Validation of Network Measures as Indicators of Defective Modules in Software Systems

Ayşe Tosun
Department of Computer Engineering,
Boğaziçi University,
Istanbul, Turkey
+90 212 359 7227
ayse.tosun@boun.edu.tr

Burak Turhan
Institute for Information Technology,
National Research Council,
Ottawa, Canada
+1 613 993 7291
Burak.Turhan@nrc-cnrc.gc.ca

Ayşe Bener
Department of Computer Engineering,
Boğaziçi University,
Istanbul, Turkey
+90 212 359 7226
bener@boun.edu.tr

ABSTRACT

In ICSE'08, Zimmermann and Nagappan show that network measures derived from dependency graphs are able to identify critical binaries of a complex system that are missed by complexity metrics. The system used in their analysis is a Windows product. In this study, we conduct additional experiments on public data to reproduce and validate their results. We use complexity and network metrics from five additional systems. We examine three small scale embedded software and two versions of Eclipse to compare defect prediction performance of these metrics. We select two different granularity levels to perform our experiments: function-level and source file-level. In our experiments, we observe that network measures are important indicators of defective modules for large and complex systems, whereas they do not have significant effects on small scale projects.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Complexity measures, Process metrics.* D.4.8 [**Performance]:** Measurements, Modeling and Prediction.

General Terms

Experimentation, Measurement, Performance.

Keywords

Code metrics, network metrics, defect prediction, public datasets.

1. INTRODUCTION

As software systems become larger and more complex, the need for effective guidance in decision making has considerably increased. Various methods/ tools are used to decrease the time and effort required for testing the software to produce high quality products [2, 3, 12]. Recent research in this context shows that defect predictors provide effective solutions to the software industry, since they can provide the developers potentially problematic areas in the software [4, 5, 6, 8]. With such intelligent oracles, resources spent for testing and bug tracing can be allocated effectively while preserving quality of the software at the same time.

Learning-based defect predictors are often built using static code attributes and the location of defects, both of which are extracted from completed projects. Static code attributes are widely accepted by many researchers, since they are easily collected from various systems using automated tools and they are practical for the purpose of defect prediction [2, 4, 5, 6, 8, 10, 11, 19]. Although successful defect predictors can be built using static code attributes, it is observed that their information content is limited [18]. Therefore, many algorithms suffer from a ceiling effect in their prediction performances such that they are unable to improve the defect detection performance using size and complexity metrics.

There are studies that focus on other factors affecting an overall software system such as development processes [7], dependencies [1], code churn metrics [15] or organizational metrics [16]. 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 [1, 14, 15]. In a recent study, Zimmermann and Nagappan also challenged the limited information content of data in defect prediction [1]. 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 recall, i.e. detection rate, is by 10% higher, when network metrics are used to find defective binaries than code complexity metrics.

In this research, we extend the study of network analysis in order to reproduce the previous work [1], validate and/or refute its results using new datasets and further improve the performance of these metrics in defect prediction by using additional methodologies. First, we evaluate both code complexity and network metrics using five additional data sets: three projects from relatively small scale embedded software of a white-goods manufacturer and two versions of Eclipse, all of which are publicly available [20]. We have designed two experimental setups. One of them is the replication of the previous study that proposed the significance of network measures in predicting critical software components [1]. The other one is that, we propose a learning-based defect prediction model to evaluate and compare with the previous models. To further improve the prediction performance of the learning based model, we incorporate a simple weighting framework based on the inter-module interactions of software systems. In our experiments we show that network metrics are important indicators of defective modules in large and complex systems. On the other hand, we argue that these metrics do not have significant effects on small scale projects.

We present our study as follows: We start with a discussion on the related work in Section 2. In Section 3, we explain the data collection process, our data sources, their complexity metrics, dependency data as well as the network metrics derived from these dependencies. Later, we present the results of replication study using new datasets and describe our defect predictor with its extension of a simple weighting framework in Section 4 as well as the results in Section 5. In Section 6, we conclude our study and discuss future research directions.

2. RELATED WORK

Size and complexity metrics such as static code attributes have been so far widely used in many defect prediction studies to predict defective modules of a system [2, 4, 5, 6, 8, 10, 11, 19]. Menzies et al. defines the current status of the research in empirical software engineering as *static* due to the fact that many complex algorithms fail to improve the performance of the defect predictors [18]. The authors discussed that researchers need to find new ways to have a better insight on the data used in defect prediction studies. Improvement on the information content of the data can be one option rather than applying complex algorithms.

Various researchers focus on critical factors that would capture additional information about the overall software system [1, 14, 15, 16]. Code churn metrics are used to examine the change in the code over time [15]. Specific results indicate that these metrics are able to discriminate failures in the system with an accuracy of 89 percent. However, software data with a version history or code churn measures are not publicly available for other researchers to reproduce the results. Recently, process metrics related with the organizational structure has been integrated into defect prediction studies to investigate the relationship between human factors, processes, organizational complexity and defects in the software [16]. Although these metrics predict failure-proneness in the software system with significant precision and recall values, it is often hard to collect such information from open source projects or from small scale companies. Especially in small scale organizations, there exists limited number of engineers where the depth of the organizational structure tree may not provide remarkable measures about the company [16].

