Information and Software Technology 69 (2016) 50–70

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Empirical analysis of network measures for effort-aware fault-proneness prediction

Wanwangying Ma, Lin Chen*, Yibiao Yang, Yuming Zhou, Baowen Xu*

State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

ARTICLE INFO

Article history:
Received 4 March 2015
Revised 30 August 2015
Accepted 4 September 2015
Available online 14 September 2015

Keywords:
Network measures
Dependency relationships
Fault-proneness prediction
Effort-aware

ABSTRACT

Context: Recently, network measures have been proposed to predict fault-prone modules. Leveraging the dependency relationships between software entities, network measures describe the structural features of software systems. However, there is no consensus about their effectiveness for fault-proneness prediction. Specifically, the predictive ability of network measures in effort-aware context has not been addressed.
Objective: We aim to provide a comprehensive evaluation on the predictive effectiveness of network measures with the effort needed to inspect the code taken into consideration.
Method: We first constructed software source code networks of 11 open-source projects by extracting the data and call dependencies between modules. We then employed univariate logistic regression to investigate how each single network measure was correlated with fault-proneness. Finally, we built multivariate prediction models to examine the usefulness of network measures under three prediction settings: cross-validation, across-release, and inter-project predictions. In particular, we used the effort-aware performance indicators to compare their predictive ability against the commonly used code metrics in both ranking and classification scenarios.
Results: Based on the 11 open-source software systems, our results show that: (1) most network measures are significantly positively related to fault-proneness; (2) the performance of network measures varies under different prediction settings; (3) network measures have inconsistent effects on various projects.
Conclusion: Network measures are of practical value in the context of effort-aware fault-proneness prediction, but researchers and practitioners should be careful of choosing whether and when to use network measures in practice.

1. Introduction

With the growing size and complexity of modern software, it is difficult and costly to test the source codes and produce high-quality software products. One cost-effective way to deal with this problem is to allocate the limited testing resources on higher risk modules that are more prone to faults before delivery. In the last decades, various kinds of metrics have been proposed to build prediction models to prioritize modules in terms of their probability of having a fault or the number of expected faults. Among these metrics, network measures derived using concepts from the Social Network Analysis field have drawn the attention of many researchers recently [1–6]. Network-based analysis treats the modules as nodes and extracts the dependencies between modules as edges to construct the software source code network. Then, the network measures obtained from it are used to build prediction models. Taking into account the interactions between modules, network measures characterize the information flow and the overall topology of the software system, which cannot be captured by code metrics.

There have been a few studies focusing on the predictive effectiveness of network measures in recent years. However, to the best of our knowledge, studies on this field do not show a conclusive picture so far. Zimmermann and Nagappan [2] first found that network measures significantly outperformed code metrics at predicting defects in Windows Sever 2003. Tosun et al. [3] argued that network measures provided no better predictive power on small-scale projects. In addition, Premraj and Herzig [4] revealed that network measures were not superior to code metrics in terms of forward-release and inter-project predictions. With the mixed results, developers and researchers have been confused about whether, when, and how they should use network measures, especially considering the extra cost involved in data collection and model training. Therefore, in this study, we aim to make a thorough investigation into the actual usefulness of network measures in effort-aware fault-proneness prediction with more subject projects and more experimental scenarios, as well as provide some practically useful information for software

* Corresponding authors. Tel.: +86 13813825166 (Lin Chen).
E-mail addresses: lchen@nju.edu.cn (L. Chen), bwxu@nju.edu.cn (B. Xu).

http://dx.doi.org/10.1016/j.infsof.2015.09.001
0950-5849/

practitioners. In this context, if at least one fault is detected in a module, the module is considered to be fault-prone and otherwise not fault-prone.

For this purpose, we conducted our study on a total number of 39 releases of 11 open-source projects: Mozilla Firefox, Eclipse, JEdit, and eight projects under Apache Foundation. First, by extracting the data dependencies and call dependencies between modules (files or classes), we constructed the source code networks at module level. Then, we empirically investigated most of the network measures proposed in the literature to analyze the correlations between individual measures and fault-proneness by using the univariate logistic regression. Finally, with the most commonly used code metrics as the baseline, we built multivariate prediction models to evaluate the effectiveness of network measures when used alone or used together with the baseline code metrics in effort-aware fault-proneness predictions.

By conducting large-scale experiments on publically available data sets, this study highlights the following three aspects and contributes to the empirical software engineering body. First, we evaluated the usefulness of network measures in effort-aware context, where the effort needed to review or test the source codes is taken into account, leading to a more realistic performance evaluations. As far as we know, no previous studies assessed the predictive ability of network measures considering effort. Second, in order to simulate actual prediction scenarios and obtain comprehensive performance evaluations, we performed our experiments under two prediction scenarios (ranking and classification) and three prediction settings (cross-validation, across-release prediction, and inter-project prediction). Third, we conducted our study on a large set of 11 subject projects varying in software size, programming language, and application domain, which makes our findings more generalizable to other software systems and reliable for practitioners to get implications and enlightenment from.

Based on the 11 projects, our results show that: (1) most network measures have a significant correlation with fault-proneness; (2) network measures produce a statistically significant improvement of the effectiveness over code metrics for most projects in cross-validation; (3) network measures in general outperform code metrics in across-release prediction; (4) network measures are not superior to code metrics when performing inter-project prediction. Our study provides valuable data for researchers and practitioners to better understand the practical usefulness of network measures. We believe that these experimental results can help the practitioners decide whether it is worth spending relatively more time in collecting network measures for fault-proneness prediction.

The rest of this paper is organized as follows. Section 2 provides the related work. Section 3 describes our study design, including the research questions, the data sources, the methods we used to collect the experimental data sets, the construction of the source code network, and the network measures together with the most commonly used code metrics that we investigated. Section 4 introduces the dependent and independent variables, presents the employed modeling technique, and describes the data analysis methods. Section 5 reports the experimental results, answers our research questions, and makes some discussion. Section 6 examines the threats to validity of our study. Section 7 concludes this paper.

2. Related work

This section discusses the existing studies focusing on fault-proneness predictive ability of network measures.

Zimmermann and Nagappan [2] proposed that network measures on the dependency relationships between binaries of Window Server 2003 were able to predict fault occurrence and number. They employed automated tools to build dependency graphs between binaries and extract network measures. They revealed that network measures could identify 10% more fault-prone binaries than complexity metrics. Their results are promising since their technique is one of the few that have helped improve the performance of product metrics.

Later, Tosun et al. [3] reproduced their work on three small-scale embedded software systems and Eclipse. They further extended it by proposing a learning-based model incorporated with a Call Graph Based Ranking Framework to predict the fault-proneness of functions and source files. Their results revealed that network measures were effective indicators of defective modules for large and complex systems, but did not provide significant predictive power on small-scale projects.

Premraj and Herzig [4] replicated and extended the study conducted by Zimmermann and Nagappan. They explored the usefulness of network measures for fault prediction on three open-source projects of different sizes, i.e., JRuby, ArgoUML, and Eclipse. Compared with code metrics, they investigated whether network measures could achieve better prediction accuracy under cross-validation, forward-release prediction, and inter-project prediction. Their results showed that though network measures outperformed code metrics in cross-validation, they provided no advantage for forward-release and inter-project predictions.

Nguyen et al. [5] validated the generalization of Zimmermann and Nagappan's conclusion to Eclipse. They also explored the rationale behind the better performance using network measures and selected three kinds of network measures with the highest impact on fault prediction. Additionally, they demonstrated that predicting bugs at class level were more useful than at higher level concerning the effort involved.

Prateek et al. [6] further compared the fault-proneness prediction performance of network measures against code metrics. They employed Logistic Regression, Support Vector Machines, and Random Forests to evaluate the effectiveness in three different scenarios, namely, cross-validation, within-project, and cross-project predictions. By conducting their experiments on 11 small-scale projects from the *Promise* repository, they found that network measures were inferior to code metrics for a majority of the analyzed projects under all prediction setups. Moreover, they investigated the most influential metrics on prediction performance for each project. They found that except *OutDegree*, the other network measures were not as prevalent in top 10 as code metrics.

Combining the strong points of [2–6], we performed a more in-depth and more comprehensive investigation on the predictive effectiveness of network measures on a large number of projects with multiple releases. Table 1 summarizes these studies together with our work in terms of the investigated projects, granularity level of the source code network, the prediction scenarios, the prediction settings, and the evaluation criteria.

Compared with the previous studies, the main difference of our work lies in the following two aspects. First, we evaluated the ability of network measures in the context of the effort-aware fault-proneness prediction, while the four studies did not take into account the effort involved in inspecting codes. Note that Nguyen et al. [5] compared the practical significance of class and package level prediction considering the effort, but they did not evaluate the model performance using the effort-aware performance indicators. Second, apart from classifying the predicted modules into two categories, i.e., fault-prone and not fault-prone, we ranked them according to their predicted probabilities of fault-proneness. We consider that the ranking ability of a prediction model is of more practical significant, since software engineers can select as many modules as the testing resources available from top of the ranking list. The four studies did not investigate the predictive effectiveness of network measures under the ranking scenario. With these unique experimental design, this

Table 1
A comparison of the related work.

		Zimmermann and Nagappan [2]	Tosun et al. [3]	Nguyen et al. [5]	Premraj and Herzig [4]	Prateek et al. [6]	This study
Studied projects	Large-scale	Window Server 2003	Eclipse(2)	Eclipse(1)	Eclipse(2)	–	Eclipse(5), Firefox(6)
	Small-scale	–	Three closed source C++ projects(2)	–	JRuby(2), ArgoUML(2)	Eleven open-source Java projects(48)	Nine open-source Java projects(28)
Granularity level		Binary	Function and file	Class and package	File	Class	Class or file
Prediction scenarios				Classification			Classification and ranking
Prediction settings			Cross-validation		Cross-validation, across-release, and inter-project predictions		
Evaluation criteria			Traditional binary classification evaluation criteria				Effort-aware performance indicators

research extends and complements the previous ones, as well as provides further empirical evidences for the practical usefulness of network measures.

3. Study design

In this section, we first present the research questions of our study. Then, we introduce the subject projects, the way that we collected and processed fault data, the construction of source code network, and the network measures as well as the baseline code metrics that we used.

3.1. Research questions

The goal of this study is to provide empirical evidence for the effectiveness of network measures in effort-aware fault-proneness prediction. Particularly, we seek to answer the following research questions:

RQ1: Is there a significant correlation between each network measure and fault-proneness?
RQ2: Are network measures effective in effort-aware fault-proneness prediction within the same release of a project?
RQ3: Are network measures effective in effort-aware fault-proneness prediction within a project?
RQ4: Are network measures effective in effort-aware fault-proneness prediction across different projects?

By analyzing the effects of individual network measures on fault-proneness, the purpose of RQ1 is to determine whether each of the network measures is a potentially useful fault-proneness predictor. Then these useful network measures are combined to build the prediction models. RQ2, RQ3, and RQ4 are set to evaluate their predictive power in various cases: cross-validation (RQ2), across-release prediction (RQ3), and inter-project prediction (RQ4). Since there are no agreed standards to assess the predictive ability, we adopt the code metrics, which are commonly used in fault-proneness predictions [7–9], as a baseline. We believe that network measures are effective only if: (1) they have a significantly better fault-proneness predictive ability than the baseline code metrics; or (2) they can significantly improve the performance of fault-proneness prediction when used together with the baseline code metrics. This is especially true when considering the expense for collecting network measures. In RQ2 to RQ4, we make in-depth comparisons between network measures and traditional code metrics.

3.2. Studied projects

We conducted our study on 11 well-known open-source projects: Firefox,[1] Eclipse,[2] JEdit,[3] Ant,[4] Camel,[5] Ivy,[6] Poi,[7] Lucene,[8] Xalan-Java,[9] Xerces,[10] and Tomcat.[11] Three categories of data were needed in this work, i.e., fault data, network measures, and code metrics (detailed in Sections 3.3, 3.5, and 3.6, respectively). Except Firefox and Eclipse, the fault data and code metrics of the other nine projects were obtained from the *Promise*[12] repository. The *Promise* data sets have been widely used in previous studies involving fault-proneness prediction [10–14], and thus are convincing data sources with high accuracy and reliability. However, the *Promise* data still have several limitations: 1) for most of the nine projects, only a few (no more than 4) releases of data have been shared on the repository, which are not adequate to perform across-release prediction; 2) the nine projects are all written in a single kind of programming language, i.e., Java; and 3) they are all small applications in size. All these limit the generalization of our study.

In order to overcome the inadequacies in the number of releases, kinds of programming languages, and range of project size, we selected Firefox and Eclipse, two popular large-scale projects commonly investigated in the literature, to complement the *Promise* data sets. With six releases of Firefox, five releases of Eclipse, and a total number of 28 releases of the nine projects on *Promise*, this work was conducted on an abundant set of subject software systems varying in application domain, size, and programming language. Firefox is a popular web browser developed by the Mozilla Foundation. Eclipse is a famous integrated development environment under Eclipse Foundation. JEdit is a mature programmer's text editor. The last eight projects are parts of the Apache Software Foundation with various functionalities. With respect to the software size, Firefox and Eclipse are large-scale projects, while the others are relatively small. Moreover, apart from Firefox which is mainly written in C/C++, the others are all Java applications.

[1] http://www.firefox.com/.
[2] https://www.eclipse.org/.
[3] http://www.jedit.org/.
[4] http://ant.apache.org/.
[5] http://camel.apache.org.
[6] http://ant.apache.org/ivy/.
[7] http://poi.apache.org/.
[8] http://lucene.apache.org/.
[9] http://xalan.apache.org/.
[10] http://xerces.apache.org/.
[11] http://tomcat.apache.org/.
[12] http://openscience.us/repo/.

3.3. Fault data collection

The fault data of these studied projects were collected in two ways: (1) mining the source repositories by ourselves for Firefox and Eclipse; and (2) using the existing data sets posted on *Promise* repository for the other nine projects.

The *Promise* data sets used in this study were collected by Jureczko and Madeyski [15] with the help of *BugInfo*[13] tool, which analyzed the logs from versioning control system (i.e., SVN[14] or CVS[15]) and identified bug fix commits based on regular expression matching technology. A commit was interpreted as a bug fix when it solved an issue reported in the bug tracking system (i.e., Bugzilla[16]), then *BugInfo* incremented the fault count for all classes that had been modified in that commit. Each instance in the *Promise* data sets represented a class with the attribute labeled "bug" indicating the number of faults in the class.

