node2defect: using network embedding to improve software defect prediction

Conference Paper · September 2018

DOI: 10.1145/3238147.3240469

CITATIONS
17

READS
1,592

7 authors, including:

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

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

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

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

node2defect: Using Network Embedding to Improve Software Defect Prediction

Yu Qu
Ministry of Education Key Lab For Intelligent Networks and Network Security, Xi'an Jiaotong University, China
quyuxjtu@xjtu.edu.cn

Ting Liu
School of Electronic and Information Engineering, Xi'an Jiaotong University, China
tingliu@xjtu.edu.cn

Jianlei Chi
Xi'an Jiaotong University, China
chijianlei@stu.xjtu.edu.cn

Yangxu Jin
Xi'an Jiaotong University, China
jyx530@stu.xjtu.edu.cn

Di Cui
Xi'an Jiaotong University, China
cuidi@stu.xjtu.edu.cn

Ancheng He
Xi'an Jiaotong University, China
hg19941996@stu.xjtu.edu.cn

Qinghua Zheng
Xi'an Jiaotong University, China
qhzheng@xjtu.edu.cn

ABSTRACT

Network measures have been proved to be useful in predicting software defects. Leveraging the dependency relationships between software modules, network measures can capture various structural features of software systems. However, existing studies have relied on user-defined network measures (e.g., degree statistics or centrality metrics), which are inflexible and require high computation cost, to describe the structural features. In this paper, we propose a new method called *node2defect* which uses a newly proposed network embedding technique, *node2vec*, to automatically learn to encode dependency network structure into low-dimensional vector spaces to improve software defect prediction. Specifically, we firstly construct a program's Class Dependency Network. Then *node2vec* is used to automatically learn structural features of the network. After that, we combine the learned features with traditional software engineering features, for accurate defect prediction. We evaluate our method on 15 open source programs. The experimental results show that in average, *node2defect* improves the state-of-the-art approach by 9.15% in terms of F-measure.

CCS CONCEPTS

• **Software and its engineering** → *Risk management*; *Maintaining software*;

KEYWORDS

Software defect, defect prediction, software metrics, network embedding

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3240469

ACM Reference Format:
Yu Qu, Ting Liu, Jianlei Chi, Yangxu Jin, Di Cui, Ancheng He, and Qinghua Zheng. 2018. node2defect: Using Network Embedding to Improve Software Defect Prediction. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3238147.3240469

1 INTRODUCTION

Software defect prediction is the process of building machine learning classifiers to predict code regions that potentially have defects [9, 19, 26]. Defect prediction techniques can provide useful guidances for code reviewers or software testers to allocate their limited resources more effectively, thus is very useful in software engineering practices [9, 19, 26].

Network measures have been proved to be useful in predicting software defects (e.g., [6, 14, 16, 19, 21, 24, 26]). Exploiting the dependency relationships between software modules, network measures can capture various structural features of software systems. Thus, they are very useful in software defect prediction. For instance, in their seminal study, Zimmermann and Nagappan [26] proposed to use network measures on the dependency graph of a software system in defect prediction. Recently, Ma et al. [16] further evaluated the predictive effectiveness of network measures in effort-aware bug prediction. Chen et al. [6] used network measures to predict high severity software bugs, and they found that most network measures were significantly related to high severity bug-proneness.

However, these network measures (e.g., degree statistics or centrality metrics) are traditionally user-defined, which has been believed to be inflexible and require human designing efforts [12]. These measures are also believed to suffer from the high computation and space cost [5]. Thus, *network embedding* (graph embedding) techniques [10, 12], which automatically learn to encode network structure into low-dimensional vector space, have drawn increasing attentions in the machine learning community and shown their advantages over user-defined measures in many machine learning tasks [10, 12].

Considering this new research trend, in this paper, we propose a new method called *node2defect* which uses a new network embedding technique, *node2vec* [11], to automatically learn structural features of software Class Dependency Network to improve software defect prediction.

