Investigating the Impact of Bug Dependencies on Bug-fixing Time Prediction

Chenglin Li, Yangyang Zhao*
School of Computer Science and Technology
Zhejiang Sci-Tech University
Hangzhou, China
lichenglin2110@163.com, yangyangzhao@zstu.edu.cn

Yibiao Yang, Yuming Zhou
State Key Lab for Novel Software Technology
Nanjing University
Nanjing, China
{yangyibao, zhouyuming}@nju.edu.cn

Liming Nie
School of Computer Science and Engineering
Nanyang Technological University
Singapore
liming.nie@ntu.edu.sg

Zuohua Ding
School of Computer Science and Technology
Zhejiang Sci-Tech University
Hangzhou, China
zouhuading@hotmail.com

***Abstract*—Background: Bug dependencies refer to the link relationships between bugs and related issues, which are commonly observed in software evolution. It has been found that bugs with bug dependencies often take longer time to be resolved than other bugs without any dependencies. Despite the potential impact of bug dependencies on bug-fixing time, previous studies use traditional metrics without considering bug dependencies to build bug-fixing time prediction models. As a result, there is currently little empirical evidence to support the use of bug dependencies in improving prediction accuracy.**

Aims: We aim to conduct a comprehensive empirical study to investigate the value of considering bug dependencies for bug-fixing time prediction.

Method: We define a set of bug dependency metrics based on bug dependencies. We first investigate the correlation between bug dependency metrics and bug-fixing time to investigate whether bugs with more complex dependencies are more time-consuming to be fixed. Next, we employ principal component analysis to study whether bug dependency metrics capture additional dimensions of a bug compared to traditional metrics. Finally, we build multivariate prediction models to explore whether considering bug dependencies can improve the effectiveness of bug-fixing time prediction.

Results: The experimental results suggest that: (1)bugs with more complex dependencies require more time to be fixed; (2) bug dependency metrics are complementary to traditional metrics; (3) considering bug dependencies can improve the effectiveness of bug-fixing time prediction.

Conclusions: These findings highlight the importance of considering bug dependencies in bug-fixing time prediction, and provide valuable insights into the potential impact of bug dependencies on software development processes.

***Index Terms*—bug dependency; bug fixing; bug-fixing time prediction; network analysis**

I. Introduction

Bug fixing is notoriously a time-consuming activity in software development, with Brady's finding that detecting and fixing bugs accounts for over 50% of developers' working time [11]. Anticipating bug-fixing time is crucial for developers to recognize critical bugs and optimize the allocation of limited resources, so as to enhance bug-fixing efficiency. Accurate estimation of bug-fixing time is also essential for software managers to plan software releases and ensure overall software quality. In pursuit of this goal, numerous studies have been conducted to evaluate and predict bug-fixing time.

In order to anticipate bug-fixing time, previous studies primarily extracted metrics from issue tracking systems to build bug-fixing time prediction models. More specifically, these metrics that serve as features of bugs are typically collected from various fields in bug reports, such as bug location, reporter and description [8, 26, 16]. These studies consistently regarded bugs as existing in isolation, while disregarding the link relationships between bugs and other related issues, which we refer to as ***bug dependency***.

As software evolves, bugs frequently become linked to other issues due to various relationships such as being *Related* or *Blocked*, thereby forming bug dependencies. In practice, developers commonly display bug dependencies in the link field (e.g. the "*Issue Links*" field in Jira) of bug reports to facilitate the bug-fixing process. Based on these links, a ***bug dependency graph*** can be created for each bug, consisting of the maximal strongly connected issues that are associated with this bug.

In software projects, bug dependencies are prevalent and complex. As supported by the findings of Lüders et al. [23, 24], approximately 35% of the issues were related to other issues, and many bug relationship graphs in their data were composed of three or more bugs, indicating that many bugs are interlinked rather than isolated from each other. Recently, Vieira et al. discovered that bugs connected to other issues require a significantly longer fixing time than other bugs without any relationships [34]. This leads us to conjecture that the bug dependency may serve as a useful indicator for predicting bug-fixing time. However, there is currently lacking empirical evidence to support this claim.

Moreover, prior studies typically utilized classification models to group bug-fixing time into categories such as slow

*Corresponding Author.

978-1-6654-5223-6/23$31.00

or fast, or into intervals such as three months or one year, solely offering developers a rough estimate of when a bug is likely to be resolved. [2, 4, 26, 1]. Nevertheless,classifying bugs into categories or intervals may not provide sufficient information for developers to accurately judge the bug-fixing time. Continuous value prediction, on the other hand, can provide developers with more precise estimates and be more useful in this regard.

To this end, we are motivated to conduct a thorough study to investigate whether considering bug dependencies can improve the effectiveness of bug-fixing time prediction under the scenario of continuous value prediction. Our main contributions are as follows:

1) This study presents the first empirical study to investigate the value of bug dependencies in bug-fixing time prediction. We propose a set of metrics to quantify bug dependencies based on bug dependency graphs from various dimensions, providing guidance for future research on utilizing bug dependencies.
2) We analyze the correlation between bug dependency metrics and bug-fixing time to determine whether bugs with more complex dependencies are more time-consuming to fix. Our results reveal that most bug dependency metrics are significantly positively correlated with bug-fixing time.
3) We utilize principal component analysis to investigate whether bug dependency metrics capture additional dimensions compared to traditional metrics. Our findings demonstrate that bug dependency metrics provide new and complementary views of bugs compared to traditional metrics.
4) We build two types of bug-fixing time prediction models, including "T" mode (built with only traditional metrics) and "T+D" model (built with both bug dependency metrics and traditional metrics). The experimental results indicate that considering bug dependencies can significantly improve the effectiveness of the bug-fixing time prediction.

Paper outline The rest of this paper is organized as follows. In Section II, we explain our research motivation. In Section III, we describe the bug dependency metrics. We present research questions and experimental methods in Section IV. In Section V, we report the detailed experimental results with respect to each of the research questions. Section VII represents related works. In Section VI, we discuss the threats to validity. Finally, Section VIII concludes the paper and outlines directions for future work.

II. Motivation

Bugs are inevitable in software projects. To effectively manage them, many projects employ an issue tracking system(IST), such as Jira and Bugzilla, which enables developers to streamline their bug-fixing processes and prioritize issues based on their severity and impact. With the growth in the number of bugs, the relationships between them have become complex. For instance, bugs can be connected to other issues through the *Issue Links* field in Jira, which is labeled with link types such as *Blocked*, *Reference* and *Require*. Fig 1 illustrates the issue links of bug report Hadoop-15418, showing two linked bug issues and recording the link types and direction of links. This field provides a convenient way to connect related issues and indicate their interdependence.

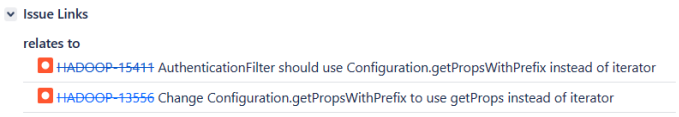

Fig. 1. The issue links of bug report Hadoop-15418 in Jira.