Nagappan and Ball, on the other hand, examined software dependencies and code churn measures while predicting failures of Windows Server 2003 [14]. They showed that those metrics are statistically significant indicators of post-release failures in the system. Furthermore, Zimmermann and Nagappan extended that research by comparing network and complexity measures, where network metrics are derived from the interactions between binaries of Windows Server 2003 [1]. They used automated tools to build dependency graphs between binaries and extract network measures. Their results provide significant conclusions: *Network metrics derived from dependency graphs are able to predict defects in the system better than the complexity metrics*. However, their results come from only one project (Windows product). Results may change on different datasets depending on the underlying system architecture.

On the other hand, transfer of the current experimental knowledge and generalization of the results is only possible with replication studies, which is one of the most difficult approaches in empirical studies [5]. In replication experiments, it is fundamental but not sufficient to reproduce the previous setting as similar as possible. Researchers must address new and deeper research questions to "increase the confidence" of their results and focus on different factors affecting the final outcome. We have been motivated by the fact that empirical software engineering is seemed to be less ideal due to isolated experiments and less generalized results in the recent research [30]. Thus, our research goals are stated as:

- Analysis of network measures as indicators of defective modules in different software systems.
- Reproduction of the previous experimental setups which focus on the relation between network measures and defect-prone components of a software product [1].
- Challenging these results on five public datasets.
- Improving the outcomes with new factors such as call graph based weighting framework and learning based oracles.

3. SOFTWARE DATA & METRICS

Zimmermann and Nagappan employed their studies on Windows Server 2003 product [1]. To externally validate their results, we used publicly available datasets from two sources, whose metrics are collected in different granularity levels. Therefore our results can easily be repeated and refuted [20]. Furthermore, the data sources we used contain diversities in terms of the size, programming language, the domain and the functionality of the software. Our results will show whether network metrics have different effects on predicting defective modules in various software systems.

We have used two public data sources: three small scale embedded software from a Turkish white goods manufacturer and two releases of an open source project, Eclipse [20]. We extract code complexity metrics (Section 3.1), dependency data (Section 3.2) and network metrics (Section 3.3) from these projects. To further extend the idea of the previous study [1], we apply a simple weighting framework, i.e. Call Graph Based Ranking Framework (CGBR) on code complexity and network (Section 4.2). Using CGBR framework, we weight modules based on their inter-module dependencies.

Data of three small scale projects, namely AR3, AR4 and AR5, have embedded controller software for white-goods, manufactured in a local company [20]. Code complexity metrics, collected from functional methods, and defect data matched with those methods can be accessed via Promise Repository [20]. In addition, we have extracted dependency data between functional methods of the projects to serve the following purposes: a) to collect network metrics with Ucinet 6 Network Analysis tool [22] as in the study of [1], b) to weight intra-module complexities, i.e. complexity metrics, with inter-module dependencies using CGBR framework. We have used our open-source metrics extraction tool, *Prest,* to obtain inter-module dependencies [23]. Prest is able to parse C, C++, Java, Jsp and PL/ SQL files to extract static code attributes of the modules (either package, class, file or method level) in any software system. In addition, it builds static call dependency matrices between modules, i.e. methods, of a project. Prest can capture both inter-module and intra-module relations inside a system.

Second data source contains releases 2.0 and 2.1 (from years 2002 and 2003) of an open source environment, Eclipse [20]. We treat these releases as two projects and named them as v2_0 and v2_1. We have used their file-level code complexity metrics from the Promise Repository [20]. We have combined pre- and post-release defect contents and matched them with the code metrics. Furthermore, we have accessed the source codes of v2_0 and v2_1 from the online web source [21], and built their dependency matrices using Prest [23]. Since Prest only builds dependencies at the functional method level, we have first computed method-level network metrics from two Eclipse projects and then converted them to source-file level values to match with the file-level complexity metrics and the defects.

General properties of all projects used in our study can be seen in Table 1. The sizes of the projects vary between 2732 LOC and 987603 LOC with different defect rates. We have used two different granularity levels in our data sources such that a module is a single functional method for the first three projects, whereas it is a source file for the last two projects. Therefore, the number of modules represents the number of methods for AR3, AR4 and AR5, whereas, it is the number of files for v2_0 and v2_1.