In *node2defect*, the Class Dependency Network (CDN) of the program under analysis is firstly constructed. Then *node2vec* is used to automatically learn structural features of CDN. For each node (i.e., each class) in CDN, *node2vec* automatically learns a vector to encode its structural feature. Then *node2defect* concatenates the class's feature vector with its traditional software engineering features, such as cohesion and coupling metrics [7], to form the new feature of this class. Then the new feature is used in software defect prediction. Experimental results show that the proposed *node2defect* approach can improve software defect prediction nontrivially. The F-measure is improved by 9.15% in average.

In summary we make the following contributions in this paper:

(1) a new network embedding technique, *node2vec*, is used to automatically learn structural features of software Class Dependency Networks; (2) we propose a new method named as *node2defect*, which exploits network embedding technique and combines with traditional software engineering metrics, to improve defect prediction; (3) experiments on 15 Java open source programs show that the proposed method can nontrivially improve defect prediction models' performances.

2 BACKGROUND

2.1 Class Dependency Network

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 [23]: $CDN_P = (V, E)$, where each node $v \in V$ represents a class in P, and the edge set E represents the class dependency relationships. 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.

Figure 1 (a) gives an illustrative example Java code snippet. For this code snippet, Figure 1 (b) shows its Class Dependency Network (CDN) [23]. Although other work [2, 8, 15] 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 Figure 1, 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.

2.2 The *node2vec* Technique

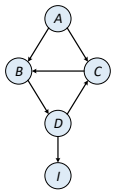
In this section, we firstly give a brief introduction on network embedding, followed by a brief discussion on *node2vec*.

For a network $N = (V, E)$, a network embedding is a mapping $f : v_i \rightarrow y_i \in \mathbb{R}^d, \forall v_i \in V$ such that $d \ll |V|$ and the function f preserves some proximity measure defined on network N [10].

The conceptual framework of *node2vec* [11] is shown in Figure 2. Briefly, *node2vec* algorithm is based on and motivated by a Natural Language Processing (NLP) model – Skip-gram [18], and then

```
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;
    }
}
```

(a)



(b)

Figure 1: A Class Dependency Network example.

proposes a flexible random walk approach to generate (sample) network neighborhoods for nodes.

Specifically, Skip-gram model is one of the two basic models of the NLP technique – *word2vec* [18] (the other one is Continuous Bag of Words model, CBOW). Skip-gram aims to maximize the co-occurrence probability among the words that appear within a sliding window. Such process is realized by solving an optimization problem using Stochastic Gradient Descent (SGD) and backpropagation on single hidden-layer feedforward neural network.

As shown in Figure 2, *node2vec* firstly samples a set of paths (walks) from the input network using their new fixed length random walk approach. Each path sampled from the network corresponds to a sentence from the corpus in NLP, where a node corresponds to a word. Then Skip-gram is applied on the paths to maximize the probability of observing a node's neighborhood conditioned on its embedding. As the authors stated [11], given the linear nature of text in NLP, the notion of neighborhood can be naturally defined using a sliding window over consecutive words. However, networks (such as CDNs in this paper) are not linear, thus a richer notion of neighborhood is needed. In a word, the key contribution of *node2vec* is in defining a flexible notion of a node's network neighborhood by providing a trade-off between Breadth-first Sampling (BFS) and Depth-first Sampling (DFS) graph searching strategies. Figure 2 also shows BFS and DFS examples given a source node C7.

A new 2^{nd} order random walk is defined in *node2vec* with two parameters p and q which control the transition probability during the walk, as shown in Figure 2. The parameters p and q are named as *return parameter* and *in-out parameter*, respectively. Interested readers are referred to the original paper [11] for more details.

It is worth noting that the time complexity of *node2vec* is $O(|V|\, d)$ [10]. Existing network measures usually have more expensive time complexities. For instance, the metric *Betweenness Centrality* has been intensively used in existing researches [6, 16, 19, 21, 24, 26]. The time complexity of Betweenness Centrality is $O(|V|\,|E|)$ on unweighted network [3]. Considering $d \ll |V|$, $|V|$ is usually smaller than $|E|$ (see the statistics in Table 1) and there are usually more than 10 metrics are computed in existing researches, so the existing methods usually require much higher computation and space cost.

