Network versus Code Metrics to Predict Defects: A Replication Study

Rahul Premraj
VU University Amsterdam
rpremraj@cs.vu.nl

Kim Herzig
Saarland University
kim@cs.uni-saarland.de

***Abstract*—Several defect prediction models have been proposed to identify which entities in a software system are likely to have defects before its release. This paper presents a replication of one such study conducted by Zimmermann and Nagappan [1] on Windows Server 2003 where the authors leveraged dependency relationships between software entities captured using social network metrics to predict whether they are likely to have defects. They found that network metrics perform significantly better than source code metrics at predicting defects. In order to corroborate the generality of their findings, we replicate their study on three open source Java projects, viz., JRuby, ArgoUML, and Eclipse. Our results are in agreement with the original study by Zimmermann and Nagappan when using a similar experimental setup as them (random sampling). However, when we evaluated the metrics using setups more suited for industrial use – forward-release and cross-project prediction – we found network metrics to offer no vantage over code metrics. Moreover, code metrics may be preferable to network metrics considering the data is easier to collect and we used only 8 code metrics compared to approximately 58 network metrics.**

I. Introduction

Defects in software get increasingly expensive to fix as the software progresses through its life-cycle [2]. Quality assurance via rigorous testing before releasing the product is crucial to keep such costs low. However, test managers are often challenged by limited resources and other factors such as the pressure of time to market the product (e.g., the product must be launched in time for the holiday season shopping). Such constraints put managers in a situation that requires them to draw from their experience and prioritize software entities such as components or source files that should be tested first to increase the likelihood of finding and resolving the most severe or most number of defects. These entities prioritized by the managers are referred to as *defect-prone entities*.

Several *defect prediction models* have been developed by researchers in order to support managers in reliably identifying defect-prone entities. The primary difference between these models is the input data or simply the *metrics* used to characterize the software entities. *Code metrics* such as age and size of the software entity [3], code complexity [4], [5], and code churn [6] have been used to predict defect-prone entities in the past. More recently, socio-technical *network metrics* have been shown to be promising too [1], [7], [8]. Network metrics treat software entities as nodes in a graph and characterize them on the basis of their dependencies with other entities. As opposed to code metrics, network metrics take into account the interactions between entities, thus modelling the flow of information in the software.

Network metrics are however new to the field of predicting defect-prone entities and remain to be evaluated across different types of software systems. In this paper, we investigate the value of network metrics for use in defect prediction models by running an external replication study [9] based on the study conducted by Zimmermann and Nagappan [1] (henceforth referred to as Z & N for sake of brevity). Our study will allow us to appraise the generality of network metrics for defect prediction. Such replication studies are essential in empirical software engineering because they serve as a critical verification step [9] and allow building upon the body of knowledge by giving insights into the conditions under which the results hold [10], [11].

In the original study, Z & N found that network metrics outperform code metrics at predicting defect-prone entities (albeit by a small yet statistically significant margin) in Windows Server 2003, which is a large C++ product. Our focus is to replicate the study on three open-source Java projects of different sizes (Section III) and compare the performance of prediction models trained on code and network metrics. Additionally, we identify the most influential metrics (both code and network) across the projects in order to investigate which ones are most dominant.

More specifically, we seek to answer the following three research questions in our study:

RQ1 *Do code and network metrics predict defect-prone software entities with comparable accuracy within the same release of a project (Sections IV-B and VI)?*

RQ2 *Do code and network metrics predict defect-prone software entities with comparable accuracy across different release in a project (Sections IV-C and VII)?*

RQ3 *Do code and network metrics predict defect-prone software entities with comparable accuracy across different projects (Sections IV-D and VIII)?*

After addressing the above research questions, we present the most influential metrics in Section IX. Thereafter, we discuss our results (Section X) and threats to the validity of our study (Section XI). Finally we conclude our paper in Section XII. In the next section, we discuss previous research closely related to our work Section II

II. Related Work

The research area of predicting defect-prone software entities has been a vibrant one over the last decade. One of the earliest attempts made was by Basili et al. [12] who validated object-oriented metrics to predict defect density. Later, Subramanyam and Krishnan [13] conducted a survey that also showed strong relationship between object-oriented metrics and defects.

Thereafter, studies investigating the value of different types of metrics for defect prediction began to emerge. For instance, Ostrand et al. [3] used file status information and code metrics such as lines of code, age, and prior faults to predict defects. Nagappan et al. [14] proposed evaluated code metrics on five Microsoft products; they also described how to systematically build defect prediction models. Zimmermann et al. [4] demonstrated that complexity and code metrics together perform well at predicting defects, suggesting that higher code complexity leads to more defects.

Besides using code-related metrics such as complexity and age, studies investigating the use of change-related metrics also surfaced. Examples of such work include Moser et al. [15] who used change metrics such as the number of revisions or refactorings to predict defects in Eclipse classes. Prinzger et al. [7] measured fragmentation of developer contributions to predict failure proneness of software modules. Recently, Hassan [16] used process complexity measures to predict defects.

Leveraging dependencies between software entities has also been explored for the purpose of defect prediction. Schröter et al. [17] showed that dependencies attributed to import statements in Java programs can be used to predict defects. Shin et al. [18] and Naggappan and Ball [19] also used dependency information for defect prediction. Dependency relationships have been used by Zimmermann et al. [20] to assess the quality of neighbouring entities.

Zimmermann et al. [1] (the original study of this replication) demonstrated that network metrics outperform source code metrics at predicting defects by conducting an evaluation on data from Windows Server 2003. Tosun et al. [21] then replicated their study on three small closed source C++ projects and on Eclipse. Their results however showed that network metrics have no significant predictive power for small-scale projects. Bird et al. [8] validated also evaluated network metrics on two more software projects: Windows Vista and Eclipse. They extended the set of network metrics to include those that reflect cross-component developer contributions. The results verified that network metrics increase defect prediction accuracy not only for Microsoft products, but also for open source projects like Eclipse.

