Using K-core Decomposition on Class Dependency Networks to Improve Bug Prediction Model's Practical Performance

Article *in* IEEE Transactions on Software Engineering · January 2019

DOI: 10.1109/TSE.2019.2892959

Yu Qu
University of California, Riverside
27 PUBLICATIONS **307** CITATIONS
SEE PROFILE

Qinghua Zheng
Xi'an Jiaotong University
500 PUBLICATIONS **6,525** CITATIONS
SEE PROFILE

Di Cui
Xidian University
18 PUBLICATIONS **185** CITATIONS
SEE PROFILE

Ting Liu
Xi'an Jiaotong University
132 PUBLICATIONS **2,384** CITATIONS
SEE PROFILE

Using K-core Decomposition on Class Dependency Networks to Improve Bug Prediction Model's Practical Performance

Yu Qu, *Member, IEEE*, Qinghua Zheng, *Member, IEEE*, Jianlei Chi, Yangxu Jin, Ancheng He, Di Cui, Hengshan Zhang, and Ting Liu, *Member, IEEE*

Abstract—In recent years, *Complex Network* theory and graph algorithms have been proved to be effective in predicting software bugs. On the other hand, as a widely-used algorithm in Complex Network theory, *k-core decomposition* has been used in software engineering domain to identify key classes. Intuitively, key classes are more likely to be buggy since they participate in more functions or have more interactions and dependencies. However, there is no existing research uses *k*-core decomposition to analyze software bugs. To fill this gap, we first use *k*-core decomposition on Class Dependency Networks to analyze software bug distribution from a new perspective. An interesting and widely existed tendency is observed: for classes in *k*-cores with larger *k* values, there is a stronger possibility for them to be buggy. Based on this observation, we then propose a simple but effective equation named as *top-core* which improves the order of classes in the suspicious class list produced by effort-aware bug prediction models. Based on an empirical study on 18 open-source Java systems, we show that the bug prediction models' performances are significantly improved in 85.2 percent experiments in the cross-validation scenario and in 80.95 percent experiments in the forward-release scenario, after using *top-core*. The models' average performances are improved by 11.5 and 12.6 percent, respectively. It is concluded that the proposed *top-core* equation can help the testers or code reviewers locate the real bugs more quickly and easily in software bug prediction practices.

Index Terms—Bug prediction, software defects, complex network, class dependency network, effort-aware bug prediction

1 Introduction

Over the past decade, *Complex Network* theory and the related graph algorithms [1], [2], [3] have been proved to be very effective in analyzing and predicting software bugs (e.g., [4], [5], [6], [7], [8], [9], [10], [11], [12], [13]). For instance, many researchers have used network metrics derived from software dependency networks to predict bugs more effectively [4], [6], [7], [8], [11], [12]. Existing researches also have designed new form of software networks [9], [13] or developed new quality metrics based on software networks [10] to improve bug prediction performances.

On the other hand, *k-core decomposition* [14] is an efficient and widely-used analyzing algorithm in Complex Network theory (for the definition of *k*-core decomposition, please refer to Section 2.1). There are many application domains of *k*-core decomposition. For instance, *k*-core decomposition has been successfully used in social network analysis [15], visualization of large networks [16], analyzing protein interaction networks [17], [18], etc.

Researchers have started using *k*-core decomposition in software engineering domain. Existing studies mainly focused on using *k*-core decomposition to identify key classes in software [19], [20], [21]. Intuitively, key classes are more likely to contain bugs since they participate in more functions or have more interactions and dependencies with other modules. However, there is no existing research using *k*-core decomposition to analyze software bugs.

To fill this gap, in this paper, we first use *k*-core decomposition to analyze software bug distribution on *Class Dependency Networks* (see Section 2.2 for more details).

Fig. 1 shows the visualizations of *k*-core decomposition and bug distribution of a widely-used open-source Java logging library – Log4j (version 1.1.3). All the networks in Fig. 1 represent Log4j's Class Dependency Network, in which each node represents a class, and the edges represent class dependency relations. Briefly speaking, *k*-core decomposition is the process in which the nodes of degree less than *k* are recursively removed from the network. Fig. 1 shows the *k*-core decomposition process on Log4j's Class Dependency Network. As shown in the figure, for each *k* value, the nodes are categorized into: *nodes in this core* and *nodes not in this core*. For the nodes in a certain core, the figure uses different colors to signify whether a node's corresponding class is buggy or not.

- Y. Qu, Q. Zheng, J. Chi, Y. Jin, A. He, D. Cui, and T. Liu are with the Ministry of Education Key Lab For Intelligent Networks and Network Security (MOEKLINNS), School of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an 710049, China. E-mail: {quyuxjtu, qhzheng, tingliu}@xjtu.edu.cn, {chijianlei, jyx530, hg19941996, cuidi}@stu.xjtu.edu.cn.
- H. Zhang is with the School of Computer Science, Xi'an University of Posts and Telecommunications, Xi'an 710121, China. E-mail: zhanghs@xupt.edu.cn.

Manuscript received 22 June 2018; revised 23 Dec. 2018; accepted 8 Jan. 2019. Date of publication 14 Jan. 2019; date of current version 11 Feb. 2021. (Corresponding author: Ting Liu.) Recommended for acceptance by T. Zimmermann. Digital Object Identifier no. 10.1109/TSE.2019.2892959

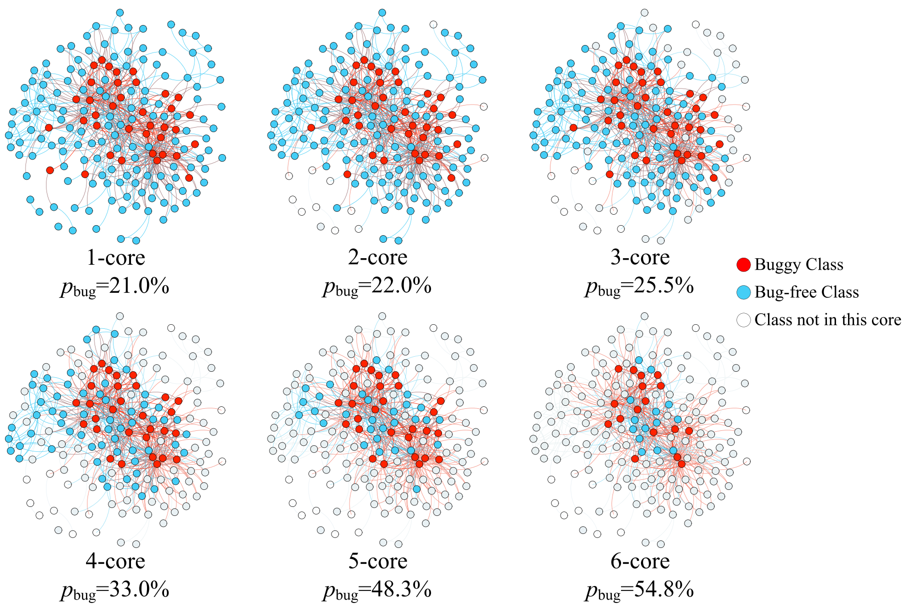


Fig. 1. The visualization of bug distribution on *k*-cores of the Class Dependency Network of Log4j, a widely-used logging library, when *k* changes from one to six (its largest possible value in Log4j's case).

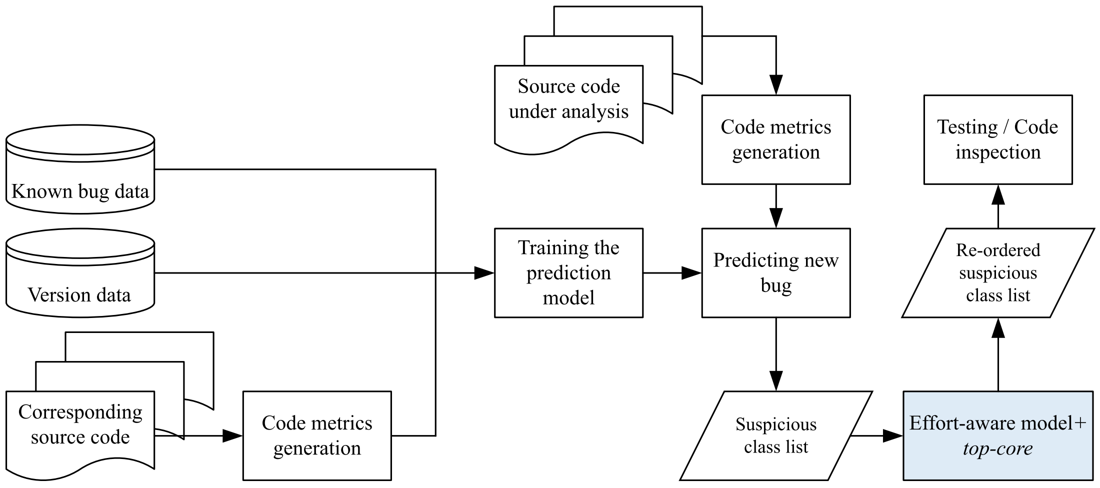

Source code under analysis
Code metrics generation
Testing / Code inspection
Known bug data
Version data
Training the prediction model
Predicting new bug
Re-ordered suspicious class list
Corresponding source code
Code metrics generation
Suspicious class list
Effort-aware model+ top-core

Fig. 2. The proposed *top-core* equation's role in bug prediction process.

Based on Fig. 1, *an interesting tendency* is observed: when k increases from one to six (the largest possible value in Log4j's case), the percentage of buggy classes (p_{bug} in Fig. 1) monotonously increases accordingly, which means that *bugs tend to cluster towards inner cores*. In other words, *for classes in k-cores with larger k values, there is a stronger possibility for them to have bugs*. We have conducted such analysis on 18 widely-used open-source software systems. Such tendency is observed in most of the cases for *all* the subject systems. We believe our study and observation will provide a new perspective to study software bugs and bug distribution. We also believe such widely existed tendency is of great value in software engineering practices.

Based on this observed tendency, we then propose a new equation named as *top-core* to improve the order of classes in the suspicious class list produced by effort-aware bug prediction models to help the testers or code reviewers locate the real bugs more quickly and easily. The intuition of *top-core* is as follows:

Bug prediction techniques can identify modules of software systems that are more likely to contain bugs. Thus, they can provide valuable aids in software engineering practices since they can guide software tester or code reviewer to allocate the limited resources to modules that are more likely to be buggy [22], [23]. As shown in Fig. 2, in software bug prediction practices, after applying some prediction model on the subject classes (when the granularity of bug prediction is *class*), the model usually outputs a suspicious class list which contains classes that are predicted to be buggy. Then practitioners can allocate their limited efforts to bug-prone classes so as to find more bugs with smaller efforts. In practice, it is usually not possible to test or inspect all the classes in the suspicious list. Thus, if an approach can improve the order of suspicious classes, it will be of great application value in bug prediction practices.

Based on the above understandings, effort-aware bug prediction models [24], [25], [26], which include the notion of effort awareness into bug prediction, have been proposed in recent years to help testers or code reviewers allocate

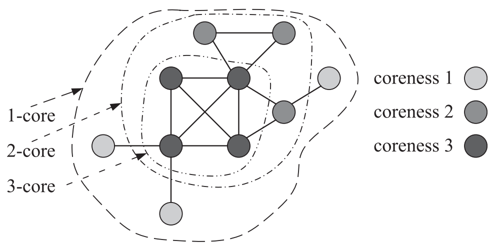


Fig. 3. *K*-core decomposition for a small network [14].

their resources more effective. Can we incorporate the observed tendency into existing effort-aware bug prediction models? In this paper, a simple but effective equation named as *top-core* is proposed to further improve the ranking of classes produced by effort-aware models. Conceptually, *top-core* increases the *relative risk*s of the classes located in the inner cores of CDN so that these classes are prioritized in the suspicious class list.

An empirical study on 18 subject systems is conducted to show the application value of *top-core*. Experimental results show that *top-core* can significantly improve the performance of bug prediction models in effort-aware scenarios. In general, when Random Forest is used as the machine learning algorithm in 96 experiments, *top-core* significantly improves existing effort-aware (baseline) model's performance in 85.2 percent experiments in the cross-validation scenario. The performance is improved by 11.5 percent in average. In the forward-release scenario, *top-core* improves baseline model's performance in 80.95 percent experiments, and the average improvement is 12.6 percent.

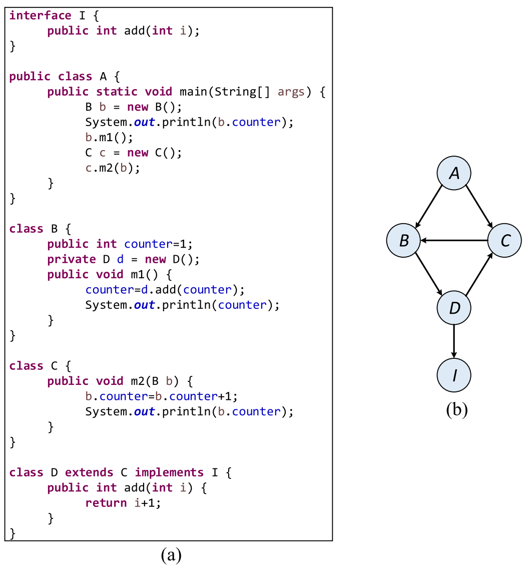
In summary we make the following contributions in this paper:

(1) We use *k*-core decomposition to analyze bug distribution on Class Dependency Networks of software. An interesting and widely existed tendency is observed: software bugs tend to cluster towards inner cores. In other words, for classes in *k*-cores with larger *k* values, there is a stronger possibility for them to be buggy.