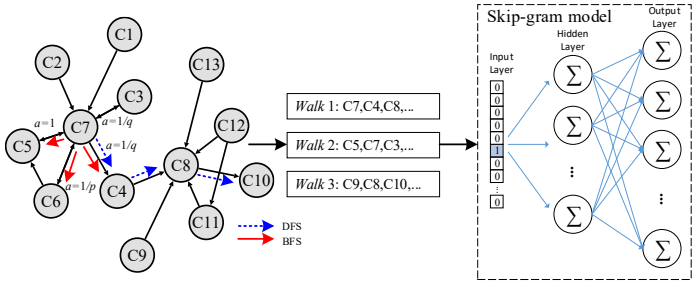


Figure 2: The conceptual framework of *node2vec*.

3 THE PROPOSED APPROACH: *NODE2DEFECT*

In this section, we introduce *node2defect*'s framework based on a real and simple example. Figure 3 shows the process of *node2defect* when a package of Ant, `org.apache.tools.ant.input`, is considered in the analyzing process of *node2defect*. For this package, Figure 3 depicts its CDN when only considering the class dependency relationships within this package (in practice, the dependency relationships within the whole system are considered). Based on the constructed CDN, *node2vec* is used to automatically learn a vector for each node (i.e., each class) in the CDN. For the simplicity of illustration, as shown in Figure 3, the vector's dimension (d) is set as 8. In experiments in this paper, we use 32 as the dimension of the learned vector. Since the learned vector only quantify the structural features of a class, but the defectiveness of a class are the consequences of many complicated and inter-played quality factors, e.g., the software architecture and the programmer's perception on the programming language. To more accurately predict defects, *node2defect* also computes traditional software engineering metrics, e.g., Lack of Cohesion in Methods (LCOM), Coupling Between Object classes (CBO), and Depth of Inheritance Tree (DIT) [7]. Then *node2defect* concatenates the learned vector with traditional metrics and uses the concatenated vector as the features that are fed into the subsequent machine learning classifiers. In the evaluation part of this paper, traditional software engineering metrics alone are used as the ***traditional approach*** and ***baseline*** metrics. Table 2 in the appendix part of this paper shows the traditional software engineering metrics used in this study.

4 EXPERIMENTS

4.1 Subject Software Systems

To evaluate the practical application of *node2defect*, a data set containing 15 large open-source and widely-used Java software systems is collected, as shown in Table 1.

Ant is a command-line tool for automating software build processes; Camel is an integration framework; DrJava is a Java development environment; GenoViz is a tool for data visualization and sharing in genomics; jEdit is a text editor; Ivy is a transitive dependency manager; Jmri is a building tool for modelling railroad computer control; Jmol is a browser-based HTML5 viewer and stand-alone viewer for chemical structures in 3D; Jppf is an open-source grid computing solution; Lucene is a searching and information retrieval library; Poi a library to process Microsoft Office files; Synapse is a high-performance Enterprise Service Bus (ESB); Tomcat is a web server and servlet container; Velocity is a template engine to reference objects defined in Java code; Xalan is a library for processing XML documents.

All the defect data of these subject systems is collected from publicly available software defect data repository. Among these subject systems, the defect data of Ant, Camel, jEdit, Ivy, Lucene, Poi, Synapse, Tomcat, Velocity, and Xalan is obtained from the tera-PROMISE data repository[1] [17]. The defect data of DrJava, Genoviz, Jmri, Jmol, and Jppf is obtained from a newly released data set[2] which has been contributed by Shippey et al. in their ESEM 2016 paper [22]. To ensure the reproducibility of this study, we use the metrics extracted and contained in these tow data sets as the traditional software engineering metrics. Table 2 in the appendix part of this paper shows the metrics in tera-PROMISE data set.

Table 1 gives the statistics of these systems and their CDNs. The data in each column in Table 1 is interpreted as follows:

Column Version shows the version of the corresponding system that is contained in the aforementioned two defect data sets. Columns SLOC, # Class, show the **S**ource **L**ines **O**f **C**ode and the number of classes of the subject systems, respectively. Column N_{CDN} and E_{CDN} show the number of nodes and edges of each CDN, respectively. In the construction processes of CDNs, only when a class has certain dependency relations with other classes, its corresponding node is added into CDN. Thus, N_{CDN} is slightly smaller than the total number of classes in Column # Class.