Since the *Promise* repository does not provide the data for Eclipse and Firefox, we had to collect them by ourselves. For the sake of consistency of fault data collection, we gathered the fault information by using the same method adopted by *BugInfo*. This approach took the following four steps:

(1) Extracted the change logs with key words (e.g. "bug" for Firefox) found in commits from the versioning control system (i.e., GIT[17] or SVN);
(2) If a log contained one or more key words and those key words were followed by numbers, this log was treated as a potential bug fix, and those numbers were seen as bug IDs;
(3) Checked those bug IDs by linking them with the bug tracking system (i.e., Bugzilla). A log was a bug fix only if it met three requirements: (a) a bug report with one of its corresponding bug IDs was found in Bugzilla; (b) the *Severity* field of the bug report was not set to "ENHANCEMENT"; and (c) the *Resolution* field was set to "FIXED" or the *Status* field set to "CLOSED";
(4) Parsed the fault fix logs to get the list of files which were modified to fix the faults, and incremented the fault count for all those files.

We developed a Java program to automate this fault data collection process for each studied release of Firefox and Eclipse. To collect faults from a particular release, we applied this program to analyze the change logs committed between this release and its next release. Each of the identified faults was then associated with each source code file (for Firefox)/class (for Eclipse). Finally, we obtained the fault data sets with the same format as *Promise* repository, i.e. each file/class was an instance, and a "bug" attribute showed how many faults it had.

Fig. 7 (in Appendix A) shows the distribution of faults of the 11 investigated projects. The *x*-axis indicates the number of faults within a module, while the *y*-axis means the number of modules. It is obvious that the number of faults is not normally distributed across any of these systems. This distribution is to be expected given that the fault data are highly unbalanced, that is, the vast majority of modules are likely to contain no faults.

3.4. Construction of the source code network

We generated the source code network at module level, and treated a single file (for C/C++ projects) or a class (for Java projects) as a module. Each module was regarded as one node in the network and the dependencies between modules were extracted as edges. A program dependency is a directed relationship between two pieces of code [1]. We considered two kinds of dependencies: the data dependency from the use of a variable to its definition, and the call dependency from the sites where a function is called to its declaration. The two kinds of dependencies were then rolled up to the module level. Since Firefox is a C/C++ software, we combined each source (.cpp/.c) file with the corresponding head file (.h) as one node and merged their dependencies. Specifically, a source (.cpp/.c) file and a header (.h) file which have the same base name were treated as one node in the network, and any dependency relationship with that source/header file was seen as an edge from or to that node. Moreover, even though there exist self-dependencies, we disregarded them in this paper.

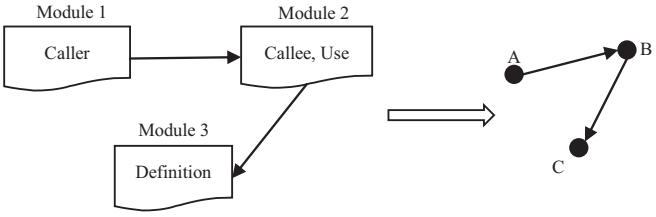


Fig. 1. A simple example of the construction of the source code network.

We used the program-understanding tool *Understand*[18] to build the source code network. For each studied release of the 11 projects, we first generated an *Understand* database which stored information about entities (such as files and classes) and references (such as function calls and variable references). We then developed a Perl script to track the dependency information between modules and generate the network of the software system at module level. Fig. 1 shows a simple example of the construction of the source code network.

Formally, we define a source code network as a directed graph $G = (V, E)$, where V is a set of nodes and $E \subseteq V \times V$ is a set of directed edges. An edge (A, B) means that *module A* depends on *module B*.

Table 2 lists some properties of these software systems involved in our study. The second column shows the number of releases we analyzed. The third column contains the main language(s) that the project is written in. Column *kLOC* indicates the project size in terms of the lines of code. Column *#modules(vertices)* lists the total number of modules (classes or files) in the project, which is also the number of vertices in the corresponding source code network. Column *#edges* shows the number of directed edges in the network. Column *#faults* tells the total number of faults found in the project.

3.5. Network measures

Each node in the source code network was evaluated from two perspectives: local and global. To view a given node locally, we considered **ego network** which consists of the node itself and every other nodes that directly connects to it. The ego network measures capture the importance of the node within its neighborhood. On the other hand, **global network** contains all the nodes and the global network measures weigh the importance of the nodes within the entire network. For each module in the software system, ego network only takes into account the direct dependencies between modules, while global network also comprises the indirect use of data and methods.

We employed *Ucinet*[19] tool to obtain ego and global network measures for all the 11 projects. Table 3 summarizes the network measures used in this paper. For each node, we computed three types of ego network: in (_in), out (_out), and undirected (_un). *In* (*out*) *ego network* only concerns incoming (outgoing) directed dependencies

[13] http://kenai.com/projects/buginfo.
[14] http://subversion.apache.org/.
[15] http://www.nongnu.org/cvs/.
[16] https://bugzilla.mozilla.org//.
[17] http://git-scm.com/.
[18] http://www.scitools.com/.
[19] https://sites.google.com/site/ucinetsoftware/home/.

Table 2
The statistics of the studied projects.

Project	#Release	Language	kLOC		#modules(vertices)		#edges		#faults
			First release	Last release	First release	Last release	First release	Last release	
Firefox	6	C/C++	2613.3	2880.7	11431	12721	93166	98434	4490
Eclipse	5	Java	815.7	1979.3	6014	12936	39971	90461	13073
Ant	5	Java	31.8	116.0	361	1053	1043	3944	637
Camel	4	Java	11.0	45.1	298	804	1074	4046	1371
Lucene	3	Java	32.6	68.0	172	312	1408	2660	1314
Xalan	2	Java	105.9	122.5	681	711	3159	2901	687
Xerces	3	Java	30.4	63.3	115	301	333	933	475
Poi	4	Java	27.7	53.1	213	410	1233	4420	1377
Jedit	5	Java	51.4	109.4	243	434	1098	1098	943
Ivy	1	Java	36.6	–	294	–	1212	–	56
Tomcat	1	Java	146.9	–	714	–	2495	–	114

Table 3
Network measures.

Category	Metric	Description
Ego network measures	Size	# nodes that ego is directly connected to
	Ties	# directed ties corresponds to the number of edges
	Pairs	# unique pairs of nodes, i.e., Size × (Size − 1)
	Density	% of possible ties that are actually present, i.e., Ties/Pairs
	nWeakComp	# weak components in the ego network
	pWeakComp	# weak components normalized by size
	2StepReach	# nodes ego can reach within two steps normalized by Size
	ReachEffic	2StepReach normalized by the sum of the size of the ego's every neighbor's ego network
	Broker	# pairs not directly connected to each other
	nBroker	Broker normalized by the number of pairs
	EgoBetween	% shortest paths between neighbors that pass through ego
	nEgoBetween	EgoBetween normalized by the size of the ego network
Global network measures	Degree	# nodes adjacent to a given node
	Clus Coef	measures the density of an node's open neighborhood
	Closeness	sum of the lengths of the shortest paths from a node to all other nodes
	Reachability	# nodes that can be reached from a given node
	Eigenvector	assigns relative scores to all nodes in the dependency graphs
	Betweenness	measures how many shortest paths between other entities it occurs
	Power	measures the connections of the nodes in one's neighborhood
	EffSize	# nodes that are connected to a node minus the average number of ties between these nodes
	Efficiency	normalizes EffiSize to the total size of the network
	Constraint	measures how strongly an node is constrained
	Hierarchy	measures the extent to which constraint a node is concentrated in the network

between the ego module and its neighboring modules, while *undirected ego network* contains the dependencies of both directions. In addition, some global metrics were computed using only the incoming or outgoing edges, indicated as *IN* and *OUT*, respectively. In all, we explored the usefulness of 53 network measures (36 ego network measures and 17 global network measures) in effort-aware fault-proneness prediction.

3.6. Code metrics

In this study, code metrics were used to provide a baseline for the performance of the prediction models built with network measures. Like fault data, we collected code metrics by two means: (1) calculating them by ourselves for Firefox and Eclipse; and (2) using the existing data shared on *Promise* repository for the other nine projects.

The *Promise* data sets mentioned in Section 3.3 also record the values of 20 kinds of code metrics for each class of the projects. These code metrics include a size metric, two structural complexity metrics [16] and a set of object-oriented metrics (OO metrics) given by Chidamber and Kemerer [17], Henderson-Sellers [18], Bansiy and Davis [19], Tang et al. [20], and Martin [21]. Note that some OO metrics related to coupling also capture the dependency relationships between modules, such as CBO, RFC, IC, CBM, CA, and CE. Since they logically overlap network measures, using these metrics as the baseline can help us to decide whether network measures are better at catching the relationship between module interaction and fault-proneness.

The code metrics for Firefox and Eclipse are not available on *Promise*, thus we calculated them by ourselves. For Eclipse, all of these 20 code metrics were collected for each class of each investigated release. However, since Firefox is mainly written in C language and most source code files do not contain any classes, it is not suitable to compute OO metrics for it. Instead, we collected the Halstead metrics [22] for Firefox. In all, a total number of 14 and 20 kinds of code metrics were computed for all releases of the C/C++ project and Java projects, respectively. Table 4 summarizes these code metrics used in our study. A more detailed description is provided in [15].

We obtained these code metrics for Eclipse and Firefox by using the aforementioned (in Section 3.4) *Understand* databases and a Perl script. Size metrics, structural complexity metrics, and OO metrics were calculated at module (file or class) level, while Halstead metrics were generated for each single function and then rolled up to module level. Note that we treated a source file and its corresponding header file as one single module in Firefox, therefore, LOC, CCMax, and CCAvg of a module referred to the sum of the lines of code, the maximum, and the average cyclomatic complexity of all nested functions in the two files, respectively.

4. Research methodology

In this section, we first introduce the dependent and independent variables investigated in our study. Then, we describe the employed

Table 4
The baseline code metrics in this study.

Category		Metric	Description
Size metric		LOC	Number of lines containing source code of a module
Structural complexity metrics		CCMax	Maximum cyclomatic complexity of all nested functions in a module
		CCAvg	Average cyclomatic complexity of all nested functions in a module
Halstead metrics (collected for Firefox only)		n1	Total number of distinct operators of a function
		n2	Total number of distinct operands of a function
		n	Total number of vocabulary of a function, $n = n1 + n2$
		N1	Total number of operators of a function
		N2	Total number of operands of a function
		N	Halstead program length, $N = N1 + N2$
		V	Halstead program volume, $V = N \times \log_2 n$
		D	Halstead program difficulty, $D = N1/n1 \times N2/n2$
		L	Halstead program level, $L = 1/D$
		E	Halstead programming effort, $E = V \times D$
		T	Programming time, $T = E/18$ s
Object-oriented metrics (not collected for Firefox)	Complexity	WMC	The overall complexity of methods implemented within a given class.
		AMC	The average method size for each class.
	Cohesion	LCOM	For each attribute in a given class, calculate the percentage of the methods in the class using that attribute.
		LCOM3	The Henderson-Sellers version of LCOM.
		CAM	The relatedness among methods of a class based upon the parameter list of the methods.
		MOA	The extent of the part-whole relationship, realized by using attributes.
	Coupling	RFC	The number of methods that can potentially be executed in response to a message being received by an object of a given class.
		CBO	The number of distinct non-inheritance-related classes on which a given class is coupled.
		IC	The number of parent classes to which a given class is coupled.
		CBM	The total number of new/redefined methods to which all the inherited methods are coupled.
		Ca	The number of classes that depend upon the measured class.
		Ce	The number of classes that the measured class is depended upon.
	Inheritance	DIT	The length of the longest path from a given class to the root in the inheritance hierarchy.
		NOC	The number of classes that directly inherit from a given class.
		MFA	The ratio of the number of methods inherited by a class to the total number of methods accessible by the member methods of the class.
	Encapsulation	NPM	The number of the methods in a class that are declared as public.
		DAM	The ratio of the number of private (protected) attributes to the total number of attributes declared in the class.

modeling technique and the data analysis methods used to answer the research questions.

4.1. Variable description

The goal of our study is to empirically evaluate the actual usefulness of network measures in effort-aware fault-proneness prediction. For our work, we refer to each module (class/file) as an instance. The binary dependent variable here is fault-proneness. A module is said to be fault-prone if it has at least one fault, while it is said to be not fault-prone if it has no fault. In this paper, identifying the fault-proneness of modules was done by logistic regression (described in Section 4.2).

The independent variables in this study consist of two sets: 53 network measures and 14 (or 20) most commonly used code metrics. All these metrics were collected at module level. With these independent variables, we are able to answer the four research questions set up in Section 3.1.

4.2. Modeling technique

In our work, logistic regression was used to predict the dependent variable (module fault-proneness) from a set of independent variables (network measures and code metrics) to determine the percent of variance in the dependent variable explained by the independent variables [7]. Logistic regression is a type of probabilistic statistical classification model in which the dependent variable can take two different values.

We decided to use logistic regression mainly for four reasons. First, logistic regression is suitable for building fault-proneness prediction models in both ranking and classification scenarios (detailed in Section 4.3.3). On the one hand, logistic regression results in a predicted probability surface, which allows us to rank the modules according to their predicted possibility of being fault-prone. On the other hand, an artificial threshold can be set to classify the modules into two categories: fault-prone and not fault-prone, i.e., a module with a predicted probability larger than the threshold is treated as fault-prone, otherwise as not. Second, logistic regression does not make any assumptions of normality, linearity, and homogeneity of variance for the independent variables [23,24]. Therefore, the distribution of the independent variables will not influence the regression model, and consequently not affect the prediction results. Third, logistic regression is fairly intuitive. The regression model (as shown in Eq. (1)) is straightforward for researchers to interpret what effect each independent variable (metric) has on the dependent variable (fault-proneness), which will be discussed in Section 4.3.1, leading to a deeper understanding of network measures. Lastly, logistic regression is a well-known strategy, widely used in the area of software engineering including fault-proneness prediction [2,3,7] and shown to be promising. Systematic literature reviews on fault prediction performance [25,26] concluded that logistic regression performed well compared with other complex modeling techniques when predicting fault-proneness.

There are two types of logistic regression: univariate logistic regression and multivariate logistic regression. The univariate analysis is used to find the individual effect of the independent variable on the dependent variable, while the multivariate analysis is used to find the combined effect of the independent variables on the dependent variable.

The multivariate logistic regression model is based on the following equation:

$$\Pr(Y = 1|X_1, X_2, \ldots, X_n) = \frac{e^{\beta_0+\beta_1X_1+\beta_2X_2+\cdots+\beta_nX_n}}{1 + e^{\beta_0+\beta_1X_1+\beta_2X_2+\cdots+\beta_nX_n}} \quad (1)$$

where $X_i (i = 1, 2, \ldots, n)$ are independent variables and Y is the dependent variable which can only take on 0 or 1. $\beta_i (i = 0, 1, \ldots, n)$ are the regression coefficients and can be estimated through the maximization of a log-likelihood. $\Pr(Y = 1|X_1, X_2, \ldots, X_n)$ represents the probability of $Y = 1$, which, in this paper, indicates a module being fault-prone. The univariate logistic regression model is a special case of the multivariate logistic regression model, where there is only one independent variable [27].