Table 1. General properties of datasets

	Code metrics	Network metrics	Total LOC	Modules	Defect Rate
AR3	29	23	5624	63	0.12
AR4	29	23	9196	107	0.18
AR5	29	23	2732	36	0.20
v2_0	198	23	796941	6729	0.013
v2_1	198	23	987603	7888	0.004

3.1 Code Complexity Metrics

We use code complexity metrics in order to provide a baseline for the performance of our defect prediction model and then, to compare their results with the network metrics. We select static code attributes such as McCabe, Halstead and LOC metrics as the *complexity metrics*, since they are widely-used, practical and easily collected through automated tools [10, 11]. Many researchers accept these attributes as significant indicators of defective modules in learning-based defect predictors [2, 4, 6, 8, 10, 11, 19]. Therefore, we also assume that static code attributes provide as much information as possible so far to make accurate predictions in software systems. A complete list of static code attributes along with their explanations from [4] can be seen in Appendix A. Similarly, various complexity metrics are collected from Eclipse projects by Schröder et al. [24] and average, sum and maximum values are extracted for each Java file in the versions 2_0 and 2_1. We have used all of these complexity metrics in our experiments.

3.2 Dependency Graph

A *software dependency* shows a directed relation between two elements of the code, i.e. binaries, files or methods [1]. Different kinds of dependency exist such as *data dependencies*, which observe the declaration and use of data values, and *call dependencies*, which observe the declaration and call of the elements [1]. Dependency graph of a software system is simply a directed graph, where the *nodes* are the elements of the software and the *edges* are the dependencies, i.e. caller-callee relations, between these elements. Zimmermann and Nagappan used an automated tool developed in Microsoft to build the dependency graph of their Windows binaries [1]. We have tracked dependency information with our metrics extraction tool, Prest [23], and stored this data as an NxN matrix which contains static caller-callee relations between methods of the projects.

3.3 Network Metrics

We have worked with Ucinet 6 tool [22] to extract *Ego* and *Global Network Metrics* both of which are called as *network metrics*. While ego metrics measure the importance of the modules within the local neighborhood, global metrics measures the importance of the modules within the entire system [1]. A complete list of network metrics is provided in Appendix B. We have extracted network metrics based on the dependencies between C methods in three embedded software systems and between Java files in two Eclipse projects.

4. EXPERIMENTS

We conducted two types of experiments in order to a) reproduce and evaluate the performance of network metrics by using new datasets, and, b) improve the prediction performance of the predictor using a Naïve Bayes model incorporated with Call Graph Based Ranking (CGBR) Framework. Both experiments have different experimental designs such that they are individually described in their subsections. However, a typical and common confusion matrix is preferred (Table 2) to assess the performance of this empirical study [25]. From the confusion matrix, all performance measures are computed as follows:

Table 2. Typical confusion matrix

Actual	**Predicted**	
	Defective	**Defect-free**
Defective	A	B
Defect-free	C	D

$$pd\ (recall) = A/(A+B)$$

$$precision = A/(A+C)$$

$$pf = C/(C+D)$$

$$bal = 1-\sqrt{(0-pf)^2+(1-pd)^2}\Big/\sqrt{2}$$

In the previous study, authors computed *recall* and *precision* measures from this matrix [1]. Both precision and recall should be close to 1 so that the model does not produce false negatives or false positives. *Precision* measures the percentage of defective modules that are classified correctly, over the modules that are predicted as defective. *Recall*, on the other hand, is the same as *probability of detection (pd)*. *Pd* measures how good our predictor is in finding actual defective modules.

Instead of *precision*, we have measured *probability of false alarm rates* (*pf)* in our previous studies [2, 8, 18, 19]. *Pf* measures the false alarms of the predictor, when it classifies defect-free modules as defective. We have computed *pf* to measure additional costs our predictors would cause. Predicting actual defective modules is the prior action in such oracles. However, increasing *pd* or *precision* would cause additional false alarms, which means we unnecessarily highlight safe regions in the software and waste significant amount of time and effort. Therefore, one of our aims during calibrating this predictor model should be decreasing this *pf* rate, while keeping high *pd* rates and high *precision* rates. We used all three measures in this study to make a comprehensive comparison of our work with the previous study.

In the ideal case, we expect from a predictor to catch maximum number of defective modules (ideally *pd* = 1, *precision* = 1). Moreover, it should give minimum false alarms by misclassifying actual defect-free modules as defective (ideally *pf* = 0). Finally, to measure how close our estimates are to the ideal case, we use *balance* measure. The ideal case is very rare, since the predictor is activated more often in order to get higher probability of detection rates [4]. This, in turn, leads to higher false alarm rates. Thus, we need to achieve a prediction performance which is as near to (1, 1, 0) in terms of (*precision*, *pd*, *pf*) rates as possible.

4.1 Replication: Logistic Regression with PCA

In the replication part, we used the same algorithms used by Zimmermann and Nagappan: logistic regression and linear regression models. Linear regression and correlation analysis show that network metrics such as *Fragmentation* and *dwOutReach* are positively correlated with defects in software systems. However, we have only displayed the results of logistic regression, since it produces likelihood probabilities instead of R-square measures of linear model. Thus, its prediction performance can be easily illustrated via recall and precision measures. Similar to Zimmermann and Nagappan, we have applied *Principal Components Analysis* to overcome the problem of inter-correlations among metrics [1]. We selected principal components whose cumulative variance is above 95 percent.