(2) Based on the observed tendency, a new equation *top-core* is proposed to rearrange the order of classes in the suspicious class list produced by effort-aware bug prediction models to help the testers or code reviewers locate the real bugs more quickly and easily.

(3) We conduct empirical study and illustrate the effectiveness of *top-core* through effort-aware bug prediction experiments on 18 open-source systems. Experimental results show that the effort-aware bug prediction models' performances are significantly improved after using *top-core*.

The rest of this paper is organized as follows. Section 2 gives background knowledge on *k*-core decomposition and Class Dependency Network. In Section 3, *k*-core decomposition is conducted on 18 subject systems' Class Dependency Networks to analyze bug distribution from a new perspective. *Top-core* equation which rearranges the order of classes in the suspicious class list produced by effort-aware bug prediction models is introduced and discussed in Section 4. Sections 5 and 6 conduct empirical study to show effectiveness of *top-core* through effort-aware bug prediction experiments. Section 7 gives related discussions. Section 8 reviews related work. Section 9 gives the conclusions and future work.



```
interface I {
      public int add(int i);
}

public class A {
      public static void main(String[] args) {
            B b = new B();
            System.out.println(b.counter);
            b.m1();
            C c = new C();
            c.m2(b);
      }
}

class B {
      public int counter=1;
      private D d = new D();
      public void m1() {
            counter=d.add(counter);
            System.out.println(counter);
      }
}

class C {
      public void m2(B b) {
            b.counter=b.counter+1;
            System.out.println(b.counter);
      }
}

class D extends C implements I {
      public int add(int i) {
            return i+1;
      }
}
```

Fig. 4. An illustrative example Java code snippet and its Class Dependency Network.

2 Background

2.1 K-core Decomposition

In this section, we briefly introduce the definitions of *k*-core and its related concepts. For a graph (or a network) $G = (V, E)$ of $|V| = n$ nodes and $|E| = e$ edges, a *k*-core of G is defined as follows [14]:

Definition 1. *A subgraph* $H = (C, E|C)$ *induced by the set* $C \subseteq V$ *is a* k-core *or a* core of order k *if and only if* $\forall v \in C$: $\text{degree}_H(v) \geq k$, *and* H *is the maximum subgraph with this property.*

A *k*-core of G can therefore be obtained by recursively removing all the nodes of degree less than *k*, until all nodes in the remaining graph have at least degree *k*. Such process is called *k-core decomposition*.

Definition 2. *A node i has* coreness c *if it belongs to the* c-core *but not to* (c+1)*-core.*

The concept of a node's coreness has been proved to be effective to quantify and measure the node's importance [27] in complex networks.

Fig. 3 shows the process of *k*-core decomposition for a small network [14]. In Fig. 3, each closed line contains the set of nodes belonging to a certain *k*-core, while colors of the nodes represents their coreness. In this paper, we call the *k*-cores with larger *k* values as "*inner cores*" when there is no ambiguity.

The time complexity of *k*-core decomposition for G is $O(n + e)$, which means that *k*-core decomposition is a highly efficient analyzing algorithm for a network.

2.2 Class Dependency Network (CDN)

In this section, the basic concept of Class Dependency Network is introduced based on an illustrative example.

For an OO program P, its Class Dependency Network CDN_P is a directed network [28]: $CDN_P = (V, E)$, where each node $v \in V$ represents a class in P, and the edge set E represents the class dependency relations. Let c_i denotes the class that v_i refers to. Then $v_i \rightarrow v_j \in E$ if and only if c_i has at least one class dependency relation with c_j.

TABLE 1
Subject Software Systems

System	Version	SLOC	# Class	N_{CDN}	E_{CDN}	$\lvert C_{RB} \bigcap C_{CDN} \rvert$	p_{bug}	Website
Camel	1.6.0	98,962	2,193	2,140	6,209	908	8.8%	camel.apache.org
DrJava	20080106	67,958	814	792	2,385	755	16.8%	drjava.org
Eclipse JDT Core	3.4	311,316	1,796	1,728	13,868	995	11.9%	www.eclipse.org/jdt/core
Equinox framework	3.4	64,301	618	570	2,484	295	22.6%	www.eclipse.org/equinox
Genoviz	6.3	110,396	855	839	3,897	766	8.6%	sourceforge.net/projects/genoviz
HtmlUnit	2.7	93,045	808	803	3,274	603	13.4%	htmlunit.sourceforge.net
Ivy	2.0	36,636	421	418	1,851	349	9.6%	ant.apache.org/ivy
Jikes RVM	3.0.0	198,496	1,906	1,851	12,073	1,203	6.7%	www.jikesrvm.org
Jmol	6.0	29,855	291	282	648	280	27.7%	jmol.sourceforge.net
Jppf	5.0	68,765	1,621	1,412	4,550	1,143	11.3%	jppf.org
Jump	1.9.0	173,759	1,891	1,854	6,672	1,837	3.9%	openjump.org
Log4j	1.1.3	11,769	199	177	591	103	21.0%	logging.apache.org
Lucene	2.4.0	35,984	460	457	1,879	308	11.4%	lucene.apache.org
Poi	3.0	53,097	511	505	2,473	436	55.4%	poi.apache.org
Synapse	1.2	46,060	573	546	1,811	250	15.6%	synapse.apache.org
Tomcat	6.0.38	166,396	1,481	1,450	6,371	812	5.3%	tomcat.apache.org
Velocity	1.6.1	26,636	254	253	1,235	228	30.8%	velocity.apache.org
Xalan	2.6.0	155,067	1,039	1,014	6,007	860	38.7%	xalan.apache.org

Fig. 4a gives an illustrative example Java code snippet. For this snippet, Fig. 4b shows its Class Dependency Network (CDN) [28]. Although other literatures [29], [30], [31] usually only gave the processes to construct the class networks, the intrinsic nature of these networks is the same as CDN's. In CDN, nodes are classes and edges represent class dependency relations. As shown in Fig. 4, these dependency relations include aggregation (A→B, A→C, and B→D), inheritance (D→C), interface implementation (D→I), parameter types (C→B) and return types.

3 CDNs and Bug Distributions

3.1 Subject Software Systems

To empirically use *k*-core decomposition to study the bug distribution on Class Dependency Networks, a data set including 18 large open-source and widely-used Java software systems is collected, as shown in Table 1.

Camel is a versatile integration framework based on known Enterprise Integration Patterns; DrJava is a lightweight Java development environment; Eclipse JDT Core is the core component of the Eclipse IDE; Equinox framework is an implementation of the OSGi core framework specification; GenoViz is a tool for data visualization and data sharing in genomics; HtmlUnit is a Java unit testing framework for testing Web based applications; Ivy is a transitive dependency manager; Jikes RVM is a flexible open testbed to prototype virtual machine technologies; Jmol is a browser-based HTML5 viewer for chemical structures in 3D; Jppf is an open-source grid computing solution; Jump is a Geographic Information System (GIS) written in Java; Log4j is a Java-based logging library; Lucene is a searching and information retrieval library; Poi is a Java API to process Microsoft Office files; Synapse is a lightweight and high-performance Enterprise Service Bus (ESB); Tomcat is a Web server and servlet container; Velocity is a Java-based template engine that provides a template language to reference objects defined in Java code; Xalan is a library for processing XML documents.

These systems exhibit a strong heterogeneity in their sizes, design principles and application domains. Most of these systems are popular and widely used in practice, especially in internet enterprise production environments.

To ensure the reproducibility of this study, all the bug data of these subject systems is collected from publicly available software bug data repositories. Among the subject systems, the bug data of Camel, Ivy, Log4j, Poi, Synapse, Tomcat, Velocity, and Xalan is obtained from the tera-PROMISE data repository[1] [32], [33]. The bug data of Eclipse JDT Core, Equinox framework, and Lucene is from the Bug prediction dataset[2] provided by D'Ambros et al [34]. While the bug data of DrJava, GenoViz, HtmlUnit, Jmol, Jikes RVM, Jppf, and Jump is obtained from a newly released data set[3] which has been contributed by Shippey et al. in their ESEM 2016 paper [35]. We used the code and process metrics extracted and contained in these three datasets as the features in bug prediction experiments, also considering the reproducibility. Table 2 shows the code and process metrics contained in the tera-PROMISE dataset and the Bug prediction dataset. The other seven subject systems' metrics are extracted using the JHawk tool[4] [35].

Table 1 first gives the basic information of these systems. Column Version shows the version of the corresponding system that is contained in the aforementioned three bug data sets. Columns SLOC, # Class, list the **S**ource **L**ines **O**f **C**ode and the number of classes of the subject systems, respectively. The last column shows the websites of these systems. The rest of the columns in Table 1 are introduced in the following section.

3.2 Bug Distribution in k-cores

The rest of the columns in Table 1 give the statistics of subject software systems' CDNs and their bugs. The data in these columns is interpreted as follows:

First, columns N_{CDN} and E_{CDN} show the number of nodes and edges of each CDN, respectively. In the construction

1. http://openscience.us/repo/
2. http://bug.inf.usi.ch/index.php
3. https://github.com/tjshippey/ESEM2016
4. http://www.virtualmachinery.com/Jhawkmetricslist.htm

TABLE 2
Code Metrics and Process Metrics in Two Datasets

Metrics Name	Symbol
Code Metrics in the tera-PROMISE dataset [32], [33]	
Average Method Complexity	AMC
Average McCabe	Avg_CC
Afferent couplings	Ca
Cohesion Among Methods of class	CAM
Coupling Between Methods	CBM
Coupling Between Object classes	CBO
Efferent couplings	Ce
Data Access Metric	DAM
Depth of Inheritance Tree	DIT
Depth of Inheritance Coupling	IC
Lack of Cohesion in Methods	LCOM
Lack of Cohesion in Methods 3	LCOM3
Lines of Code	LOC
Maximum McCabe	Max_CC
Measure of Function Abstraction	MFA
Measure of Aggregation	MOA
Number of Children	NOC
Number of Public Methods	NPM
Response for a Class	RFC
Weighted Methods per Class	WMC
Code Metrics and Change Metrics in the Bug prediction dataset [34]	
CK metrics (i.e., CBO, DIT, LCOM, NOC, RFC, WMC)	
Lines of Code	LOC
Number of other classes referenced by the class	FanOut
Number of other classes that reference the class	FanIn
Number of Attributes	NOA
Number of Public Attributes	NOPA
Number of Private Attributes	NOPRA
Number of Attributes Inherited	NOAI
Number of Methods	NOM
Number of Public Methods	NOPM
Number of Private Methods	NOPRM
Number of Methods Inherited	NOMI
Number of revisions	NR
Number of times file has been refactored	NREF
Number of times file was involved in bug-fixing	NFIX
Number of authors who committed the file	NAUTH
Lines added and removed (sum, max, average)	LINES
Codechurn (sum, maximum and average)	CHURN
Change set size (maximum and average)	CHGSET
Age and weighted age	AGE

processes of CDNs, only when a class has certain dependency relation with other classes, its corresponding node is added to CDN. Thus, N_{CDN} is slightly smaller than the total number of classes in column # Class.

Second, column $|C_{RB} \bigcap C_{CDN}|$ shows the number of classes that appear both in CDN and the defect data sets, in the latter they have *R*ecords to signify they are *B*uggy or not. For each program, there are also some classes appeared in bug data set but not in CDN. We have manually investigated these classes and observed that most of them are classes whose SLOC equals to zero. Column p_{bug} is the percentage of buggy classes in the whole CDN.

Based on the CDNs and bug data of the subject software systems, it is possible for us to study the bug distribution on *k*-cores of CDNs. The *k*-core decomposition processes have been carried out on all the 18 subject systems' CDNs.

Fig. 5 shows the buggy classes' percentages (p_{bug}) in *k*-cores when *k* changes from one to its largest possible value in the corresponding CDN (similar to the process shown in Fig. 1). For each k value, the number of nodes (classes) in the corresponding *k*-core is given in the parentheses. A few *k* values which can be easily inferred are omitted, considering the space of *x*-axis.

Based on Fig. 5, a very interesting *tendency* can be observed: for all the subject systems, in most of the cases, when *k* increases, the percentage of buggy classes increases accordingly, which means that *software bugs tend to cluster towards inner cores* in CDN. Such tendency does not hold only for DrJava, Equinox framework, Genoviz, Tomcat, and Xalan when *k* is close to its corresponding largest value. Nevertheless, even for these five systems, the trend also exhibits itself at most of the *k* values. We think the reason for such exceptions might be that there are only quite a few nodes in inner cores for these systems, so the statistical significance no longer applies. For instance, in the figure of DrJava, the numbers of nodes in 6-core and 7-core are 91 and 12 respectively, and 12 is quite a small number to exhibit the aforementioned statistical tendency.

We believe this observation is interesting and also very useful in software engineering practices. For instance, it might be useful in bug prediction, bug localization, test case prioritization, etc. In the following part of this paper, we propose an approach based on such conclusion to re-order the suspicious class list produced by effort-aware bug prediction models. Experimental results have shown that this approach is useful in improving the practical applications of bug prediction techniques.

4 *Top-Core*: The Proposed Approach

Based on the observations in Section 3.2, we propose an equation called *top-core*, which rearranges the order of classes in the suspicious class list produced by effort-aware bug prediction models to help the testers or code reviewers locate the real bugs more quickly and easily. Since the proposed equation is based on effort-aware bug prediction models. In this section, we first give a brief introduction on effort-aware bug prediction, followed by the introduction and discussion on the proposed *top-core* equation.