Odds ratio is the most commonly used measure to quantify the magnitude of the correlation between the independent and dependent variables in a logistic regression model. When a logistic regression is calculated, the regression coefficient is the estimated increase in the log odds of the dependent variable per unit increase in the value of the independent variable [27]. In other words, the exponential function of the regression coefficient is the odds ratio associated with a one-unit increase in the independent variable. For a given independent variable X_i the odds that $Y = 1$ at $X_i = x$ denotes the ratio of the probability that $Y = 1$ to the probability that $Y = 0$ at $X_i = x$, i.e.,

$$\text{Odds}(Y = 1|X_i = x) = \frac{\Pr(Y = 1|..., X_i = x, ...)}{\Pr(Y = 0|..., X_i = x, ...)} \quad (2)$$

An odds ratio equal to 1 means that the independent variable does not affect the dependent variable. An odds ratio larger than 1 indicates that the independent variable positively associates with the dependent variable, while an odds ratio smaller than 1 indicates that the independent variable negatively associates with the dependent variable.

4.3. Data analysis methods

In our empirical study, univariate logistic regression was used to assess the correlations between individual network measures and fault-proneness. Multivariate logistic regression was employed to build prediction models with network measures or/and code metrics, in order to predict module fault-proneness under the cross-validation, across-release, and inter-project predictions. In particular, two effort-aware indicators, CE and ER, were exploited to quantify the prediction performance in ranking and classification separately. The Wilcoxon signed-rank test and the Cliff's δ served to compare the performance between different models to assess whether network measures are more effective than code metrics. In the following part, we detail these data analysis methods for answering the four research questions.

4.3.1. Univariate logistic regression analysis for RQ1

In order to explore whether there is a significant correlation between each network measure and fault-proneness, we employed univariate logistic regression to examine the effect of each metric separately, identify which metrics are significantly related to fault-proneness, and thus to determine which network measures are potentially useful fault-proneness predictors.

A critical step in logistic regression is to identify influential observations whose presence would greatly change the regression results [28]. The small subset of influential observations in the data could have a disproportionate impact of the estimation, so that the model-estimates are very likely to be based primarily on this data subset rather than on the majority of the data [28], which will lead to a bias in modeling the relationship between individual network measures and fault-proneness. Therefore, when performing univariate analysis, we checked for the presence of influential observations by using the Cook's distance [28]. Cook's distance is a measure of how much the residuals of all observations would change if a particular observation were excluded from the calculation of the regression coefficients. A large Cook's distance indicates that excluding an observation from computation of the regression statistics changes the coefficients substantially. The observation with a Cook's distance equal to or larger than 1 was regarded as an influential observation [28] and was hence excluded for the analysis.

The following statistics were evaluated for each single network measure:

- *Coefficient*. It is the estimated regression coefficient and represents the change in the logit for each unit change in the independent variable. The larger the absolute value of the coefficient, the stronger the impact of the independent variable (a given network measure) on the possibility of a module being fault-prone.
- *p-value*. It relates to the statistical hypothesis and tells whether the corresponding coefficient is significant or not. The Wald test [29] was used to evaluate the significance of an independent variable by examining whether or not removing it from the model will substantially harm the fit of that model. We set the significance level $\alpha = 0.05$ to assess the p-value we obtained.
- *Odds ratio*. As with the coefficient, it quantifies the effect of a given network measure on fault-proneness and allows us to compare the effects of individual measures.

4.3.2. Multivariate logistic regression analysis for RQ2, RQ3, and RQ4

To evaluate the predictive ability of the combined network measures, we used multivariate logistic regression to construct models that could optimize the explanation of fault-proneness and analyze the effect of network measures when used together. The multivariate logistic regression model could tell us how well network measures interpret fault-proneness.

Since not all the network measures are relevant to module fault-proneness, a key step in multivariate regression is to identify and remove unneeded, irrelevant, and redundant metrics that do not contribute to or may in fact decrease the accuracy of the predictive model. This is precisely the aim of the variable selection techniques, which allow to 1) improve the prediction performance of the model; 2) simplify the model for an easier interpretation; and 3) speed up modeling time and provide faster and more cost-effective predictors [30]. In statistics, the most popular form of variable selection methods is stepwise regression, and among which we adopted backward elimination, the simplest procedure, to determine which network measures to be included in the regression model in our study. Backward stepwise regression involves starting with all candidate variables, testing the deletion of each variable using a chosen model comparison criterion, deleting the variable that improves the model the most by being deleted, and repeating this process until no further improvement is possible.

Besides, one common problem in multivariate logistic regression analysis is multicollinearity which refers to a situation where two or more independent variables in a multiple regression model are highly linearly related. In the presence of multicollinearity, the estimated regression coefficients may be inaccurate and unreliable, which can adversely affect the study on how individual independent variables contribute to an understanding of the dependent variable. In this study, we computed the variance inflation factors (VIF) of each metric to examine multicollinearity between the variables of the model. We removed all variables with VIF larger than 10, which is the recommended cut-off value to deal with multicollinearity in a regression model [31].

Furthermore, all results were checked for the presence of influential observations by using the Cook's distance. The observation with a Cook's distance equal to or larger than 1 was regarded as an influential observation and was hence excluded for the analysis.

In all, we distinguished three types of prediction models: (1) NM (using only the 53 network measures); (2) CM (using only the baseline code metrics); and (3) AM (using the combination of network measures and code metrics). The construction of these models involves three main steps: (1) metric selection by using backward stepwise regression; (2) eliminating the metrics with VIF $>$ 10; and (3) removing the observations with *Cook's distance* $\geq$1.

In order to answer RQ2 to RQ4, we compared the predictive power of two pairs of models: (1) NM vs. CM; and (2) AM vs. CM. We adopted the following three prediction settings to simulate the realistic use and to obtain a thorough comparison:

- *Cross-validation for RQ2*. Cross-validation is performed within the same release of a project. The data set is split between training and testing sets using stratified sampling. In our study, for a given release of a project, we used 10 times 10-fold cross-validation to evaluate the effectiveness of the prediction models. More specifically, the samples from one release were randomly split into 10 equal size of subsamples. With each one fold being test set and the remaining nine folds training set, the validation process was then repeated 10 rounds. We ran 10 times of the entire process to get 10 $\times$ 10 prediction results.
- *Across-release prediction for RQ3*. Across-release prediction is performed within the same project. It is supposed to be more suitable and practical for industrial use where past project data are exploited to predict fault-prone modules in its future releases. In our study, we trained the prediction models on each release of a project and applied them to all their follow-up releases. That is, if a project had n releases under experiment, we got $n \times (n-1)/2$ prediction values for NM/CM/AM.
- *Inter-project prediction for RQ4*. Inter-project prediction is performed across different projects. The models trained on one project are used to predict the module fault-proneness in a new project. In our study, we trained the prediction models on the latest release of each project and tested them on the latest release of all other projects. Hence, if m project were under evaluation, we had $m \times (m-1)$ prediction values for NM/CM/AM.

4.3.3. Model evaluation criteria

We applied the prediction models in two typical application scenarios: ***ranking*** and ***classification***. In both scenarios, we evaluated the effectiveness of network measures in the context of effort-aware fault-proneness prediction where the effort to inspect the modules was taken into consideration. We used the source lines of code (LOC) in a module as a measure of the effort required to inspect it [32–34]. The intuition is that a larger module takes a longer time to review than a smaller module, hence one should prioritize smaller modules if the possibility of being fault-prone is the same. Therefore, in effort-aware fault-proneness prediction, the modules are ordered according to their relative risk, which is defined as the ratio of the predicted possibility of fault-proneness by logistic regression model to its LOC [34–36]. In the ranking scenario, modules are ranked in order from the most to the least relative risk. Software practitioners could simply select from the list as many potential high-risk modules as available resources for software quality enhancement will allow. In the classification scenario, modules are first classified into two categories: high-risk and low-risk. After that, those modules classified as high-risk are targeted for software quality enhancement.

Next we introduce the evaluation measures used in ranking and classification scenarios to quantify the prediction performance.

(1) CE in ranking scenario.

We evaluated the effort-aware ranking effectiveness of a fault-proneness prediction model with the help of the cost-effectiveness (CE) curve [37]. Fig. 2 shows an example of the CE curve. In this diagram, the x-axis is the cumulative percentage of LOC of the modules selected from the ranking list and the y-axis is the cumulative

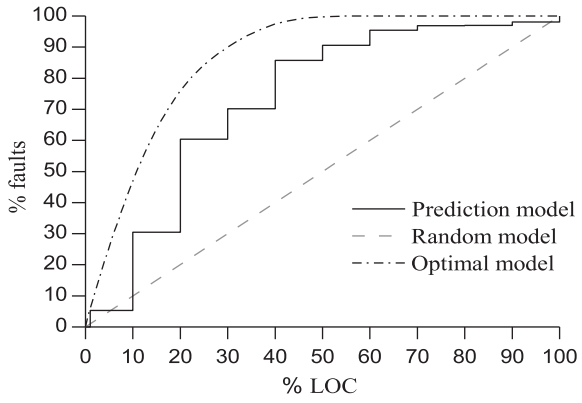


Fig. 2. An example CE curve.

percentage of faults found in the selected modules. Each fault-proneness prediction model corresponds to one CE curve which represents the actual percentage of faults given a percentage of lines of code of the classes selected to focus verification according to the ranking procedure. The overall cost-effectiveness of fault predictive models can be evaluated based on the area under the CE curve. We used the following formula [37] to quantify the effort-aware ranking performance of a model m:

$$\mathrm{CE}_\pi = \frac{\mathrm{Area}_\pi(m) - \mathrm{Area}_\pi(\mathrm{random})}{\mathrm{Area}_\pi(\mathrm{optimal}) - \mathrm{Area}_\pi(\mathrm{random})} \tag{3}$$

where $\mathrm{Area}_\pi(x)$ is the area under the CE curve of model x (m, *random* or *optimal*) for a given π percentage of LOC. In the random model, modules are randomly selected to inspect. In the optimal model, modules are sorted in decreasing order according to their actual fault densities. A larger value of CE_π means a better ranking effectiveness. A negative CE_π indicates that the model is inferior to the random model. Depending on the amount of resources available for inspecting modules, the cut-off value π will vary between 0 and 1. In practice, practitioners are more interested in the ranking performance of a prediction model at the top fraction. Therefore, we chose the CE_π at $\pi = 0.1$ and 0.2 to evaluate the performance of each model. In addition, we reported the CE_π at $\pi = 1.0$ in order to provide a more complete picture of the ranking performance.

(2) ER in the classification scenario.

We used the effort reduction measure ER (originally called LOC inspection reduction measure LIR) proposed in [38] to evaluate the effort-aware classification effectiveness of a fault-proneness prediction model. The ER measure calculates the ratio of the reduced source lines of code to inspect by using a classification model compared with a random selection to achieve the same recall of faults. In the classification scenario, only those modules predicted as high-risk will be inspected for software quality enhancement.

In order to get a comprehensive understanding of ER, we first introduce the inspection effort measure *Effort* (originally called LI in [38]). *Effort* is the ratio of LOC in the modules predicted as high-risk to the total lines of code in the system. We assume that the system under analysis consists of n modules. Let l_i be the LOC in module i and f_i be the number of faults in it, $1 \leq i \leq n$. For a given prediction model m, let p_i be 1 if module i is predicted as high-risk by m, and 0 otherwise, $1 \leq i \leq n$. Then the inspection effort measure of a classification model m is defined below:

$$\mathrm{Effort}(m) = \frac{\sum_{i=1}^{n} l_i \times p_i}{\sum_{i=1}^{n} l_i} \tag{4}$$

To achieve the same recall of faults as the model m, the inspection effort of a random model is $\mathrm{Effort}(\mathrm{random}) = \sum_{i=1}^{n} f_i \times p_i / \sum_{i=1}^{n} f_i$. Then the effort reduction measure of the model m is defined

as follows:

$$\mathrm{ER}(m) = \frac{\mathrm{Effort}(\mathrm{random}) - \mathrm{Effort}(m)}{\mathrm{Effort}(\mathrm{random})} \quad (5)$$

Note that in classification scenario, our goal is to classify each module into two categories, while logistic regression model results in a predicted probability surface. Therefore, a choice of threshold above which a given module is predicted to be high-risk is required. The traditional default is to simply use a threshold of 0.5 as the cut-off, but this does not necessarily result in the highest prediction accuracy, especially for highly unbalanced data sets [39]. In this study, we employed receiver operating characteristic (ROC) curve to determine the classification threshold, and this method is called balanced-pf-pd (BPP) method.

The ROC curve graphically illustrates the performance of a binary classifier at various discrimination threshold, with *pd* (probability of detection) as the *y*-axis and *pf* (probability of false alarm) as the *x*-axis [40]. A good model will achieve a high *pd* while *pf* is still relatively small. Hence, the closer a point in the ROC curve is to the perfect classification point (0, 1), the better the model performs. In this context, the degree of balance between *pf* and *pd* is evaluated by the "balance" metric balance $= 1 - \sqrt{(0 - pf)^2 + (1 = pd)^2}/2$ [40]. For a given training data set, BPP chooses the threshold having the maximum "balance".

For the simplicity of presentation, the effort reduction measure under the BPP threshold is called "ER-BPP". In addition, to provide a complete picture of the classification performance, we also adopted the "ER-AVG" metric [41] which evaluates the average effort reduction of a classification model over all possible thresholds on the test data set.

4.3.4. Performance comparison criteria

To determine whether network measures are effective in effort-aware fault-proneness prediction, we compared their performance with the baseline code metrics by using the Wilcoxon signed-rank test [42]. It is a non-parametric statistical hypothesis test used to compare whether two matched groups of data are identical. The paired samples in our study are the prediction results (CEs and ERs) produced by NM/AM and CM on every project. In particular, we employed the Benjamini–Hochberg (BH) corrected *p*-values [43] to assess the test outcomes at the significance level of 0.05. If the *p*-value obtained from the Wilcoxon signed-rank test was lower than 0.05, the two compared models (NM/AM vs. CM) were considered to achieve significantly different prediction performance. Together with the median values of CEs/ERs, we were able to draw a conclusion about whether network measures are better than code metrics.