Table I
Summary of subject projects

Project	Release #	Release date	#files	LOC
JRuby	1.0	Jun. 29, 2007	517	73,343
	1.1	Mar. 29, 2008	550	95,008
ArgoUML	0.24	Feb. 15, 2007	1,480	155,547
	0.26	Sep. 30, 2008	1,752	186,372
Eclipse	2.1	Jun. 27, 2003	7,900	975,292
	3.0	Sep. 14, 2004	6,614	1,296,622

Our replication study shares one commonality with the studies by Tosun et al. [21] and Bird et al. [8] — Eclipse is a common subject project. However, there are many differences in our studies which warrant the work presented in this paper. For instance, in comparison to Tosun et al. [21], who used a tool called Prest to compute network metrics, we used UCINET (same as Z & N) in order to have a closer replication to the original study. Also, our experimental setup to predict defects across different releases for a project has been designed to be more suited (or accessible) for industrial use in comparison to their work (see Section IV-C). Additionally, the smaller projects used by Tosun et al. for evaluations were closed source and C++ based, while ours are open source and Java based. We believe that these differences will increase our combined knowledge on the worth of network metrics at predicting defects.

One of the key differences between our study and that by Bird et al. [8] is the level of granularity at which we predict defects for Eclipse. While Bird et al. chose to predict defects at the plug-in level, we intently chose to work at a finer level of granularity, i.e., source files. The latter is more useful to developers because it suggests highly targeted defect-prone entities as compared to plug-ins which comprise up to several hundred source files.

III. Data Collection

This section presents the subject projects used for our study and the relevant data that we collected, including our methods, to conduct the experiments.

A. Subject projects

We selected three Java projects to perform the experiments on: JRuby, ArgoUML, and Eclipse. The reason to choose these projects is that they differ substantially in size ranging from small (JRuby), medium (ArgoUML) to large (Eclipse). The difference allows us to investigate whether the prediction models trained using the code or network metrics are sensitive to project size. Moreover, the research community

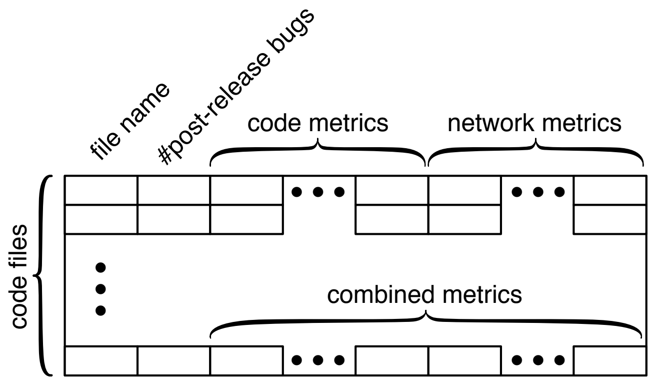

Figure 1. Data collected from subject projects

is familiar with these projects from previous defect prediction studies [4], [22]. A brief summary of the projects used is presented in Table I.

B. Data collected

Our experiments required three sets of data from each project release: *post-release bug data*, *code metrics*, and *socio-technical network metrics* (see Figure 1). We ran our experiments across three sets of metrics: (a) code metrics, (b) network metrics, and (c) combined metrics including both code and network metrics.

Z & N predicted defect-prone entities in their study at the binary level, which is the smallest entity to which bugs could be mapped. The closest equivalents of binaries in Java projects are packages or `jar` files. We consider these entities too course grained to predict defects for; hence chose to map bugs to individual source files and predict defects in files. All data for our study has been collected and aggregated at the file level. The following sections present the methods and tools used to compute the data.

C. Post-release bugs

We leveraged the publicly available bug databases of JRuby and ArgoUML to compute the number of post-release failures observed in the projects. In the case of Eclipse, we used the post-release bug data set made available by Zimmermann et al. [4] in the PROMISE repository. We followed the steps outlined by Zimmermann et al. [4] to gather post-release bugs for JRuby and ArgoUML too. These steps are outlined below:

1) We first identified bug fixes in the projects' history by analyzing their log entries from their version control systems. Any commit message in the log adhering to patterns such as `[bug|issue|fixed]:?\\s*#?\\s?(\\d+)` or `http://[^/]+/issues/show_bug.cgi?id=(\\d+)` (identified using regular expressions) were considered a bug fix candidate.
2) Next, we extracted references to bug ids (identifiers such as #14562) from the candidate commit messages and verified them against the project's bug tracking system to confirm that they are indeed valid bug identifiers. Bug reports with valid matches to identifiers in the commit messages and marked as either *closed* or *resolved* were then linked to the corresponding commit.
3) In order to select only post-release bugs, we filtered reports that were reported before the respective release was made public and not reported after the release date of the subsequent public release.
4) In the last step we associated the bug reports from the above step to files by identifying the files changed in the corresponding commit. Associating bug identifiers to files and counting them as distinct per file gives us the port-release bug data for each project's release.

D. Code metrics

Next, we computed source code metrics for all files shipped within each project release using a commercial tool called *Understand* (Version 2.0, Build 505, http://www.scitools.com/). Understand computes both classical and object-oriented source code metrics for Java and C/C++ projects. Z & N used an in-house Microsoft tool to compute metrics on the products. We selected code metrics to compute using Understand that were as close as possible to their metrics. Table II presents the list of code metrics used in this study including a small description of each.