4.1 Effort-Aware Bug Prediction Models

In recent years, several effort-aware bug prediction models [24], [25], [26] have been proposed to help testers or code reviewers allocate their resources more effectively. For instance, Mende and Koschke [24] for the first time proposed two models to include the notion of effort awareness into bug prediction models. One of the proposed models is R_{ad}. For a certain class c belongs to the class set C of the system under analysis, the relative risk $R_{ad}(c)$ is defined as:

$$R_{ad}(c) = p(c) \cdot \left(1 - \frac{E(c)}{E_{max}}\right),$$

where $p(c)$ is the probability that class c to be buggy, $E(c)$ is the effort (which is measured using Lines Of Code) required to inspect c, and E_{max} is the maximum value of $E(x)$, $\forall x \in C$. Then the classes belong to C are ranked according to R_{ad}. By using this model, the high-risk classes with less inspection effort are prioritized in the suspicious class list. Thus, this model significantly improved the cost effectiveness of bug prediction models.

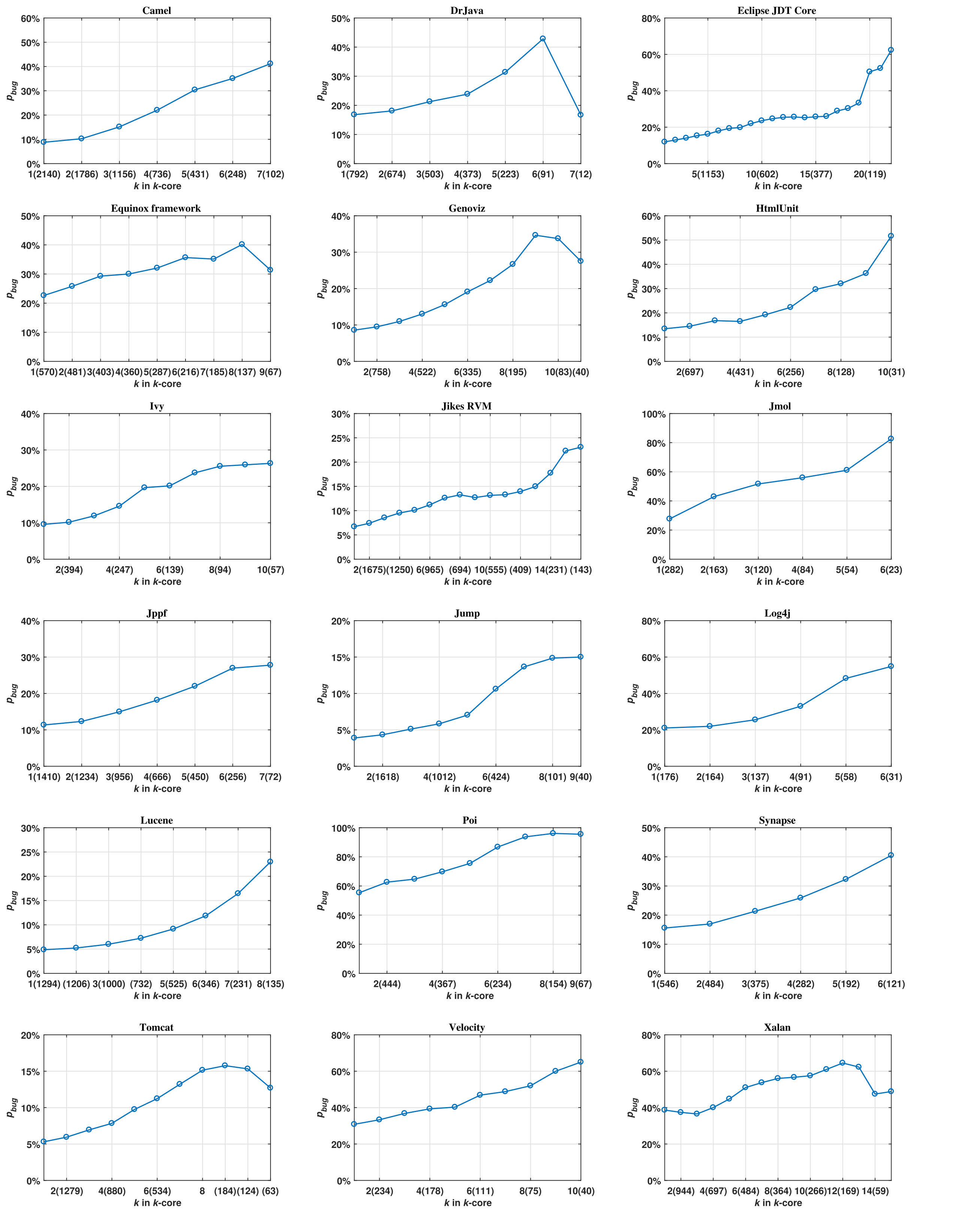


Fig. 5. The buggy classes' percentages in different k-cores of the 18 subject software systems. For each k value, the number of nodes (classes) in the corresponding k-core is given in the parentheses. A few k values which can be easily inferred are omitted, considering the space of x-axis.

Table 3 gives the definitions of three effort-aware bug prediction models [36]. In experiments, we have evaluated the performances of these three models, and observed that the model R_{ee} [26] achieved the best performance (both by itself and with *top-core*).

4.2 The Proposed *Top-Core* Equation

As mentioned in the previous section, effort-aware bug prediction models take the efforts to inspect or test the software modules into consideration, thus can improve the cost effectiveness of bug prediction models. On the other hand, as

TABLE 3
Definitions of Three Effort-Aware Bug Prediction Models [36]

Model	Prediction target	Relative risk
R_{ad} [24]	$p(c)$	$p(c) \cdot \left(1 - \frac{E(c)}{E_{max}}\right)$
R_{dd} [24], [25]	$Y(c)$	$\frac{Y(c)}{E(c)}$
R_{ee} [26]	$p(c)$	$\frac{p(c)}{E(c)}$

Note that: $p(c)$ is the probability that class c to be buggy; $Y(c)$ is the binary prediction whether class c is buggy or not; $E(c)$ is the effort required to inspect c, and E_{max} is the maximum value of $E(x), \forall x \in C$.

observed in Section 3.2 and Fig. 5, classes that are located in the inner cores of CDN are more likely to be buggy. Can we incorporate such knowledge into existing effort-aware bug prediction models? In this section, we propose a simple but effective equation – *top-core*, to further improve the ranking of classes based on the output of effort-aware models.

In *top-core*, for a certain effort-aware model R, the improved relative risk of class c is:

$$R_{top-core}(c) = R(c) \cdot coreness(c), \tag{1}$$

where $coreness(c)$ is the coreness of c in CDN, after applying *k*-core decomposition on this CDN. For instance, when the effort-aware model R_{ee} [26] (see definitions in Table 3) is used, the improved relative risk of c is:

$$R_{top-core}(c) = \frac{p(c) \cdot coreness(c)}{E(c)}. \tag{2}$$

Then classes in the suspicious class list are ranked using the improved relative risk $R_{top-core}(c)$. By using this simple equation, *top-core* further prioritizes the classes in the inner cores of CDN in the suspicious class list produced by effort-aware models, by increasing the relative risks of the classes located in the inner cores. The intuition of the multiplication in Equation 1 is similar to the intuitions of the division and multiplication used in the R_{ad}, R_{dd}, and R_{ee} models, which are used to increase the relative risks of the classes with less inspection effort. Considering effort-aware models already incorporate the predicted probability of c to be buggy (e.g., $p(c)$ in Equation (2)) and the efforts required to inspect or test c (e.g., $E(c)$ in Equation (2)). Equation (1) can produce a more reasonable ranking of classes. Such ranking includes the prediction results, the efforts, and the information of bug distribution in *k*-cores, thus can produce a more effective suspicious class list.

In a word, *top-core* rearranges the suspicious class list produced by effort-aware bug prediction model, since classes in inner cores have stronger possibilities to be buggy, as observed in Section 3.2 and Fig. 5. The purpose of this equation is to help the testers or code reviewers locate the real bugs more quickly and easily. In practice, when there are usually limited resources for testing or code inspection, the testers or code reviewers cannot test or inspect all the suspicious classes. The equation is especially useful under such circumstances.

5 Empirical Study

5.1 Experiment Design

To evaluate the effectiveness of the proposed *top-core* approach, we designed and conducted thorough bug prediction experiments applying the approach to all the 18 subject software systems. In our experiment, we investigated the effectiveness of *top-core* in the following two bug prediction scenarios: (1) cross-validation; (2) forward-release.

1) Cross-Validation. Cross-validation is a most widely-used method to evaluate bug prediction models [37]. We used threefold (3*3) cross-validation in this experiment. Specifically, in each single run of this experiment, the original bug data set is randomly split into three parts, two thirds of the instances are used to train the bug prediction model, and the rest of the instances are used to evaluate the model. For each subject system, we repeated the 3*3 cross-validation for 10 times to reduce the bias caused by the random split of instances and hence get a more realistic estimate of the proposed approach's performance.

2) Forward-Release. Forward-release prediction has been considered more suitable and practical when evaluating bug prediction approaches [12], which uses the bug data in the earlier release to predict the bug proneness in the later release of a software project. For each subject system, the experiment was repeated for 10 times to reduce the bias caused by the randomness in the experiment. It has to be noticed that the bug data of the project Eclipse JDT Core, Equinox framework and Lucene, which is from the "Bug prediction dataset", and the bug data of the project Tomcat, which is from the "tera-PROMISE dataset" (see Section 3.1), only have records for a single version. Thus, these projects cannot be used in the forward-release prediction scenario. Table 4 gives the versions of the 14 subject systems in forward-release experiment. For each system, the version used in training the bug prediction model is first given, followed by the version used in the testing process of the bug prediction model.

TABLE 4
The Versions of the Subject Systems Used in Forward-Release Prediction Scenario

Systems	Versions	Systems	Versions
Camel	1.4, 1.6	Jppf	4.2, 5.0
DrJava	20080106, 20090821	Jump	1.8, 1.9
Genoviz	6.2, 6.3	Log4j	1.0, 1.1
HtmlUnit	2.6, 2.7	Poi	2.5, 3.0
Ivy	1.4, 2.0	Synapse	1.1, 1.2
Jikes RVM	2.0, 3.0	Velocity	1.5, 1.6
Jmol	5.0, 6.0	Xalan	2.5, 2.6

5.2 Bug Prediction Models

To evaluate the proposed *top-core* equation, two widely-used machine learning techniques have been used to build the bug prediction models: Random Forest (RF) and Logistic Regression (LR). Implementation of these techniques is based on the open-source Python machine learning framework scikit-learn.[5] The *grid search* function of scikit-learn was used for tuning these two machine learning techniques [38].

1) Random Forest (RF). Random Forest [39] is a forest of many decision trees. It is an ensemble learning method which outputs the class which is the mode of the output of

5. http://scikit-learn.org/stable/

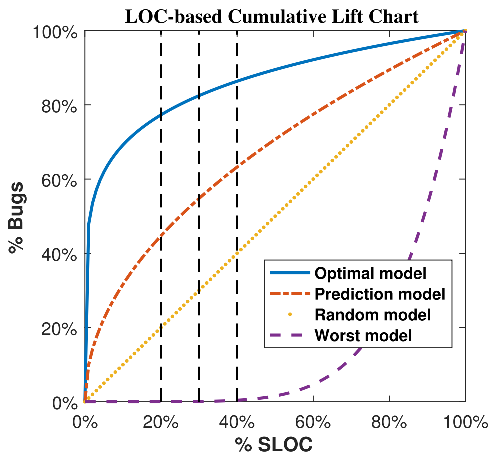


Fig. 6. LOC-based cumulative lift chart.

all individual decision trees. It can overcome the overfitting problem of decision tree.

2) Logistic Regression (LR). Logistic Regression [40] is a widely-used classification technique which measures the relationship between the binary dependent variable and one or more independent variables by estimating probabilities using a logistic function. One main benefit of using logistic regression over other types of statistical models is that its parameters have intuitive and interpretable meanings [41].

Random Forest and Logistic Regression have been widely used in software bug prediction researches [22], [42], [43], [44], [45], [46] and have been believed to have good bug prediction performances. For instance, in a systematic study on bug prediction techniques, Hall et al. concluded that Naive Bayes and Logistic Regression seem to be the techniques that are performing relatively well, although they are relatively simple compared with other machine learning techniques [22].

5.3 Evaluation Metric

A widely-used evaluation metric in effort-aware bug prediction is P_{opt} [47], which is defined as $1 - \Delta_{opt}$, where Δ_{opt} is the area between the LOC-based cumulative lift charts of the optimal model and the prediction model, as shown in Fig. 6. In LOC-based cumulative lift chart, the x-axis is considered as the cumulative percentage of SLOC of the classes selected from the suspicious class list and the y-axis is the cumulative percentage of real bugs found in these selected classes. In the optimal model, all the classes are sorted by their actual bug density in descending order. On the other hand, the worst model is built by sorting all the classes according to their actual bug density in ascending order. P_{opt} can be normalized as follows [25]:

$$P_{opt}(m) = 1 - \frac{Area(optimal) - Area(m)}{Area(optimal) - Area(worst)},$$

where $Area(optimal)$, $Area(m)$ and $Area(worst)$ represent the areas under the curves corresponding to the best model, the prediction model m and the worst model, respectively. In most of the previous researches, P_{opt} is usually measured at the effort=20% point (the leftmost vertical line in Fig. 6). In this study, P_{opt} is also measured at the effort=30% and 40% point (the other two vertical lines in Fig. 6) to provide a more thorough evaluation. P_{opt} has been widely used as the major evaluation metric in previous related works [25], [47], [48], [49], [50], [51].