Since many bugs are interconnected with other issues, they often form a network of dependencies that can complicate the process of fixing them. After manually examining bug reports from several large open-source projects, we discover that in the bug dependency graphs, bugs are not simply connected one-to-one but often have many-to-many relationships with other bugs, forming multiple connections between different bugs. Based on this finding, we conjecture that bugs with complex dependencies may take longer to fix. This conjecture is supported by a real example shown in Figure 2.

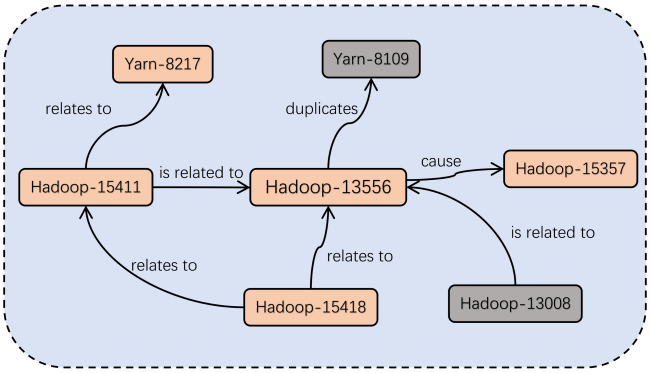

Fig. 2. An example of bug dependency graph

Fig 2 illustrates the actual bug dependency graph of bug Hadoop-13556, which consists of 7 nodes and 7 directed edges. Each node in the graph represents an issue with a unique issue id. Specifically, the orange nodes represent bug issues, while the grey ones represent other non-bug issues. The edge between two nodes represents the presence of dependencies between them, and the type of dependency is annotated on the edge, which is labeled in the link field by developers. As can be seen from Fig 2, Hadoop-13556 is the bug with the highest incoming and outgoing degrees, which occupies the most complex position in the graph. Hence, it can be considered the central bug in this graph. As a preliminary attempt to investigate whether bugs with more complex dependencies take longer to fix, we manually check the fixing time of these bugs. Consistent with our conjecture, Hadoop-13556 consumes the longest fixing time of 409.23 days. In contrast, other bugs take less time to fix. For example, Hadoop-15418, which has direct dependencies on two other bugs, takes 175.01 days to fix. And Hadoop-15357, which is directly linked to only one bug, costs just 8.16 days to fix. In terms of this example, it is clear that there are significant

TABLE I
DESCRIPTION OF THE STUDIED NETWORK ANALYSIS METRICS.

type	metrics	description
Ego network	Size	# alters that ego is directly connected to
	Ties	# ties in the ego network
	Pairs	# pairs of alters in the ego network
	Density	% possible ties that are actually present
	nWeakComp	# weak components in the ego network
	pWeakComp	# weak components normalized by size
	2StepReach	# nodes ego can reach within two steps
	ReachEffic	2StepReach normalized by sum of alters' size
	Broker	# pairs not directly connected to each other
	nBroker	Broker normalized by the number of pairs
	EgoBetw	% all shortest paths across ego
	nEgoBetw	normalized EgoBetween (by ego size)
Global network	EffSize	# bugs minus the average degree of bugs
	Efficiency	effsize divided by number of bugs.
	Constraint	The extent to which ego is constrained.
	Hierarchy	measures how the constraint measure is distributed across neighbors.
	Degree	# issues adjacent to a given bug
	Degree centrality	Centrality score according to the degrees
	Closeness	How close to other vertices.
	Eigenvector centrality	Centrality score according to eigenvector.
	Betweenness centrality	Centrality score according to betweenness.

differences in the fixing time of these bugs with different levels of dependency complexity, and the bugs located in more complex positions in the bug dependency graph require longer time to fix.

Despite of the above preliminary observations, there is a lack of statistical evidence to determine whether the degree of bug dependency is significantly correlated with bug-fixing time, or whether considering bug dependency can significantly improve the predictability of fixing time. To bridge this gap, we conduct a comprehensive study to investigate the impact of bug dependencies on bug-fixing time and the practical implications of considering bug dependency for bug-fixing time prediction.

III. BUG DEPENDENCY METRICS

In order to study bug dependency, we attempt to quantify the bug dependency in two ways. Firstly, since the maximal strongly connected issues that are associated with a bug form an independent dependency graph (like Figure 2), we can measure bug dependency using *network analysis metrics* based on the graph. Additionally, we propose a set of metrics that quantify the degree of dependency in the graph using a counting method, which is named *dependency counting metrics*. We provide a detailed description of the two types of bug dependency metrics as follows.

Network analysis metrics Network analysis metrics were first proposed by Wasserman et al. in social sciences research [36], where they were used to describe complex interrelationships between people. In this study, we use these network metrics to describe the interrelationships between issues in the bug dependency graph. As shown in Table I, the network metrics

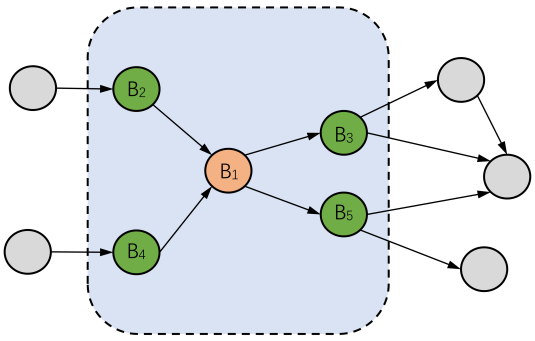

Fig. 3. An example of an ego network.

consist of two dimensions: Ego network metrics and Global network metrics, which differ in the analysis scope. Ego network metrics focus on the importance of a bug to its neighboring issues, while global network metrics provide a more complete depiction of bug dependencies by describing the importance of bugs in the overall dependency graph. These two types of metrics are used to provide a more complete depiction of bug dependencies from different perspectives. In Table I, the names and descriptions of these metrics are shown in the second and third columns respectively.

(1) *Ego network metrics*: Ego network analysis focuses on the node (also called the ego in the network) and how it is connected to its neighbors. These metrics provide a localized view of bug dependencies. Figure 3 illustrates an example of an ego network, which includes the ego node B_1, adjacent nodes that depend on the B_1 (i.e. B_2 and B_4), nodes on which B_1 depends (i.e. B_3 and B_5), and inter-dependencies among these nodes. In this example, the scope of ego network analysis would be the subgraph within the blue box of Figure 3. In our study, we consider each bug and its direct neighbors, as well as the dependencies between them, for ego network analysis. Ego network metrics include 12 metrics, which are presented in the top half of Table I. These metrics describe the importance of a bug to its neighboring issues in the bug dependency graph. we compute them in an undirected way to describe the influence and connectivity strength of the bug on its neighboring nodes.