In the experimental design, we have separated two thirds of each dataset as training and the rest as testing sets. After 50 random splits, *recall (pd)*, *precision*, *pf* and *balance (bal)* measures in percent, using only complexity metrics (CM), only network metrics (NM) and both of them (ALL), can be seen in Table 3. We took the average of 50 random iterations and represented in percent. We can interpret the results in Table 3 as follows:

- Network metrics are not significant indicators of defective modules in small-scale projects, predicting around 30% of defective modules in AR3, AR4 and AR5, whereas they compete with the complexity metrics in Eclipse versions, around 70% detection rates.
- Therefore, as opposed to the findings in the previous study [1], the performance of network metrics in defect prediction changes with respect to size and complexity of software systems.
- Although the performance of logistic regression in v2_0 and v2_1 are quite satisfactory in terms of *pd*, *pf* and *bal*, it could not go beyond 40% prediction accuracy in three embedded software.
- One threat to internal validity of these results could be the algorithm. Logistic regression may fail to fit on embedded software data. Therefore, we have decided to reproduce those experiments with a learning based model.

Table 3. Results (%) of the logistic regression with PCA

		pd	**precision**	**pf**	**bal**
AR3	**CM**	37	30	10	55
	NM	26	34	10	47
	ALL	34	33	10	53
AR4	**CM**	61	46	12	71
	NM	19	5	18	41
	ALL	45	47	12	60
AR5	**CM**	44	48	15	59
	NM	51	60	12	64
	ALL	49	62	12	63
avg	**CM**	**47**	**41**	**12**	**62**
	NM	**32**	**33**	**14**	**51**
	ALL	**43**	**47**	**12**	**59**
v2_0	**CM**	69	56	19	74
	NM	70	61	23	73
	ALL	70	59	19	75
v2_1	**CM**	68	44	9	76
	NM	69	56	19	73
	ALL	69	49	14	76
avg	**CM**	**69**	**50**	**14**	**75**
	NM	**70**	**59**	**21**	**73**
	ALL	**70**	**54**	**16**	**76**

4.2 Extension: Naïve Bayes Predictor Using CGBR (Our Proposed Model)

To observe different factors that may affect the prediction performance; we have focused on increasing the information content using additional metrics extracted from dependencies. Moreover, we have used Naïve Bayes classifier, since recent study shows that learning-based defect predictors with a Naïve Bayes classifier outperform other machine learning methods with an average (71%, 25%) in terms of (pd, pf) [4]. One of the reasons for the success of Naïve Bayes algorithm is that it manages to combine signals coming from multiple attributes [2, 4]. It simply uses attribute likelihoods derived from historical data to make predictions for the modules of a software system [4, 25].

As the inputs to the model, we have collected complexity and network metrics together with the actual defect information from five software systems. Since complexity metrics are highly skewed and Naïve Bayes assumes Gaussian distribution of data, we have log-filtered all metrics before applying the model on the datasets [4]. Our model outputs a posterior probability indicating whether a module, either a method or a file depending on the data source, is defect-prone or not.

We have incorporated a Call Graph Based Ranking Framework to our prediction model before the metrics are fed into the system. This framework adjusts the data by multiplying each module with a rank derived from call dependency matrices. Our previous work on CGBR Framework shows that inter-module call dependencies adjust the complexity metrics significantly so that the defect prediction model produces fewer false alarms [26]. The essential algorithm of the CGBR framework is explained in subsection 4.2.1 in detail.

In our experimental design, we have kept one project as the test set and used random sampling to select 90 percent of the other projects in the same data source to form the training set. We have repeated this procedure 100 times for five projects to overcome the ordering effects [17, 25]. Then, we have presented the performance measures of the experiments after

(100 randomize orderings for the training set) x (5 projects) = 500 iterations.

4.2.1 Call Graph Based Ranking Framework

We have collected static caller - callee relations as *dependency data* to weight modules based on their interactions. In the previous research, Kocak et al. [26] and Turhan et al. [27] used these call dependencies to increase the information content of the static code attributes. These calls are represented as 1's or 0's in a matrix, depending on which module calls the others or is called by the other modules. Since these call dependencies represent structural complexities, they succeeded to build a model that not only use intra-module complexities derived from complexity metrics, but also inter-module complexities. Their results show the effectiveness of dependencies on adjusting static code attributes by taking into account the structural complexity of projects.