5.4 Method to Handle the Class-Imbalance Problem

In software bug prediction, the bug datasets usually tend to contain much more bug-free instances (majority) than buggy instances (minority), which can also be observed in Table 1. Such *class-imbalance problem* [52] is common and can greatly influence the performance of bug prediction models. In this study, a newly proposed method, SMOTUNED [53], is used to handle the class imbalance problem. According to this recent study, in the overwhelming majority of papers (85 percent), software engineering researches use SMOTE (Synthetic Minority Oversampling Technique) [54] to fix data imbalance [53]. SMOTUNED uses Differential Evolution (DE) [55] to automatically explore the parameter space of SMOTE, and it can achieve significant improvement over SMOTE [53]. We have re-implemented SMOTUNED based on referencing to the original implementation provided by its authors[6] [53]. It should be noticed that SMOTUNED is only applied on the training dataset in the cross-validation and forward-release scenarios.

5.5 Baseline Methods and the Variant of *Top-Core*

In the empirical study, the effort-aware models in Table 3 with the two machine learning techniques in Section 5.2 are treated as the *first baseline method*. In Complex Network theory, the network measures *Betweenness Centrality* [56] and *PageRank* [57] are also widely used as metrics to quantify a node's importance. They have also been used as features in software bug prediction [4], [7]. In this study, these two network measures are also used as *baseline methods*. They are used by simply replacing $coreness(c)$ in Equation 1 with the Betweenness Centrality and PageRank values of c in CDN.

Based on the definitions in Section 2.1, it can be noticed that the concept of *coreness* is closely related to the node's *degree* in CDN. To thoroughly investigate the effects of *k*-core decomposition, we should also compare *top-core* equation with the concept of node degree. Thus, we also proposed a variant equation of *top-core* using the degree concept. The variant equation replaces the *coreness* metric in Equation 1 with the *degree* metric. The variant equation is named as *top-degree*. For instance, when the effort-aware model R_{ee} is used, *top-degree* is formalized as:

$$R_{top-degree}(c) = \frac{p(c) \cdot degree(c)}{E(c)}, \tag{3}$$

where $degree(c)$ is the node degree of c in CDN. In experiments, *top-degree* is also evaluated. The performances of all the methods are evaluated using the P_{opt} metric introduced in Section 5.3.

6 Experimental Results

In this section, the results of the aforementioned experiments are given. First, the results and discussions of the

6. http://tiny.cc/smotuned

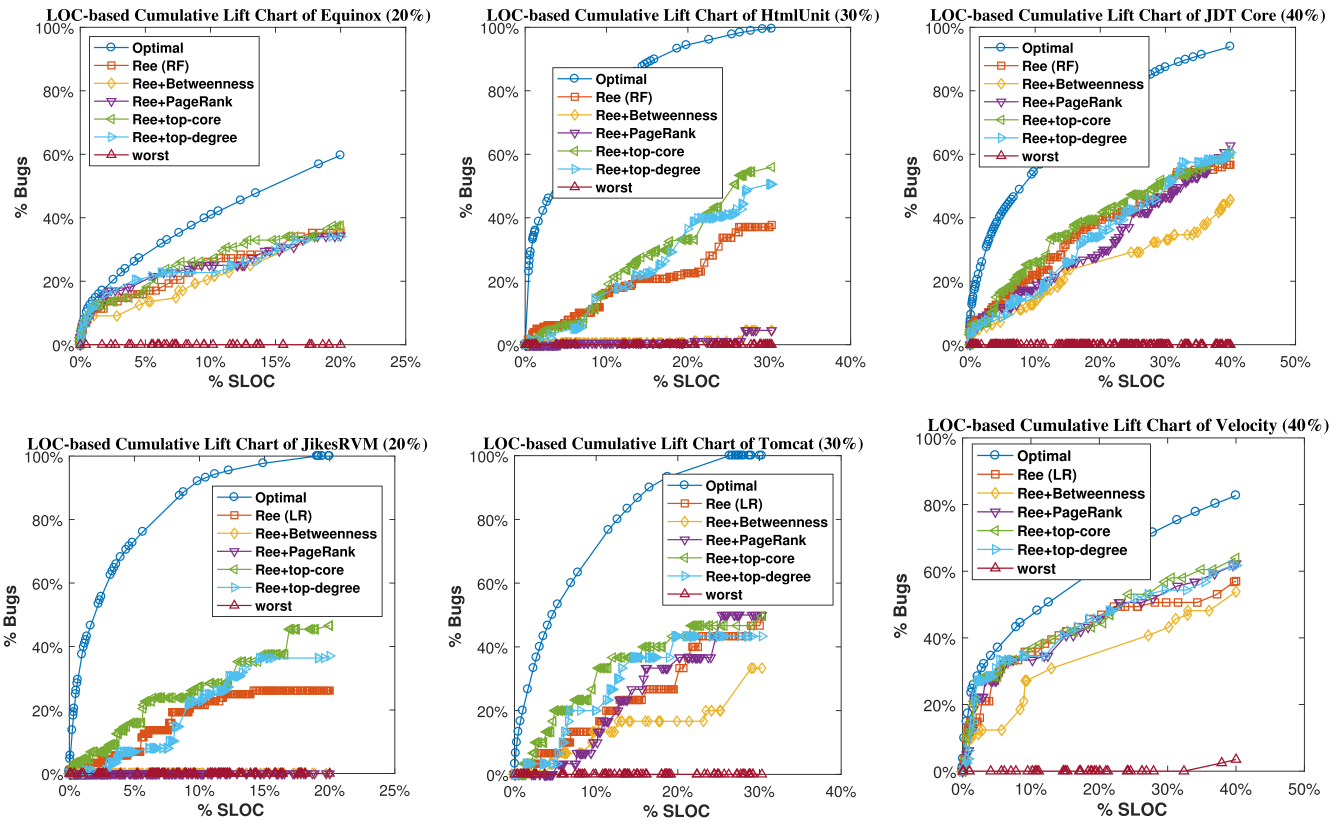


Fig. 7. LOC-based cumulative lift charts of six subject systems using Random Forest (RF) and Logistic Regression (LR) at different efforts levels in the *cross-validation* scenario.

cross-validation scenario are shown, followed by the results and discussions of the forward-release scenario.

6.1 Cross-validation

Fig. 7 shows the LOC-based cumulative lift charts (see the definitions in Section 5.3), in one simulation in the cross-validation scenario, of six subject systems using the two machine learning techniques and the effort-aware model R_{ee} with different methods at different efforts levels, respectively.

For each subject system, the figure shows the curves of the optimal model, the worst model, the R_{ee} model, and the models that combine R_{ee} with different measures using Equation (1) (e.g., "R_{ee}+top-core" and "R_{ee}+top-degree" in Fig. 7), respectively. Based on Fig. 7, it can be observed that the models that combine R_{ee} with *top-core* and *top-degree* usually achieve the best performances, compared with the models using Betweenness Centrality (abbreviated to "Betweenness" or "Between.") and PageRank (abbreviated to "Page."), and R_{ee} alone.

Fig. 7 shows the results of one simulation. As mentioned in Section 5.1, each threefold (3*3) cross-validation experiment has been conducted for 10 times to reduce the bias caused by the randomness. Fig. 8 gives the box plots of P_{opt}s in 10 experiments of these six subject systems at different efforts levels, respectively. It can be observed that in these Figures, the models R_{ee}+top-core and R_{ee}+top-degree still achieve the best performance, which is in line with the results in Fig. 7.

Tables 6, 7, 8 list the mean values of P_{opt}s in the aforementioned experiments for all the 18 subject systems at different efforts levels, respectively. For each system, the tables first give the mean values of P_{opt}s of different models. Largest P_{opt} value in each row is in bold. In each experiment, the Wilcoxon signed-rank test is used to investigate the statistical significance of the differences of: 1) top-degree versus R_{ee} (e.g., degree versus R_{ee}), 2) top-core versus R_{ee}, and 3) top-degree versus top-core. In each test, $>$ means the former is statistically greater than the latter, and $<$ vice versa. The $=$ means there is no statistical difference between the two group of results, which is to say the null hypothesis in the Wilcoxon signed-rank test cannot be rejected. ** means the 0.01 significant level, which represents that the p-value<0.01 in the Wilcoxon signed-rank test. While * means the 0.05 significant level. The row Average in each table gives the average values of P_{opt}s of different models. Also the percentage of difference of a certain model compared with R_{ee} is shown. The last row in each table summarizes the Win/Tie/Loss analysis results when comparing each measure with the baseline model R_{ee} based on quantitative results and statistical tests.

We take Table 6 as an example to analyze the results in Tables 6 , 7, 8. Based on Table 6, it can be observed that for the 18 subject systems, *top-core* significantly improves the effort-aware model R_{ee} for 14 subject systems (Win: 14). There is only one subject system for which the performance is worse than the original R_{ee} model (Loss: 1). In average, *top-core* improves the model's performance by 11.02 percent. Although *top-degree* achieves better performance in term of the average value (11.81 percent). Based on the the Win/Tie/Loss results, it can be observed that *top-degree* is *less stable* than *top-core* since for the former, there are only nine

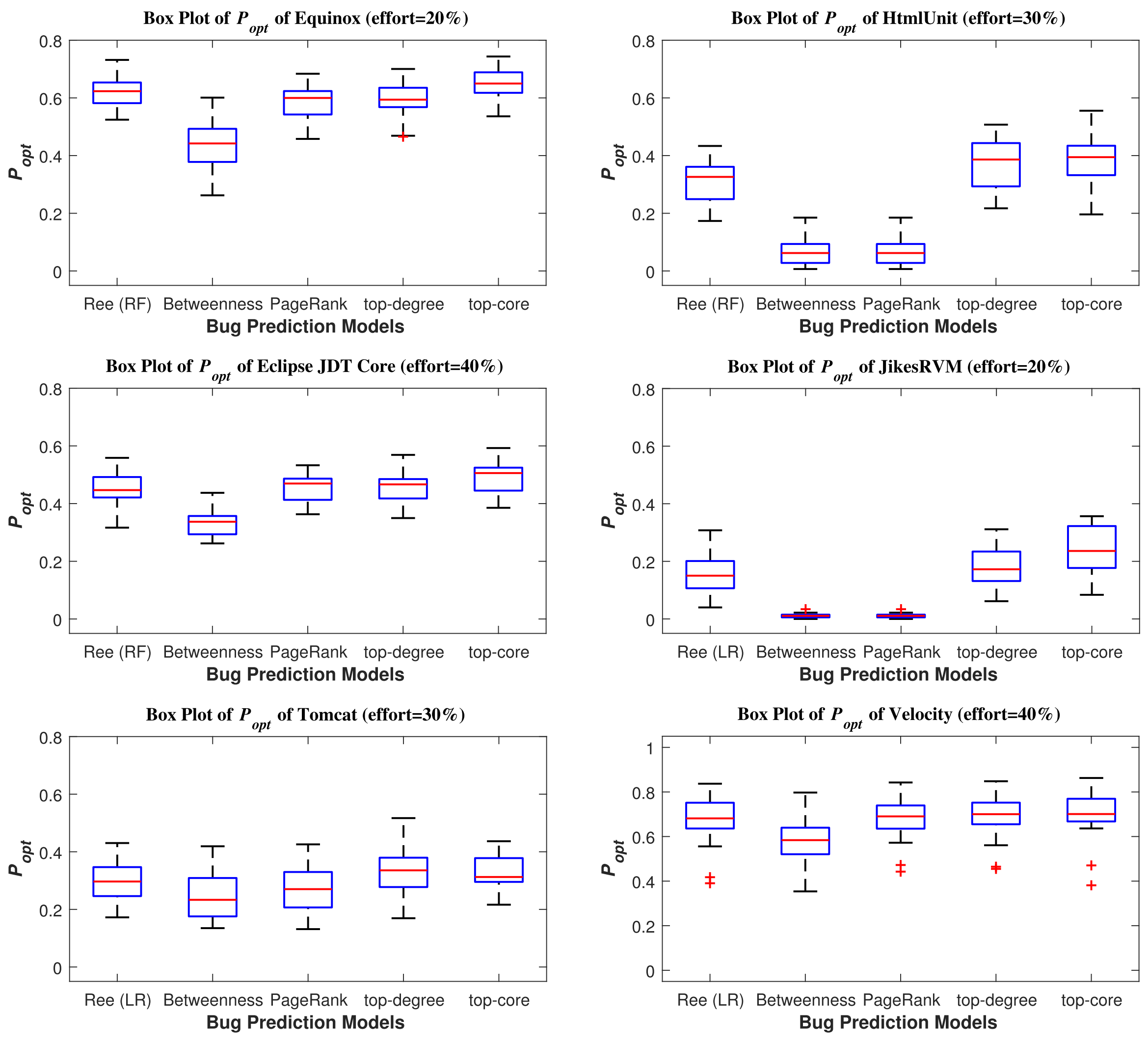


Fig. 8. Box plots of P_{opt}s in six experiments using Random Forest (RF) and Logistic Regression (LR) at different efforts levels in the *cross-validation* scenario.

systems (Win: 9) for which the performance is improved. There are also four subject systems (Loss: 4) for which the performance of *top-degree* is worse than the R_{ee} model.

Table 7 shows the results of P_{opt}s when using Logistic Regression at the efforts level 30 percent. And Table 8 gives the results when using Random Forest at the efforts level 40 percent. Similar results can be observed in these two tables. For instance, in Table 8, there are 16 subject systems (Win: 16) for which the performance is statistically significantly improved when using *top-core*.