(2) *Global network metrics*: In contrast to Ego network metrics, global network analysis focuses on the overall importance of bugs in the dependency graph. We employ two types of global network metrics, namely, structural hole metrics and centrality metrics. The structural hole metrics reflect the influence of nodes in a network, and measure the closeness of bug dependencies. More specifically, a bug that is closely linked to other issues will have a higher value of the structural hole metric. Figure 4 depicts an example of a structural hole. In the left sub-figure, A_1, B_1, and C_1 are mutually connected, so their influence in the network is equal. However, in the right sub-figure, the absence of a connection between nodes B_2 and C_2 results in a structural hole. As a result, node A_2 has more influence in the dependency graph than B_2 and C_2. In our study, structural hole metrics include EffSize, Efficiency, Constraint, and Hierarchy. On the other hand, the centrality metrics describe the position of a node in the network, including five metrics: Degree, Degree centrality,

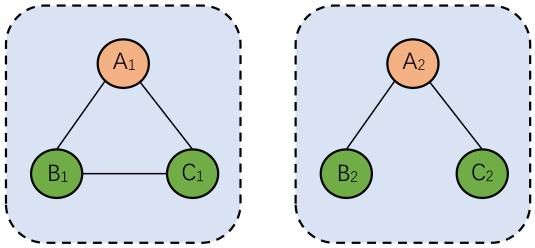

Fig. 4. An example of a structural hole.

Closeness, Eigenvector centrality, and Betweenness centrality. Generally, a bug related to more other issues will have a higher value of the centrality metric.

Dependency counting metrics In addition to network analysis metrics, we propose a set of dependency counting metrics, which are collected in a counting way based on the bug dependency graph. The names and descriptions of these metrics are illustrated in Table II. These metrics capture information on the number, direction, type, and extent of bug dependencies. In our study, they are used in conjunction with network metrics to provide a more comprehensive description of bug dependencies.

To illustrate the computations of the proposed dependency counting metrics, we take the bug Hadoop-13356 in Figure 2 as an example. For this bug, both BC and EDC are equal to 7, as the graph comprises of 7 nodes and edges. A higher value for BC and EDC indicates a more complex dependency relationship between nodes. DD measures the density of the entire dependency graph, which is 1/6 (=7/(7×6)) for this bug. Hadoop-13356 has 3 inward and 2 outward dependencies, resulting in ID, OD and TD values of 3, 2, and 5, respectively. IDT and ODT count the number of types of inward and outward dependencies. For Hadoop-13356, there is one type of inward dependency and two different types of outward dependency, resulting in IDT, ODT and TT values of 1, 2,and 3(=1+2), respectively. Bugs with more types of dependencies may take longer to fix, as developers may need different amounts of time to deal with different types of dependencies. ICD and OCD count the number of cross-project dependencies, and TCD is the sum of them. In this case, Hadoop-13356 points to Yarn-8109 from another project, so ICD and OCD are 0 and 1 respectively, resulting in TCD value of 1 (=0+1). The existence of cross-project dependencies may increase the complexity of fixing bugs, as developers may need to deal with bugs from different projects simultaneously, potentially resulting in longer fix time.

TABLE II
THE DESCRIPTION OF DEPENDENCY COUNTING METRICS.

dependency counting metrics	description
Bug count(BC)	# nodes in the bug dependency graph.
Entire dependencies count (EDC)	# edges in the bug dependency graph.
Dependencies density (DD)	the number of nodes divided by the maximum number of edges. EDC/(BC*(BC-1))
Inward dependencies (ID)	# dependencies from other issues.
Outward dependencies (OD)	# dependencies on other issues.
Total dependencies(TD)	Sum of ID and OD.
In-dependencies type (IDT)	# types of Inward dependencies.
Out-dependencies type(ODT)	# types of Outward dependencies.
Total type (TT)	Sum of IDT and ODT.
In-cross dependencies (ICD)	# dependencies from issues in other project.
Out-cross dependencies (OCD)	# dependencies on issues in other projects.
Total cross dependencies (TCD)	Sum of ICD and OCD.

IV. Methodology

In this section, we first present the research questions in our study, followed by the introduction of the experimental methods with respect to each question. Then, we describe the data collection process.

A. Research questions

RQ1: Are bugs with more complex dependencies more time-consuming to fix?

Each bug, acting as a node within a dependency graph formed by its associated interconnected issues, occupies a distinct position within the graph. The objective of RQ1 is to investigate whether bugs with more intricate dependencies take longer time to fix. To answer RQ1, we examine the statistical correlation between bug dependency metrics and bug-fixing time. Bug dependency metrics provide insight into the complexity and importance of a bug. The higher a bug's dependency metric value, the more central and complex its position within the dependency graph. If the bug dependency metrics are significantly positively correlated with bug-fixing time, it suggests that bugs with more complex dependencies require more time to fix.

RQ2: Does considering bug dependencies capture additional dimensions of a bug compared to ignoring them?

The aim of RQ2 is to study whether considering bug dependencies is crucial for describing the holistic attributes of a bug. To answer RQ2, we examine whether bug dependency metrics are redundant with respect to the metrics without considering dependencies, so as to provides insights into the unique contribution of bug dependency metrics in interpreting bugs. If the bug dependency metrics are able to capture additional dimensions, it is necessary to collect bug dependency metrics to help developers comprehensively understand a bug and make more accurate predictions about the time and effort required to fix it.

RQ3: Are bug dependencies useful for improving the performance of bug-fixing time prediction?

If the results of RQ1 show that bug dependency metrics are not correlated with bug-fixing time, or the results of RQ2 show they are redundant with other metrics, there is no need to collect bug dependency metrics in practice. Otherwise, further study is needed on the value of bug dependencies on bug-fixing time prediction. RQ3 is to determine whether considering bug dependencies can improve the effectiveness of bug-fixing time prediction. Understanding the value of bug dependency metrics in predicting bug-fixing time is crucial for effective bug triage and resource allocation in software development.

B. Experimental methods

To address RQ1, we examine the relationship between bug dependency metrics and bug-fixing time. Since our metrics consist of continuous values, we utilize Pearson correlation analysis to measure their correlation with bug-fixing time. The Pearson correlation coefficient is a widely accepted method of determining the linear relationship between variables. In our study, we investigate whether each bug dependency metric exhibits a significant positive correlation with bug-fixing time at significance levels of 0.1, 0.05, and 0.01.

TABLE III
THE DETAILED DESCRIPTION OF TRADITIONAL METRICS.

metrics	desription
Priority (1 metric)	The level of importance of the bug.
Average fixing time of priority (1 metric)	Average fixing-time of previous bugs of the same priority.
Length of summary and description (1 metric)	Text length of the summary and description.
Open bugs (5 metrics)	#open bugs for the last 7 days, 30 days, 6 months, 1 year, and life time the project.
Fixed bugs (5 metrics)	#bugs fixed in the project in the last 7 days, 30 days, 6 months, 1 year, and all previous bugs. They indicate the bug fix rate for the project.
Average Fix time (5 metrics)	Average fixing-time of bugs for the last 7 days, 30 days, 6 months, 1 year, and lifetime.
Reporter's open bugs (1 metric)	#bugs that the bug reporter has ever reported.
Reporter's average fixing-time (1 metric)	Average fixing-time for bugs previously reported by the bug's reporters.
Reporter Popularity (3 metrics)	(1) Overall Popularity: The ratio of bugs fixed by the reporter to all bugs in the project. (2) Recent Popularity: The ratio of bugs fixed by the reporter to all bugs within a year. (3) Relative Recent/Overall Popularity: The ranking of the above metrics relative to other reporters in the project.