Note that Understand computes source code metrics at class and method level. Metrics marked in Table II as being computed on method or class level were aggregated to file level by taking the sum of the metric values for all entities declared within the same file.

E. Network metrics

Network metrics essentially map the flow of information in the software system by leveraging dependency relationships between source code entities. That is which software entity "relies on" which other entities by means of calling them or being called by them. In Java projects, software entities would typically refer to Java classes defined in files and the task then is to detect information flow between objects. Once informational flow dependencies between objects are known, including the files in which the classes are defined, it is straightforward to aggregate dependency relationships between files (as needed for our study).

We used Java byte code analysis to detect outgoing method calls and object usages. Collecting dependency data on byte code profits from type resolution performed by the Java compiler and thus, is more accurate than analyzing source code [24]. But this gain in accuracy requires additional effort to map dependencies between files. Compiled class files do not necessarily have the same name as the

Table II
LIST OF CODE AND NETWORK METRICS COMPUTED FOR OUR STUDTY. FOR A FULLER DESCRIPTION OF NETWORK METRICS, PLEASE REFER TO [23]

Code metrics	**Z&N Metric**	**Description**
At the class level:		
NumInstVar	GlobalVariables	# instance variables
CountLineCode	Lines	# source code lines
NumDeclMethod	ClassMethods	# local methods
CountClassDerived	SubClasses	# immediate subclasses
MaxInheritanceTree	InheritanceDepth	inheritance tree depth
CountClassCoupled	ClassCoupling	# classes used as type, data, or member
At the method level:		
CountInput	FanIn	# inputs a function uses. Inputs include parameters and global variables that are used in the function.
CountOutput	FanOut	# outputs that are SET. This can be parameters or global variables.
CyclomaticStrict	Complexity	Strict cyclomatic complexity.
Network metrics:		
Ego-network metrics (computed each for incoming, outgoing, and undirected dependencies; descriptions adapted from Z & N [1]):		
Size	Size	# nodes connected to the ego network
Ties	Ties	# directed ties corresponds to the number of edges
Pairs	Pairs	# ordered pairs is the maximal number of directed ties, i.e., Size $\times$ (Size - 1)
Density	Density	% of possible ties that are actually present, i.e., Ties/Pairs
WeakComp	WeakComp	# weak components in neighborhood
nWeakComp	nWeakComp	# weak components normalized by size, i.e., WeakComp/Size
TwoStepReach	TwoStepReach	% nodes that are two steps away
ReachEfficency	ReachEfficency	Normalizes TwoStepReach by size, i.e., TwoStepReach/Size. High reach efficiency indicates that egoÕs primary contacts are influential in the network
Brokerage	Brokerage	# 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	nBrokerage	Brokerage normalized by the number of pairs, i.e., Brokerage/Pairs
EgoBetween	EgoBetween	% shortest paths between neighbors that pass through ego
nEgoBetween	nEgoBetween	Betweenness normalized by the size of the ego network
Structural metrics (descriptions adapted from Z & N [1]):		
EffSize	EffSize	# entities that are connected to an entity minus the average number of ties between these entities
Efficiency	Efficiency	Normalizes the effective size of a network to the total size of the network
Constraint	Constraint	Measures how strongly an entity is constrained by its neighbors
Hierarchy	Hierarchy	Measures how the constraint measure is distributed across neighbors. When most of the constraint comes from a single neighbor, the value for hierarchy is higher
Centrality metrics (computed each for incoming, outgoing, and undirected dependencies; descriptions adapted from Z & N [1]):		
Degree	Degree	# dependencies for an entity
nDegree	(none)	# dependencies for an entity normalized by number of entities
Closeness	Closeness	Sum of the lengths of the shortest paths from an entity (or to an entity) from all other entities
Reachability	dwReach	# entities that can be reached from a entity (or which can reach an entity)
Eigenvector	Eigenvector	assigns relative scores to all entities in the dependency graphs.
nEigenvector	(none)	assigns relative scores to all entities in the dependency graphs normalized by number of entities
Information	Information	Harmonic mean of the length of paths ending at an entity.
Betweenness	Betweenness	Measure for a entity on how many shortest paths between other entities it occurs
nBetweenness	(none)	Betweenness normalized by the number of entities

source file that the class was declared. Declaring two top-level classes A and B in one source file `A.java` and compiling it will result in two byte code files `A.class` and `B.class`. Thus, analyzing byte code to get object dependencies will result in class dependencies instead of file dependencies. To map classes back to their source files, we parsed the source code using the JDT framework to determine the file in which a class was declared in. Having class dependencies based on byte code analysis and knowing which classes were declared in which source files allowed us to merge the two data sets and then determine the dependencies that exist between the source files.

Once the dependencies between files were known, we computed several network metrics using the *UCINET tool* (same as Z & N). The list of metrics computed is largely the same as in the original study and they have been listed along with a short description in Table II. Due to paucity of space in this paper, we refer the reader to the original study to get a fuller description of the network metrics. Note that in rare cases, we were unable to compute the metrics using the tool. For instance the tool crashed when computing *information centrality* for Eclipse. Despite emails to the company for support, we received none and hence we excluded the metric in the Eclipse network data set.

IV. EXPERIMENTAL SETUPS

This section describes the three experimental setups used to answer the research questions presented in the introduction.

A. *Classification and regression*

Z & N built two types of prediction models: classification and regression. Classification models predict whether a software entity will have defects or not, i.e., classify them as *defect-prone* or *not defect-prone*. The regression models, on the other hand, predict the number of defects expected be found in the software entity. Considering the wide scope of our replication, we restricted ourselves to conducting only classification in this paper. We believe that the general trend in performance of code and network metrics at predicting defects will emerge already using the classification models. In the future, we plan to extend our replication to include regression models too.

