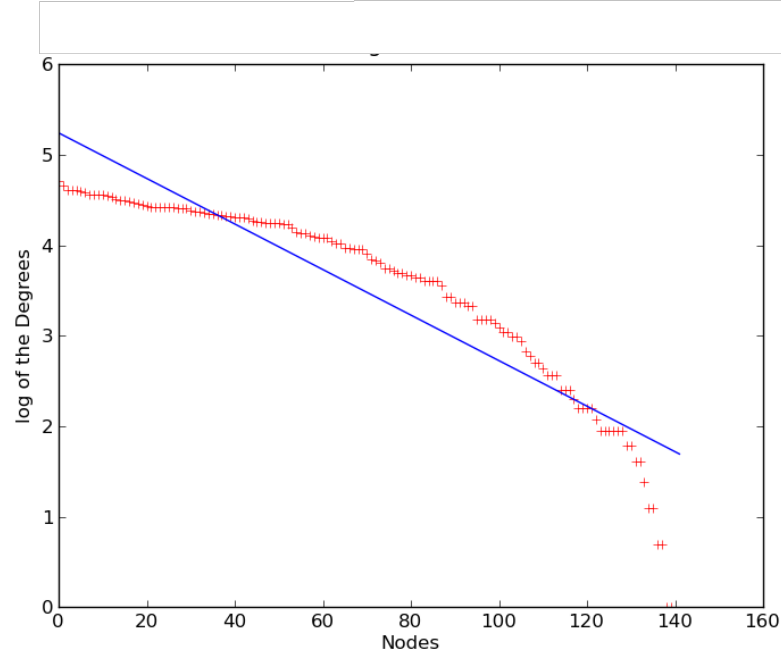


Figure 6.1: Degree Distribution of the Nodes in The Enterprise Software Dataset Collaboration Network with log-normal scale

mean relative error 0.27. As seen in Figure 6.2, 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 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 [8]. 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.

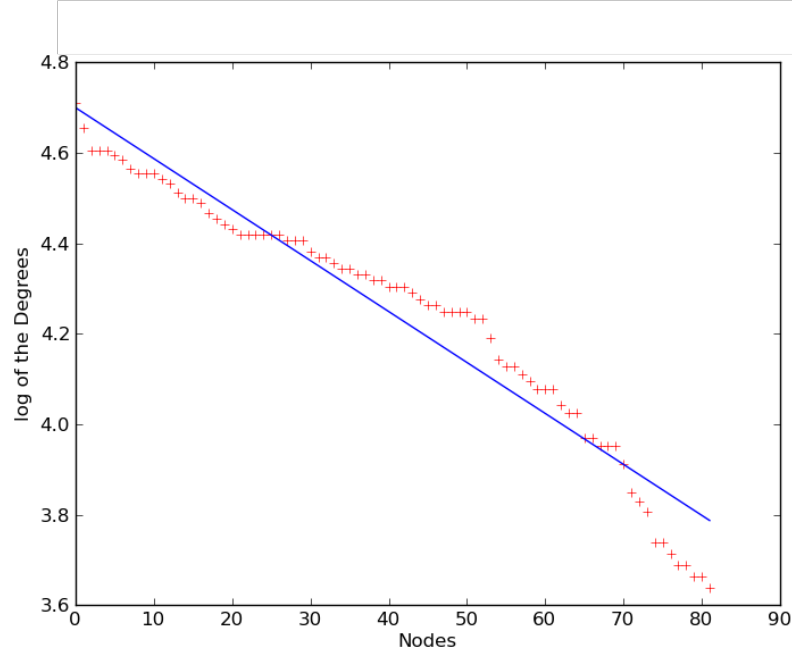


Figure 6.2: Degree Distribution of the Nodes in Enterprise Software Collaboration Network with log-normal scale with top 80 nodes: notice a closer fit

6.2.2. 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 [8].