We have added the CGBR framework to our model in order to further quantify the effects of dependencies on complexity and network metrics. An overview of our defect prediction model with a schema representing the flow can be seen in Figure 1. On the left side of the schema, Naïve Bayes classifier is applied to log-filtered data which is adjusted with CGBR weights. To calculate these weights for each module of the software, we have composed an *NxN* matrix, M, where rows represent all modules that call others and columns represent modules that are called by the others. After constructing call dependency matrices, we have utilized a web link based paging algorithm, PageRank, to calculate ranks, i.e. weights, for the modules of all projects [28]. The basic flow for producing CGBR values is illustrated in Figure 2. Inspired from the web page ranking algorithms, we have treated each interaction between modules as a hyperlink from one page to another. We have assigned equal ranks to the modules of a project and iteratively increased module ranks using PageRank algorithm. After achieving CGBR values, we have normalized these values into the range between 0.1 and 1.0. Finally, we have adjusted complexity and network metrics by multiplying the metrics of each module with its corresponding rank value. In order to assign optimum CGBR ranks to the modules, we have iterated our PageRank algorithm 40 times until it converges.

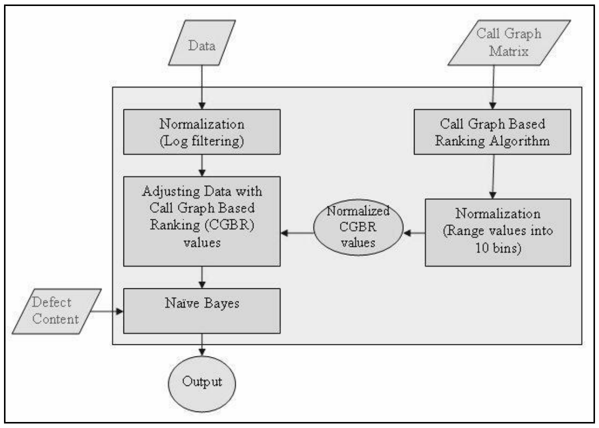
Data
Call Graph Matrix
Normalization (Log filtering)
Call Graph Based Ranking Algorithm
Adjusting Data with Call Graph Based Ranking (CGBR) values
Normalized CGBR values
Normalization (Range values into 10 bins)
Defect Content
Naïve Bayes
Output

Figure 1. An overview of the proposed model (Section 4.2)

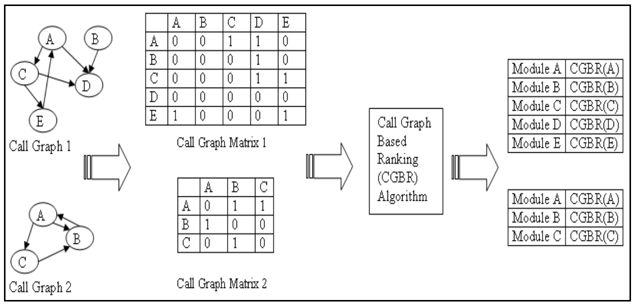

Figure 2. Producing CGBR values (Section 4.2.1)

5. RESULTS

The results of our experiments on three local projects and two Eclipse releases are presented in Table 4 and Table 5 respectively. Tables include five consequent experiments completed with complexity metrics (CM), network metrics (NM) and combination of them (ALL). We have conducted our experiments twice by adjusting the metrics with CGBR and without CGBR framework for two different data sources. First, we have measured the performance of our model using only complexity metrics. Then, we have adjusted these metrics with CGBR values to see the effects of using inter-module interactions. After setting the baseline with complexity metrics, we have used the same procedure for network metrics alone to observe their defect detection performance. Finally, we applied CGBR framework onto all metrics and evaluated the performance.

Table 4 shows that complexity metrics (CM) adjusted with CGBR framework increases *pd* slightly, whereas it increases *precision* significantly when compared to CM alone. On the other hand, the adjustment with CGBR significantly decreases false alarm rates for the local projects. When we observe the experiments done with network metrics (NM), it is clearly seen that using NM dramatically decreases detection rates, including precision, and increases false alarms. Therefore, network metrics in small scale projects could not provide significant information about the defect contents as in the case of complexity metrics.

Table 4. Performance measures (%) for small projects

		CM	CM+ CGBR	NM	CM+ NM	ALL+ CGBR
AR3	**pd**	86	***88***	13	87	86
	precision	67	***73***	27	69	67
	pf	42	***33***	35	40	43
	bal	69	***75***	34	70	68
AR4	**pd**	50	***55***	55	55	50
	precision	74	***85***	47	79	67
	pf	18	***10***	62	15	25
	bal	62	***67***	46	66	60
AR5	**pd**	***100***	88	50	***100***	50
	precision	78	***83***	52	78	78
	pf	29	18	47	29	***14***
	bal	79	***85***	51	79	63
Avg	**pd**	*79*	*77*	39	***80***	66
	precision	72	***79***	45	74	71
	pf	30	***20***	48	28	27
	bal	74	***78***	45	76	69