To address RQ2, we investigate whether bug dependency metrics are redundant with respect to metrics without considering dependencies. In this study, we refer to metrics that do not consider dependencies as *traditional metrics*, which are listed in Table III. Table III illustrates the name and description of each class of metrics, along with the number of metrics in each class shown in brackets. There are total 23 traditional metrics collected from bug reports, which are proposed by Marks et al. [26] and widely used to study bug-fixing time [39, 12, 34, 1, 33]. In this study, we choose these metrics as the baseline metrics.

To demonstrate the redundancy of bug dependency metrics with respect to traditional metrics, we utilize principal component analysis(PCA) to determine whether bug dependency metrics can capture additional dimensions of information that traditional metrics cannot. PCA is often used to reduce data dimensions by converting a set of potentially correlated variables into a set of linearly uncorrelated variables through an orthogonal transformation. The transformed set of variables is called a principal component(PC). We retain only the PCs with eigenvalues greater than 1 for interpretation. In particular, we use the results of the rotated component matrix to map the independent variables to PCs. If at least one PC consists solely of bug dependency metrics, it indicates that the bug dependency metrics are not redundant with respect to traditional metrics.

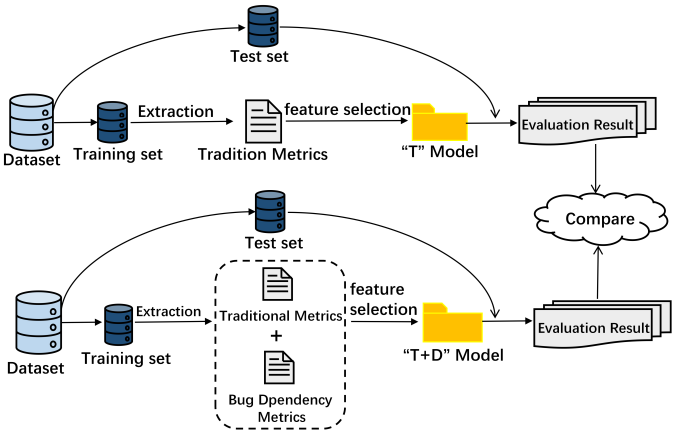

Fig. 5. The analysis method for RQ3 in our study.

For RQ3, Figure 5 illustrates an overview of the analysis method. In order to answer RQ3, we build two types of multivariate linear regression models: (1)"T" model (as the baseline model, using only traditional metrics); (2)"T+D" model (using both bug dependency metrics and traditional metrics). When building these models, we first normalize the metrics, and use the p value as the criterion to perform a forward stepwise variable selection. Then, we choose to use linear regression analysis to construct prediction models due to its simplicity, proven effectiveness and widespread use. After building these models, we conduct 20 times two-fold cross-validation to compare their predictive effectiveness. Consequently, each model has 20×2=40 predictive effectiveness results. Specifically, we evaluate "T+D" model and "T+D" model using three effectiveness indicators: Mean Absolute Error (MAE), Mean Square Error (MSE), and Root Mean Square Error (RMSE), which provide different perspectives on the accuracy of the predicted results and are commonly used in performance evaluation. With these results, We utilize Wilcoxon's signed-rank test to examine whether "T+D" model outperforms "T" model at significance levels of 0.1, 0.05, and 0.01. Additionally, we presented the improvement of "T+D" model over "T" model".

C. Data collection

We use the dataset created by Vieira et al. [35], which includes bug reports of 55 large open-source projects spanning the period from 2009 to 2018. We select this dataset as the primary source of our experimental data for two main reasons. Firstly, it contains comprehensive information that is essential for our study, such as bug creation and resolution time, issue links, and various attributes that have been used in previous studies on bug-fixing time prediction. Secondly, the projects in this dataset leverage Jira to manage their bugs. Jira offers an "*issue links*" field, which facilitates the

TABLE IV
THE DETAILED INFORMATION OF THE STUDIED NINE PROJECTS.

project	#bug	#bug related to others	%bugs that are interdependent with other issues
Derby	1083	670	61.87%
Hadoop	2861	1522	53.20%
Hbase	6693	1818	27.16%
Hdfs	3214	1661	51.68%
Hive	7105	2807	39.51%
Solr	2249	872	38.77%
Spark	6380	1567	24.56%
Ww	725	206	28.41%
Yarn	2090	931	44.55%
average	3600	1339	41.08%

tracking of dependencies between issues, enabling us to create a dependency graph for each bug. Based on this dataset, we conduct additional data filtering to ensure the quality and relevance of the data, and collect bug-dependency metrics to facilitate our analysis. The details are outlined below.

To ensure the availability of rich bug dependency data, we perform filtering on the dataset provided by Vieira et al. Specifically, for each project, we count the number of bugs that are interconnected with other issues, as well as their proportion to all bugs. We select the projects with high proportion as the experimental projects in this study. Based on this criterion, we finally select nine projects and record their details in Table IV. Each row represents the project name, the total number of bugs, the number of bugs with dependencies and their proportion to all bugs. As shown in Table IV, the proportion of bugs with dependencies ranges from 24.56% to 61.87% across the selected nine projects, with six projects exceeding 35%. On average, each project has 3600 bugs, of which 1339 have dependencies, accounting for 41.08% of the total number of bugs.

Furthermore, to quantify bug dependencies, we collect bug dependency metrics for each bug in two steps. First, we construct a bug dependency graph by treating bugs and their connected issues as nodes and drawing edges between them. Second, we collect bug dependency metrics for each bug based on its dependency graph, including network analysis metrics and dependency counting metrics. We utilize the NetworkX Python library to construct dependency graphs, which allows the creation, manipulation, and analysis of the structure, dynamics, and functions of complex networks. To collect network metrics, we used built-in functions in the NetworkX library that cover the studied network analysis metrics. Additionally, we develop scripts to collect dependency counting metrics based on NetworkX library. These scripts have undergone a double review process by authors with extensive experience in software development.

Finally, we calculate bug-fixing time as the difference between creation time and resolution time. We set the bug-fixing time as the dependent variable and the bug-related features (i.e. bug dependency metrics and traditional metrics) as independent variables. With the above process of data collection, we obtain the experimental dataset[1].

[1] https://github.com/linchengLi-l/BugDependency-Dataset

V. Experimental results

In this section, we report the detailed experimental results with respect to each question. Section V-A presents the results of the correlation analysis between the bug dependency metrics and bug-fixing time(RQ1). Section V-B shows the results of whether bug dependency metrics are redundant to traditional metrics(RQ2). Section V-C reveals the effect of considering bug dependency on improving the performance of bug-fixing time prediction (RQ3).