B. *Stratified repeated holdout setup (RQ1)*

In order to perform the experiments, the data available from the projects must be sampled into two sets — training and test sets. The former set is to be used to train the prediction models and fine tune them; the latter set is set aside to evaluate the performance of the trained prediction model.

Several possible experimental setups are available to sample the data. Zimmermann and Nagappan [1] used a setup called *random repeated holdout setup* where a specified proportion of the data (66% in their case) is randomly sampled for training the model and the remainder is set aside for testing the model. They repeatedly sampled their data 300 times in order to generate 300 independent training and test sets, and 300 prediction models to evaluate the metrics. The advantage of repeated random sampling is that it reduces bias—a single "lucky" (or "unlucky") sample may lead to a good (or bad) result.

We adopted a similar setup called *stratified repeated holdout setup* to sample our data. In this setup the data is sampled to preserve the proportion of positive and negative instances in the data in both training and test sets. Such a split improves the representation of each type of instance in the sets and is known to reduce sampling error. To exemplify, suppose that 30% of files in a project were known to have defects, then the data was sampled such that files with defects comprised 30% of the data in both the training and the test set.

Similar to Z & N, we sampled our data 300 times and used 66% of the sampled data for training the models and the remaining data for testing. Note that the data from a single project and release is used in this experimental setup. Hence, for each project and release, we sampled the data separately and trained and evaluated the models.

C. *Forward-release prediction setup (RQ2)*

Forward prediction setup requires trained data from an older release of a project to be evaluated on the next immediate release. For instance, in the case of the Eclipse project, data from release 2.1 will be trained to predict defect-prone entities in release 3.0; likewise data from JRuby 1.0 will be used to predict defects in JRuby 1.1 and so on. This setup is closest to what can be deployed in the real world where past project data is used to identify defect-prone entities in on-going or future releases. It has been previously applied in several papers [4], [25]. Note that this setup was not used in the original study because the authors had access to data from only one release of Windows Server 2003.

D. *Cross-project prediction setup (RQ3)*

The cross prediction setup entails using data from a release of one project to identify defect-prone entities in a release from another project. The rationale behind evaluating this setup is to verify whether defect prediction models are transferable from one project to another. If the results are promising, it will suggest that projects with little or no data from the past can leverage data from other projects for prediction purposes. This setup has been previously used in [5] and again, it has not been used in the original study.

V. Experimental steps (to reproduce)

With the availability of large data sets in software engineering and access to sophisticated prediction modelling tools, quantitatively intensive empirical investigations are becoming increasingly complex to validate and reproduce [26]. Similar observations have been made in other research disciplines too [27]. Primary reasons cited for this inability to validate and reproduce the experiments are that the data and/or tools and scripts are not made publicly available.

In this section, we present the steps taken to execute our experiments. Importantly, we complement the steps with the statistical packages (and functions) used to perform the experiments so as to allow independent validation and the reproduction of our work, if needed. To support reproduction and replication of this paper, our data and scripts have been made available on the *PROMISE* website.

We conducted our experiments using *R statistical software* [28], which is a popular open-source tool used amongst statisticians and machine learners. Several user-contributed packages exist to perform various modelling tasks — we chose Max Kuhn's R package *caret* [29] for our experiments because it provides a wrapper function to several machine learning algorithms available in other packages, thus providing us access to a multitude of models and keeping our code smaller. Where applicable, we indicate the name of the function used from the caret package at the beginning of the step below.

For the sake of brevity, we present the steps below used to perform classification of files as defect-prone or not defect-prone (Section IV-A) using the stratified repeated hold-out setup (Section IV-B). These steps can be easily adapted to conform to other experimental setups presented earlier in the paper or even regression. The steps were executed for each project and each of the three metrics sets to arrive at the results presented in the following sections of the paper.

Step 1: `createDataPartition()`: Generate 300 training and test set from the data using stratified sampling (note that the following steps were run on each pair of training/test set).

Step 2: `nearZeroVar()`: Independently assess and remove numerical input attributes from the training set that have near zero variance (essentially a single value) to avoid any undue influence on the models. The same attributes were then removed from the test set.

Step 3: `findCorrelation()`: Remove input attributes from the training set that correlate with other attributes with $\rho > .90$ to avoid any undue influence on the models. The attributes that were correlated highly with more number of columns were picked to be removed. The same attributes were then removed from the test set.

Step 4: `preProcess()` and `predict()`: Rescale the training data using the *center* and *rescale* to minimize the effect of large values on the prediction model. We additionally experimented with performing principal component analysis on our data (similar to Z & N), but this often led to inferior results. Hence we restricted ourselves to normalize our data by centering and rescaling it. The corresponding test data was centered and rescaled accordingly using the `predict()` function.

Step 5: `train()` We used several prediction models for our experiments. These are listed in Table III. Each model offers one or more parameters that can be tuned to optimize performance. This is internally handled by the `train()` function when the number of values (`tuneLength`) to validate is specified. We set this number to 5.

Step 6: `extractPrediction()` Each trained model was evaluated against the test data. The evaluation measures that we computed include *precision*, *recall*, and *F-measure*.

VI. Stratified repeated holdout results (RQ1)

Results from the stratified repeated holdout experimental setup (see Section IV-B) are presented in Figure 2 (plotted using R package `ggplot2` [31]). Panels across the x-axis in the figure represent the subject projects. The six prediction models listed in Table III were run on 300 stratified random samples on the three metrics sets: code, network, and all metrics for each project. Precision, recall, and F-measure values from each run were recorded; their distributions have been plotted as boxplots in the figure. Note that for a given project and metrics set, we plotted values from the model that gave the best result (i.e., the highest average of the 300 values). The corresponding best performing model is indicated at the bottom of each plot in parenthesis. To exemplify, in the case of JRuby 1.0, *rpart* gave the highest average F-measure using code metrics, while *nb* and *treebag* performed best when using network and all metrics respectively. Similarly, *svmRadial* gave the highest average precision values using all three metrics sets for JRuby 1.0.