In general, when we summarize the results of Random Forest (of which efforts level 30 percent's results are not shown), there are 54 (18×3) experiments in total, and in 85.2 percent $((14 + 16 + 16)/54)$ experiments, *top-core* statistically significantly outperforms baseline methods, while only in 5.6 percent $(3/54)$ experiments *top-core* introduces performance degradations. In average, bug prediction model's performance is improved by 11.5 percent.

To provide a more thorough evaluation on the proposed equation. Similar to Tables 6, 7, 8, Table 5 shows the results when using the effort-aware model R_{dd} and Random Forest at efforts level 20 percent. Due to the limited space, statistical results are omitted in Table 5. Briefly speaking, similar results can be observed: based on the mean values of P_{opt}s,

TABLE 5
Comparison of P_{opt} When Using R_{dd} and Random Forest in the Cross-Validation Scenario (effort=20%)

System	R_{dd}	top-degree	top-core
Camel	0.348	**0.398**	0.376
DrJava	0.333	**0.443**	0.380
Eclipse JDT Core	0.327	0.324	**0.363**
Equinox Framework	0.559	0.540	**0.594**
Genoviz	0.342	**0.350**	0.342
HtmlUnit	0.256	0.287	**0.312**
Ivy	0.191	0.200	**0.203**
Jikes RVM	0.137	**0.242**	0.217
Jmol	0.335	**0.474**	0.382
Jppf	0.366	0.362	**0.389**
Jump	0.159	**0.162**	**0.162**
Log4j	0.384	**0.603**	0.480
Lucene	0.378	**0.424**	0.404
Poi	0.535	0.493	**0.564**
Synapse	0.344	**0.358**	0.345
Tomcat	**0.244**	0.212	0.231
Velocity	0.453	0.447	**0.469**
Xalan	**0.498**	0.448	0.471
Average	**0.344**	**0.376 +9.30%**	**0.371 +7.85%**
Win/Tie/Loss		**7/9/2**	**12/5/1**

TABLE 6
Comparison of P_{opt} When Using Random Forest in the Cross-Validation Scenario (effort=20%)

System	R_{ee}	Betweenness	PageRank	top-degree	top-core	degree vs. R_{ee}	core vs. R_{ee}	degree vs. core
Camel	0.423	0.323	0.415	**0.492**	0.466	>, **	>, **	>, **
DrJava	0.299	0.224	0.224	**0.403**	0.334	>, **	>, **	>, **
Eclipse JDT Core	0.384	0.233	0.356	0.343	**0.411**	<, **	>, **	<, **
Equinox Framework	0.622	0.438	0.582	0.592	**0.649**	<, **	>, **	<, **
Genoviz	0.237	0.104	0.125	**0.355**	0.352	>, **	>, **	=
HtmlUnit	0.238	0.050	0.050	0.270	**0.300**	>, **	>, **	<, **
Ivy	0.192	0.228	0.221	**0.246**	0.221	>, **	>, *	=
Jikes RVM	0.199	0.007	0.007	0.192	**0.266**	=	>, **	<, **
Jmol	0.285	0.206	0.207	**0.470**	0.377	>, **	>, **	>, **
Jppf	0.354	0.108	0.108	0.335	**0.372**	=	=	<, **
Jump	0.264	0.145	0.130	**0.344**	0.324	>, **	>, **	>, **
Log4j	0.395	0.349	0.388	**0.682**	0.521	>, **	>, **	>, **
Lucene	0.443	0.461	0.498	**0.514**	0.470	>, **	>, **	>, **
Poi	0.623	0.237	0.392	0.560	**0.628**	<, **	=	<, **
Synapse	0.439	0.408	**0.467**	0.464	0.446	=	=	=
Tomcat	0.259	0.186	0.197	0.276	**0.282**	=	>, **	=
Velocity	0.553	0.432	0.551	0.560	**0.586**	=	>, **	<, **
Xalan	**0.654**	0.414	0.601	0.562	0.607	<, **	<, **	<, **
Average	**0.381**	**0.253** **−33.60%**	**0.307** **−19.42%**	**0.426** **+11.81%**	**0.423** **+11.02%**			
Win/Tie/Loss		**0/4/14**	**1/5/12**	**9/5/4**	**14/3/1**			

TABLE 7
Comparison of P_{opt} When Using Logistic Regression in the Cross-Validation Scenario (effort=30%)

System	R_{ee}	Betweenness	PageRank	top-degree	top-core	degree vs. R_{ee}	core vs. R_{ee}	degree vs. core
Camel	0.431	0.356	0.455	**0.514**	0.489	>, **	>, **	>, **
DrJava	0.276	0.211	0.211	**0.440**	0.333	>, **	>, **	>, **
Eclipse JDT Core	0.396	0.305	0.415	0.416	**0.425**	>, *	>, **	=
Equinox Framework	0.592	0.472	0.576	0.600	**0.632**	=	>, **	<, **
Genoviz	0.398	0.245	0.266	0.477	**0.483**	>, **	>, **	=
HtmlUnit	0.311	0.082	0.082	0.372	**0.375**	>, **	>, **	=
Ivy	0.253	**0.292**	0.236	0.258	0.252	=	=	=
Jikes RVM	0.211	0.011	0.011	0.264	**0.315**	>, **	>, **	<, **
Jmol	0.346	0.283	0.327	**0.529**	0.423	>, **	>, **	>, **
Jppf	0.374	0.120	0.120	0.410	**0.421**	>, **	>, **	=
Jump	0.231	0.213	0.196	**0.382**	0.320	>, **	>, **	>, **
Log4j	0.413	0.501	0.558	**0.655**	0.505	>, **	>, **	>, **
Lucene	0.472	0.529	0.549	**0.568**	0.509	>, **	>, **	>, **
Poi	0.640	0.309	0.485	0.594	**0.647**	<, **	=	<, **
Synapse	0.435	0.437	**0.474**	0.451	0.429	=	=	=
Tomcat	0.300	0.241	0.267	0.330	**0.333**	=	>, *	=
Velocity	0.642	0.522	0.633	0.654	**0.665**	=	>, *	=
Xalan	**0.664**	0.471	0.642	0.600	0.648	<, **	<, **	<, **
Average	**0.410**	**0.311** **−24.15%**	**0.361** **−11.95%**	**0.473** **+15.37%**	**0.456** **+11.22%**			
Win/Tie/Loss		**2/4/12**	**4/4/10**	**11/5/2**	**14/3/1**			

top-core and *top-degree* significantly improve the baseline model's performance. Based on the Win/Tie/Loss results, it can be concluded *top-core* performs better than *top-degree* when R_{dd} is used.

In a word, we can conclude that in the cross-validation scenario, *top-core* and its variant model *top-degree* usually significantly outperform the baseline methods. In most of the experiments, the mean value of P_{opt} is statistically significantly improved after using the proposed *top-core* equation. Although *top-degree* performs slightly better than *top-core* in terms of average P_{opt} values, we think average values are coarse-grained measures which might lose some important information. The Win/Tie/Loss results show that *top-core* outperforms *top-degree*, and *top-degree* is less stable than *top-core*. Thus, we recommend practitioners to use *top-core* in software defect prediction practices.

6.2 Forward-Release

Discussions on the experimental results in the forward-release scenario are similar to those in the cross-validation scenario. Similarly, Fig. 9 shows the LOC-based cumulative lift charts, in one simulation, of three subject systems using

TABLE 8
Comparison of P_{opt} When Using Random Forest in the Cross-Validation Scenario (effort=40%)

System	R_{ee}	Betweenness	PageRank	top-degree	top-core	degree vs. R_{ee}	core vs. R_{ee}	degree vs. core
Camel	0.476	0.394	0.518	**0.571**	0.539	$>$, **	$>$, **	$>$, **
DrJava	0.367	0.244	0.244	**0.540**	0.440	$>$, **	$>$, **	$>$, **
Eclipse JDT Core	0.450	0.337	0.457	0.459	**0.489**	$=$	$>$, **	$<$, **
Equinox Framework	0.614	0.502	0.612	0.639	**0.655**	$>$, **	$>$, **	$<$, **
Genoviz	0.412	0.295	0.340	**0.519**	0.507	$>$, **	$>$, **	$=$
HtmlUnit	0.372	0.088	0.088	0.453	**0.465**	$>$, **	$>$, **	$<$, *
Ivy	0.271	0.340	0.313	**0.343**	0.333	$>$, **	$>$, **	$=$
Jikes RVM	0.295	0.009	0.010	0.389	**0.393**	$>$, **	$>$, **	$=$
Jmol	0.363	0.311	0.381	**0.549**	0.447	$>$, **	$>$, **	$>$, **
Jppf	0.470	0.164	0.176	0.500	**0.541**	$>$, **	$>$, **	$<$, **
Jump	0.385	0.301	0.301	**0.501**	0.447	$>$, **	$>$, **	$>$, **
Log4j	0.509	0.531	0.579	**0.739**	0.606	$>$, **	$>$, **	$>$, **
Lucene	0.557	0.587	0.637	**0.652**	0.602	$>$, **	$>$, **	$>$, **
Poi	**0.678**	0.369	0.560	0.652	0.673	$<$, *	$=$	$<$, **
Synapse	0.485	0.492	**0.530**	0.518	0.501	$=$	$>$, *	$=$
Tomcat	0.360	0.293	0.337	**0.404**	0.391	$>$, *	$>$, **	$=$
Velocity	0.633	0.530	0.651	**0.664**	**0.664**	$>$, *	$>$, **	$=$
Xalan	**0.710**	0.527	0.690	0.640	0.685	$<$, **	$<$, **	$<$, **
Average	**0.467**	**0.351** **−24.84%**	**0.412** **−11.78%**	**0.541** **+15.85%**	**0.521** **+11.56%**			
Win/Tie/Loss		**2/3/13**	**5/5/8**	**14/3/1**	**16/1/1**			

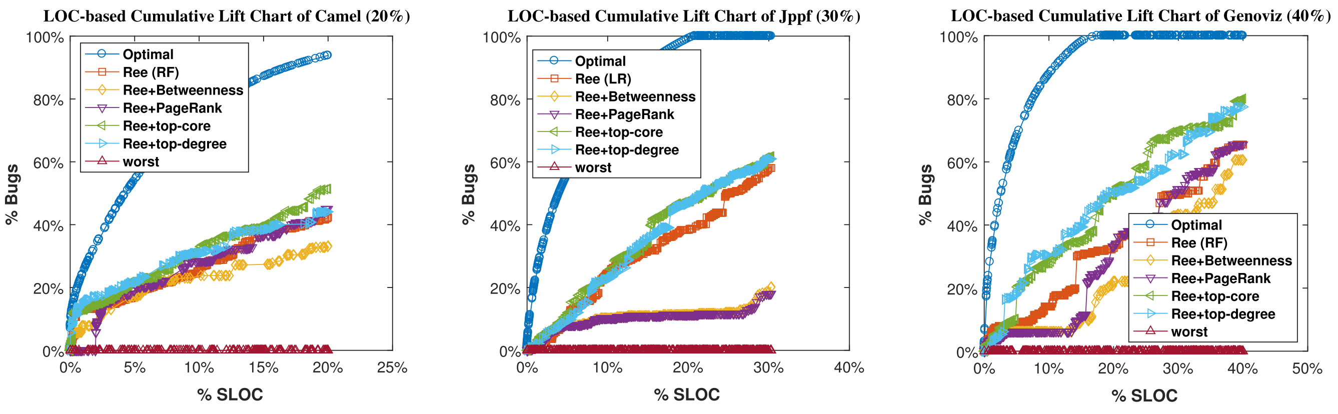


Fig. 9. LOC-based cumulative lift charts of three subject systems using Random Forest (RF) and Logistic Regression (LR) at different efforts levels in the *forward-release* scenario.

different models at different efforts levels, respectively. Similar to the results in Fig. 7, it can be observed that the models that combine R_{ee} with *top-core* and *top-degree* usually achieve the best performances, compared with other baseline network measures, and R_{ee} alone.

Similar to the experiments in the cross-validation scenario, experiments in the forward-release scenario have been conducted for 10 times to reduce the bias caused by the randomness in the experiments. Fig. 10 shows the box plots of P_{opt}s in 10 experiments of these three subject systems, respectively.

Based on Fig. 10, it can be concluded that for these three subject systems, the models R_{ee}+top-degree and R_{ee}+top-core still achieve better performances, which is in line with Fig. 9.

Similar to Tables 6 , 7, 8, Table 9 gives the mean values of P_{opt}s for 10 simulations of each subject systems in the forward-release scenario when using Random Forest and Logistic Regression. Since the only randomness in the forward-release scenario is in the SMOTUNED algorithm, there is little difference between results in each simulation. Thus, most of the results' differences are statistically significant between each pair of models. So the statistical tests' results are omitted in Table 9.

In Table 9, the mean values of P_{opt}s are given at different efforts levels. Still, we conduct similar analysis on Table 9: in each row for one of the machine learning algorithms, the largest P_{opt} value is in bold. In Table 9, the results of *top-core* and *top-degree* are in gray background when they are smaller than the results of R_{ee}, which means that the two equations introduce degradations in the corresponding experiments.