A. RQ1: Are bugs with more complex dependencies more time-consuming to fix?

To answer RQ1, we employ the Pearson correlation coefficient to investigate the correlation between bug dependency metrics and bug-fixing time. The experimental results are illustrated in Table V. The first column of Table V denotes two categories of bug dependency metrics, i.e. network analysis metrics and dependency counting metrics. The second column lists the name of each metric. In terms of columns six to fourteen, each column corresponds to one dataset, which displays the correlation coefficients between the metrics and bug-fixing time within the corresponding dataset, along with the significance levels of 0.01 (denoted by ***), 0.05 (denoted by **), and 0.1 (denoted by *) . Particularly, the non-significant correlations are indicated by "/". In addition, columns third to fifth summarize the total count of positive (in the "+" column), negative (in the "-" column), and non-significant (in the "/" column) correlations between each metric and bug-fixing time across all nine datasets. These three columns provide a statistical overview of the relationship between different metrics and bug-fixing time for the nine projects

For each metric, we consider it is significantly correlated with bug-fixing time if it shows a significant correlation with bug-fixing time in most datasets. From Table V, we observe that most of bug dependency metrics are positively correlated with bug-fixing time in most datasets. More specifically, the correlation coefficients mainly fall within the range of 0.1 to 0.3. And a majority of metrics exhibit a significant positive correlation with bug-fixing time across more than five datasets. This suggests that bugs with more complex dependencies tend to require more time to fix. However, we also observe that the metrics, 2stepReach, ReachEffic, Broker, and nBroker, highlighted in grey, are not significantly positively correlated with bug-fixing time at most projects. Conceptually, these four metrics evaluate the impact of studied bugs on other pairs of bugs that form links. For instance, the Broker metric measures the number of pairs of unconnected bugs that a bug connects as an intermediate point. These metrics may not be intuitive enough to reflect the complexity of the studied bugs within the network structure. This may be the reason why they are not significantly and positively correlated with bug-fixing time.

Conclusion 1: The results of RQ1 reveal that the majority of the bug dependency metrics are significantly positively correlated with bug-fixing time. The observations suggest that bugs located in more complex and central positions in the

TABLE V
PEARSON CORRELATION BETWEEN THE BUG-FIXING TIME AND BUG DEPENDENCY METRICS.

type	metrics	+	-	/	Derby	Hadoop	Hbase	Hdfs	Hive	Solr	Spark	Ww	Yarn
Network analysis metrics	Size	9	0	0	0.253***	0.139***	0.120***	0.128***	0.160***	0.178***	0.217***	0.128***	0.287***
	Ties	9	0	0	0.251***	0.139***	0.120***	0.128***	0.159***	0.177***	0.216***	0.128***	0.285***
	Pairs	9	0	0	0.141**	0.133***	0.115***	0.119***	0.122***	0.147***	0.214***	0.110***	0.281***
	Density	7	1	1	0.061*	/	0.065***	0.041**	0.099***	0.114***	0.120***	0.104***	-0.217***
	WeakComp	9	0	0	0.254***	0.139***	0.120***	0.130***	0.163***	0.178***	0.217***	0.124***	0.295***
	nWeakComp	9	0	0	0.219***	0.100***	0.099***	0.100***	0.147***	0.167***	0.181***	0.123***	0.225***
	2StepReach	3	0	6	/	/	0.003**	/	/	0.040**	0.025**	/	/
	ReachEffic	2	0	7	/	/	0.039***	/	/	0.028**	/	/	/
	Broker	0	0	9	/	/	/	/	/	/	/	/	/
	nBroker	1	0	8	/	/	/	/	/	/	0.028**	/	/
	EgoBetw	8	0	1	0.084***	0.112***	0.105***	0.069***	0.101***	0.088***	0.161***	/	0.268***
	nEgoBetw	9	0	0	0.186***	0.122***	0.107***	0.091***	0.112***	0.095***	0.162***	0.067*	0.267***
	Dgree	9	0	0	0.253***	0.139***	0.120***	0.128***	0.160***	0.178***	0.217***	0.128***	0.287***
	EffecSize	9	0	0	0.205***	0.126***	0.107***	0.135***	0.136***	0.118***	0.17***	0.108***	0.302***
	Constraint	5	0	4	/	/	0.041***	0.061***	0.056***	0.070***	0.054***	/	/
	Efficiency	9	0	0	0.146***	0.08***	0.072***	0.106**	0.101***	0.098**	0.102***	0.081**	0.131***
	Hierarhcy	8	0	1	0.159***	0.066***	0.086***	0.076***	0.13***	0.142***	0.154***	0.112***	/
	DCentrality	8	0	1	0.166***	0.070***	0.086***	0.074***	0.126***	0.146***	0.160***	0.128***	0.088***
	Closeness	7	0	2	0.107***	0.037*	0.063***	/	0.097***	0.108***	0.142***	0.089**	/
	Eigenvector	5	0	4	/	/	0.041***	/	0.060***	0.093***	0.109***	0.074**	/
	BetwCentrality	8	0	1	0.159***	0.100***	0.066***	0.097***	0.094***	0.09***	0.116***	/	0.181***
Dependency counting metrics	BC	9	0	0	0.154***	0.101***	0.108***	0.108***	0.125***	0.143***	0.196***	0.118***	0.183***
	EDC	9	0	0	0.155***	0.101***	0.108***	0.107***	0.123***	0.144***	0.197***	0.121***	0.181***
	DD	9	0	0	0.143***	0.097***	0.105***	0.103***	0.111***	0.128***	0.191***	0.114***	0.186***
	ID	9	0	0	0.203***	0.110***	0.093***	0.068***	0.130***	0.156***	0.186***	0.065*	0.128***
	OD	9	0	0	0.194***	0.102***	0.091***	0.127***	0.114***	0.106***	0.139***	0.123**	0.207***
	IDT	9	0	0	0.189***	0.104***	0.092***	0.050***	0.119***	0.159***	0.178***	0.079***	0.103***
	ODT	9	0	0	0.146***	0.101***	0.084***	0.115***	0.116***	0.117**	0.121***	0.121***	0.178***
	ICD	7	0	2	/	0.047***	0.052***	0.038**	0.033***	/	0.052***	0.038***	0.089***
	OCD	7	0	2	/	0.073***	0.075***	0.054***	0.053***	0.048**	0.037***	/	0.121***
	TD	9	0	0	0.256***	0.139***	0.119***	0.128***	0.161***	0.176***	0.216***	0.128***	0.214***
	TT	9	0	0	0.229***	0.136***	0.116***	0.109***	0.157***	0.188***	0.199***	0.134***	0.181***
	TCD	7	0	2	/	0.080***	0.083***	0.060***	0.058***	0.050***	0.040***	/	0.134***

dependency graph take longer to fix, which is in line with our conjecture.

B. RQ2: Does considering bug dependencies capture additional dimensions of a bug compared to disregarding them?

To answer RQ2, we use principal component analysis(PCA) to determine whether bug dependency metrics are redundant to traditional metrics. Figure 6 displays the rotated components resulting from PCA for the nine datasets. In Figure 6, each subplot corresponds to one distinct data set, and consists of multiple bars. Each bar represents a principal component (PC), with the horizontal axis showing the PC's ordinal number and the corresponding y-axis displaying the cumulative percentage of variances explained by the corresponding PC. There are three categories of PCs: (1) PCs comprising solely traditional metrics (highlighted in gray); (2) PCs comprising solely bug dependency metrics (highlighted in blue); and (3) PCs comprising a combination of both types of metrics (highlighted in orange).