The black line in the middle of each boxplot indicates the median value of the distribution; the mean is plotted as a red dot on the boxplot. Larger median and mean values indicate better performance of the metrics set for the project based on the respective evaluation measure. Note that the red coloured horizontal lines connecting the means across the boxplots do not have any statistical meaning — they have been added to aid visual comparison of the performance of the metrics set. An upward sloping horizontal line between two boxplots indicates that the metrics set on the right performs better than the one of the left and vice versa.

Additionally, we performed a non-parametric statistical test (Kruskal-Wallis) to statistically compare the results from the use of two pairs of metrics sets: (a) code vs. network and (b) network vs. combined. For each set, we used the results from the best performing prediction model (same as in Figure 2).

Observing Figure 2, it is apparent that results obtained from using code metrics are inferior to those from using network metrics. In fact, all statistical tests performed to compare results from the two metrics sets showed a statistically significant difference ($p < .0001$), thus confirming that network metrics perform better than code metrics when using the stratified holdout setup, irrespective of the project or evaluation measure. Mean values of some evaluation measures using network metrics were notably high: precision

Table III
List of models used for experimentation

Model*	**Description**
k-nearest neighbour (*knn*)	*This model finds k training instances closest in Euclidean distance to the given test instance and predicts the class that is the majority amongst these training instances.*
Logistic regression (*multinom*)	*This is a generalized linear model using a logit function and hence suited for binomial regression, i.e. where the outcome class is dichotomous.*
Naïve Bayes (*nb*)	*Applying Bayes' theorem, this is a simple probabilistic classifier assuming strong independence.*
Recursive partitioning (*rpart*)	*A variant of decision trees, this model can be represented as a binomial tree and popularly used for classification tasks.*
Support vector machines (*svmRadial*)	*This model classifies data by determining a separator that distinguishes the data with the largest margin. We used the radial kernel for our experiments.*
Tree Bagging (*treebag*)	*Another variant of decision trees, this model uses bootstrapping to stabilize the decision trees.*

* For a fuller understanding of these models, we advise the reader to refer to specialized machine learning texts such as by Wittig and Frank [30].

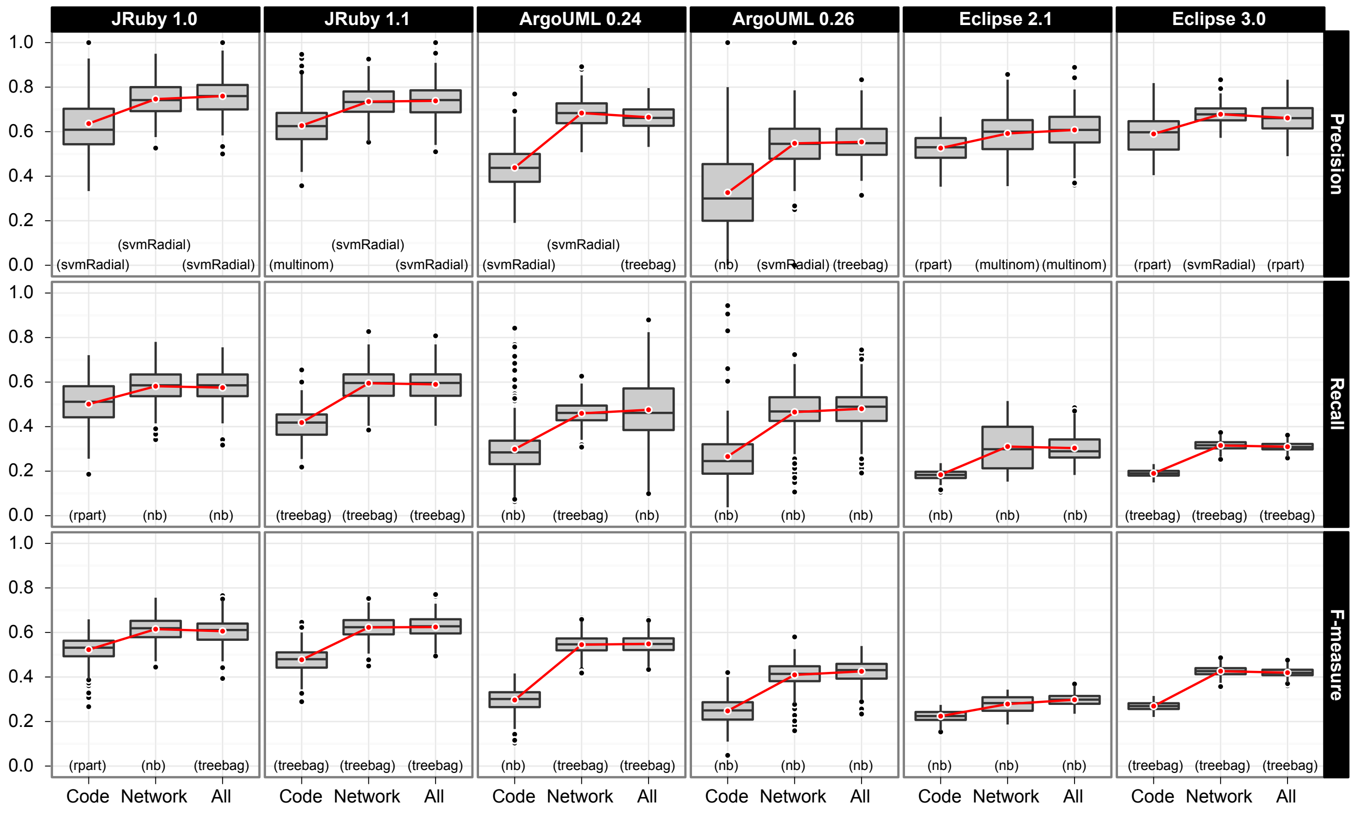

Figure 2. Results from the repeated holdout experimental setup. Note that the 'All' label refers to the combined metrics.

close to 0.8 for both releases of JRuby and approximately 0.6 for both releases of Eclipse. Recall values for JRuby and ArgoUML also hovered around 0.6, but were relatively lower for Eclipse.