Furthermore, we used the Cliff's δ effect size to evaluate whether the magnitude of the difference between the prediction performances of two models is important from the viewpoint of practical application [37]. Since the testing resources are limited, when deciding whether to use network measures, we have to consider the cost of generating the source code network, collecting the large number of metrics, and building the prediction models. If the promotion in predictive effectiveness is trivial compared with code metrics, network measures are not preferable because they are harder to collect, more in numbers, and slower to build the prediction models with, especially for large-scale projects. That is why we take into consideration the effect size, which is a simple way of quantifying the practical difference between two groups. Of all kinds of effect sizes, the Cliff's δ statistic is the most direct and simple variety of a non-parametric one, which considers the ordinal properties of the data [44]. By convention, the magnitude of the difference is considered either trivial ($|\delta| < 0.147$), small (0.147–0.33), moderate (0.33–0.474), or large (>0.474) [45].

Specifically, we chose to adopt the Wilcoxon signed-rank test rather than the pair-*t* test, and use Cliff's δ instead of other kinds of effect sizes, because the non-parametric nature of the chosen methods reduces the influence of the data characteristics such as distribution shape, differences in dispersion, and extreme values [46,47]. We tested the distribution of the prediction results (CEs and ERs) and found that they were not normally distributed in most cases, hence it was more suitable to apply the two non-parametric methods to them. A detailed normality test results are shown in Appendix B.

5. Results and discussion

This section presents the results of our empirical study with respect to each research question, together with some discussion.

5.1. Correlation (RQ1)

We employed univariate logistic regression to answer RQ1. For each of the 53 network measures, we tested its correlation with module fault-proneness for every investigated release of the 11 projects.

Table 5 summarizes the results of the univariate analysis for Firefox, Eclipse, Ant, and Camel. Due to the limited space, the results for the other seven projects are listed in Appendix C. The column *Metric* shows the independent variable used for the univariate logistic regression. The figures in parentheses indicate the number of releases of the corresponding project under analysis. The column *p* shows the number of releases for which the univariate regression models have significant coefficients by the likelihood-ratio test. A total number of 53 tests ran on each release of all the studied projects. Since multiple tests on the same data set might result in spurious statistically significant results, we used the Benjamini–Hochberg correction of *p*-values to control false discovery. In addition, for each project, if the regression models built with a specific network measure had no significant estimated coefficients for all the releases, the corresponding entry in the column *p* is marked in gray. The column *C* indicates the sign of the significant coefficients. The sign '–' means negative coefficients, while the blank means positive coefficients. The column OR lists the average odds ratio of the network measures.

From Table 5, we find that most of the investigated network measures have a significant correlation with fault-proneness because the *p*-values obtained from the Wald tests are smaller than the significance level (0.05) for at least one release of each project. In addition, most network measures are positively related to fault-proneness, since their estimated regression coefficients are positive. It means that the larger the values of these network measures for a module are, the more likely the module is to be fault-prone. However, only a few odds ratios are pronouncedly larger than 1.0, such as *nBroker, InCloseness*, and *OutCloseness*, which indicates that only a few network measures make outstanding contributions to the estimated probability of fault-proneness.

Based on the findings above, we answer RQ1:

The univariate logistic regression analysis results confirm that most network measures are significantly related to the occurrence of faults in modules. They are useful predictors for module fault-proneness and can be used to build the prediction models.

5.2. Cross-validation (RQ2)

In order to answer RQ2, we first built three prediction models (NM, CM, and AM) on the latest release of each project investigated in our study. Then, for each model, we ran 10 times 10-fold cross-validation to obtain 100 prediction values for a project. Finally, we used the Wilcoxon signed-rank test and the Cliff's δ to compare the predictive effectiveness of two pairs of models, i.e. (1) NM vs. CM; and (2) AM vs. CM, under ranking and classification scenarios.

Tables 6–8 show the median CEs at different cut-off values (π = 0.1, 0.2, and 1.0) and the median ER-BPPs/ER-AVGs obtained

Table 5
Univariate logistic regression analysis for individual network measures.

Metric	Firefox(6)			Eclipse(5)			Ant(5)			Camel(4)		
	p	*C*	OR	*p*	*C*	OR	*p*	*C*	OR	*p*	*C*	OR
Size	6		1.00	5		1.01	3		1.06	4		1.07
Ties	6		1.00	5		1.00	3		1.02	3		1.04
Pairs	0		/	5		1.00	1		1.00	4		1.00
Density	3		1.02	4	–	0.98	1	–	0.99	1	–	0.98
nWeakComp	2		1.02	5		1.04	0		/	4		1.15
pWeakComp	1	–	0.98	5	–	0.98	4	–	0.98	1	–	0.99
2StepReach	6		1.07	5		1.08	5		1.04	4		1.10
ReachEffic	2	–	0.99	5	–	0.99	2	–	0.98	0		/
Broker	0		/	5		1.00	1		1.00	4		1.00
nBroker	6		127.73	5		68.03	2		23.76	2		27.71
EgoBetween	3		1.00	5		1.00	2		1.01	4		1.02
nEgoBetween	6		1.04	2	–	0.99	0		/	0		/
Size_in	2		1.00	5		1.01	1		1.03	4		1.07
Ties_in	0		/	5		1.00	1		1.02	3		1.05
Pairs_in	0		/	5		1.00	0		/	4		1.00
Density_in	6		1.01	2	–	0.99	0		/	1	–	0.99
nWeakComp_in	1		1.02	5		1.03	0		/	4		1.13
pWeakComp_in	2		1.01	5	–	0.99	0		/	1		1.01
2StepReach_in	6		1.10	5		1.06	3		1.03	4		1.17
ReachEffic_in	3		1.01	5	–	0.99	0		/	2		1.01
Broker_in	0		/	5		1.00	0		/	4		1.00
nBroker_in	6		20.72	3	–	0.92	2		6.75	2		36.91
EgoBetween_in	2		1.00	3		1.00	1		1.06	0		/
nEgoBetween_in	6		1.04	5		1.01	0		/	3	–	0.98
Size_out	6		1.07	5		1.07	5		1.24	2		1.10
Ties_out	6		1.01	5		1.02	5		1.11	0		/
Pairs_out	6		1.00	5		1.00	5		1.02	2		1.02
Density_out	4		1.02	3		1.00	1		1.04	0		/
nWeakComp_out	6		1.17	5		1.22	1		1.37	2		1.32
pWeakComp_out	5	–	0.97	5	–	0.99	2		0.99	2		1.00
2StepReach_out	6		1.07	5		1.08	4		1.03	3		1.07
ReachEffic_out	4	–	0.99	3	–	1.00	1	–	0.99	2		1.01
Broker_out	6		1.00	5		1.00	5		1.04	2		1.04
nBroker_out	6		71.09	5		24.81	4		45.99	2		12.16
EgoBetween_out	6		1.00	2		1.00	1	–	1.03	0		/
nEgoBetween_out	6		1.04	4		1.01	0	–	/	3	–	0.98
Degree	6		1.00	5		1.01	3		1.06	4		1.07
OutDegree	6		1.07	5		1.07	5		1.24	2		1.10
InDegree	2		1.00	5		1.01	1		1.03	4		1.07
ClusCoef	3	–	0.26	5	–	0.12	1	–	0.16	1	–	0.08
Eigenvec	6	–	0.00	5		3.68E+17	4		3.07E+13	4		2.49E+32
Power	3	–	1.00	0		/	4		1.00	1	–	0.99
InCloseness	6		2.44E+54	4		5.42E+23	1		57.88	4		2.09E+20
OutCloseness	6		9.53E+54	2		8.18E+06	3		9.13E+52	1		1.08E+23
OutdwReach	6		1.01	5		1.00	4		1.08	2		1.02
IndwReach	6		1.00	5		1.00	2	–	0.99	4		1.02
Betweenness	6		1.00	5		1.00	2		1.00	3		1.00
UnBetweenness	1		1.00	5		5.68E+12	2		4.95	4		1.91
EffSize	6		1.00	5		1.01	2		1.05	4		1.08
Efficiency	6		8.42	4		3.68	0		/	1		3.03
Constraint	3	–	0.10	5	–	0.01	4	–	0.04	3	–	0.25
Hierarchy	0		/	5	–	0.18	1	–	0.16	2	–	0.19
Indirects	6		92.65	5		5.92	3		96.37	1		6.82

Table 6
The cross-validation performance of CM.

	$CE_{0.1}$	$CE_{0.2}$	$CE_{1.0}$	ER-BPP	ER-AVG
Firefox	0.413	0.376	0.438	0.380	0.805
Eclipse	0.246	0.244	0.235	0.556	0.515
Ant	0.039	0.038	0.027	0.067	−0.105
Camel	0.171	0.181	0.284	0.490	0.325
Ivy	−0.156	−0.200	−0.060	0.076	−0.605
JEdit	0.013	0.458	0.744	−1.000	−0.124
Lucene	0.636	0.643	0.659	0.727	0.533
Poi	0.505	0.505	0.520	0.591	0.414
Tomcat	0.151	0.196	0.415	0.340	−0.036
Xalan	0.628	0.639	0.686	0.762	0.685
Xerces	0.588	0.596	0.638	0.861	0.576

from 10 times 10-fold cross-validation in each project for CM, NM, and AM respectively. The underlined figure in Tables 7 and 8 means that NM/AM obtains a lower median value than CM. The figure is followed by "∗", if the Wilcoxon signed-rank test shows that the difference between NM/AM and CM is significant at the significance level of 0.05. The background colors from the lightest to the darkest indicate the effect sizes from trivial to large.

From Tables 6 to 8, we have the following observations:

(1) Ranking performance.
- From a statistically significant point of view, the median values and the outcomes of the Wilcoxon signed-rank test indicate that both NM and AM provide better ranking performance than CM at all cutoff values for Firefox, Eclipse, Camel,

Table 7
The cross-validation performance of NM.

	$CE_{0.1}$	$CE_{0.2}$	$CE_{1.0}$	ER-BPP	ER-AVG
Firefox	0.458*	0.443*	0.545*	0.698*	0.780*
Eclipse	0.275*	0.281*	0.305*	0.544	0.536*
Ant	0.012*	−0.025*	0.044	0.040	−0.168
Camel	0.362*	0.360*	0.589*	0.608*	0.509 *
Ivy	−0.170*	−0.174	0.029	−0.030	−0.490
JEdit	0.096*	0.461*	0.801*	0.443	0.222*
Lucene	0.636	0.655	0.671	0.657*	0.546
Poi	0.471	0.507*	0.595*	0.477*	0.422
Tomcat	0.159	0.178	0.251*	0.406	0.047*
Xalan	0.678*	0.664*	0.695*	0.769*	0.684
Xerces	0.696*	0.645*	0.670*	0.833	0.542

Table 8
The cross-validation performance of AM.

	$CE_{0.1}$	$CE_{0.2}$	$CE_{1.0}$	ER-BPP	ER-AVG
Firefox	0.474*	0.465*	0.591*	0.679*	0.784*
Eclipse	0.283*	0.293*	0.328*	0.564*	0.539*
Ant	0.025	0.100*	0.188*	0.194*	0.007*
Camel	0.391*	0.381*	0.593*	0.614*	0.501*
Ivy	−0.126	−0.076*	0.159*	0.024	−0.203*
JEdit	0.102	0.507	0.796	−1.000	0.217*
Lucene	0.672	0.660	0.665	0.722	0.537
Poi	0.472	0.509	0.568*	0.412*	0.416*
Tomcat	0.183*	0.283*	0.437*	0.472*	0.184*
Xalan	0.677*	0.670*	0.688*	0.773*	0.683
Xerces	0.722*	0.669*	0.747*	0.678*	0.444*

Xalan, and Xerces. For Poi, NM or AM outperforms CM at $\pi = 0.2$ or 1.0.

- For Ant, Ivy, and Tomcat, network measures alone are inferior to code metrics, since NM achieves significantly lower median values than CM at $\pi = 0.1$, 0.2, or 1.0. However, network measures improve the ranking effectiveness when adding them to code metrics, because AM produces significantly higher median values than CM.
- For JEdit, NM significantly outperforms CM, but AM cannot improve the ranking performance of CM.
- For Lucene, both NM and AM have comparable ranking ability with CM, because all the *p*-values obtained from the Wilcoxon signed-rank test for comparing CEs between NM/AM and CM are larger than 0.05.
- Considering Cliff's δ, the effect sizes are large only for Firefox, Eclipse, and Tomcat. For most projects, the small effect sizes indicate that the performance differences between NM/AM and CM are negligible.

  Recall that we decide that network measures are effective if they alone or together with the baseline code metrics have a significantly better fault-proneness predictive ability than code metrics. We conclude that for most projects (10 out of 11, except Lucene), network measures are effective under the ranking scenario in the context of the effort-aware cross-validation.

(2) Classification performance.

- Only for Camel, NM and AM are better performers than CM in terms of both ER-BPPs and ER-AVGs.
- For Eclipse, Ant, Ivy, JEdit, Tomcat, and Xalan, NM or/and AM achieve significantly higher ER-BPPs or/and ER-AVGs than CM.
- For Lucene, Poi, and Xerces, CM outperforms NM or/and AM, since the *p*-values obtained from the Wilcoxon signed-rank tests are significant for comparing ER-BPPs or/and ER-AVGs.
- For Firefox, NM and AM show significantly higher ER-BPPs but lower ER-AVGs than CM.
- Considering Cliff's δ, the effect sizes remain trivial or small in most cases, which implies that there are not notable differences in the classification ability between network measures and code metrics.

In all, for 8 out of 11 projects (except Lucene, Poi, and Xerces), network measures are effective in the classification scenario of the effort-aware cross-validation.

Furthermore, in order to get an overall comparison across all these studied projects, apart from reporting the results for each system, we combine them: for each prediction model (CM, NM, and AM) with respect to each performance measure (CEs at $\pi = 0.1$, 0.2, and 1.0, ER-BPP, and ER-AVG), we merge the 100 prediction values obtained for every system, resulting in a set of 1100 data points to represent all the 11 projects. The Wilcoxon signed-rank test and the Cliff's δ are also used to compare the overall performance between CM and NM/AM. We visually display the overall performance by way of a "mini boxplot": a bar indicates the first–third quartile range, and a circle the median. Table 9 shows the median values and the mini boxplots of the ranking and classification performance under cross-validation, together with the Cliff's δ of the comparisons between NM/AM and CM. The "*" following the Cliff's δ means that the difference between the comparison pair is significant at the significance level of 0.05.

The results of the overall comparison show that NM and AM significantly outperform CM both in ranking and classification scenarios, but the effect sizes are always trivial to small.