As shown in Figure 6, in the nine datasets, the metrics are grouped into 10 to 13 different principal components, all of which explain at least 84% of the variances. Specifically, each dataset has a minimum of three principal components consisting of solely bug dependency metrics, with one dataset having up to six. Notably, in the Hadoop dataset, half of the principal components are exclusively comprised of bug dependency metrics. In addition, across all datasets, the first principal component is exclusively composed of bug dependency metrics. The first principal component is typically considered the most important in principal component analysis as it explains the largest variance in the data. This observation highlights the importance of bug dependency metrics in capturing different properties. Furthermore, we examine the metrics contributing to the first principal component to explore the commonalities, and find that *Size*, *Ties*, *Degree*, and *nWeakComp* appeared in the first principal component of all datasets.The fact that these metrics consistently belong to the first principal component suggests that they play a significant role in capturing the primary variability and underlying structure of the data.

In contrast, the third type of principal component, which incorporates both types of metrics and is highlighted in orange, is infrequently observed in the analyzed datasets. As can be seen from Figure 6, this type of PC is only present once in the Hbase dataset. The possible reason for this may be the significant differences in the calculation methods of bug dependency metrics and traditional metrics, which capture distinct features of bugs. Therefore, they are less likely to appear in the same principal component.

Conclusion 2: The results from principal component analysis reveal that, for each project, at least three principal components consist solely of bug dependency metrics. This indicates that the bug dependency metrics can capture addi-

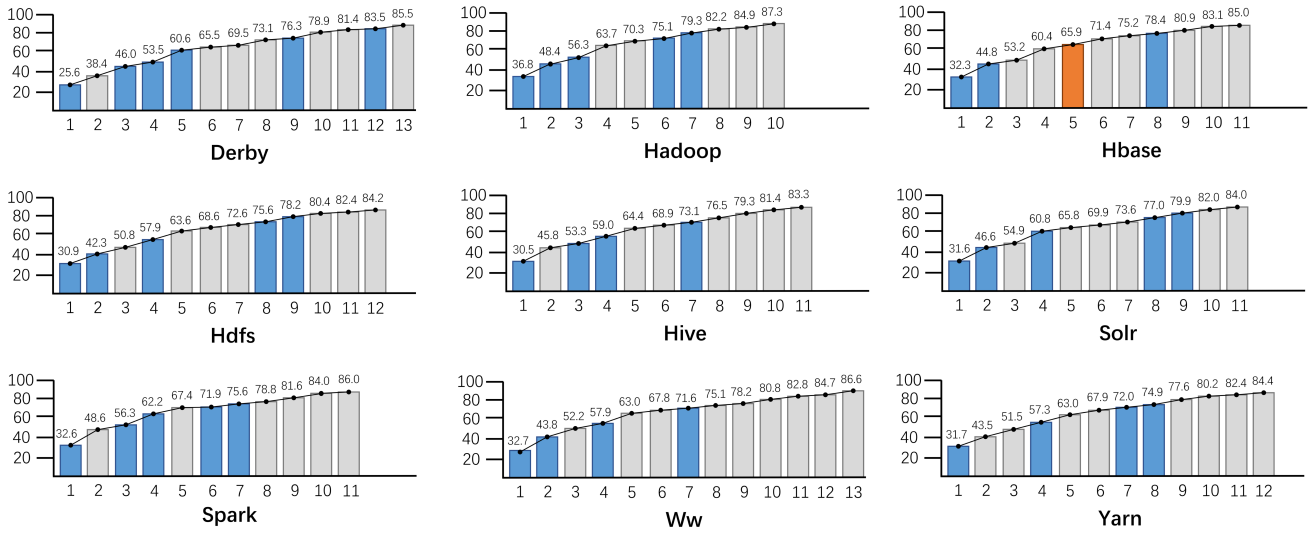

Fig. 6. The rotated components on nine datasets from PCA.

tional dimensions of a bug compared to traditional metrics which disregard bug dependencies.

C. RQ3:Are bug dependencies useful for improving the performance of bug-fixing time prediction?

For RQ3, we conduct 20 times two-fold cross-validation for "T+D" and "T" models and evaluate the performance improvement for "T+D" model over "T" model. The improvement denotes the reduction in MAE, MSE, and RMSE. Figure 7 illustrates the distribution of improvement percentages in the performance of "T+D" model compared to "T" model across nine projects. Specifically, each subplot is divided into three parts, depicting the percentages of the improvement in MAE, MSE, and RMSE, respectively. In the subplots, each point represents the outcome of a single validation, and its corresponding y-axis represents the improvement percentage of the "T+D" model for the validation.

From Figure 7 we observe that, for a majority of validations, the improvement of "T+D" model is positive. It suggests the "T+D" model outperforms the "T" model in most comparisons. Although there are cases that the improvement percentage is less than zero, the number of such cases was relatively small compared to the overall results.

In addition, to evaluate the significance of the improvement, we conduct Wilcoxon's signed rank test on the evaluation indicators for each dataset to determine whether the "T+D" model significantly outperforms the "T" model. We report the average improvement in prediction performance of the "T+D" model versus the "T" model, as well as the p-value for the significance of the difference in results between the two models. Table VI presents the evaluation results of the "T+D" model and the "T" model for each dataset. The first column lists the names of the datasets. Columns 2, 4, and 6 show the percentage reduction in MAE, MSE, and RMSE achieved by the "T+D" model compared to the "T" model. A higher percentage reduction indicates better performance of the "T+D" model. Columns 3, 5, and 7 show the p-values and significance levels of the Wilcoxon signed rank test, which measures the statistical significance of the difference between the two models in terms of MAE, MSE, and RMSE.

The results in Table VI clearly demonstrate that the "T+D" model significantly outperforms the "T" model in all studied datasets. The "T+D" model achieves lower MAE, MSE, and RMSE values than the "T" model, with an average reduction of 2.69%, 4.68%, and 2.44%, respectively. These differences are statistically significant, with p-values less than 0.01 in most cases, indicating that the "T+D" model's performance improvement over the "T" model is not due to chance.

Furthermore, When building these models, we perform a forward stepwise variable selection to identify the metrics that contribute most to the predictive model. By examining the outcomes of the variable selection, we consistently observe that five metrics, namely *Density*, *nWeakComp*, *IDT*, *ODT*, and *ID*, are frequently selected across the nine datasets. This suggests that these metrics play a greater role in build "T+D" models compared to other metrics, potentially leading to improved performance of the models. This finding provide valuable insights for future predictive model construction, allowing for a reduction in the collection of redundant metric data.

Conclusion 3: The results of RQ3 show that incorporating both bug dependency metrics and traditional metrics in bug-fixing time prediction models leads to better prediction performance than using only the traditional metrics. It suggests that considering bug dependency can significantly improve the effectiveness of bug-fixing time prediction.

VI. Validity

In this section, we discuss the most critical threats to the validity of our study, including Construct validity, Internal validity, and External validity.