Table 5. Performance measures (%) for large systems

		CM	CM+ CGBR	NM	CM+ NM	ALL+ CGBR
V2_0	**pd**	***70***	65	61	68	67
	precision	66	65	64	66	68
	pf	36	35	35	35	***32***
	bal	***67***	65	63	66	***67***
V2_1	**pd**	81	***82***	79	82	***82***
	precision	63	***64***	56	63	***64***
	pf	48	***47***	62	48	***47***
	bal	63	***64***	54	64	***64***
Avg	**pd**	***76***	74	70	75	75
	precision	64	64	59	64	***65***
	pf	42	41	49	42	***40***
	bal	66	66	59	65	***67***

The results in Table 5 show similar performances using complexity or network metrics. One of them is not significantly better than the other and they reach, on the average 70% to 76% detection rates. Moreover when we use both metrics adjusted with CGBR framework, we would further improve the prediction performance for Eclipse data. Therefore, we can conclude that network metrics are also successful indicators of defective modules as the complexity metrics in large and complex systems.

In summary, we have analyzed the effect of network metrics on defect proneness in different software systems. We have challenged the fact that network measures produce better pd rates than complexity measures in predicting defective modules. Finally, we have reached two different conclusions: a) Complexity metrics adjusted with CGBR framework are effective indicators of defective modules in small projects with (*77, 79, 20, 78*) in terms of (*pd, precision, pf, bal*), compared to (*39, 45, 48, 35*) using network metrics. b) Network metrics produces similar results in large and complex systems, i.e., (*70, 59, 49, 59*), when compared to (*76, 64, 42, 66*) using complexity metrics alone in terms of (*pd, pf, bal*). These results support our claim that improving information content using intra-module dependencies would also improve the prediction performance of defect predictors. Moreover, we have accomplished that network metrics would have different effects on defect proneness in different software systems: Larger and more complex systems tend to have significant relationship between intra-module dependencies and defects compared to small-scale projects.

5.1 Threats to Validity

One threat to validity of our results can be the level of the granularity that is different in two data sources. We have performed our experiments in two different data sources whose metrics are available at function and file levels. Therefore, when network metrics are collected at function-level, one cannot clearly claim that network metrics are effective indicators of defective modules in large and complex systems. To avoid such threat, we need to enrich our software data repositories to replicate this study with different size of projects on various granularity levels, separately. Another threat to validity can be the type of defect data, i.e. pre-release vs. post-release that is used interchangeably during the experiments. Small projects have only pre-release test defects, whereas Eclipse projects consist of all defect sets. We are aware of the fact that correlation between post-release defects and the network measures should present different conclusions in contrast to the correlation between pre-release defects and the network metrics. Therefore, we need to focus on the correlation between pre-release and post-release defects while conducting future experiments.

6. CONCLUSION & FUTURE WORK

In this research, we have carried out an empirical evaluation of network metrics by taking the study of Zimmermann and Nagappan [1] as the baseline. We have measured the effects of network metrics on defect prediction performance. For this, we have collected code complexity metrics, dependency data and network metrics from two public data sources: a white-goods manufacturer located in Turkey and an open source environment. To extend the previous study, we have incorporated a simple weighting framework (CGBR) that measures inter-module complexities.

Previous research indicates that network metrics are effective indicators of critical binaries in a Windows product and their prediction performance is 10% higher than complexity metrics

[1]. We reproduced this study using the same experimental design and extending it with new methodologies. Both replication experiments and our results reveal that network metrics are significant indicators of defective modules in large and complex systems. On the other hand, they do not provide significant effect on small scale projects.

In empirical studies, we cannot assume that the results present a general trend beyond the environments and the systems software data is gathered [14]. It would be more confident to support our claims when we observe similar behaviors in different applications. We try to extend the data sources by adding different projects with varying characteristics to support our claims with higher confidence. Furthermore, it is one of our future research directions to expand this public test-bed with additional projects from a large and complex system taken from a local GSM company. Currently, static code attributes of various projects from this GSM company are publicly available in the Promise Repository [20]. However, network metrics should be separately extracted to validate the performance of network metrics on new large-scale projects.

7. ACKNOWLEDGMENT

This research is partly supported by Turkish Scientific Research Council, TUBITAK, under grant number EEEAG108E014, and Bogazici University under grant number DPT 2007K120610 TAM.

8. APPENDIX A: Static Code Attributes

The list of static code attributes with their definitions can be seen in Table A.1 [4].