Based on the findings from the individual and overall comparisons, we answer RQ2:

> *On the whole, network measures (alone or together with code metrics) are more effective than code metrics under the cross-validation evaluation. However, considering the effect size, network measures do not improve the prediction performance substantially.*

5.3. Across-release prediction (RQ3)

In order to answer RQ3, we evaluated the across-release predictive effectiveness of network measures on four projects: Firefox, Eclipse, Ant, and JEdit which had five or more releases investigated in this study. The three types of prediction models (NM, CM, and AM) built on every earlier release of the four projects were applied to each future release of the same project. That is, we got $n \times (n-1)/2$ prediction values for NM, CM, and AM each, where n was the number of studied releases in a project. Specifically, we excluded the other seven projects because they each had less than five releases of data posted on the *Promise* repository, producing at most 6 prediction values, which were not adequate to get a statistical conclusion. Finally, the Wilcoxon signed-rank test and the Cliff's δ were used to compare the predictive effectiveness of two pairs of models, i.e. (1) NM vs. CM; and (2) AM vs. CM, under ranking and classification scenarios.

Fig. 3 shows the boxplots which describe the distribution of the CEs at different cut-off values and the ER-BPPs/ER-AVGs obtained from the across-release predictions for CM, NM, and AM. For each model, the boxplot presents the median (the horizontal line within the box), the 25th, and the 75th percentiles (the lower and upper sides of the box) of the prediction results in terms of CEs and ERs. The figures in parentheses indicate the number of prediction values for NM/CM/AM.

From Fig. 3, we have the following observations:

(1) Ranking performance.

- Only for Firefox, NM and AM gain significantly advantage over CM with *p*-values lower than 0.05 at all cutoff values. The effect sizes are all large in terms of Cliff's δ ($0.493 \leq |\delta| \leq 0.920$). For Eclipse, NM and AM outperform CM only at $\pi = 1.0$, and the effect sizes are large ($|\delta| = 0.660$ and 0.800).

Table 9
Overall ranking/classification performance under cross-validation.

			median	mini boxplot	Cliff's δ	
					NM vs. CM	AM vs. CM
ranking	$CE_{0.1}$	CM	0.310			
		NM	0.364		0.056*	0.103*
		AM	0.398			
	$CE_{0.2}$	CM	0.316			
		NM	0.382		0.070*	0.121*
		AM	0.416			
	$CE_{1.0}$	CM	0.408			
		NM	0.489		0.102*	0.178*
		AM	0.528			
classifi	ER-BPP	CM	0.511			
cation		NM	0.571		0.051*	0.067*
		AM	0.572			
	ER-AVG	CM	0.437			
		NM	0.487		0.039*	0.05*
		AM	0.472			
				0.032 0.724		

- For Ant and JEdit, NM and AM perform as well as CM. Though the median CEs of CM are higher (lower) than those of NM or AM for Ant (JEdit), the differences are however undistinguishable from a statistically significant point of view. The effect sizes are always trivial to small ($0.081 \leq |\delta| \leq 0.322$).

(2) Classification performance.

- Only for Firefox, NM and AM have significantly higher ER-BPPs and ER-AVGs than CM. The effect sizes for comparing ER-BPPs are large ($|\delta| = 0.933$ and 0.929), while for ER-AVGs are moderate ($|\delta| = 0.396$ and 0.387).
- For Eclipse, Ant, and JEdit, NM and AM exhibit comparable classification effectiveness with CM. Though Fig. 3 shows that differences lie in ER-BPPs (ER-AVGs) between CM and NM/AM, the conclusions of the Wilcoxon signed-rank test indicate that the differences are not significant. The effect sizes are trivial to small ($0.025 \leq |\delta| \leq 0.313$).

In general, from the results for individual projects, it is difficult to demonstrate that one prediction model is better than another in ranking or classification scenario under effort-aware across-release prediction, as witnessed by the many failures (p-value > 0.05) of the Wilcoxon signed-rank test. Hence, like in Section 5.3, we also conducted an overall comparison across all the four studied projects. For each prediction model (CM, NM, and AM) with respect to each performance measure (CEs at $\pi = 0.1$, 0.2 and 1.0, ER-BPP, and ER-AVG), we merged the prediction values obtained for each system, resulting in a set of 45 data points to represent all the four projects. Table 10 shows the overall ranking and classification performance under across-release prediction.

The results of the overall comparison show that NM and AM significantly outperform CM in ranking scenario, but the effect sizes are always small. Under the classification scenario, NM and AM have significantly higher ER-BPPs than CM, but comparable ER-AVGs. The effect sizes are moderate for comparing ER-BPPs, but trivial for ER-AVGs. In addition, though CM, NM, and AM obtain obviously higher median ER-AVGs than CEs, the ER-AVGs vary a lot across different projects. This observation shows that the classification ability of both network measures and code metrics are less stable compared with their ranking ability.

Based on these findings, we answer RQ3:

In general, network measures are more effective than code metrics under the across-release prediction. However, considering the effect size, network measures do not notably improve the prediction performance, especially when ranking fault-prone modules.

5.4. Inter-project prediction (RQ4)

In order to answer RQ4, the three types of prediction models (NM, CM, and AM) built on the latest release of each investigated Java project were applied to the other nine Java projects. That is, we got $10 \times 9 = 90$ prediction values for NM, CM, and AM each. Again, the Wilcoxon signed-rank test and the Cliff's δ were used to compare the predictive effectiveness of two pairs of models, i.e. (1) NM vs. CM; and (2) AM vs. CM, under ranking and classification scenarios. Note that we did not carry out the inter-project prediction on Firefox, because the code metrics used to build CM for Firefox were different from those used for other projects.

Fig. 4 shows the boxplots which describe the distribution of the CEs at different cut-off values and the ER-BPPs/ER-AVGs obtained from 90 inter-project predictions for CM, NM, and AM.

From Fig. 4, we have the following observations:

In both ranking and classification scenarios, CM has higher median CEs (at $\pi = 0.1$, 0.2, and 1.0), ER-BPPs, and ER-AVGs than both NM and AM. The Wilcoxon signed-rank test shows there are no significant differences between the performance of CM and NM. However, when comparing AM with CM, the p-values are lower than the significance level. It indicates that network measures alone have comparable predictive ability with code metrics, but when adding network measures to code metrics, the predictive effectiveness is significantly reduced. In terms of Cliff's δ, the effect sizes are all trivial ($0.023 \leq |\delta| \leq 0.117$) for comparing NM with CM, while they are all small ($0.177 \leq |\delta| \leq 0.210$) for comparing AM with CM.

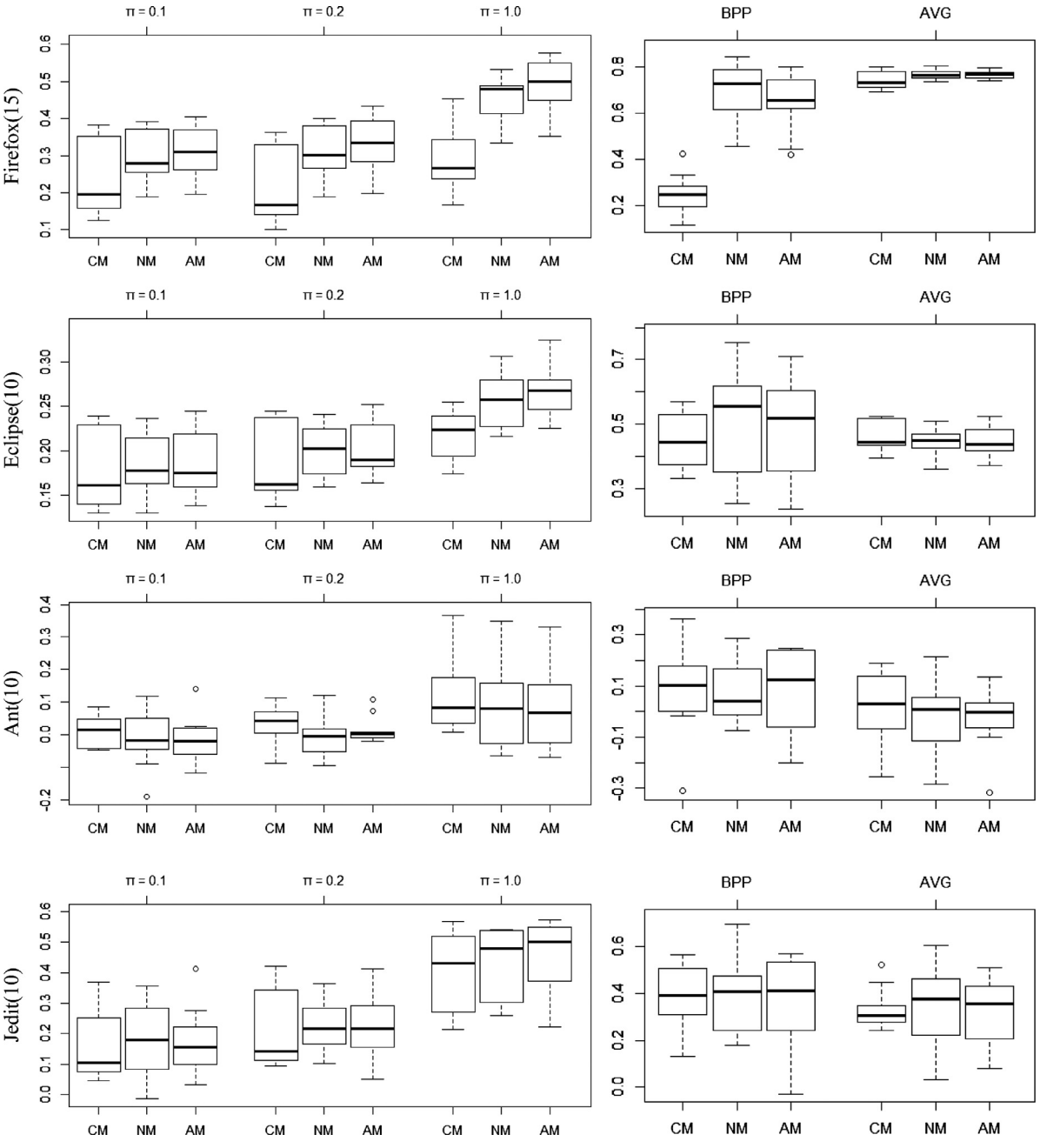


Fig. 3. Ranking/classification performance under across-release prediction.

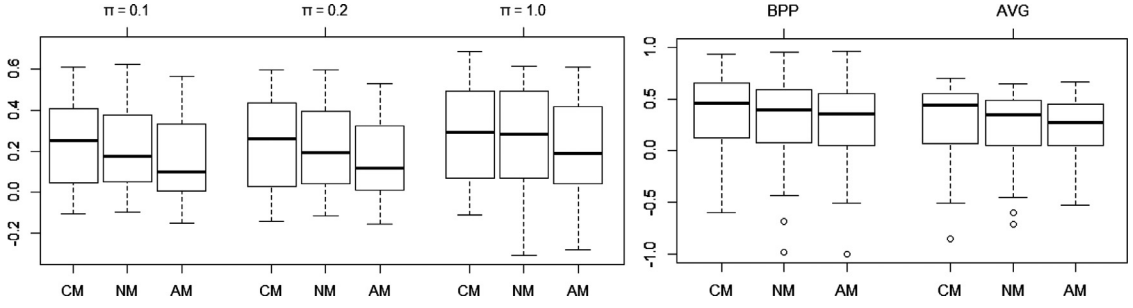


Fig. 4. Ranking/classification performance under the inter-project prediction.

Table 10
Overall ranking/classification performance under across-release prediction.

			median	mini boxplot	Cliff's δ	
					NM vs. CM	AM vs. CM
ranking	$CE_{0.1}$	CM	0.151			
		NM	0.188		0.155*	0.157*
		AM	0.181			
	$CE_{0.2}$	CM	0.154			
		NM	0.205		0.227*	0.231*
		AM	0.198			
	$CE_{1.0}$	CM	0.242			
		NM	0.307		0.264*	0.309*
		AM	0.352			
classifi cation	ER-BPP	CM	0.296			
		NM	0.457		0.369*	0.354*
		AM	0.447			
	ER-AVG	CM	0.436			
		NM	0.461		0.042	0.021
		AM	0.438			
				0.081 0.752		

Based on the observations above, we answer RQ4:

Overall, network measures are not more effective than code metrics under the inter-project prediction.

5.5. Discussion

In this paper, we examine the effectiveness of network measures for effort-aware fault-proneness prediction. Next, we will make some discussion on our experimental results.

(1) *Different network measures have different effects on fault-proneness*

From Section 5.1, we find that there are huge differences in the effects among various kinds of network measures. Some network measures are not related to the fault-proneness of modules for most studied projects, such as Pairs_in, pWeakComp_out, and ReachEffic_out. In addition, a majority of network measures have an odds ratio very close to 1.0, which indicates that they have very small impact on the fault-proneness. Only a few network measures have an odd ratio significantly larger than 1.0, such as nBroker, Eigenvec, and OutCloseness. They are practically useful indicators for fault-proneness. Since collecting and training models with the large number of useless network measures cost plenty of time, such results are valuable for practitioners to direct their effort toward those actually useful network measures.

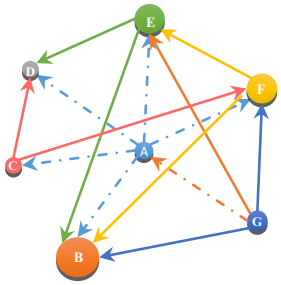
Moreover, the influential network measures may help us find some vulnerable structural features that the faulty modules have in common. As we know, each kind of network measure describes the software structure from a certain perspective. For example, OutCloseness evaluates how difficult to get to other modules in the whole network from a given module. A module with short distances to all other ones will have a high OutCloseness. Since we know from Section 5.1 that the univariate regression model built with OutCloseness has a very large odds ratio, which indicates that the modules with high values of this metric are likely to be fault-prone. Consequently, we have the reason to believe that the modules closer to the rest ones in the source code are more prone to faults. An intensive study into these influential network measures is meaningful and useful to identify bad software design, and thus to improve the software quality.

(2) *Network measures are better than code metrics in most cases within the project*

From the results of cross-validation and across-release prediction, we conclude that within the same project, network measures are more effective than code metrics on the whole and they are better fault-proneness indicators for most investigated projects. We conjecture that it is because network measures can catch more fault patterns in the source code of these projects compared with code metrics.

Network measures treat each module as a node and emphasize the dependency relationships between modules. They weigh the importance of a module based on the centrality of its location in its neighborhood or the whole source code network from various aspects, such as the distance from it to other modules, the size or density of its neighboring modules, and the amount of information it exchanges. These structure features are claimed to be related with fault-proneness, e.g., the central module of a star-shaped sub-network is found to be fragile [2]. Similarly, when analyzing the fault data and the source code networks of the investigated projects, we have two findings supporting the opinion: (1) the modules with a complex neighborhood where modules have complicated dependencies with others are fault-prone; and (2) the "broker" modules lying between many pairs of others are vulnerable. These location-related properties, which cannot be captured by code metrics directly and comprehensively, are very useful for predicting faults in certain projects, causing code metrics to miss some fault-prone modules identified by network measures.