Construct Validity The main threat to construct validity lies in the accuracy of the independent and dependent variables. In our study, we investigate the impact of bug dependency on bug

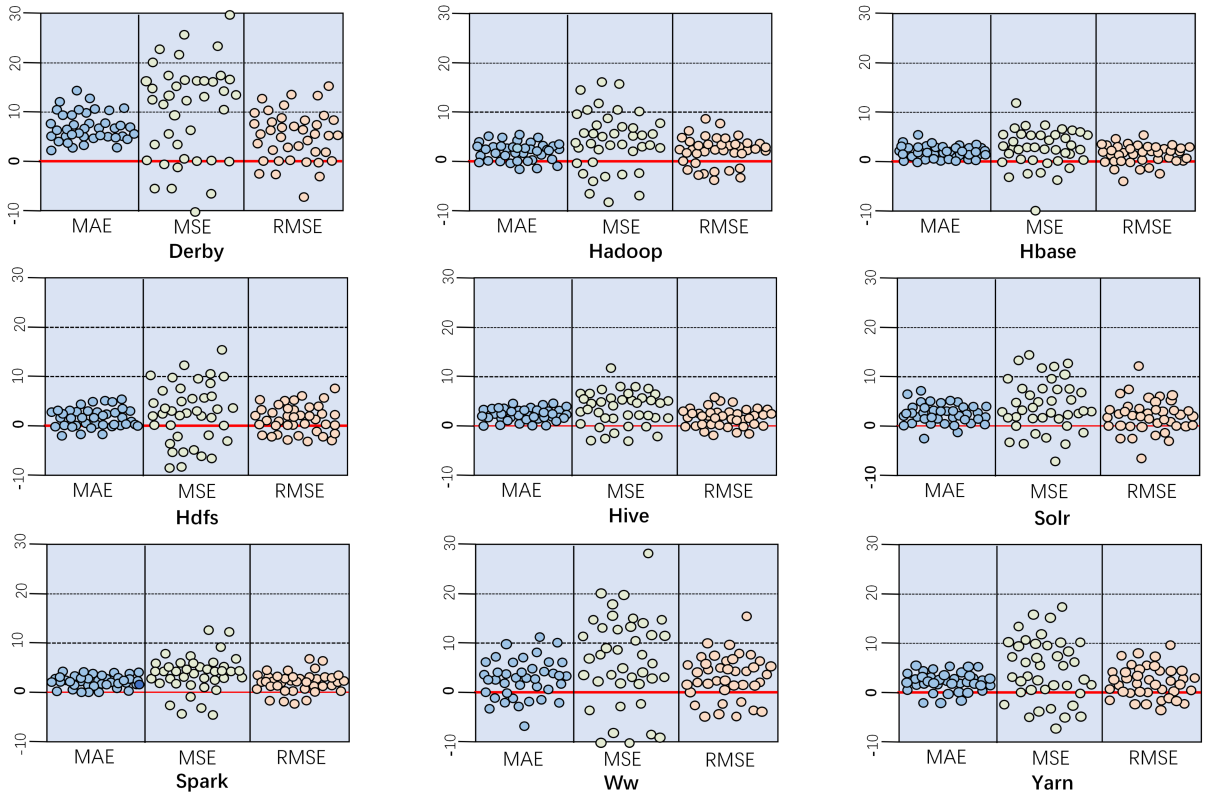

Fig. 7. The result of comparison in MAE, MSE and RMSE

TABLE VI
THE IMPROVEMENT IN TERMS OF THE PREDICTION PERFORMANCE.

	MAE		MSE		RMSE	
	↑%	P	↑%	P	↑%	P
Derby	7.29%	<0.01	10.29%	<0.01	5.44%	<0.01
Hadoop	1.93%	<0.01	4.40%	<0.01	2.28%	<0.01
Hbase	1.81%	<0.01	2.92%	<0.01	1.50%	<0.01
Hdfs	1.36%	<0.01	2.32%	0.019	1.22%	0.018
Hive	2.24%	<0.01	3.80%	<0.01	1.93%	<0.01
Solr	2.44%	<0.01	4.25%	<0.01	2.21%	<0.01
Spark	1.98%	<0.01	3.55%	<0.01	1.81%	<0.01
Ww	2.78%	<0.01	6.02%	<0.01	3.17%	<0.01
Yarn	2.42%	<0.01	4.59%	<0.01	2.38%	<0.01
average	2.69%		4.68%		2.44%	
	9/0/0		9/0/0		9/0/0	

fix-time prediction. The results rely on the quality of research data. Therefore, ensuring the correctness and completeness of bug dependency data is critical. In this study, we collect bug dependency metrics based on the link attribute in the bug report from Jira. These links are added by the developer and indicate the relationships between issues. Jira records more detailed link data than other issue tracking systems, and many previous studies have collected data from it [34, 24, 23, 27]. In terms of the the link attribute, there are human factors that may potentially compromise the accuracy and integrity of our data. For example, human error may occur when developers manually add link attributes and mistakenly provide incorrect information. While this type of error cannot be completely eliminated, we have taken steps to mitigate this threat by collecting data from Apache projects. These projects are known for their higher quality and maturity, which helps to minimize the impact of human error on the data. Furthermore, during the calculation of bug dependency metrics, we use the NetworkX library to build the bug dependency graph, which is widely used in many previous studies to explore networks [19, 29, 31]. NetworkX is a Python library that facilitates the creation, manipulation, and analysis of complex network graphs, including their structure, dynamics, and functionality. The library includes functions that assist us in computing the metrics. Therefore, we believe the construct validity of the independent variables in our study can be considered as satisfactory. In terms of the dependent variable, consistent with many previous studies, we measure the bug-fixing time as the difference between the time of bug creation and resolution, i.e., resolved time minus created time. This is a widely used and accepted approach for measuring bug-fixing time, as it captures the duration between the identification of a bug and its resolution, which is an important factor in software development and maintenance. As a result, we believe the dependent variable used in this study is reasonable and acceptable.

Internal Validity In our study, the potential threat to internal validity is the presence of duplicated issues in the bug dependency graphs, which may introduce noise into the data. In Jira, duplicated issues are connected through "*duplicate*" or "*clone*" links. During the construction of the bug dependency graph, we choose to retain these two types of links to preserve

the original structure of the network. In practice, developers may unintentionally or intentionally attach links to duplicates instead of the original bug. Removing these links may result in the loss of dependencies between the original bug and other issues. Additionally, some "*duplicate*" or "*clone*" links may serve as important bridges between two sub-network graphs, and deleting them could impact the metric values in the network, ultimately compromising the authenticity of the data. Furthermore, previous studies have investigated bug dependency without removing "*duplicate*" or "*clone*" links,and obtained reliable conclusions, as demonstrated in the work of Vieira et al. [34]. Consistently, we retain these two types of link labels in our study to capture the full picture of the bug dependencies and better reflect the complexity of bugs.

External Validity An external validity threat to this study is the potential challenge of generalizing our findings to all software systems. In order to mitigate this external threat, we conduct our investigation using nine project datasets that are representative of different domains. Additionally, each project has been active for a long time and contains a significant amount of bug data. However, all of the studied projects are open-source and coded in Java. Therefore, the conclusions may not be generalized to other kinds of software, especially closed-source systems or systems developed using other programming languages. To mitigate this threat, we will try to replicate our study using a wide variety of systems in future work.