Column $|C_{RD} \bigcap C_{CDN}|$ 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 **D**efective or not. p_{Defect} is the percentage of defective classes in $C_{RD} \bigcap C_{CDN}$. The last column shows the websites of these systems.

4.2 Experiment Design

In the evaluation, we investigated the effectiveness of *node2defect* in the following two defect prediction experiments:

(1) Cross-validation

Cross-validation is a most widely-used method to evaluate defect prediction models. We used three-fold 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

[1] http://openscience.us/repo/
[2] https://github.com/tjshippey/ESEM2016

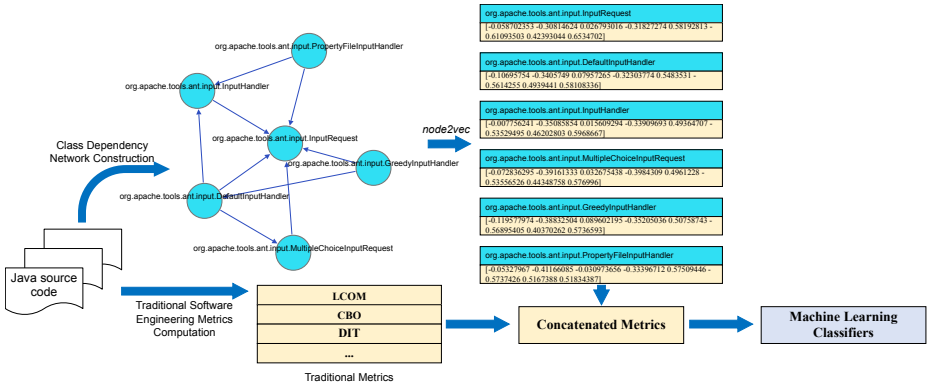


Figure 3: *node2defect*'s framework. A Java library and command-line tool – Ant's CDN is depicted as an illustrating example.

Table 1: Subject software systems

System	Version	SLOC	# Class	N_{CDN}	E_{CDN}	$\lvert C_{RD} \bigcap C_{CDN} \rvert$	p_{Defect}	Website
Ant	1.7.0	93,520	1,068	1,060	3,770	739	22.5%	https://ant.apache.org/
Camel	1.6.0	98,962	2,193	2,140	6,209	907	20.7%	https://camel.apache.org/
DrJava	20080106	67,958	814	792	2,385	515	20.4%	http://drjava.org/
Genoviz	6.0	91,307	704	693	3,279	633	32.4%	https://sourceforge.net/projects/genoviz/
jEdit	4.1	69,272	622	614	1,806	308	25.6%	http://jedit.org/
Ivy	2.0	36,636	421	418	1,851	349	11.5%	http://ant.apache.org/ivy/
Jmri	2.4	206,551	2,241	2,206	6,941	2,020	23.3%	http://jmri.sourceforge.net/
Jmol	6.0	29,855	291	282	648	158	43.7%	http://jmol.sourceforge.net/
Jppf	5.0	68,765	1,621	1,412	4,550	1,025	15.4%	http://jppf.org/
Lucene	2.4.0	35,984	460	457	1,879	340	60.1%	http://lucene.apache.org/
Poi	3.0	53,097	511	505	2,473	435	64.1%	http://poi.apache.org/
Synapse	1.2	46,060	573	546	1,811	250	34%	http://synapse.apache.org/
Tomcat	6.0.38	166,396	1,481	1,450	6,371	812	9.5%	http://tomcat.apache.org/
Velocity	1.6.1	26,636	254	253	1,236	228	34.2%	http://velocity.apache.org/
Xalan	2.6.0	155,067	1,039	1,014	6,007	884	45.5%	http://xalan.apache.org/

rest of the instances are used to evaluate the model. For each subject system, we repeated the experiment for 30 times to reduce the bias caused by the random split of instances. The machine learning algorithm Random Forest [4] is used in this experiment.