The advantage of network measures is evident in our experimental results. In across-release prediction, network measures identified 6.3% more real fault-prone modules than code metrics for Firefox, Eclipse, Ant, and JEdit. A concrete example is the fault-prone class "org.gjt.sp.jedit.print.BufferPrintable" in JEdit 4.0. NM could pick it out while CM were not able to. Fig. 5 shows the local network of this class. "BufferPrintable" has dependencies with six classes which have many interactions with others, e.g., class "org.gjt.sp.jedit.jEdit"



A: org.gjt.sp.jedit.print.BufferPrintable

B: org.gjt.sp.jedit.jEdit

C: org.gjt.sp.jedit.textarea.ChunkCache

D: org.gjt.sp.jedit.syntax.SyntaxStyle

E: org.gjt.sp.jedit.GUIUtilities

F: org.gjt.sp.jedit.Buffer

G: org.gjt.sp.jedit.print.BufferPrinter

Fig. 5. The local network of the class "*org.gjt.sp.jedit.print.BufferPrintable*".

is coupled with 128 classes. From the network perspective, "BufferPrintable" has a complex neighborhood and takes a great deal of responsibility for exchanging information among its neighbors, causing it vulnerable to faults. However, from the object-oriented perspective, "BufferPrintable" is of relatively low coupling (cbo = 0) and high cohesion (lcom3 = 0.711), and thus a good design with little possibility of being fault-prone. Consequently, faults in this kind of classes can be identified by network measures but not by code metrics.

However, network measures were not always superior to code metrics within projects. For some projects such as Lucene, Poi, and Xerces, code metrics achieved significantly higher ERs when classifying modules in cross-validation. Unlike network measures, code metrics also take into account the information inside a module, e.g., CCAvg measures the average cyclomatic complexity of all nested functions in it. The inconsistent comparison results imply that for some systems, the dependency relationships between modules have more influence on their fault-proneness, while for others, the internal structural features would be equally or more important fault-proneness indicators.

(3) *Network measures do not show advantage over code metrics across projects*

From the results of inter-project prediction, it is obvious that network measures are not superior to code metrics for predicting fault-proneness. We argue that it is the long-lasting obstacle of the inter-project prediction, i.e., the different data characteristics between different projects [48], that drain the advantage of network measures away. Since the studied software systems are developed for different applications in different environments by different developers using different programming languages and coding styles, their source code networks have different structures and the characteristics of network measures differ from one project to another. Fig. 6 depicts the mean values of the 53 network measures in the nine small-scale projects. Although the nine projects are of comparable software size and written in the same programming language, the network features for some of them vary a lot with the metric values in different distributions. For example, Ant 1.7 and Poi 3.0 have significantly different mean values of most network measures.

The various data features across software systems lead to an inconsistent performance of network measures in our study, which is shown as the following three aspects.

(1) The same kind of network measure may have different effects on the fault-proneness among different projects. A network measure which is significantly related to fault-proneness in a project may have no or little impact in another project, shown as the large differences of *p*-values/ORs across different projects for nWeakComp, Efficiency, and Hierarchy in RQ1.

(2) The network measures selected to build the prediction models by the backward stepwise regression are different across projects. Table 11 shows the models of Poi 3.0 and Ivy 2.0. It is clear that only nWeakComp_out is in both models.

Table 11
The network measures selected to build the prediction models for Poi 3.0 and Ivy 2.0.

	Coefficient	Std. error	Pr(>|z|)
Poi 3.0			
(Intercept)	−0.66	0.56	0.24
Pairs	0.00	0.00	0.00
Density	−0.07	0.01	0.00
nWeakComp	−0.10	0.05	0.03
ReachEffic	−0.03	0.01	0.00
nBroker	1.86	0.98	0.06
nEgoBetween	−0.04	0.02	0.04
ReachEffic_in	0.03	0.01	0.00
nBroker_in	0.55	0.35	0.12
nWeakComp_out	0.64	0.20	0.00
Eigenvec	57.30	10.97	0.00
UnBetweenness	1.42	0.78	0.07
Constraint	2.79	1.58	0.08
Ivy 2.0			
(Intercept)	−2.82	0.50	0.00
pWeakComp	−0.01	0.01	0.16
Size_out	0.13	0.05	0.00
nWeakComp_out	0.40	0.19	0.04
EgoBetween_out	−0.01	0.00	0.01

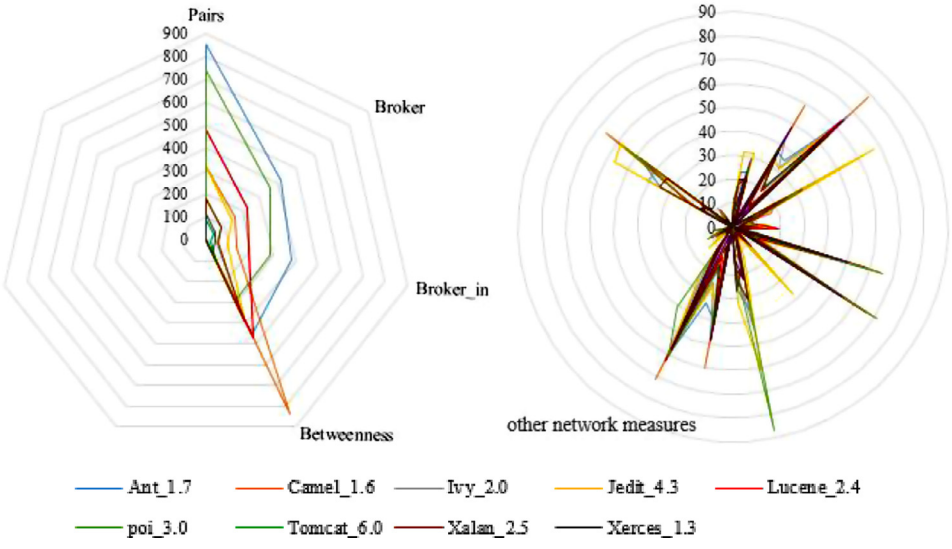


Fig. 6. The mean values of the 53 network measures in the nine small-scale projects.

(3) When carrying out inter-project prediction, the performance of a prediction model varies a lot when it is transferred to different projects. For example, the NM trained with Poi 3.0 suited Lucene 2.4 well (with CE0.1 = 0.624, CE0.2 = 0.596, CE1.0 = 0.606, ER-BPP = 0.695, and ER-AVG = 0.619). But, the same prediction model performed extremely badly when it was transferred to Ivy 2.0 (with CE0.1 = −0.021, CE0.2 = −0.057, and CE1.0 = −0.006, ER-BPP = −0.689, and ER-AVG = −0.367). The results suggest that a subset of network measures selected to build the regression model and to characterize one project may fail to capture the salient features of other projects.

Furthermore, when observing the distributions of network measures in Ivy 2.0, Poi 3.0, and Lucene 2.4, we find that Poi 3.0 and Lucene 2.4 share many similarities in the data features, e.g., the comparable mean values for most network measures. However, when comparing Ivy 2.0 with Poi 3.0, we see great differences of data distribution lying between them. These observations suggest that, when performing inter-project predictions, it is more suitable to use the models trained on the projects with as many similarities as possible.

In summary, the various data features and the unstable performance of network measures make it a big challenge to find a training project presenting the same fault pattern with the target project, which was pointed out to be a key solution to achieve an excellent performance for inter-project prediction [14]. Thus, network measures are not so effective across different projects as within the same project.

In addition, the unstable prediction performance also confirms that one kind of metrics cannot capture the structure features of all projects [9]. A further investigation on the effectiveness of network measures for different categories (in terms of software size, programming language, application domain, and etc.) of software systems needs to be carried out.

(4) *Make the choice by considering the prediction scenarios from practical use*

Previous studies have not drawn conclusions about when and how network measures should be used, which is desirable for software practitioners. Therefore, at the end of the discussion section, we intend to provide some practical advices about the use of network measures. Before that, we first compare our findings with those of previous studies [2–6] in the following part. We find that some experimental results in our study confirm their conclusions, but others are not consistent.

Consistent findings. First, our work indicates that network measures are superior to code metrics when predicting faults within the same release of large-scale projects, which is consistent with [2–5]. Second, we obtain the same inter-project prediction results as Premraj and Herzig [4] and Prateek et al. [6] that network measures are not more effective when transferred to a new project.

Inconsistent findings. Premraj and Herzig [4] concluded that network measures provided no advantage over code metrics for forward-release prediction. It is not true in our study since network measures significantly outperform code metrics for Firefox. In addition, for some small-scale projects, the statistical superiority of network measures over code metrics in cross-validation is in conflict with Tosun et al. [3] and Prateek et al. [6]. They revealed that network measures did not provide significant predictive power on small-scale projects. On the contrary, our results suggest that network measures are preferable for Camel, Xerces, and Xalan in cross-validation.

These differences in findings are possibly due to the different model evaluation criteria. Nguyen et al. proposed that deciding the performance of a prediction model depended on the choice of evaluation measure [5]. The effort-aware performance indicators (CEs and ERs) used in this study assess the predictive effectiveness with the effort involved in inspecting the code taken into account. The traditional classification performance indicators, such as precision and recall, are interested in whether the prediction model can accurately identify more fault-prone modules. To determine which to use is not a trivial matter and should be based on the objective of the prediction. More studies on the predictive effectiveness of network measures need to be carried out to compare the inconsistent results.

In addition, it is noteworthy that apart from the Wilcoxon signed-rank test, we also calculated the effect size to quantify the difference of the prediction performance between the two metric sets, and it was not employed by the aforementioned studies. The effect size helps to decide whether network measures are cost-effective especially when considering their expensive cost of collection and being trained with. Reviewing the predictive effectiveness of network measures and code metrics for small-scale projects, we find that NM outperforms CM with trivial or small effect sizes for most investigated projects under the cross-validation and across-release prediction. The results imply that network measures offer no decided advantage over code metrics in practice. The extra expense to collect network measures and train the models does not pay off. On the other hand, from the cross-validation and across-release prediction results of the two large-scale projects: Firefox and Eclipse, we find that the effect sizes are large in most cases when comparing NM/AM with CM. We have the reason to believe that it is worth sacrificing extra time to collect and use the network measures for Firefox and Eclipse.

After the discussion about the actual usefulness of network measures with considering the prediction costs, we give the following suggestions for practical use based on the findings of both previous and this studies:

- Cross-validation and across-release prediction. For small-scale projects, network measures are more suitable for fault-proneness prediction, if the software practitioners have sufficient time to collect and build models with network measures in testing stage. For large-scale projects, using network measures is a better choice worthy to be spent more time on.
- Inter-project prediction. It is suggested to examine the source code network features of the trained and target projects first. If the data characteristics of network measures are much similar in the two systems, network measures are preferable to identify fault-prone modules in the target project.

We believe that our experimental results can provide valuable data for the practical usefulness of network measures, as well as offer the practitioners some meaningful implications of fault-proneness prediction.

6. Threats to validity

In this section, we discuss the most important threats to the construct, internal, and external validity of our study following common guidelines for empirical studies [49].

6.1. Construct validity

Construct validity is the degree to which the variables used in a study accurately measure the concept they purport to measure. The most important threat to the construct validity of our study is the accuracy of the dependent and independent variables.

The dependent variable in this study is a binary variable that represents the fault-proneness of modules. The first threat concerns the way we collected faults and linked them with versioning system files. We only selected the faults with references in change logs such as "#623441", which would lead to false negatives in the fault data set since some developers did not leave references for faults in change logs. In addition, we classified the faults into each release based on the date that they were modified. For example, if a fault was fixed

during version 9.0 and 10.0, the number of faults in version 9.0 increased by 1. We could not decide whether this fault was introduced in previous versions, e.g. version 8.0, which may also result in false negatives. However, Kim et al. [50] discovered that false negatives in fault data do not affect fault prediction performance in a significant manner.

The independent variables in this study are the network measures and the widely-used code metrics. To obtain the network measures, we first collected the dependency information between modules and constructed the source code networks. The degree of accuracy of network construction depends on the tool we used. Although *Understand* is a mature commercial tool and has been used to collect the metric data in many previous studies [51–54], some features of C/C++ cannot be handled perfectly by *Understand*, for example, function calls via pointers. *Understand* associates dependency between the function and where its address is taken. However, table or array driven setups where the taking of the address may be away from where it is actually used. In spite of this, *Understand* can still work pretty well in other types of code. In future work, other static tools will be adopted to compare and validate the source code network construction. In addition, the merging of the source (.cpp/.c) and header (.h) files in Firefox also influenced the building of source code networks. We combined a source file and its corresponding header file as one single node in the network. Specifically, we treated the header file with the same base name of a source file as its corresponding one, however this is not always the case for all C/C++ projects. To check the reliability of merging, we randomly picked up 100 source files in Firefox 6.0 and found that their corresponding header files all had the same base names (including path names). Therefore, we believe that the merging method in our study would barely affect the accuracy of networks.

6.2. Internal validity

Internal validity reflects the extent to which a causal conclusion based on a study is warranted. The first threat to the internal validity of our study is the unknown effect of the deviation of the independent variables from the normal distribution. In logistic regression, there is no assumption related to normal distribution. Therefore, we did not take into account whether the independent variables followed a normal distribution, and the raw data were used to build the logistic regression models. However, previous studies suggested that applying the log transformation to the independent variables to make them close to a normal distribution might lead to a better model [40]. To eliminate this threat, we applied the log transformation to the network measures and code metrics of Firefox and reran the analysis. We found that the conclusions for RQ2 and RQ3 did not change significantly before and after the log transformation.

The second threat concerns the identification and deletion of influential observations in logistic regression. According to Belsley et al. [28], regression diagnostics comprises a collection of methods used in the identification of influential points and multicollinearity. Notably, the detection of influential observations has received a great deal of attention in the statistical literature. Influential observations arise from two fundamentally distinct sources [28]. First, they may be the result of measurement or recording errors which are just bad data, detrimental to model estimation. Second, they may be legitimately occurring extreme observations which reflect the true distribution of the innovations process, exhibiting heteroscedasticity, skewness, or leptokurtosis. Such observations may contain abnormal sample information that is nevertheless valuable to accurate model estimation. In literature, the Cook's distance has always been used to check the presence of the influence observations [55], which is also adopted in our study. In all, we have identified a total number of 23 influential observations in 26 multivariate regression models used in RQ3 and RQ4.