6.2.3. Network Evolution

6.3. Kronecker Networks

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 [20]. 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 use stochastic Kro-

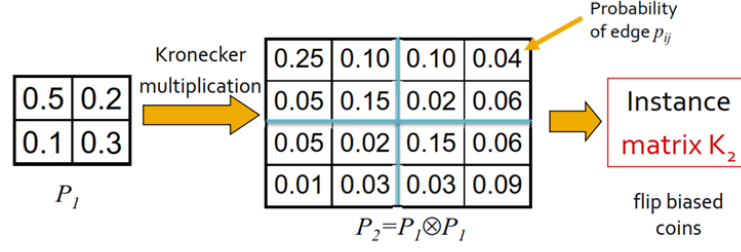


Figure 6.3: One iteration of the adjacency matrix with Stochastic Kronecker generator

necker 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 6.3 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[22].

We use KronFit algorithm to estimate the parameters of G_1 [22]. Details of the algorithm can be seen in Figure 6.4. The actual code of the implementation is 2000+ LOC so it is not practical to attach to this document. For this reason, we provide the source code in the following link: <http://d1.dropbox.com/u/695188/kronecker.cpp> 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.

Network evolution is the underlying dynamics of how a graph evolves over time. Researchers have tackled this problem since 1970s,[96, 97, 98, 99, 100, 101]. Concepts such as "rich gets richer" have been proposed in order to explain the evolutionary dynamics of the networks [8]. Rich gets richer or preferential attachments explains the degree distribution by a simple probabilistic model. In the model, a node with high degree is proportionally more likely to get even more connections.

We use network densification and estimate the node increase speed by fitting node count over time with a polynomial of degree three [21, 102]. Network densification

Algorithm 1 KRONFIT algorithm

Input: integer N_1 , and graph G on $N = N_1^k$ nodes
Output: MLE parameters $\hat{\Theta}$ ($N_1 \times N_1$ matrix)
initialize $\hat{\Theta}_1$
while not converged **do**
 evaluate gradient: $\frac{\partial}{\partial \Theta_t} l(\hat{\Theta}_t)$
 update parameters: $\hat{\Theta}_{t+1} = \hat{\Theta}_t + \lambda \frac{\partial}{\partial \Theta_t} l(\hat{\Theta}_t)$
end while
return $\hat{\Theta} = \hat{\Theta}_t$

Algorithm 2 Calculating log-likelihood and gradient

Input: Parameter matrix Θ , and graph G
Output: Log-likelihood $l(\Theta)$, and gradient $\frac{\partial}{\partial \Theta} l(\Theta)$
for $t = 1$ **to** T **do**
 $\sigma^{(t)} := \text{SamplePermutation}(G, \Theta)$
 $l_t = \log P(G|\sigma^{(t)}, \Theta)$
 $\text{grad}_t := \frac{\partial}{\partial \Theta} \log P(G|\sigma^{(t)}, \Theta)$
end for
return $l(\Theta) = \frac{1}{T} \sum_t l_t$, $\frac{\partial}{\partial \Theta} l(\Theta) = \frac{1}{T} \sum_t \text{grad}_t$

Figure 6.4: KronFit Algorithm [22]

concept can be explained with the following equation:

$$e(t) \propto n(t)^\alpha \quad (6.1)$$

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 [21]. (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.)

REFERENCES

1. W. Everett and S. Honiden, “Reliability and safety of real-time systems,” *Software, IEEE*, vol. 12, no. 3, pp. 13–16, May.
2. B. Nuseibeh, “Ariane 5: Who dunnit?” *Software, IEEE*, vol. 14, no. 3, pp. 15–16, May-June 1997.
3. A. Puntambekar, *Software Engineering And Quality Assurance*. Technical Publications, 2010.
4. E. S. Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O’reilly and Associates, 1999.
5. U. Riss and A. Rickayzen, “Challenges for business process and task management,” ...*Knowledge Management*, 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
6. J. H. Miller, *Complex Adaptive Systems: An Introduction to Computational Models of Social Life*. Princeton University Press, 2007.
7. 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.
8. D. Easley and J. M. Kleinberg, *Networks , Crowds , and Markets : Reasoning about a Highly Connected World*, draft ed. Cambridge University Press, 2010.
9. 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>

10. A.-L. Barabási and E. Bonabeau, “Scale-free networks,” pp. 50–59, 2003. [Online]. Available: <http://elibrary.ru/item.asp?id=7821500>
11. 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. 845–54, Aug. 2009. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pubmed/19380275>
12. 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>
13. 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>
14. 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>
15. 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>
16. D. Jannach, M. Zanker, A. Felfernig, and G. Friedrich, *Recommender Systems: An Introduction*, 1st ed. Cambridge University Press, 2010.