If we treat each row in Table 9 when using Random Forest as an independent experiment, then we have 42 (14×3) experiments in total. Then there are 80.95 percent (34/42) experiments in which *top-core* outperforms R_{ee}, and there are 71.4 percent (30/42, see the Win/Tie/Loss results in the last row in Table 9) experiments in which *top-degree* outperforms R_{ee}. For the experiments in which *top-core* or *top-degree* introduce performance degradations, the average degradation of *top-degree* (compared with R_{ee}) is

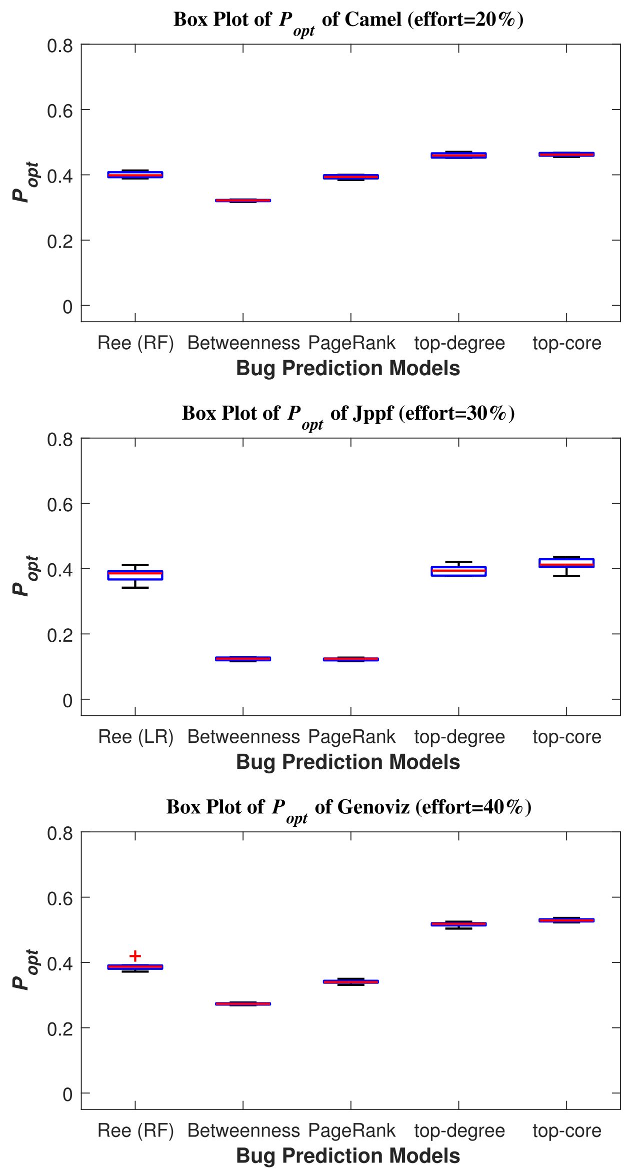


Fig. 10. Box plots of P_{opt}s in three experiments using Random Forest (RF) and Logistic Regression (LR) at different efforts levels in the *forward-release* scenario.

11.75 percent, while the average degradation of *top-core* is 8.04 percent.

Results of Logistic Regression are similar to those of Random Forest. There are 80.95 percent (34/42) experiments in which *top-core* outperform R_{ee}. The average degradations of *top-degree* and *top-core* are 10.21 and 7.67 percent, respectively.

Generally speaking, results in Table 9 are not as significant as those in Table 6, 7, 8, since there are 19 percent experiments in which the original effort aware model R_{ee} performs better than *top-core*. However, there are still 80.95 percent experiments in which *top-core* achieves a better performance, and in average for Random Forest, bug prediction model's performance is improved by 12.6 percent when using *top-core*.

Considering the performance degradations, still we recommend using *top-core* in practice since it introduces acceptable performance degradations (8.04 and 7.67 percent) in a few situations. Moreover, the Win/Tie/Loss results shows that *top-core* outperforms *top-degree* in the forward-release scenario.

To sum up, based on the experimental results in the cross-validation and forward-release scenarios, it can be concluded that the proposed *top-core* equation can significantly improve the practical application of effort-aware bug prediction models. Although *top-degree* outperforms *top-core* in terms of the average value of P_{opt}, the Win/Tie/Loss results signify that *top-core* outperforms *top-degree* in both scenarios. Thus, we recommend practitioners always use *top-core* in software defect prediction practices.

7 Discussion

7.1 The Widely Existed Tendency

In this paper, we report an interesting and widely existed tendency: for classes in k-cores of CDN with larger k values, there is a stronger possibility for them to have bugs. We have also conducted the analysis on some newly published software bug datasets. For instance, Table 10 shows the basic statistics on three subject system from a new software bug data repository provided by Ferenc et al. [58] in 2018, which is called the GitHub Bug Dataset. Fig. 11 shows the buggy classes' percentages in different k-cores of these three subject software systems. It is interesting to observe that these subject systems also exhibit such tendency. We believe the proposed *top-core* equation is also useful for this bug data repository.

Although the proposed *top-core* equation in this paper is mainly based on effort-aware models. In our ongoing research, it is observed that the basic idea of *top-core* can also be combined with unsupervised models, e.g., ManualDown and ManualUp [59]. Promising preliminary results have been observed. For instance, it can improve ManualDown in the bug classification scenario. We also encourage the community and practitioners to further evaluate applications of the observed tendency and the proposed method.

7.2 Comparison of the Notions of Coreness and Degree

As discussed earlier, the notion of *coreness* is closely related to the node's *degree*. It is also observed that, the variant equation *top-degree* indeed outperforms *top-core* in terms of average P_{opt} values, although the former is less stable and the latter has better Win/Tie/Loss results.

It is necessary to investigate whether the observed tendency still exists when degree is analyzed in a similar way. Fig. 12 shows the results of such analysis on Camel, Synapse, and Velocity. In each figure, the x-axis is the node's degree in CDN, the y-axis is the percentage of buggy classes in the classes whose degree is greater than or equal to the corresponding degree. It can be noticed that there is no clear common tendency. Similar results exist for other subject systems.

We think the performance difference between the notions of coreness and degree is partially because of that coreness uses a more *conservative* way to quantify the importance of a node in CDN. For instance, the largest node degree in Camel's CDN is 497 (as shown in Fig. 12), whereas the largest k value in k-core decomposition on Camel's CDN is only 7 (as shown in Fig. 5).

TABLE 9
Comparison of P_{opt} in the Forward-Release Scenario

System	P_{opt}s when using Random Forest in the forward-release scenario					P_{opt}s when using Logistic Regression in the forward-release scenario				
	R_{ee}	Between.	Page.	top-degree	top-core	R_{ee}	Between.	Page.	top-degree	top-core
effort=20%										
Camel	0.400	0.321	0.393	0.460	**0.462**	0.382	0.332	0.382	**0.453**	0.434
DrJava	0.174	**0.205**	**0.205**	0.140	0.144	0.136	**0.176**	**0.176**	0.122	0.125
Genoviz	0.211	0.102	0.131	**0.374**	0.351	0.238	0.101	0.125	0.366	**0.383**
HtmlUnit	0.187	0.083	0.083	0.237	**0.243**	0.212	0.070	0.068	0.241	**0.247**
Ivy	0.182	0.205	0.239	**0.264**	0.222	0.162	0.169	0.188	**0.219**	0.195
Jikes RVM	0.133	0.004	0.004	0.124	**0.160**	0.142	0.009	0.009	**0.211**	0.184
Jmol	0.198	0.249	0.237	**0.455**	0.289	0.188	0.293	0.244	**0.456**	0.247
Jppf	0.264	0.094	0.094	0.275	**0.301**	0.307	0.122	0.122	0.313	**0.340**
Jump	0.212	0.150	0.130	**0.359**	0.252	0.084	0.068	0.061	**0.210**	0.120
Log4j	0.455	0.349	0.388	**0.676**	0.575	0.412	0.372	0.397	**0.675**	0.525
Poi	**0.583**	0.229	0.382	0.538	**0.583**	0.571	0.210	0.389	0.546	**0.582**
Synapse	0.396	0.400	**0.458**	**0.458**	0.418	0.345	0.383	0.442	**0.444**	0.371
Velocity	0.512	0.421	0.531	0.550	**0.570**	0.494	0.430	0.518	0.549	**0.566**
Xalan	**0.634**	0.428	0.613	0.582	0.615	**0.625**	0.434	0.608	0.586	0.610
Average	**0.324**	**0.231** **−28.70%**	**0.278** **−14.20%**	**0.392** **+20.99%**	**0.370** **+14.20%**	**0.307**	**0.226** **−26.38%**	**0.266** **−13.36%**	**0.385** **+25.41%**	**0.352** **+14.66%**
effort=30%										
Camel	0.442	0.360	0.456	**0.512**	0.508	0.418	0.357	0.440	**0.493**	0.470
DrJava	0.223	**0.304**	0.303	0.170	0.181	0.200	**0.258**	**0.258**	0.155	0.159
Genoviz	0.295	0.187	0.245	0.443	**0.456**	0.364	0.202	0.249	0.454	**0.471**
HtmlUnit	0.318	0.096	0.095	0.326	**0.336**	0.317	0.072	0.068	0.334	**0.344**
Ivy	0.200	0.275	0.278	**0.299**	0.280	0.186	0.208	0.210	**0.236**	0.220
Jikes RVM	0.181	0.005	0.005	**0.265**	0.248	0.202	0.009	0.009	**0.322**	0.262
Jmol	0.264	0.283	0.302	**0.457**	0.364	0.268	0.320	0.329	**0.453**	0.332
Jppf	0.338	0.105	0.104	0.364	**0.388**	0.382	0.123	0.123	0.394	**0.413**
Jump	0.276	0.251	0.250	**0.423**	0.327	0.145	0.156	0.150	**0.306**	0.201
Log4j	0.530	0.454	0.501	**0.699**	0.609	0.463	0.464	0.507	**0.692**	0.578
Poi	**0.631**	0.320	0.494	0.609	0.620	**0.620**	0.316	0.490	0.591	0.617
Synapse	0.418	0.446	0.475	**0.467**	0.427	0.377	0.431	**0.454**	0.444	0.396
Velocity	0.569	0.485	0.592	0.612	**0.624**	0.561	0.496	0.585	0.608	**0.616**
Xalan	**0.676**	0.494	0.658	0.617	0.656	**0.663**	0.493	0.661	0.622	0.657
Average	**0.383**	**0.290** **−24.28%**	**0.340** **−10.53%**	**0.447** **+16.71%**	**0.430** **+12.27%**	**0.369**	**0.279** **−24.39%**	**0.324** **−12.20%**	**0.436** **+18.16%**	**0.410** **+11.11%**
effort=40%										
Camel	0.478	0.394	0.507	**0.553**	0.543	0.440	0.388	0.484	**0.536**	0.506
DrJava	0.294	**0.385**	0.384	0.204	0.218	0.264	**0.322**	**0.322**	0.187	0.191
Genoviz	0.387	0.273	0.340	0.517	**0.529**	0.451	0.308	0.348	0.533	**0.540**
HtmlUnit	0.407	0.109	0.107	0.432	**0.443**	0.392	0.079	0.071	0.413	**0.419**
Ivy	0.243	0.321	0.335	**0.358**	0.335	0.226	0.269	0.262	**0.290**	0.270
Jikes RVM	0.228	0.005	0.005	**0.383**	0.330	0.252	0.017	0.016	**0.424**	0.357
Jmol	0.323	0.317	0.354	**0.474**	0.409	0.287	0.340	0.345	**0.433**	0.364
Jppf	0.426	0.152	0.153	0.442	**0.475**	0.439	0.171	0.172	0.468	**0.484**
Jump	0.343	0.338	0.342	**0.487**	0.419	0.201	0.223	0.194	**0.334**	0.263
Log4j	0.532	0.504	0.564	**0.727**	0.613	0.512	0.518	0.568	**0.721**	0.602
Poi	**0.656**	0.384	0.560	0.643	0.649	**0.648**	0.374	0.544	0.624	0.643
Synapse	0.454	0.474	0.503	**0.493**	0.460	0.402	0.447	0.459	**0.444**	0.410
Velocity	0.627	0.524	0.636	**0.674**	0.667	0.621	0.529	0.636	**0.673**	0.660
Xalan	**0.701**	0.545	0.698	0.655	0.689	**0.691**	0.546	0.698	0.658	0.687
Average	**0.436**	**0.338** **−22.48%**	**0.392** **−10.09%**	**0.503** **+15.37%**	**0.484** **+11.01%**	**0.416**	**0.324** **−22.12%**	**0.366** **−12.02%**	**0.481** **+15.63%**	**0.457** **+9.86%**
Win/Tie/Loss		**10/4/28**	**17/4/21**	**30/2/10**	**34/0/8**		**12/6/24**	**19/7/16**	**31/2/9**	**34/0/8**

Another issue should be discussed is that as observed in Fig. 3, *k*-core decomposition ignores the directions of the edges in CDN. Whether the information of direction is useful for the proposed approach needs further investigation. In this paper, we report the experimental results of the tera-PROMISE dataset in the cross-validation scenario when the *in-degree* or *out-degree* measures are considered separately.