The decision to delete influential observations ultimately depends on the purpose of the model. Since the aim of our regression models is predicting, i.e. building models with past data to predict the fault-proneness of modules in new versions or new projects, then it must be decided whether deleting points would create a resample that is more typical of the training data, and so the testing ones. Although influential observations could help to build more accurate models, they are likely to indicate overfitting or the need for truncating the range of highly skewed variables. We manually inspected the 23 influential observations identified in our study and found that they all had some extremely low or high metric values compared with other observations, and thus were special cases which could not reflect the general character of the majority [56]. They were not good for prediction, therefore we excluded them from building the models.

The third threat is the modeling technique we chose in our study. We used logistic regression to build the prediction models and came to several conclusions based on the performance of these models. However, the related work [6] also adopted some learning-based approaches, such as Support Vector Machines and Random Forests, to predict fault-proneness. In order to examine the impact of the modeling technique on our conclusions, we repeated the experiments by using three popular kinds of machine learning methods, i.e., Random Forests, Support Vector Machines, and Classification and Regression Trees. They produced very similar results to logistic regression in most cases, and did not change our original conclusions on the whole.

6.3. External validity

External validity concerns the possibility to generalize our findings to other systems. We have studied multiple releases of 11 long-lived and popular projects varying in size, programming language, and application domain. The data sets collected from these systems are large enough to draw statistically meaningful conclusions. We believe that our study makes a significant contribution to the software engineering body of empirical knowledge about the usefulness of network measures in effort-aware fault-proneness prediction. Nevertheless, we cannot assume that our results generalize beyond the specific environment where they were conducted. Further validation on a larger set of software systems is desirable.

7. Conclusions

In this paper, we make an in-depth exploration into the effectiveness of network measures in effort-aware fault-proneness prediction. By conducting our study on multiple releases of 11 open-source projects with various sizes, programming languages and application domains, we find that most network measures are significantly and positively related to fault-proneness, but only a few of them have a notably effect. This is an important finding, as it can guide the researchers to concentrate their effort on these impactful metrics. This result also suggests that the modules with lower values of these network measures tend to be of higher quality, which is helpful for practitioners in software design and refactoring.

We further evaluated the predictive ability of network measures both in ranking and classification scenarios. Our results indicate that, although network measures almost offer no improvement in inter-project prediction compared with code metrics, they do improve the prediction performance in most cases within the same project. These findings are practically useful, as they can help practitioners to decide whether and when to use network measures for fault-proneness prediction.

Another observation in this study is that the performance of network measures varies across different projects, probably dependent on the various software structures. Network measures

appear to be more effective for large and complex projects, while they are unstable across small-scale projects. This result implies that practitioners should take into account the software features when using network measures. A future study is needed to evaluate the effectiveness of network measures for different categories of projects.

Acknowledgments

The authors would like to thank the editor and anonymous reviewers for very helpful suggestions in greatly improving the quality of this paper. This work is supported by the National Key Basic Research and Development Program of China (2014CB340702), the National Natural Science Foundation of China (61472175, 61432001, 91318301, 61472178, 61170071), and the National Natural Science Foundation of Jiangsu Province (BK20130014). All support is gratefully acknowledged.

Appendix A. The distribution of fault data

Fig. 7 shows the distribution of fault data for each studied release of Firefox, Eclipse, Ant, Camel, Ivy, jEdit, Lucene, Tomcat, Poi, Xerces, and Xalan, i.e., the number of modules that contain different number (1, 2, 3, 4, 5, and more than 5) of faults. The *x*-axis means the number of faults within a module (class/file), while the *y*-axis means the number of modules.

Appendix B. The normality tests of the prediction results

Tables 12–14 show the results of the normality tests. Specifically, we used the Shapiro–Wilk test to examine whether the prediction

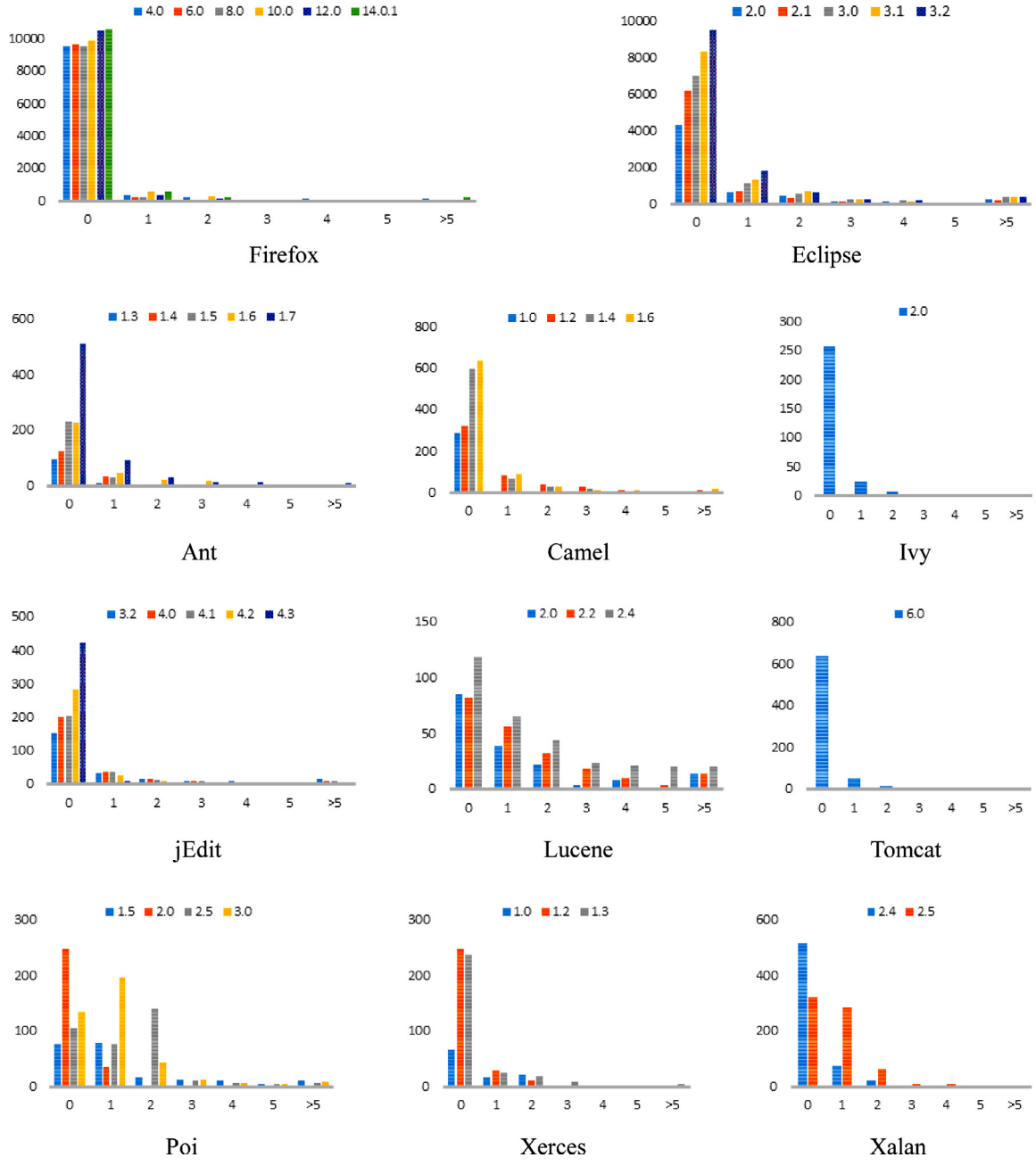


Fig. 7. The distribution of fault data.

Table 12
The *p*-values obtained from the normality tests for prediction results in RQ2.

		Firefox	Eclipse	Ant	Camel	Ivy	JEdit	Lucene	Poi	Tomcat	Xalan	Xerces
CM	ER-BPP	0.195	0.004	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.930	0.000
	ER-AVG	0.472	0.164	0.000	0.009	0.000	0.000	0.115	0.355	0.081	0.023	0.000
	$CE_{0.1}$	0.050	0.584	0.620	0.159	0.000	0.000	0.021	0.153	0.564	0.159	0.000
	$CE_{0.2}$	0.516	0.299	0.145	0.111	0.004	0.000	0.011	0.635	0.842	0.130	0.001
	$CE_{1.0}$	0.591	0.592	0.882	0.065	0.763	0.000	0.000	0.001	0.000	0.024	0.000
NM	ER-BPP	0.195	0.004	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.930	0.000
	ER-AVG	0.472	0.164	0.000	0.009	0.000	0.000	0.115	0.355	0.081	0.023	0.000
	$CE_{0.1}$	0.050	0.584	0.620	0.159	0.000	0.000	0.021	0.153	0.564	0.159	0.000
	$CE_{0.2}$	0.516	0.299	0.145	0.111	0.004	0.000	0.011	0.635	0.842	0.130	0.001
	$CE_{1.0}$	0.591	0.592	0.882	0.065	0.763	0.000	0.000	0.001	0.000	0.024	0.000
AM	ER-BPP	0.195	0.004	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.930	0.000
	ER-AVG	0.472	0.164	0.000	0.009	0.000	0.000	0.115	0.355	0.081	0.023	0.000
	$CE_{0.1}$	0.050	0.584	0.620	0.159	0.000	0.000	0.021	0.153	0.564	0.159	0.000
	$CE_{0.2}$	0.516	0.299	0.145	0.111	0.004	0.000	0.011	0.635	0.842	0.130	0.001
	$CE_{1.0}$	0.591	0.592	0.882	0.065	0.763	0.000	0.000	0.001	0.000	0.024	0.000

Table 13
The *p*-values obtained from the normality tests for prediction results in RQ3.

		Firefox	Eclipse	Ant	JEdit
CM	ER-BPP	0.745	0.562	0.393	0.576
	ER-AVG	0.109	0.062	0.355	0.054
	$CE_{0.1}$	0.004	0.034	0.196	0.013
	$CE_{0.2}$	0.004	0.008	0.777	0.028
	$CE_{1.0}$	0.283	0.529	0.079	0.171
NM	ER-BPP	0.745	0.562	0.393	0.576
	ER-AVG	0.109	0.062	0.355	0.054
	$CE_{0.1}$	0.004	0.034	0.196	0.013
	$CE_{0.2}$	0.004	0.008	0.777	0.028
	$CE_{1.0}$	0.283	0.529	0.079	0.171
AM	ER-BPP	0.745	0.562	0.393	0.576
	ER-AVG	0.109	0.062	0.355	0.054
	$CE_{0.1}$	0.004	0.034	0.196	0.013
	$CE_{0.2}$	0.004	0.008	0.777	0.028
	$CE_{1.0}$	0.283	0.529	0.079	0.171

Table 14
The *p*-values obtained from the normality tests for prediction results in RQ4.

	CM	NM	AM
ER-BPP	0.001	0.001	0.001
ER-AVG	0.000	0.000	0.000
$CE_{0.1}$	0.001	0.001	0.001
$CE_{0.2}$	0.000	0.000	0.000
$CE_{1.0}$	0.002	0.002	0.002

values (in terms of ER-BPP, ER-AVG, CE0.1, CE0.2, and CE1.0) of the three types of regression models (i.e., CM, NM, and AM) were normally distributed. The *p*-value smaller than 0.05 indicates that the prediction results do not follow the normal distribution, and are marked in gray in the tables.

Appendix C. The results of univariate logistic regression analysis

Table 15 lists the results of univariate logistic regression analysis for JEdit, Lucene, Poi, Xalan, Xerces, Ivy, and Tomcat. The column *Metric* shows the independent variable used for the univariate logistic regression. The figures in parentheses indicate the number of releases of the corresponding project under analysis. The column *p* shows the number of releases for which the univariate regression models have significant coefficients by the likelihood-ratio test. A total number of 53 tests ran on each release of all the studied projects. Since multiple tests on the same data set might result in spurious statistically significant results, we used the Benjamini–Hochberg correction of *p*-values to control false discovery. In addition, for each project, if the regression models built with a specific network measure had no significant estimated coefficients for all the releases, the corresponding entry in the column *p* is marked in gray. The column *C* indicates the sign of the significant coefficients. The sign '–' means negative coefficients, while the blank means positive coefficients. The column OR lists the average odds ratio of the network measures.

Table 15
Univariate logistic regression analysis for individual network measures.

Metric	JEdit(5)			Lucene(3)			Poi(4)			Xalan(2)			Xerces(3)			Ivy(1)			Tomcat(1)		
	p	*C*	OR	*p*	*C*	OR	*p*	*C*	OR	*p*	*C*	OR	*p*	*C*	OR	*p*	*C*	OR	*p*	*C*	OR
Size	4		1.05	3		1.10	1		1.20	2		1.03	1		1.03	1		1.05	1		1.06
Ties	4		1.01	3		1.06	1		1.03	2		1.01	1		1.02	1		1.01	1		1.02
Pairs	4		1.00	2		1.01	1		1.01	2		1.00	1		1.01	1		1.00	1		1.00
Density	2	–	0.98	1	–	0.98	1	–	0.95	1	–	0.98	2	–	0.95	0		/	1	–	0.99
nWeakComp	3		1.39	1		1.52	2		1.34	2		1.20	0		/	1		1.27	1		1.17
pWeakComp	4	–	0.98	2	–	0.99	2	–	0.98	2	–	0.99	1		1.01	1	–	0.98	1	–	0.99
2StepReach	5		1.07	3		1.04	3		1.06	2		1.04	0		/	1		1.05	1		1.15
ReachEffic	1	–	0.98	2	–	0.98	3	–	0.98	1	–	0.99	1		1.03	1	–	0.97	1	–	0.99
Broker	4		1.00	2		1.02	1		1.01	2		1.00	1		1.02	1		1.01	1		1.00

(*continued on next page*)

Table 15 (*continued*)