Using combined metrics, on the other hand, gave no advantage over using network metrics. In only 4 of the 18 cases in Figure 2, results from using combined metrics performed statistically better than using network metrics alone ($p < .05$); these cases include precision for JRuby 1.0, F-measure for ArgoUML 0.26, and precision and F-measure for Eclipse 2.1. It is important to note that the means differed marginally in these cases. In all other cases, network metrics either outperformed using all metrics or performed at least as well (confirmed using statistical testing). The flat slopes of the lines connecting the mean values of results from network and all metrics also visually support this result. We draw from these results that using solely network metrics is a better option because the results are as good (or even better) than using combined metrics and the time to build these models is shorter because of smaller data sets.

Another noteworthy observation in Figure 2 is that different models give best results for different projects, metrics sets, and evaluation measures. Currently we cannot explain what causes such marked differences in performance across models, but this is a necessary area of future research, i.e., how to determine which model amongst a set of candidates is likely to perform best on a given data set.

☛ *Network metrics outperform code metrics at predicting defects when using the stratified holdout setup.*
☛ *Using all metrics together offers no improvement in prediction accuracy over using network metrics alone.*
☛ *Generally, higher accuracy can be observed for the smaller projects in comparison to ECLIPSE.*

VII. Forward-release prediction results (RQ2)

A rather different picture can be seen in Figure 3 which presents the results from the forward prediction setup (Section IV-C). Each panel in the figure presents the value of the evaluation measures derived from training the prediction model on an older release of the project to predict defects in the newer release.

In the case of JRuby, precision values from using all three metrics sets are similar. But some differences can be seen in recall values where network metrics outperform other metrics sets. Overall, however, it appears that both code and network metrics can be used to predict defects with comparable performance in the case of JRuby. We performed an ANOVA statistical test to independently compare the precision, recall, and f-measure values derived from the prediction models across the three data sets. The tests

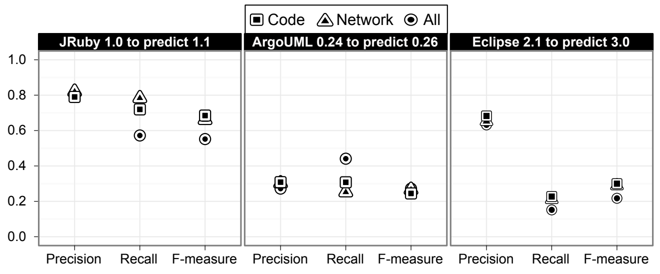

Figure 3. Results from forward prediction

showed no significant differences, in that the results from using the three metrics sets are comparable.

Similar inferences can be drawn for both ArgoUML and Eclipse. Minor differences can be seen in the precision, recall, and F-measure values using the different metrics sets. The only exception to this are the recall values in ArgoUML where using all metrics performed notably better than the other sets. The same test, as for JRuby, was performed for these two projects; no statistically significant differences were to be found.

Recall that this setup would be the most realistic scenario if defect predictions models were to be applied in an industrial setting, in that data and records about defects from past project releases are used to predict defects in the current release. Our results suggest that using either code or network metrics would deliver comparable results, however there may be a slight preference for code metrics because code metrics are easier to collect, they are fewer in number as compared to network metrics, and as a result, the time required to train the models with them is shorter.

- ☛ *All three metrics sets appear to have comparably prediction accuracy.*
- ☛ *Using code metrics may be best given that the time to train the prediction models is substantially lower than the other sets because of fewer attributes in the data set and they are easier to collect.*

VIII. Cross-project prediction results (RQ3)

The results from our last experimental setup, i.e., predicting defects across different projects, is presented in Figure 4. The panels across the x-axis indicate the project used to train the prediction models, while the panels across the y-axis indicate the projects on which the models were evaluated. For brevity sake, we only used the latest releases of the projects to conduct these experiments.

Generally, results from using code and network metrics can be again seen to be comparable. But in select cases, there are noteworthy differences. For instance, the precision value derived using network data from JRuby 1.1 to predict defects in Eclipse 3.0 is substantially lower than that from code metrics, but recall and F-measure are similar. On the

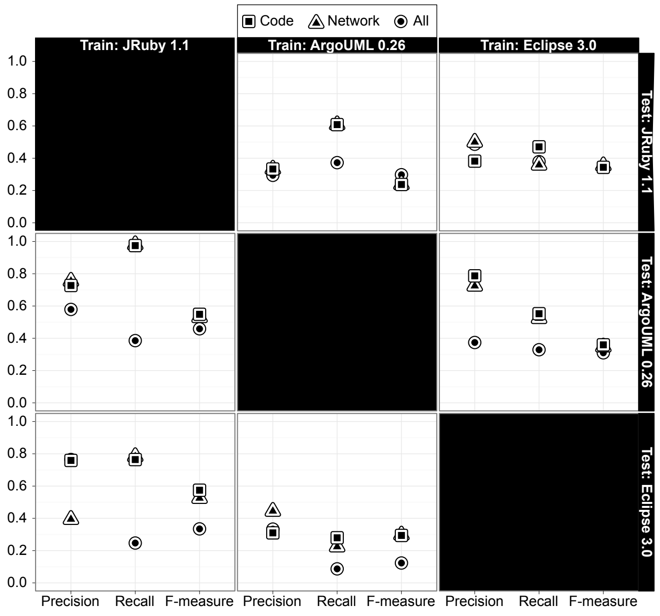

Figure 4. Results from cross project prediction

contrary, using network data outperforms code data when using ArgoUML to predict defects in Eclipse.