Table A1. Static code attributes

Name	**Description**
Branch Count	Number of branches in a given module.
Operators	Total number of operators found in a module.
Operands	Total number of operands found in a module.
Unique Operators	Number of unique operators found in a module.
Unique Operands	Number of unique operands found in a module.
Executable of Lines of Code	Source lines of code that contain only code and white space.
Lines of Comment	Source lines of code that are purely comments
Lines of Code and Comment	Lines that contain both code and comment.
Blank Lines	Lines with only white space.
Lines of Code	Total number of lines in a module.
Halstead Vocabulary	*n = number of unique operands+number of unique operators*
Halstead Length	*N =operands + operators*
Halstead Volume	*V = N*log(n)*
Halstead Level	*L = V*/V*
Halstead Difficulty	D = 1/L
Halstead Programming Effort	*E = V / L*
Halstead Error Estimate	*B = V / S*
Halstead Programming Time	*T = E / 18*
Cyclomatic Complexity - V(g)	*V(g) = edge count - node count + 2*num. unconnected parts in g*
Cyclomatic Density - Vd(g)	*V(g) / executable lines of code*
Decision Density - Dd(g)	*condition count / decision count*
Module Design Complexity - Iv(g)	*Iv(g) = call pairs*
Design Density - Id(g)	Id(g) = Iv(g) / V(g)
Normalized Cyc. Comp. - NormV(g)	*Norm V(g) = V(g) / lines of code*
Call Pairs	Number of calls to other functions in a module.
Condition Count	Number of conditionals in a given module.
Decision Count	Number of decision points in a given module.
Edge Count	Number of edges found in a given module.
Formal Parameter Count	Number of parameters to a given module.

9. APPENDIX B: Network Metrics

Network metrics described below are collected from dependency graphs using Ucinet 6 tool [22].

Table B1. Network metrics from [1,27]

Name	***Description***
Size	*The size of the ego network is the number of nodes.*
Ties	*The number of directed ties corresponds to the number of edges.*
Pairs	*The number of ordered pairs is the maximal number of directed ties, i.e. Size x (Size-1).*
Density	*The percentage of possible ties that are actually present, i.e. Ties/Pairs.*
WeakComp	*The number of weak components (=sets of connected modules) in neighborhood.*
n WeakComp	*The number of weak components normalized by size, i.e., WeakComp/Size.*
TwoStepReach	*The percentage of nodes that are two steps away.*
ReachEfficiency	*The reach efficiency normalizes TwoStepReach by size, i.e., TwoStepReach/Size. High reach efficiency indicates that ego's primary contacts are influential in the network.*
Brokerage	*The number of pairs not directly*

	connected. The higher this number, the more paths go through ego, i.e., ego acts as a "broker" in its network.
nBrokerage	*The Brokerage normalized by the number of pairs, i.e., Brokerage/Pairs.*
EgoBetween	*The percentage of shortest paths btw neighbors that pass through ego.*
nEgoBetween	*The Betweenness normalized by the size of the ego network.*
EffSize	*The effsize is the number of modules that are connected to a module X minus the average number of ties between these modules.*
Efficiency	*Efficiency norms the effective size of a network to the total size of the network.*
Constraint	*Constraint measures how strongly a module is constrained by its neighbors. The idea is that neighbors that are connected to other neighbors can constrain a module.*
Hierarchy	*Hierarchy measures how the constraint measure is distributed across neighbors.*
Eigenvector	*Eigenvector centrality is similar to Google's PageRank value; it assigns relative scores to all modules in the dependency graphs.*
Fragmentation	*Proportion of mutually reachable nodes*
Betweenness	*Betweenness centrality measures for a module on how many shortest paths between other modules it occurs.*
Information	*Information centrality is the harmonic mean of the length of paths ending at a module.*
Power	*Power based on the notion of "dependency."*
Closeness (in/out)	*Closeness is the sum of the lengths of the shortest (geodesic) paths from a module (or to a module) from all other modules.*
Degree	*The degree measures the number of dependencies for a module.*
dwReach (int/out)	dwReach is the number of modules that can be reached from a module (or which can reach a module).

10. REFERENCES

[1] Zimmermann, T. and Nagappan, N. 2008. Predicting Defect Using Network Analysis on Dependency Graphs. In Proceedings of the International Conference on Software Engineering (ICSE'08), Leipzig, Germany, 531-540.

[2] Tosun, A., Turhan, B. and Bener, A. 2008. Ensemble of Software Defect Predictors: A Case Study. In Proceedings of the International Conference on Empirical Software Engineering and Measurement (ESEM), Keiserslautern, Germany, 318-320.

[3] Adrion, R.W., Branstad, A.M. and Cherniavsky, C.J. 1982. Validation, Verification and Testing of Computer Software. ACM Computing Surveys, June 1982, Vol. 14, No. 2, 159-192.

[4] Menzies, T., Greenwald, J. and Frank, A. 2007. Data Mining Static Code Attributes to Learn Defect Predictors. IEEE Transactions on Software Engineering, January 2007, Vol. 33, No. 1, 2-13.

[5] Mendonca, M.G., Maldonado, J.C., De Oliveira, M.C.F., Carver, J., Fabbri, S.C.P.F., Shull, F., Travassos, G.H., Hohn, E.N. and Basili, V.R. 2008. A Framework for Software Engineering Experimental Replications. 13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2008), 203-212.