(2) Adaptive Selection of Classifiers in bug prediction (ASCI) [20]

ASCI is an adaptive defect prediction method, which has been recently proposed by Nucci et al [20]. ASCI can dynamically select among a set of machine learning classifiers the one which is more suitable to predict a class's defectiveness. The original implementation of ASCI has integrated five different machine learning classifiers [1], namely Logistic Regression (LOG), Naive Bayes (NB), Radial Basis Function Network (RBF), Multi-Layer Perceptron (MLP), and Decision Trees (DTree). Thus, ASCI can provide more thorough evaluation on *node2defect*. We adopted ASCI's original implementation, in which 10-fold cross-validation is used.

To evaluate the defect prediction model's performance, we use widely adopted metrics in literature [25]: (1) the F-measure, which is the harmonic mean of precision and recall, and (2) the Area Under the Curve (AUC), which quantifies the overall ability of a prediction model to discriminate between defective and clean classes. 100% AUC represents an ideal prediction result.

4.3 Experimental Results and Discussion

Figure 4 and 5 show the experimental results of the cross-validation and ASCI experiments, respectively. In each figure, the average (mean) values of F-measure and AUC are shown. Based on Figure 4, it can be observed that in the cross-validation experiment, the defect prediction model's performance is improved for all the subject systems, when *node2defect* is used. The value of F-measure is improved by 9.2% in average, and the value of AUC is improved by 3.86% in average.

Similarly, based on Figure 5, it can observed that in the ASCI experiment, the defect prediction model's performance is improved for *almost* all the subject systems, when *node2defect* is used. The value of F-measure only declines slightly for Ivy and Velocity, and the value of AUC also only declines slightly for Jmri and Tomcat, when *node2defect* is used. The value of F-measure is improved by 9.1% in average, and the value of AUC is improved by 5.6% in average. We also used the *Wilcoxon signed-rank tests* to evaluate the differences between results of approaches. The *p-values* of the Wilcoxon signed-rank tests confirm that the differences between results of approaches are significant at the 0.05 level.

In a word, in the experiments, the proposed approach – *node2defect* improves the performances of defect prediction models in most of the cases. And the improvements are nontrivial (greater than 9% in terms of F-measure), which mean that network embedding techniques are indeed helpful in software defect prediction, as they can learn the structural features of software modules automatically, which can reflect the ways how the modules depends on and interact with each other.

Here we give a possible explanation for these experimental results. As discussed in [11], prediction tasks on nodes in networks

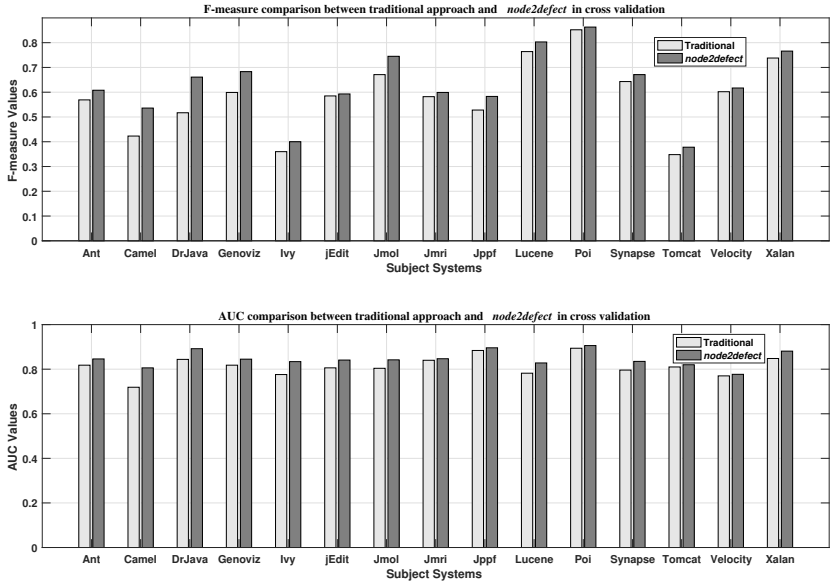


Figure 4: F-measure and AUC comparisons between traditional approach and *node2defect* in cross validation.