Surprisingly, using combined metrics are often seen to lead to poorer results than using code or network metrics alone. This suggests that using more number of metrics to a data set should be done with caution after careful investigation of their influence on the intended outcome.

Similar to the forward-release prediction setup, we used ANOVA to test for statistical differences between the precision, recall, and f-measure values across the three metrics sets. We found no statistical differences, except in the case of using Eclipse to predict defects in JRuby— code metrics gave significantly lower F-measures in comparison to the other metrics sets. Hence, we observed no added advatage of using one metrics set over the other. Note that this inference is different from what we draw from Figure 4 because the latter plots only the best accuracy value from across the six models used while the statistical tests takes into account values from all models.

- ☛ *Statistically, all three metrics sets appear to have comparably prediction accuracy, but when we select and compare results using the best accuracy values, code and network metrics sets appear to have an advantage over combined metrics.*
- ☛ *Using code metrics may be best given that the time to train the prediction models is substantially lower than the other sets because of fewer attributes in the data set and they are easier to collect.*

IX. Influential metrics

The R package *caret* allows computing the importance of individual metrics using the `filterVarImp` function. The function computes a ROC curve by first applying a series of cutoffs for each metric and then computing the sensitivity and specificity for each cutoff point. The importance of the metric is then determined by computing the area under the ROC curve.

We used the combined metrics set from each project release to compute variable importance so as to be able to compare the importance of code and network metrics together. We considered the top-10 most influential metrics for each metrics set for examination. The results show that all top-10 most influential metrics were network metrics for 3 out of the 6 metrics sets. Code metrics figured in the list only for JRuby 1.0 (*CountDeclFunction*), ArgoUML 0.26 (*CountDeclFunction*) and Eclipse 2.1 (*CountLineCode, CountDeclFunction*). For these projects, the remaining list comprised of network metrics only, thus suggesting their general dominance.

No patterns with respect to the presence or ranking of network metrics were observed in the influential metrics across the projects. For instance, while a network metric is found to be most influential in one project, it may not even be present in the top-10 list for another project. A more detailed analysis to investigate the underlying reasons for this is beyond the scope of this paper, however we intend to examine this matter in our future work.

X. Discussion

Our results portray a mixed picture of the value of network metrics over code metrics at predicting defect-prone entities. In the stratified random sampling method (similar to Z & N), network metrics undoubtedly perform much better than code metrics (see Figure 2). This result is in concordance with the original study. In fact, the difference in accuracy between the two metrics sets is more pronounced in our results in comparison to Z & N— network metrics in our study increased prediction accuracy in the order of 20–25%; this increase was approximately 10% in the original study.

However, results from using the cross-release and cross-prediction setups show no added value of using network metrics, i.e., the performance was comparable to code metrics. Better performance in these setups is more crucial because they are more likely to be applied for prediction in industry. Moreover, our experience draws us to consider code metrics preferable because they are easier to collect, fewer in numbers, and faster to train prediction models with. An important point to note is that in order to closely align our study with Z & N, we used a similar (and small) set of code metrics for our study. However, previous studies (e.g. [4]) have used a larger set of code metrics for defect prediction. Using such an elaborate set of code metrics in our study may have affected our results; however this is only a speculation and remains to be verified.

Tosun et al. [21] noted in their replication that network metrics do not perform well for small sized projects. Our results contradict their conclusions; we observed that the network metrics performed especially well for our smaller projects in comparison to Eclipse. We cannot compare our results with Bird et al. [8] who used a similar cross-project prediction setup as ours to predict defects in Eclipse, however they did so at the plug-in-level. We predicted defects at the file level which is a finer granularity. We intently chose this level because it pin-points which files to test first; predicting defects at higher granularities increases recall, but also lowers the worth of the prediction to some extent.

A noteworthy observation from our results is that gauging the performance of a prediction model depends upon the choice of evaluation measure. Hence the decision to choose one prediction model as most desirable to use is not an easy one. In the future, perhaps using an ensemble of models to predict defects based on say, majority voting, may be a viable alternative than using strictly one model.

XI. Threats to Validity

Like any other empirical study of this kind, ours too has threats to validity. We identified two noteworthy threats. First and foremost, we had to apply heuristics to compute the number of post-release bugs in a file. While we applied the same technique as other contemporary studies to do so, there is a chance that the count of bugs for some files may be an approximation. Second, we restricted the number of code metrics used in our study to align it closely with Z & N. However, using a larger set of code metrics for comparison may impact our results, but it is a speculation at this stage.

XII. Conclusions and Consequences

In this paper, we have presented a replication study based on the work by Zimmermann and Naggappan to investigate whether social network metrics computed on source code can deliver better accuracy in identifying defect-prone entities than traditionally used source code metrics. We observe mixed results in that using a setup similar to Zimmermann and Naggappan to predict defects, we see a clear advantage of using network metrics. However, when the setup was changed to suit usage in an industrial environment, we observed no added value in using network metrics over code metrics. In fact, considering that code metrics are easier to collect and fewer in number, they may be even preferable to network metrics.