Table 11 shows results of this experiment. Entries in Table 11 are similar to those in Table 6 , 7, 8. It should be noticed that the mean values of P_{opt}s of R_{ee} and *top-core* are slightly different from those in Table 6, such difference is because of the randomness in cross-validation. Generally, based on Table 11, it can be concluded that using in-degree or out-degree alone cannot improve the performance of effort-aware bug prediction model. It is also interesting to observe that for two subject systems, i.e., Ivy and Log4j, in-degree outperforms coreness, while there is no subject system for which out-degree outperforms coreness. Such results are in accordance with previous experimental results. For instance, Nguyen et al. [7] observed that in their

TABLE 10
Subject Software Systems in a New Bug Data Repository – the GitHub Bug Dataset [58]

System	Version	SLOC	# Class	N_{CDN}	E_{CDN}	$\lvert C_{RB} \bigcap C_{CDN} \rvert$	p_{bug}	Website
Elasticsearch	fe86edd	355,963	5,440	5,382	26,719	5,335	4.4%	www.elastic.co/products/elasticsearch
MapDB	5b4700c	40,312	239	226	509	158	2.7%	www.mapdb.org
mcMMO	eb359c5	26,483	320	320	1,633	311	11.9%	github.com/mcMMO-Dev/mcMMO

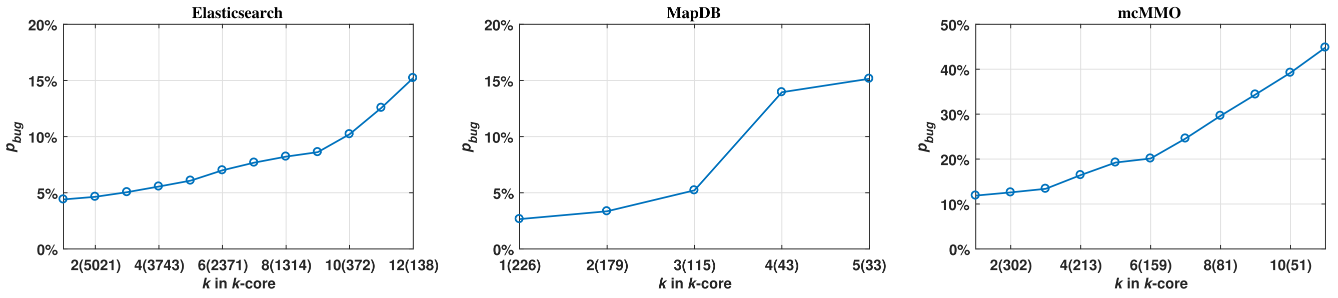


Fig. 11. The buggy classes' percentages in different k-cores of three subject software systems in the GitHub Bug Dataset.

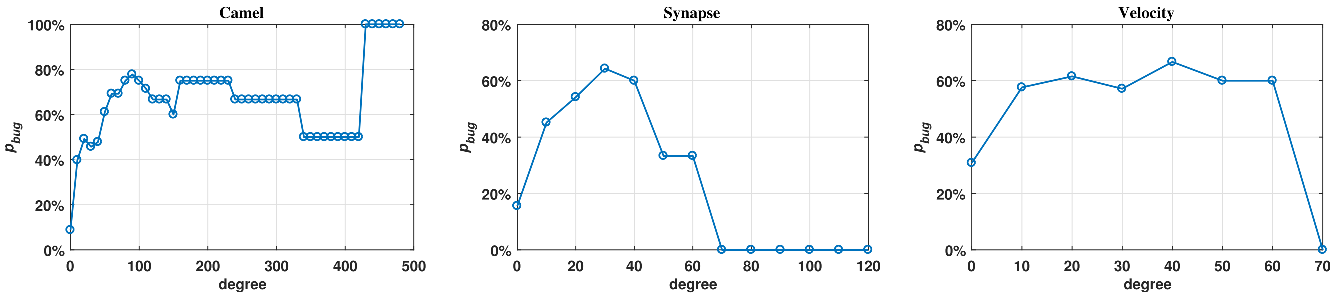


Fig. 12. The buggy classes' percentages in classes whose degree is greater than or equal to the value in x-axis.

experiments in-degree achieved the best recall among several network measures. It is also interesting for future work to investigate other network measures' performances using the proposed Equation 1 in effort-aware bug prediction.

8 Related Work

8.1 Using Complex Network Theory in Bug Prediction

Over the past decade, *Complex Network* theory and related graph algorithms [1], [2], [3] have been successfully used in software bug prediction domain. Related researches can be classified into *code-based network* and *change-based network* in software bug prediction.

In the first category, Zimmermann and Nagappan [4] proposed to use network measures on the dependency graph of a software system in bug prediction. They found that network measures could improve the prediction performance by 10 percent in predicting Windows Server 2003 operating system's post-release bugs. Later, Tosun et al. [6] reproduced Zimmermann and Nagappan's work on three small scale embedded software and two versions of the Eclipse project. Their results revealed that network measures were important indicators of defective modules for large and complex systems, but they did not have significant predictive power on small-scale projects. Premraj and Herzig [8] further replicated and extended these studies, they evaluated network measures' effectiveness based on three open-source projects. They investigated network measures' performances in cross-validation, forward-release, and cross-project scenarios. Their results showed that although network measures outperformed code metrics in cross-validation, they provided no advantage for forward-release and cross-project scenarios. Recently, Ma et al. [12] further evaluated the predictive effectiveness of network measure in effort-aware bug prediction. Chen et al. [11] used network measures to predict high severity software bugs, and they found that most network measures were significantly related to high severity bug-proneness. To sum up, these studies have used network metrics derived from software code dependency networks to improve bug prediction performances. These studies showed that network metrics are usually more effective predictors for software bugs, compared with product and complexity metrics.

There are other researches further exploiting software networks' characteristics in bug prediction. For instance, in our previous study [10], we proposed two new class cohesion metrics based on community structures of software Call Graphs. Experiments showed that these new metrics are effective in predicting software bugs. Recently, Li [13] proposed a network model (Tri-Relation Network) integrating developer contribution, module dependency, and developer collaboration relations, to improve bug prediction.

On the other hand, in the *change-based* network category, Pinzger et al. proposed to represent developer contributions in evolution with a developer-module network named as *contribution network* [5]. Then network centrality measures was used to measure the degree of fragmentation of

TABLE 11
Comparison of P_{opt} Using *in-degree* and *out-degree* Alone in the Cross-Validation Scenario (effort=20%)

System	R_{ee}	in-degree	out-degree	top-core	in-degree vs. core	out-degree vs. core
Camel	0.424	0.458	0.447	**0.470**	$=$	$<$, *
Ivy	0.190	**0.266**	0.230	0.205	$>$, **	$=$
Log4j	0.399	**0.631**	0.532	0.526	$>$, **	$=$
Poi	0.623	0.541	0.499	**0.628**	$<$, **	$<$, **
Synapse	0.434	0.366	0.375	**0.443**	$<$, **	$<$, **
Tomcat	0.264	0.238	0.276	**0.285**	$<$, **	$=$
Velocity	0.556	0.448	0.547	**0.589**	$<$, **	$<$, **
Xalan	**0.652**	0.444	0.544	0.607	$<$, **	$<$, **
Average	**0.443**	**0.424** −**4.29%**	**0.431** −**2.71%**	**0.469** +**5.87%**		

developer contributions. Experiments on Microsoft Windows Vista binaries showed that network centrality measures are significant indicator for failure-prone binaries [5]. Later, Herzig et al. proposed to group software changes into *change genealogies* [9], which are graphs of changes reflecting their mutual dependencies. They observed that compared to prediction models based on code dependency network or code complexity metrics, change genealogy based prediction methods can achieve better performance [9].

Recently, with the successful applications of network embedding techniques [60] in many machine learning tasks, researches started applying network embedding techniques in software bug prediction. Network embedding can automatically encode software networks into low-dimensional vector spaces. Recently, followed Herzig et al.'s work [9], Loyola and Matsuo proposed to use two models originated from the Natural Language Processing (NLP) area—SkipGram and CBOW, to automatically generate feature representations from change genealogy graph [61]. Recently, we also proposed to use a new network embedding technique, *node2vec* [62], to automatically learn structural features of CDN into low dimensional vector space [63]. Experimental results showed that the proposed approach can improve bug prediction nontrivially.

Generally speaking, existing researches have shown that Complex Network theory, concepts, and metrics are of great value in software bug prediction. These studies have laid a good foundation for us to further use *k*-core decomposition, which is an efficient analyzing algorithm in Complex Network theory, in software bug prediction. Our study provide a new perspective for bug prediction. Instead of proposing or leveraging new metrics in software dependency networks, we use *k*-core decomposition to analyze bug distribution on Class Dependency Networks, and observe a new tendency. Based on this observation, an equation that rearranges the suspicious class list produced by bug prediction models is proposed. The proposed equation complements existing researches.

8.2 Applications of *k*-Core Decomposition in Software Engineering and Other Domains

As discussed in Section 1, there are many application areas of *k*-core decomposition including but not limited to social network analysis [15], visualization of large networks [16], analyzing protein-protein interaction networks [17], [18], etc.

Researchers in software engineering have already started using *k*-core decomposition. Meyer et al. [19] applied *k*-core decomposition on Class Dependency Networks to identifying important classes. They studied three open-source Java projects over a 10-year period and showed that *k*-core decomposition could identify key classes of the corresponding software. Recently, Pan et al. [21] proposed a more accurate software network considering both the coupling direction and coupling strength, then proposed a generalized *k*-core decomposition method to more accurately identify key classes. Pan et al. [20] also constructed weighted software networks from real-world Java software systems. They investigated static and evolving properties of the weighted *k*-core structure and observed many interesting features. They also applied the weighted *k*-core decomposition method to identify the key classes. Experiments showed that their approach outperformed other nine approaches.

In a word, in recent years, researches in software engineering have started using *k*-core decomposition. However, these studies mainly focused on using *k*-core decomposition to identify key classes in software. To the best of our knowledge, our work is the first one that uses *k*-core decomposition to understand software bug distribution from a new perspective. Experimental results also show that, the proposed *top-core* equation can indeed help the testers or code reviewers locate the real bugs more quickly and easily.

8.3 Effort-Aware Bug Prediction

As discussed in Section 4.1, in recent years, several effort-aware bug prediction models have been proposed [24], [25], [26], aimed to help testers or code reviewers allocate their resources more effectively. Mende and Koschke for the first time proposed two models to include the notion of effort awareness into bug prediction models [24]. Experiments have shown that both models improve the cost effectiveness of bug prediction models significantly [24]. Kamei et al. used the proposed models to evaluate different metrics' performance in effort-aware bug prediction [25]. We used the same evaluation metric in [25]. Later, many related researches evaluated bug prediction models from the effort-aware perspective [64], [65], or proposed new effort-aware models [26], [66]. As discussed earlier, in our experiments, effort-aware models in Table 3 are used and it is observed that R_{ee} [26] has the best performance. The proposed *top-core* equation is observed to significantly improve existing effort-aware models' performances. We recommend future research always use the proposed equation in effort-aware bug prediction practices.

9 Conclusion

In this paper, we have used k-core decomposition on Class Dependency Networks to analyze software bug distribution from a new perspective. An interesting and widely existed tendency has been observed: measurement results have shown that classes in k-cores with larger k values have stronger possibility to be buggy. In other words, bugs tend to cluster towards inner cores. Based on this observation, a simple but effective equation named as *top-core* has been proposed. *Top-core* rearranges the order of classes in the suspicious class list produced by effort-aware bug prediction models, by prioritizing the suspicious classes in k-cores with larger k values. Experiments on 18 subject systems with two machine learning algorithms—Random Forest and Logistic Regression, have been conducted to evaluate the effectiveness of the proposed equation. Experimental results have shown that, in 96 (54+42) effort-aware bug prediction experiments in which Random Forest is used, *top-core* significantly improved effort-aware bug prediction model's performance in 85.2 percent experiments in the cross-validation scenario and in 80.95 percent experiments in the forward-release scenario. In average, the bug prediction model's performance is improved by 11.5 and 12.6 percent, respectively. It is concluded that the proposed *top-core* equation can help the testers or code reviewers locate the real bugs more quickly and easily. All code and experimental results are available at: https://github.com/XJTU-SE/top-core.

In the future, we plan to further leverage the observed tendency in other software engineering practices that are related to software quality and bugs, such as test case prioritization, fault localization, etc. We also plan to combine the idea of *top-core* with unsupervised models.

Acknowledgments

This work is partially supported by the National Key Research and Development Program of China (2016YFB0800202), National Natural Science Foundation of China (61602369, 61632015, 61772408, U1766215, 61721002, 61833015), Ministry of Education Innovation Research Team (IRT_17R86), Shaanxi Province postdoctoral research project funding (2016BSHEDZZ108), consulting research project of Chinese academy of engineering "The Online and Offline Mixed Educational Service System for 'The Belt and Road' Training in MOOC China", and Project of China Knowledge Centre for Engineering Science and Technology.