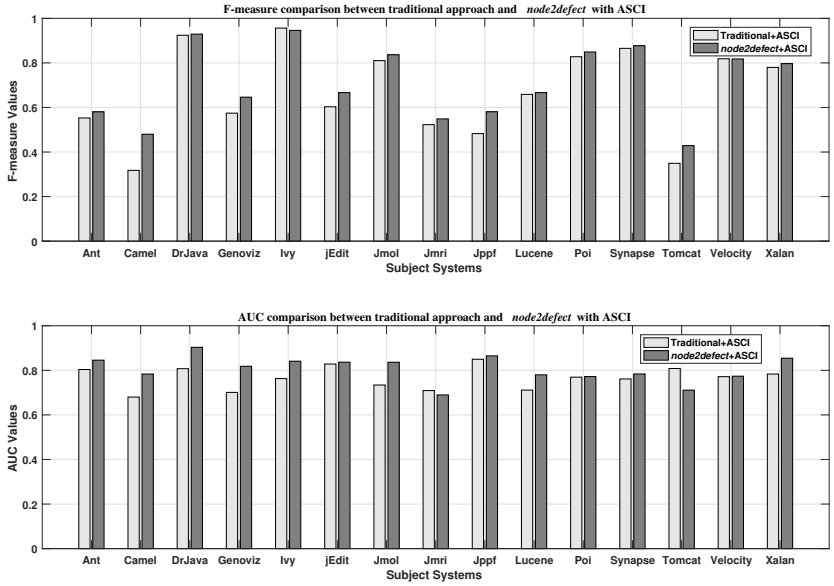


Figure 5: F-measure and AUC comparisons between traditional approach and *node2defect* with ASCI.

often shuttle between two kinds of similarities: *homophily* and *structural equivalence*. Homophily means that nodes belonging to similar communities should be embedded closely. Structural equivalence means that nodes that have similar structural roles should be embedded closely. Among the two sampling strategies, BFS leads to structural equivalence and DFS leads to homophily [11].

On the other hand, as discussed by Nguyen et al. [19], the network metric *nWeakComp in*, which is the number of unconnected components in the incoming ego network, is a good defect predictor. It means that for a certain class, if all the dependent classes are completely independent from each other, then the risk of failure for this class is higher than when they are interdependent. As shown in Figure 2, node C7 and C8 have high *nWeakComp in* scores (which is 4, repectively). From network embedding perspective, C7 and C8 plays similar roles in their communities (they are *hub*s of their corresponding communities), thus their structural equivalence can be captured by BFS and finally be reflected in the embedding vectors.

To sum up, network embedding techniques can capture similar structural properties of classes, thus classes with similar structural defect risks are close to each other in the low dimensional vector

Table 2: Traditional software engineering metrics

Metrics Name	Symbol
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

space. Such properties might be one of the reasons that network embedding can improve defect prediction.

5 CONCLUSION

In this paper, we have proposed a new method called *node2defect* which uses a newly proposed network embedding technique based on random walk, *node2vec*, to automatically learn structural features of software Class Dependency Networks to improve software defect prediction. Network embedding technique can encode classes' dependency relationships into low-dimensional vector space, thus provide a new perspective for defect prediction. *node2defect* also combines the learned structural vector with traditional software engineering metrics to predict defects more accurately. Experimental results based on 15 open source Java programs have shown that *node2defect* can improve the state-of-the-art approach, in which the traditional software engineering metrics are used alone, by 9.15% in terms of F-measure. In the future, we plan to use more network embedding techniques in software defect prediction. We also plan to study the approach that are suitable for cross-version and cross-project defect prediction.

A APPENDIX: TRADITIONAL METRICS

Table 2 shows the metrics of subject systems in tera-PROMISE data set [13], which are used as traditional software engineering metrics in this study. The other five subject systems' traditional metrics are extracted using the JHawk tool[3] [22].

ACKNOWLEDGMENTS

This work is partially supported by the National Natural Science Foundation of China (61602369, 61472318, 61632015, 61721002), Ministry of Education Innovation Research Team (IRT_17R86), National Key Research and Development Program of China (2016YFB0800202), Shaanxi Province postdoctoral research project funding (2016BSHED ZZ108), Project of China Knowledge Centre for Engineering Science and Technology.