Given this sharp contrast in results across the different experimental setups, it is clear that more investigation is needed in this area to comment on the generality of the findings of the original and our study. A serious challenge to this end is to be able to align the studies to allow a close replication. We hope that the open manner in which we share our data and scripts on the PROMISE website will foster such replication in the future by independent researchers.

References

[1] T. Zimmermann and N. Nagappan, “Predicting defects using network analysis on dependency graphs,” in *Procs. of the Int. Conf. on Software Engineering*. ACM, May 2008, pp. 531–540.

[2] S. McConnell, *Code Complete*, 2nd ed. Microsoft Press, 2004.

[3] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, “Where the bugs are,” in *Procs. of the Int. Symposium on Software Testing and Analysis*. ACM, July 2004, pp. 86–96.

[4] T. Zimmermann, R. Premraj, and A. Zeller, “Predicting defects for Eclipse,” in *Procs. of the Workshop on Predictor Models in Software Engineering*, May 2007.

[5] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, “Cross-project defect prediction,” in *Procs. of the European Software Engineering Conf. and the Symposium on the Foundations of Software Engineering*. ACM, 2009, pp. 91–100.

[6] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” in *Procs. of the Int. Conf. on Software Engineering*. ACM, 2005, pp. 284–292.

[7] M. Pinzger, N. Nagappan, and B. Murphy, “Can developer-module networks predict failures?” in *Procs. of the Int. Symposium on Foundations of Software Engineering*. ACM, 2008, pp. 2–12.

[8] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy, “Putting it all together: Using socio-technical networks to predict failures,” in *Procs. of the Int. Symposium on Software Reliability*, 2009.

[9] A. Brooks, M. Roper, M. Wood, J. Daly, and J. Miller, “Replication’s role in software engineering,” in *Guide to Advanced Empirical Software Engineering*. Springer, 2008, pp. 365–379.

[10] F. J. Shull, J. C. Carver, S. Vegas, and N. Juristo, “The role of replictions in empirical software engineering,” *Empirical Software Engineering*, vol. 13, no. 2, pp. 211–218, April 2008.

[11] V. R. Basili, F. Shull, and F. Lanubile, “Building knowledge through families of experiments,” *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 456–473, 1999.

[12] V. R. Basili, L. C. Briand, and W. L. Melo, “A validation of object-oriented design metrics as quality indicators,” *IEEE Trans. Softw. Eng.*, vol. 22, no. 10, pp. 751–761, 1996.

[13] R. Subramanyam and M. S. Krishnan, “Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects,” *IEEE Trans. Softw. Eng.*, vol. 29, no. 4, pp. 297–310, 2003.

[14] N. Nagappan, T. Ball, and A. Zeller, “Mining metrics to predict component failures,” in *Procs. of the Int. Conf. on Software Engineering*. ACM, 2006, pp. 452–461.

[15] R. Moser, W. Pedrycz, and G. Succi, “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction,” in *Procs. of the Int. Conf. on Software Engineering*. ACM, 2008, pp. 181–190.

[16] A. E. Hassan, “Predicting faults using the complexity of code changes,” in *Procs. of the Int. Conf. on Software Engineering*. IEEE Computer Society, 2009, pp. 78–88.

[17] A. Schröter, T. Zimmermann, and A. Zeller, “Predicting component failures at design time,” in *Procs. of the Int. Symposium on Empirical Software Engineering*. ACM, 2006, pp. 18–27.

[18] Y. Shin, R. Bell, T. Ostrand, and E. Weyuker, “Does calling structure information improve the accuracy of fault prediction?” *Procs. of the Int. Workshop on Mining Software Repositories,*, pp. 61–70, 2009.

[19] N. Nagappan and T. Ball, “Using software dependencies and churn metrics to predict field failures: An empirical case study,” in *Procs. of the Int. Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, 2007, pp. 364–373.

[20] T. Zimmermann, N. Nagappan, K. Herzig, R. Premraj, and L. Williams, “An empirical sudy on the relation between dependency neighborhoods and failures,” in *Procs. of the Int. Conf. on Software Testing, Verification and Validation*. IEEE, 2001.

[21] A. Tosun, B. Turhan, and A. Bener, “Validation of network measures as indicators of defective modules in software systems,” in *Procs. of the Int. Conf. on Predictor Models in Software Engineering*. ACM, 2009, pp. 1–9.

[22] J. Ratzinger, T. Sigmund, and H. C. Gall, “On the relation of refactorings and software defect prediction,” in *Procs. of the Int. Working Conf. on Mining Software Repositories*. ACM, 2008, pp. 35–38.

[23] S. Borgatti, *Social Network Analysis*, AnalyticTech. [Online]. Available: http://www.analytictech.com

[24] B. Dagenais and L. Hendren, “Enabling static analysis for partial Java programs,” in *Procs. of the Conf. on Object-oriented Programming, Systems, Languages and Applications*. ACM, 2008, pp. 313–328.

[25] T. Holschuh, M. Paeuser, K. Herzig, T. Zimmermann, R. Premraj, and A. Zeller, “Predicting defects in sap java code: An experience report,” in *Procs. of the Int. Conf. on Software Engineering*, May 2009.

[26] G. Robles, “Replicating MSR: A study of the potential replicability of papers published in the mining software repositories proceedings,” in *Procs. of the Working Conf. on Mining Software Repositories*, May 2010, pp. 171 –180.

[27] T. Hothorn and F. Leisch, “Case studies in reproducibility,” *Briefings in Bioinformatics*, 2011.

[28] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, 2010.

[29] M. Kuhn, *caret: Classification and Regression Training*, 2011, R package version 4.76.

[30] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, 2nd ed., J. Gray, Ed. Morgan Kaufmann, 2005.

[31] H. Wickham, *ggplot2: elegant graphics for data analysis*. Springer, 2009.