[6] Challagula, U.B.V., Bastani, B.F., Yen, L. and Paul, A.R. 2005. Empirical Assessment of Machine Learning based Software Defect Prediction Techniques. In Proceedings of the 10th IEEE International Workshop on Object-Oriented Real Time Dependable Systems, Sedona, USA, 2-4 February 2005, 263-270.

[7] Fenton, E.N. and Neil, M. 1999. A Critique of Software Defect Prediction Models. IEEE Transactions on Software Engineering, September/October 1999, Vol. 25, No. 5, 675-689.

[8] Turhan, B. and Bener, A. 2009. Analysis of Naive Bayes' Assumptions on Software Fault Data: An Empirical Study. Data and Knowledge Engineering Journal, Vol. 68, No. 2, 78-290.

[9] Ostrand, T. J., Weyuker, E. J. and Bell, R. M. 2005. Predicting the Location and Number of Faults in Large Software Systems. IEEE Transactions on Software Engineering, April 2005, Vol. 31, No. 4, 340-355.

[10] Halstead, M.H. 1977. Elements of Software Science, Elsevier, New York.

[11] McCabe, T. 1976. A Complexity Measure. IEEE Transactions on Software Engineering, Vol.2, No.4, 308-320.

[12] Fagan, M. 1976. Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal, Vol. 15, No. 3, 182-211.

[13] Nagappan, N. and Ball, T. 2005. Static Analysis Tools as Early Indicators of Pre-Release Defect Density. In Proceedings of the International Conference on Software Engineering (ICSE), St. Louise, USA, 580-586.

[14] Nagappan, N. and Ball, T. 2007. Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. In Proceedings of the First International

Symposium on Empirical Software Engineering and Measurement (ESEM), Madrid, Spain, 364-373.

[15] Nagappan, N. and Ball, T. 2005. Use of Relative Code Churn Measures to Predict System Defect Density. In Proceedings of the International Conference on Software Engineering (ICSE), St.Louis, USA, 284-292.

[16] Nagappan, N., Murphy, B. and Basili, V.R. 2008. The Influence of Organizational Structure on Software Quality: An Empirical Case Study. In Proceedings of the International Conference on Software Engineering (ICSE), Leipzig, Germany, 521-530.

[17] Hall, M. and Holmes, G. 2003. Benchmarking Attribute Selection Techniques for Discrete Class Data Mining. IEEE Transactions on Knowledge and Data Engineering, Vol. 15, No. 6, 1437-1447.

[18] Menzies, T., Turhan, B., Bener, A., Gay, G., Cukic, B. and Jiang, Y. 2008. Implications of Ceiling Effects in Defect Predictors. In Promise Workshop (part of the 30th International Conference on Software Engineering), Germany, 47-54.

[19] Turhan, B., Menzies, T., Bener, A., Distefano, J. 2008. On the Relative Value of Cross-company and Within-Company Data for Defect Prediction. accepted for publication in Empirical Software Engineering Journal, DOI: 10.1007/s10664-008-9103-7.

[20] Boetticher, G., Menzies, T. and Ostrand, T. 2007. PROMISE Repository of empirical software engineering data http://promisedata.org/ repository, West Virginia University, Department of Computer Science.

[21] Eclipse project archived downloads page: http://archive.eclipse.org/eclipse/downloads

[22] Ucinet tool download page: http://www.analytictech.com/ucinet/ucinet.htm.

[23] Prest. 2009. Department of Computer Engineering, Bogazici University, http://code.google.com/p/prest/.

[24] Schröder, A., Zimmermann, T., Premraj, R. and Zeller, A. 2006. If Your Bug Database Could Talk… In Proceedings of the International Symposium on Empirical Software Engineering (ISESE), Brazil, 18-20.

[25] Alpaydin, E. 2004. Introduction to Machine Learning, MIT Press, Massachusetts.

[26] Turhan, B., Bener, A. and Kocak, G. 2008. Data Mining Source Code for Locating Software Bugs: A Case Study in Telecommunication Industry. accepted for publication in Expert Systems with Applications Journal, DOI: 10.1016/j.eswa.2008.12.028.

[27] Kocak, G. 2008. Software Defect Prediction Using Call Graph Based Ranking (CGBR) Framework. MS Thesis, Boğaziçi University, Turkey.

[28] Brin, S. and Page, L. 1998. The anatomy of a large-scale hypertextual search engine. Computer Networks and ISDN Systems, 107-117.

[29] Heeger, D. 1998. Signal Detection Theory.

http://www.cns.nyu.edu/~david/handouts/sdt/sdt.html.

[30] Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., Jones, P.W., Hoaglin, D.C., El Emam, K., and Rosenberg, J. 2002. Preliminary guidelines for empirical research in software engineering. IEEE Transactions on Software Engineering, Vol. 28, No. 8, 721-734.