[3]http://www.virtualmachinery.com/Jhawkmetricslist.htm

REFERENCES

[1] Ethem Alpaydin. 2004. *Introduction to Machine Learning*. MIT Press,. 28 pages.
[2] Gareth Baxter, Marcus Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan Tempero. 2006. Understanding the shape of Java software. *Acm Sigplan Notices* 41, 10 (2006), 397–412.
[3] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *Journal of mathematical sociology* 25, 2 (2001), 163–177.
[4] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (2001), 5–32.
[5] Hongyun Cai, Vincent W Zheng, and Kevin Chang. 2018. A comprehensive survey of graph embedding: problems, techniques and applications. *IEEE Transactions on Knowledge and Data Engineering* (2018).
[6] Lin Chen, Wanwangying Ma, Yuming Zhou, Lei Xu, Ziyuan Wang, Zhifei Chen, and Baowen Xu. 2016. Empirical analysis of network measures for predicting high severity software faults. *Science China Information Sciences* 59, 12 (2016), 122901.
[7] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on software engineering* 20, 6 (1994), 476–493.
[8] Giulio Concas, Michele Marchesi, Cristina Monni, Matteo Orrù, and Roberto Tonelli. 2017. Software Quality and Community Structure in Java Software Networks. *International Journal of Software Engineering and Knowledge Engineering* 27, 07 (2017), 1063–1096.
[9] Sergio Di Martino, Filomena Ferrucci, Carmine Gravino, and Federica Sarro. 2011. A genetic algorithm to configure support vector machines for predicting fault-prone components. In *International Conference on Product Focused Software Process Improvement*. Springer, 247–261.
[10] Palash Goyal and Emilio Ferrara. 2018. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems* 151 (2018), 78–94.
[11] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *Acm Sigkdd International Conference on Knowledge Discovery & Data Mining*. 855–864.
[12] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation Learning on Graphs: Methods and Applications. *arXiv preprint arXiv:1709.05584* (2017).
[13] Marian Jureczko and Lech Madeyski. 2010. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. ACM, 9.
[14] Yihao Li. 2017. *Applying Social Network Analysis to Software Fault-Proneness Prediction*. Ph.D. Dissertation. University of Texas at Dallas.
[15] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. 2008. Power laws in software. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 18, 1 (2008), 2.
[16] Wanwangying Ma, Lin Chen, Yibiao Yang, Yuming Zhou, and Baowen Xu. 2016. Empirical analysis of network measures for effort-aware fault-proneness prediction. *Information and Software Technology* 69 (2016), 50–70.
[17] Tim Menzies, Rahul Krishna, and David Pryor. 2015. The Promise Repository of Empirical Software Engineering Data. http://openscience.us/repo. North Carolina State University, Department of Computer Science.
[18] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).
[19] Thanh HD Nguyen, Bram Adams, and Ahmed E Hassan. 2010. Studying the impact of dependency network measures on software quality. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 1–10.
[20] Dario Di Nucci, Fabio Palomba, Rocco Oliveto, and Andrea De Lucia. 2017. Dynamic Selection of Classifiers in Bug Prediction: An Adaptive Method. *IEEE Transactions on Emerging Topics in Computational Intelligence* 1, 3 (2017), 202–212.
[21] Rahul Premraj and Kim Herzig. 2011. Network Versus Code Metrics to Predict Defects: A Replication Study. In *International Symposium on Empirical Software Engineering and Measurement*. 215–224.
[22] Thomas Shippey, Tracy Hall, Steve Counsell, and David Bowes. 2016. So You Need More Method Level Datasets for Your Software Defect Prediction?: Voilà!. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 12.
[23] Lovro Šubelj and Marko Bajec. 2011. Community structure of complex software systems: Analysis and applications. *Physica A: Statistical Mechanics and its Applications* 390, 16 (2011), 2968–2975.
[24] Ayşe Tosun, Burak Turhan, and Ayşe Bener. 2009. Validation of network measures as indicators of defective modules in software systems. In *Proceedings of the 5th international conference on predictor models in software engineering*. ACM, 5.
[25] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. 2016. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.
[26] Thomas Zimmermann and Nachiappan Nagappan. 2008. Predicting defects using network analysis on dependency graphs. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. IEEE, 531–540.