Metric	JEdit(5)			Lucene(3)			Poi(4)			Xalan(2)			Xerces(3)			Ivy(1)			Tomcat(1)		
	p	*C*	OR	*p*	*C*	OR	*p*	*C*	OR	*p*	*C*	OR	*p*	*C*	OR	*p*	*C*	OR	*p*	*C*	OR
nBroker	3		47.33	1		58.94	1		5.74E+04	1		154.00	1		11392.56	0		/	1		24.19
EgoBetween	4		1.02	2		1.03	2		1.06	2		1.00	1		1.01	0		/	1		1.02
nEgoBetween	4		1.04	0		/	3		1.06	1		1.03	2		1.05	0		/	0		/
Size_in	4		1.04	2		1.10	0		/	2		1.02	0		/	0		/	1		1.03
Ties_in	4		1.01	2		1.06	2		1.03	1		1.01	1		1.02	0		/	1		1.02
Pairs_in	4		1.00	1		1.02	0		/	0		/	0		/	0		/	0		/
Density_in	1		1.01	1	–	0.98	1	–	0.96	0		/	0		/	0		/	0		/
nWeakComp_in	0		/	1		1.61	1		1.86	2		1.14	0		/	0		/	0		/
pWeakComp_in	1	–	0.99	0		/	3	–	1.00	1	–	1.00	2		1.01	0		/	0		/
2StepReach_in	4		1.03	2		1.04	2		1.15	1		1.04	1		1.05	0		/	0		/
ReachEffic_in	1	–	0.99	0		/	1	–	0.98	0		/	2		1.01	0		/	0		/
Broker_in	4		1.00	1		1.05	0		/	0		/	0		/	0		/	0		/
nBroker_in	1		12.64	1		20.92	1		383.57	1		20.10	2		22.40	0		/	1		4.09
EgoBetween_in	4		1.03	0		/	1		1.14	1		1.00	0		/	0		/	1		1.03
nEgoBetween_in	5		1.04	0		/	0		/	2		1.02	0		/	1		1.05	1		1.04
Size_out	5		1.12	3		1.21	4		1.22	2		1.10	2		1.18	1		1.15	1		1.21
Ties_out	5		1.03	2		1.17	2		1.17	2		1.02	2		1.11	1		1.05	1		1.08
Pairs_out	5		1.01	2		1.03	2		1.04	2		1.01	2		1.01	1		1.01	1		1.01
Density_out	0		/	0		/	4	–	0.99	1	–	0.99	2	–	0.98	0		/	0		/
nWeakComp_out	4		2.18	3		1.47	3		1.84	1		1.43	1		1.31	1		1.75	1		1.57
pWeakComp_out	2	–	0.99	0		/	3	–	1.00	0		/	0		/	0		/	0		/
2StepReach_out	5		1.06	2		1.05	3		1.06	2		1.03	1	–	0.96	1		1.05	1		1.16
ReachEffic_out	1		1.01	0		/	1	–	0.99	0		/	0		/	0		/	0		/
Broker_out	5		1.02	2		1.07	2		1.11	2		1.01	2		1.02	1		1.01	1		1.02
nBroker_out	5		654.50	3		11.12	3		156.75	1		66.23	1		16.68	1		66.13	1		43.54
EgoBetween_out	5		1.05	0		/	1		1.20	2		1.03	0		/	1		1.09	1		1.08
nEgoBetween_out	5		1.05	0		/	0		/	1		1.04	1		1.05	0		/	1		1.03
Degree	4		1.05	3		1.10	1		1.20	2		1.03	1		1.03	1		1.05	1		1.06
OutDegree	5		1.12	3		1.21	4		1.22	2		1.10	2		1.18	1		1.15	1		1.21
InDegree	4		1.04	3		1.08	0		/	2		1.02	0		/	0		/	1		1.03
ClusCoef	4	–	0.18	1	–	0.05	1	–	0.00	1	–	0.06	2	–	0.02	0		/	1	–	0.10
Eigenvec	5		2.87E+16	3		3.69E+46	3		7.25E+31	2		2.52E+06	1	–	0.00	1		6.99E+06	1		1.28E+06
Power	3	–	0.99	0		/	3		1.17	0		/	1		1.00	1		1.15	1	–	1.00
InCloseness	3	–	0.06	1		13.39	1	–	0.00	1		1.02E+05	2		64.33	0		/	0		/
OutCloseness	5		99.54	2		4.12E+13	1		1.87E+13	2		5.66E+04	1		1.38E+03	1		158.93	1		1.19E+25
OutdwReach	5		1.04	2		1.06	1		1.03	2		1.01	1		1.04	1		1.04	1		1.03
IndwReach	1		1.03	2		1.01	0		/	2		1.01	2		1.02	0		/	0		/
Betweenness	5		1.00	2		1.01	2		1.00	2		1.00	2		1.01	0		/	1		1.00
UnBetweenness	4		1.30	3		8.56	2		2.78	2		1.34	3		1.15	1		1.89	1		1.60
EffSize	4		1.05	3		1.12	1		1.27	2		1.03	1		1.04	1		1.05	1		1.06
Efficiency	2		49.98	1		3.08	1		25.27	1		3.74	1		49.44	0		/	0		/
Constraint	4	–	0.06	2	–	0.16	2	–	0.02	2	–	0.23	1	–	0.02	1	–	0.02	1	–	0.08
Hierarchy	2		3.57	1	–	0.29	0		/	0		/	0		/	0		/	1	–	0.36
Indirects	3		25.18	2		59.95	3		53.51	2		34.39	0		/	1		100.76	1		53.66

References

[1] T. Zimmermann, N. Nagappan, Predicting defects with program dependencies, in: Proceedings of the Third International Symposium on Empirical Software Engineering and Measurement, 2009, pp. 435–438.

[2] T. Zimmermann, N. Nagappan, Predicting defects using network analysis on dependency graphs, in: Proceedings of the 30th International Conference on Software Engineering, 2008, pp. 531–540.

[3] A. Tosun, B. Turhan, A. Bener, Validation of network measures as indicators of defective modules in software systems, in: Proceedings of the Fifth International Conference on Predictor Models in Software Engineering, 2009, pp. 1–9.

[4] R. Premraj, K. Herzig, Network versus code metrics to predict defects: a replication study, in: Proceedings of the Fifth International Symposium on Empirical Software Engineering and Measurement, 2011, pp. 215–224.

[5] T.H.D. Nguyen, B. Adams, A.E. Hassan, Studying the impact of dependency network measures on software quality, in: Proceedings of the 26th International Conference on Software Maintenance, 2010, pp. 1–10.

[6] S. Prateek, A. Pasala, L.M. Aracena, Evaluating performance of network metrics for bug prediction in software, in: Proceedings of the 20th Asia-Pacific Software Engineering Conference, 2013, pp. 124–131.

[7] V.R. Basili, L.C. Briand, W.L. Melo, A validation of object-oriented design metrics as quality indicators, IEEE Trans. Softw. Eng. 22 (10) (1996) 751–761.

[8] R. Subramanyam, M.S. Krishnan, Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects, IEEE Trans. Softw. Eng. 29 (4) (2003) 297–310.

[9] N. Nagappan, T. Ball, A. Zeller, Mining metrics to predict component failures, in: Proceedings of the 28th International Conference on Software Engineering, 2006, pp. 452–461.

[10] Z. He, F. Shu, Y. Yang, M. Li, Q. Wang, An investigation on the feasibility of inter-project defect prediction, Autom. Softw. Eng. 19 (2) (2012) 167–199.

[11] F. Peters, T. Menzies, L. Gong, H. Zhang, Balancing privacy and utility in cross-company defect prediction, IEEE Trans. Softw. Eng. 39 (8) (2013) 1054–1068.

[12] B. Turhan, A. Tosun Misirli, A. Bener, Empirical evaluation of the effects of mixed project data on learning defect predictors, Inf. Softw. Technol. 55 (6) (2013) 1101–1118.

[13] P. He, B. Li, X. Liu, J. Chen, Y. Ma, An Empirical Study on Software Defect Prediction with a Simplified Metric Set, vol. 59, Wuhan University, 2014, pp. 170–190.

[14] L. Chen, B. Fang, Z. Shang, Y. Tang, Negative samples reduction in cross-company software defects prediction, Inf. Softw. Technol. 62 (2015) 67–77.

[15] M. Jureczko, L. Madeyski, Towards identifying software project clusters with regard to defect prediction, in: Proceedings of the Sixth International Conference on Predictive Models in Software Engineering, 2010, pp. 1–10.

[16] T.J. McCabe, A complexity measure, IEEE Trans. Softw. Eng. SE-2 (4) (1976) 308–320.

[17] S.R. Chidamber, C.F. Kemerer, A metrics suite for object oriented design, IEEE Trans. Softw. Eng. 20 (6) (1994) 476–493.

[18] B. Henderson-Sellers, Object-oriented Metrics: Measures of Complexity, Prentice Hall, 1996.

[19] J. Bansiya, C.G. Davis, A hierarchical model for object-oriented design quality assessment, IEEE Trans. Softw. Eng. 28 (1) (2002) 4–17.

[20] M.-H. Tang, M.-H. Kao, M.-H. Chen, An empirical study on object-oriented metrics, in: Proceedings of the Sixth International Software Metrics Symposium, 1999, pp. 242–249.

[21] R. Martin, OO design quality metrics, Qual. Eng. 8 (4) (1996) 537–542.

[22] M. Halstead, Elements of Software Science, Elsevier, 1977.

[23] A. Bayaga, Multinomial logistic regression: usage and application in risk analysis, J. Appl. Quant. Methods 5 (2) (2010) 288–297.

[24] C.-Y.J. Peng, K.L. Lee, G.M. Ingersoll, An introduction to logistic regression analysis and reporting, J. Educ. Res. 96 (1) (2002) 3–14.

[25] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, A systematic literature review on fault prediction performance in software engineering, IEEE Trans. Softw. Eng. 38 (6) (2012) 1276–1304.
[26] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, Benchmarking classification models for software defect prediction: a proposed framework and novel findings, IEEE Trans. Softw. Eng. 34 (4) (2008) 485–496.
[27] D.W. Hosmer, S. Lemeshow, Applied Logistic Regression (2000) Wiley Series in Probability and Statistics.
[28] D.A. Belsley, E. Kuh, R.E. Welsch, Regression Diagnostics: Identifying Influential Data and Sources of Collinearity, John Wiley & Sons, 2005.
[29] F.E. Harrell, Regression Modeling Strategies: With Applications to Linear Models, Logistic Regression, and Survival Analysis (2001) Springer Series in Statistics.
[30] I. Guyon, A. Elisseeff, An introduction to variable and feature selection, J. Mach. Learn. Res. 3 (2003) 1157–1182.
[31] M.H. Kutner, C. Nachtsheim, J. Neter, Applied Linear Regression Models, fourth ed., McGraw-Hill, Irwin, 2004.
[32] T. Mende, R. Koschke, Revisiting the evaluation of defect prediction models, in: Proceedings of the Fifth International Conference on Predictor Models in Software Engineering, 2009, pp. 1–10.
[33] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, A. Bener, Defect prediction from static code features: current results, limitations, new approaches, Autom. Softw. Eng. 17 (4) (2010) 375–407.
[34] T. Mende, R. Koschke, Effort-aware defect prediction models, in: Proceedings of the 14th European Conference on Software Maintenance and Reengineering, 2010, pp. 107–116.
[35] Y. Kamei, S. Matsumoto, A. Monden, K.I. Matsumoto, B. Adams, A.E. Hassan, Revisiting common bug prediction findings using effort-aware models, in: Proceedings of the 26th IEEE International Conference on Software Maintenance, 2010, pp. 1–10.
[36] Y. Kamei, E. Shihab, B. Adams, A.E. Hassan, A. Mockus, A. Sinha, et al., A large-scale empirical study of just-in-time quality assurance, IEEE Trans. Softw. Eng. 39 (6) (2013) 757–773.
[37] E. Arisholm, L.C. Briand, E.B. Johannessen, A systematic and comprehensive investigation of methods to build and evaluate fault prediction models, J. Syst. Softw. 83 (1) (2010) 2–17.
[38] Y. Shin, A. Meneely, L. Williams, J.A. Osborne, Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities, IEEE Trans. Softw. Eng. 37 (6) (2011) 772–787.
[39] E.A. Freeman, G.G. Moisen, A comparison of the performance of threshold criteria for binary classification in terms of predicted prevalence and kappa, Ecol. Model. 217 (1-2) (2008) 48–58.
[40] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, IEEE Trans. Softw. Eng. 33 (1) (2007) 2–14.
[41] Y. Zhou, B. Xu, H. Leung, L. Chen, An in-depth study of the potentially confounding effect of class size in fault prediction, ACM Trans. Softw. Eng. Methodol. 23 (1) (2014) 1–51.
[42] J.D. Gibbons, D.A. Wolfe, Nonparametric statistical inference, in: Technometrics, 2003, pp. 196–221.
[43] Y. Benjamini, Y. Hochberg, Controlling the false discovery rate: a practical and powerful approach to multiple testing, J. R. Stat. Soc. Ser. B (Methodol.) 57 (1) (1995) 289–300.
[44] G. MacBeth, E. Razumiejczyk, R. Ledsema, Cliff's delta calculator: a non-parametric effect size program for two groups of observations, Univ. Psychol. 10 (2) (2012) 545–555.
[45] J. Romano, J. Kromrey, J. Coraggio, J. Skowronek, Appropriate statistics for ordinal level data: should we really be using *t*-test and Cohen's *d* for evaluating group differences on the NSSE and other surveys? in: Annual Meeting of the Florida Association of Institutional Research, 2006, pp. 1–3.
[46] M. Hess, J. Kromrey, Robust confidence intervals for effect sizes: a comparative study of Cohen's *d* and Cliff's delta under non-normality and heterogeneous variances, Annual Meeting of the American Educational Research Association (2004) 1–30.
[47] J.D. Kromrey, K.Y. Hogarty, Analysis options for testing group differences on ordered categorical variables: an empirical investigation of type I error control and statistical power, Multiple Linear Regression Viewpoints 25 (1998) 70–82.
[48] D. Wahyudin, R. Ramler, S. Biffl, A Framework for Defect Prediction in Specific Software Project Contexts, Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 4980, LNCS, 2011, pp. 261–274.
[49] R.K. Yin, Case Study Research: Design and Methods, Essential Guide to Qualitative Methods in Organizational Research, Sage Publications, 2009.
[50] S. Kim, H. Zhang, R. Wu, L. Gong, Dealing with noise in defect prediction, in: Proceedings of the 33rd International Conference on Software Engineering, 2011, pp. 481–490.
[51] Y. Zhou, B. Xu, H. Leung, On the ability of complexity metrics to predict fault-prone classes in object-oriented systems, J. Syst. Softw. 83 (4) (2010) 660–674.
[52] Y. Zhou, H. Leung, B. Xu, Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness, IEEE Trans. Softw. Eng. 35 (5) (2009) 607–623.
[53] K. Pan, S. Kim, E. Whitehead Jr., Bug classification using program slicing metrics, in: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation, 2006, pp. 31–42.
[54] A.G. Koru, J. Tian, Comparing high-change modules and modules with the highest measurement values in two large-scale open-source products, IEEE Trans. Softw. Eng. 31 (8) (2005) 625–642.
[55] Y. Yang, Y. Zhou, H. Lu, L. Chen, Z. Chen, B. Xu, H. Leung, Z. Zhang, Are slice-based cohesion metrics actually useful in effort-aware post-release fault-proneness prediction? An empirical study, IEEE Trans. Softw. Eng. 41 (4) (2015) 331–357.
[56] F.E. Harrell, K.L. Lee, D.B. Mark, Multivariable prognostic models: issues in developing models, evaluating assumptions and adequacy, and measuring and reducing errors, Stat. Med. 15 (4) (1996) 361–387.