References

[1] D. J. Watts and S. H. Strogatz, "Collective dynamics of small-worldnetworks," *Nature*, vol. 393, no. 6684, 1998, Art. no. 440.
[2] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Sci.*, vol. 286, no. 5439, pp. 509–512, 1999.
[3] D. Chakrabarti and C. Faloutsos, "Graph mining: Laws, generators, and algorithms," *ACM Comput. Surveys*, vol. 38, no. 1, 2006, Art. no. 2.
[4] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proc. ACM/IEEE 30th Int. Conf. Softw. Eng.*, 2008, pp. 531–540.
[5] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?" in *Proc. 16th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2008, pp. 2–12.
[6] A. Tosun, B. Turhan, and A. Bener, "Validation of network measures as indicators of defective modules in software systems," in *Proc. 5th Int. Conf. Predictor Models Softw. Eng.*, 2009, Art. no. 5.
[7] T. H. Nguyen, B. Adams, and A. E. Hassan, "Studying the impact of dependency network measures on software quality," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2010, pp. 1–10.
[8] R. Premraj and K. Herzig, "Network versus code metrics to predict defects: A replication study," in *Proc. Int. Symp. Empirical Softw. Eng. Meas.*, 2011, pp. 215–224.
[9] K. Herzig, S. Just, A. Rau, and A. Zeller, "Predicting defects using change genealogies," in *Proc. IEEE 24th Int. Symp. Softw. Rel. Eng.*, 2013, pp. 118–127.
[10] Y. Qu, X. Guan, Q. Zheng, T. Liu, L. Wang, Y. Hou, and Z. Yang, "Exploring community structure of software call graph and its applications in class cohesion measurement," *J. Syst. Softw.*, vol. 108, pp. 193–210, 2015.
[11] L. Chen, W. Ma, Y. Zhou, L. Xu, Z. Wang, Z. Chen, and B. Xu, "Empirical analysis of network measures for predicting high severity software faults," *Sci. China Inf. Sci.*, vol. 59, no. 12, 2016, Art. no. 122901.
[12] W. Ma, L. Chen, Y. Yang, Y. Zhou, and B. Xu, "Empirical analysis of network measures for effort-aware fault-proneness prediction," *Inf. Softw. Technol.*, vol. 69, pp. 50–70, 2016.
[13] Y. Li, "Applying social network analysis to software fault-proneness prediction," Ph.D. dissertation, Univ. Texas Dallas, Richardson, TX, USA, 2017.
[14] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "k-core decomposition: a tool for the visualization of large scale networks," in *Advances in Neural Information Processing Systems 18*. Cambridge, MA, USA: MIT Press, 2006, p. 41.
[15] C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis, "Evaluating cooperation in communities with the k-core structure," in *Proc. Int. Conf. Adv. Social Netw. Anal. Mining*, 2011, pp. 87–93.
[16] Y. Zhang and S. Parthasarathy, "Extracting analyzing and visualizing triangle k-core motifs within networks," in *Proc. IEEE 28th Int. Conf. Data Eng.*, 2012, pp. 1049–1060.
[17] G. D. Bader and C. W. Hogue, "An automated method for finding molecular complexes in large protein interaction networks," *BMC Bioinf.*, vol. 4, no. 1, 2003, Art. no. 2.
[18] S. Wuchty and E. Almaas, "Peeling the yeast protein network," *Proteomics*, vol. 5, no. 2, pp. 444–449, 2005.
[19] P. Meyer, H. Siy, and S. Bhowmick, "Identifying important classes of large software systems through k-core decomposition," *Adv. Complex Syst.*, vol. 17, no. 07n08, 2014, Art. no. 1550004.
[20] W. Pan, B. Li, J. Liu, Y. Ma, and B. Hu, "Analyzing the structure of java software systems by weighted k-core decomposition," *Future Generation Comput. Syst*, vol. 83, pp. 431–444, 2018.
[21] W. Pan, B. Song, K. Li, and K. Zhang, "Identifying key classes in object-oriented software using generalized k-core decomposition," *Future Generation Comput. Syst.*, vol. 81, pp. 188–202, 2018.
[22] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1276–1304, Nov./Dec. 2012.
[23] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia, "A developer centered bug prediction model," *IEEE Trans. Softw. Eng.*, vol. 44, no. 1, pp. 5–24, Jan. 2018.
[24] T. Mende and R. Koschke, "Effort-aware defect prediction models," in *Proc. 14th Eur. Conf. Softw. Maintenance Reengineering*, 2010, pp. 107–116.
[25] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2010, pp. 1–10.
[26] Y. Yang, Y. Zhou, H. Lu, L. Chen, Z. Chen, B. Xu, H. Leung, and Z. Zhang, "Are slice-based cohesion metrics actually useful in effort-aware post-release fault-proneness prediction? an empirical study," *IEEE Trans. Softw. Eng.*, vol. 41, no. 4, pp. 331–357, Apr. 2015.
[27] M. Kitsak, L. K. Gallos, S. Havlin, F. Liljeros, L. Muchnik, H. E. Stanley, and H. A. Makse, "Identification of influential spreaders in complex networks," *Nature Phys.*, vol. 6, no. 11, pp. 888–893, 2010.
[28] L. Šubelj and M. Bajec, "Community structure of complex software systems: Analysis and applications," *Physica A: Statistical Mech. Appl.*, vol. 390, no. 16, pp. 2968–2975, 2011.

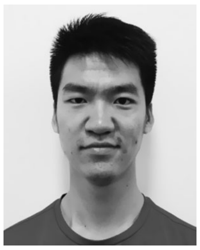
[29] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero, "Understanding the shape of java software," *ACM SIGPLAN Notices*, vol. 41, no. 10, pp. 397–412, 2006.
[30] P. Louridas, D. Spinellis, and V. Vlachos, "Power laws in software," *ACM Trans. Softw. Eng. Methodology*, vol. 18, no. 1, 2008, Art. no. 2.
[31] G. Concas, M. Marchesi, C. Monni, M. Orrù, and R. Tonelli, "Software quality and community structure in java software networks," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 27, no. 7, pp. 1063–1096, 2017.
[32] T. Menzies, R. Krishna, and D. Pryor, "The promise repository of empirical software engineering data," Dept. Comput. Sci., North Carolina State Univ., 2015. [Online]. Available: http://openscience.us/repo
[33] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proc. 6th Int. Conf. Predictive Models Softw. Eng.*, 2010, Art. no. 9.
[34] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Proc. 7th IEEE Working Conf. Mining Softw. Repositories*, 2010, pp. 31–41.
[35] T. Shippey, T. Hall, S. Counsell, and D. Bowes, "So you need more method level datasets for your software defect prediction?: Voilà!" in *Proc. 10th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.*, 2016, Art. no. 12.
[36] Y. Guo, M. Shepperd, and N. Li, "Bridging effort-aware prediction and strong classification: A just-in-time software defect prediction study," in *Proc. 40th Int. Conf. Softw. Eng.: Companion Proc.*, 2018, pp. 325–326.
[37] Y. Zhao, Y. Yang, H. Lu, J. Liu, H. Leung, Y. Wu, Y. Zhou, and B. Xu, "Understanding the value of considering client usage context in package cohesion for fault-proneness prediction," *Automated Softw. Eng.*, vol. 24, no. 2, pp. 393–453, 2017.
[38] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "The impact of automated parameter optimization on defect prediction models," *IEEE Trans. Softw. Eng.*, 2018, doi: 10.1109/TSE.2018.2794977.
[39] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.
[40] D. W. Hosmer Jr, S. Lemeshow, and R. X. Sturdivant, *Applied Logistic Regression*, vol. 398. Hoboken, NJ, USA: Wiley, 2013.
[41] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows," in *Proc. ACM/IEEE 32nd Int. Conf. Softw. Eng.*, 2010, pp. 495–504.
[42] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, Jul./Aug. 2008.
[43] B. Caglayan, A. Tosun, A. Miranskyy, A. Bener, and N. Ruffolo, "Usage of multiple prediction models based on defect categories," in *Proc. 6th Int. Conf. Predictive Models Softw. Eng.*, 2010, Art. no. 8.
[44] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 481–490.
[45] S. Herbold, A. Trautsch, and J. Grabowski, "A comparative study to benchmark cross-project defect prediction approaches," *IEEE Trans. Softw. Eng.*, vol. 44, no. 9, pp. 811–833, Sep. 2018.
[46] F. Zhang, A. E. Hassan, S. McIntosh, and Y. Zou, "The use of summation to aggregate software metrics hinders the performance of defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 5, pp. 476–491, May 2017.
[47] T. Mende and R. Koschke, "Revisiting the evaluation of defect prediction models," in *Proc. 5th Int. Conf. Predictor Models Softw. Eng.*, 2009, Art. no. 7.
[48] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Trans. Softw. Eng*, vol. 39, no. 6, pp. 757–773, Jun. 2013.
[49] A. Monden, T. Hayashi, S. Shinoda, K. Shirai, J. Yoshida, M. Barker, and K. Matsumoto, "Assessing the cost effectiveness of fault prediction in acceptance testing," *IEEE Trans. Softw. Eng.*, vol. 39, no. 10, pp. 1345–1357, Oct. 2013.
[50] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, "Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 157–168.
[51] W. Fu and T. Menzies, "Revisiting unsupervised learning for defect prediction," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 72–83. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106257
[52] K. E. Bennin, J. Keung, P. Phannachitta, A. Monden, and S. Mensah, "Mahakil: Diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 44, no. 6, pp. 534–550, Jun. 2018.
[53] A. Agrawal and T. Menzies, "Is better data better than better data miners?: On the benefits of tuning smote for defect prediction," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 1050–1061.
[54] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *J. Artif. Intell. Res.*, vol. 16, no. 1, pp. 321–357, 2002.
[55] R. Storn and K. Price, "Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces," *J. Global Optimization*, vol. 11, no. 4, pp. 341–359, 1997.
[56] U. Brandes, "A faster algorithm for betweenness centrality," *J. Math. Sociology*, vol. 25, no. 2, pp. 163–177, 2001.
[57] S. Ioana , "A PageRank based recommender system for identifying key classes in software systems," in *IEEE 10th Jubilee Int. Symp. Appl. Comput. Intell. Informat. (SACI)*, 2015, pp. 495–500.
[58] R. Ferenc, Z. Tóth, G. Ladányi, I. Siket, and T. Gyimóthy, "A public unified bug dataset for java," in *Proc. 14th Int. Conf. Predictive Models Data Analytics Softw. Eng.*, 2018, pp. 12–21.
[59] Y. Zhou, Y. Yang, H. Lu, L. Chen, Y. Li, Y. Zhao, J. Qian, and B. Xu, "How far we have progressed in the journey? an examination of cross-project defect prediction," *ACM Trans. Softw. Eng. Methodology*, vol. 27, no. 1, 2018, Art. no. 1.
[60] P. Goyal and E. Ferrara, "Graph embedding techniques, applications, and performance: A survey," *Knowl.-Based Syst.*, vol. 151, pp. 78–94, 2018.
[61] P. Loyola and Y. Matsuo, "Learning feature representations from change dependency graphs for defect prediction," in *Proc. IEEE 28th Int. Symp. Softw. Rel. Eng.*, 2017, pp. 361–372.
[62] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2016, pp. 855–864.
[63] Y. Qu, T. Liu, J. Chi, Y. Jin, D. Cui, A. He, and Q. Zheng, "node2defect: Using network embedding to improve software defect prediction," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 844–849.
[64] X. Tan, X. Peng, S. Pan, and W. Zhao, "Assessing software quality by program clustering and defect prediction," in *Proc. 18th Working Conf. Reverse Eng.*, 2011, pp. 244–248.
[65] Y. Tang, F. Zhao, Y. Yang, H. Lu, Y. Zhou, and B. Xu, "Predicting vulnerable components via text mining or software metrics? an effort-aware perspective," in *Proc. IEEE Int. Conf. Softw. Quality Rel. Secur.*, 2015, pp. 27–36.
[66] A. Panichella, C. V. Alexandru, S. Panichella, A. Bacchelli, and H. C. Gall, "A search-based training algorithm for cost-aware defect prediction," in *Proc. Genetic Evol. Comput. Conf.*, 2016, pp. 1077–1084.

Yu Qu received the BS and PhD degrees from Xi'an Jiaotong University, Xi'an, China, in 2006 and 2015, respectively. He is now a post-doctoral researcher with the Department of Computer Science and Technology, Xi'an Jiaotong University. His research interests include trustworthy software and applying complex network and machine learning theories to analyzing software systems. He is a member of the IEEE.

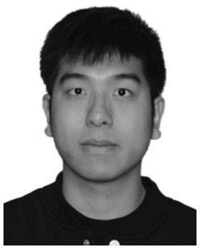
Qinghua Zheng received the BS and MS degrees in computer science and technology from Xi'an Jiaotong University, Xi'an, China, in 1990 and 1993, respectively, and the PhD degree in systems engineering from Xi'an Jiaotong University, Xi'an, China, in 1997. He was a post-doctoral researcher with Harvard University in 2002. Since 1995 he has been with the Department of Computer Science and Technology, Xi'an Jiaotong University, and was appointed director of the Department in 2008 and Cheung Kong professor in 2009. His research interests include computer network security, intelligent e-learning theory and algorithm, multimedia e-learning, and trustworthy software. He is a member of the IEEE.

Jianlei Chi received the BS degree in computer science and technology from Harbin Engineering University, China, in 2014. He is currently working toward the PhD degree in the Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, China. His research interests include trustworthy software, software testing, and software behavior analysis.

Yangxu Jin is working toward the MS degree in the Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, China. Her research interests include trustworthy software, defect prediction of software system.

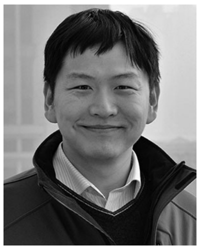


Ancheng He is working toward the MS degree in the Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, China. His research interests include trustworthy software, security analysis of android system.



Di Cui is working toward the PhD degree in the Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, China. His research interests include trustworthy software, architecture recovery of software system.

Hengshan Zhang received the PhD degree from the Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, China, in 2016. He is currently a lecturer with the School of Computer Science, Xi'an University of Posts and Telecommunications, Xi'an, China. His research interests include computing with words, group decision making, information aggregation, and trustworthy software.



Ting Liu received the BS degree in information engineering and the PhD degree in system engineering from the School of Electronic and Information, Xi'an Jiaotong University, in 2003 and 2010, respectively. He is a professor with Xi'an Jiaotong University, China. He was a visiting professor with Cornell University during 2016 to 2017. His researches include software engineering and cyber-physical system. He is a member of the IEEE.