17. 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
18. 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.
19. F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2nd Edition)*. Addison-Wesley Professional, 1995.
20. 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>
21. 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>
22. J. Leskovec, "Kronecker Graphs : An Approach to Modeling Networks," *Journal of Machine Learning Research*, vol. 11, pp. 985–1042, 2010.
23. 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.
24. 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
25. 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>
 26. 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/csd/doi/10.1109/ICPC.2010.46>
 27. 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>
 28. 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>
 29. 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>
 30. B. Caglayan, A. Misirli, and A. Miranskyy, “Factors characterizing reopened issues: a case study,” in *Promise 2012*, 2012, pp. 1–10. [Online]. Available:

<http://dl.acm.org/citation.cfm?id=2365327>

31. 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
32. T. Zhang and B. Lee, “An Automated Bug Triage Approach: A Concept Profile,” *LNCS 7389*, pp. 505–512, 2012.
33. 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>
34. 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/>
35. 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>
36. 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>
37. 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>
38. 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.
 39. 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.
 40. F. Akiyama, “An example of software system debugging,” *Information Processing*, vol. 71, pp. 353–379, 1971.
 41. M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. New York, NY, USA: Elsevier Science Inc., 1977.
 42. 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.
 43. —, “Automating algorithms for the identification of fault-prone files,” pp. –, 2007.
 44. 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.
 45. 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.
 46. S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, “Benchmarking classification models for software defect prediction: A proposed framework and novel findings,”

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, vol. 34, pp. 485–496, 2008.

47. 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.
48. 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.
49. N. Nagappan and T. Ball, “Using software dependencies and churn metrics to predict field failures: An empirical case study,” pp. –, 2007.
50. 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.
51. 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.
52. T. Menzies, B. Turhan, A. B. Bener, G. Gay, B. Cukic, and Y. Jiang, “Implications of ceiling effects in defect predictors,” pp. –, 2008.
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. G. Çalkl and 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>
 60. 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.
 61. 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.
 62. 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>

63. D. Jannach, M. Zanker, A. Felfernig, and G. Friedrich, *Recommender Systems: An Introduction*, 1st ed. Cambridge University Press, 2010.
64. 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
65. 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>
66. 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>
67. J. M. Kleinberg, “The Small-World Phenomenon : An Algorithmic Perspective,” Cornell University, Tech. Rep., 1999.
68. —, “Navigation in a small world,” *Nature*, vol. 406, no. 6798, p. 845, Aug. 2000. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pubmed/16907213>
69. 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>
70. 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>

71. 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>
72. 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>
73. 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>
74. 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>
75. 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
76. 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.
77. 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

78. 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>
79. 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>
80. 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>
81. 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>
82. 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>
83. 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>
84. 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>
85. 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>
 86. 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>
 87. B. Caglayan, A. T. Misirli, G. Calikli, A. Bener, T. Aytac, and B. Turhan, “Dione: an integrated measurement and defect prediction solution,” in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE ’12. New York, NY, USA: ACM, 2012, pp. 20:1–20:2. [Online]. Available: <http://doi.acm.org/10.1145/2393596.2393619>
 88. “Android Web Page, <http://www.android.com/>.” [Online]. Available: <http://www.android.com/>
 89. “Gartner Report on Third Quarter of Market Share: Mobile Communication Devices by Region and Country, 3Q11,” 2011. [Online]. Available: <http://www.gartner.com/it/page.jsp?id=1848514>
 90. 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>
 91. 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>
92. 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.
 93. K. Xu, “How has the literature on GINI’s Index evolved in the past 80 years,” *China Economic Quarterly*, pp. 1–41, 2003.
 94. T. DeMarco and T. Lister, *Peopleware: Productive Projects and Teams*, 2nd ed. Dorset House Publishing Company, 1999.
 95. E. Alpaydin, *Introduction to Machine Learning*. MIT Press, 2004.
 96. 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://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5175362>
 97. 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>
 98. 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>
 99. 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>
 100. 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>
101. 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>
 102. 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>