VII. Related works

A. Bug fix-time

Many previous studies focused on predicting bug-fixing time using a variety of metrics and methods. Hooimeijer et al. used the reputation of the bug opener and other metrics to build a linear model to predict the bug-fixing time [20]. Guo et al. found that bugs reported by those openers with better reputations are more likely to be fixed and consume less time [17]. Bhattacharya et al. investigated the correlation between bug-fixing time and bug opener's reputation. The results indicated that the bug opener's reputation did not significantly correlate with bug-fixing time and additional metrics were needed to improve the accuracy of prediction [8]. Subsequently, Marks et al. proposed a predictive model for the bug-fixing time by digging deeper into bug reports and mining 49 different metrics to build the prediction model [26]. These metrics have been used in many subsequent studies. Zhang et al. collected information on the submitter, owner, severity, priority, category, etc., and used Markov models to predict the bug fix time of real software [39]. Following this, Habayeb et al. noted the temporal nature of certain developer activities' frequency of occurrence and used an invisible Markov model and developer activity time series approach to predict the time of bug-fixing [18]. Assar and Akbarinasaji et al. discuss the precision of resolve time prediction and generalisability of previous studies by duplication study [30, 5, 2]. Sepahvand et al. used semantic relations of fixing activities to find their long-term dependencies to predict the time using deep learning [32]. Mamedov et al. proposed an approach to predict time-to-resolve for defect reports [25]. In 2022, Renan et al. used the link information of bugs and investigated that bug-fixing time were significantly higher for bugs with links to other bugs [34].

B. Bug dependency

Before our study, dependency have been applied to software research in other areas of study. Bavota et al. investigated the evolution of cross-dependency between different projects [6]. Decan et al. studied the evolution of package dependency networks in seven ecosystems [15]. Previous studies have defined bug dependencies at different grain levels. Pogdurski et al. analyzed dependencies from code fragments in programs [28]. In 2008, Zimmermann et al. used the Ucinet6 tool to build dependency at the code fragment grain level and extracted network metrics to predict defects [40]. Similarly, Yang et al. built bug dependency Clusters at the function-level grain and to predict bug defects [37]. There are many other studies that also use bug dependency clusters to conduct empirical study [7, 9, 10]. Cui et al. investigate bug dependency structure to understand software defects and their fixes [13, 14]. Wei et al. proposed a bug localization framework based on dependencies between classes level [38]. Jahanshahi et al. and Almhana et al. created dependency data at the bug reporting grain level and proposed tools to help bug triage based on these data [21, 3, 22].

VIII. Conclusion

In this paper, we conduct a comprehensive study on the impact of bug dependencies on bug-fixing time prediction. First, we perform Pearson correlation analysis to study the correlation between bug dependency metrics and bug-fixing time. The results indicate that most bug dependency metrics are significantly positively correlated with bug-fixing time. Next, we utilize principal component analysis to investigate whether bug dependency metrics capture additional information than traditional metrics. Our findings suggest that bug dependency metrics are complementary to traditional metrics. Finally, we compare the performance of prediction models built with traditional metrics and bug dependency metrics against those models built with solely traditional metrics for predicting bug-fixing time. The results demonstrate that considering bug dependencies significantly improves the effectiveness of bug-fixing time prediction. The findings help us to better understand the potential impact of bug dependencies on software development processes, and provide valuable insights for developing more accurate bug-fixing time prediction models.

Acknowledgement

This work was supported by the National Nature Science Foundation of China (Grant No. 62132014, 62072194 and 62172205), and Zhejiang Provincial Key Research and Development Program of China (No.2022C01045) and Zhejiang Provincial Natural Science Foundation of China(No.LQ23F020020).

References

[1] W Abdelmoez, Mohamed Kholief, and Fayrouz M Elsalmy. "Bug fix-time prediction model using naıve bayes classifier". In: *2012 22nd International Conference on Computer Theory and Applications (ICCTA)*. IEEE. 2012, pp. 167–172.

[2] Shirin Akbarinasaji, Bora Caglayan, and Ayse Bener. "Predicting bug-fixing time: A replication study using an open source software project". In: *journal of Systems and Software* 136 (2018), pp. 173–186.

[3] Rafi Almhana and Marouane Kessentini. "Considering dependencies between bug reports to improve bugs triage". In: *Automated Software Engineering* 28 (2021), pp. 1–26.

[4] Pranjal Ambardekar, Anagha Jamthe, and Mandar Chincholkar. "Predicting defect resolution time using cosine similarity". In: *2017 International Conference on Data and Software Engineering (ICoDSE)*. IEEE. 2017, pp. 1–6.

[5] Saıd Assar, Markus Borg, and Dietmar Pfahl. "Using text clustering to predict defect resolution time: a conceptual replication and an evaluation of prediction accuracy". In: *Empirical Software Engineering* 21 (2016), pp. 1437–1475.

[6] Gabriele Bavota et al. "How the apache community upgrades dependencies: an evolutionary study". In: *Empirical Software Engineering* 20 (2015), pp. 1275–1317.

[7] Árpád Beszédes et al. "Empirical investigation of SEA-based dependence cluster properties". In: *Science of Computer Programming* 105 (2015), pp. 3–25.

[8] Pamela Bhattacharya and Iulian Neamtiu. "Bug-fix time prediction models: can we do better?" In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. 2011, pp. 207–210.

[9] David Binkley and Mark Harman. "Locating dependence clusters and dependence pollution". In: *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE. 2005, pp. 177–186.

[10] David Binkley et al. "Uncovering dependence clusters and linchpin functions". In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2015, pp. 141–150.

[11] Fiorenza Brady. *Cambridge University report on cost of software faults, Press release, 2013*. 2013.

[12] Morakot Choetkiertikul et al. "Predicting the delay of issues with due dates in software projects". In: *Empirical Software Engineering* 22 (2017), pp. 1223–1263.

[13] Di Cui et al. "Investigating the impact of multiple dependency structures on software defects". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 584–595.

[14] Di Cui et al. "Towards characterizing bug fixes through dependency-level changes in Apache Java open source projects". In: *Science China Information Sciences* 65.7 (2022), p. 172101.

[15] Alexandre Decan, Tom Mens, and Philippe Grosjean. "An empirical comparison of dependency network evolution in seven software packaging ecosystems". In: *Empirical Software Engineering* 24 (2019), pp. 381–416.

[16] Emanuel Giger, Martin Pinzger, and Harald Gall. "Predicting the fix time of bugs". In: *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*. 2010, pp. 52–56.

[17] Philip J Guo et al. "Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows". In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 2010, pp. 495–504.

[18] Mayy Habayeb et al. "On the use of hidden markov model to predict the time to fix bugs". In: *IEEE Transactions on Software Engineering* 44.12 (2017), pp. 1224–1244.

[19] Aric Hagberg, Pieter Swart, and Daniel S Chult. *Exploring network structure, dynamics, and function using NetworkX*. Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[20] Pieter Hooimeijer and Westley Weimer. "Modeling bug report quality". In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. 2007, pp. 34–43.

[21] Hadi Jahanshahi et al. "DABT: A dependency-aware bug triaging method". In: *Evaluation and Assessment in Software Engineering*. 2021, pp. 221–230.

[22] Hadi Jahanshahi et al. "Wayback Machine: A tool to capture the evolutionary behavior of the bug reports and their triage process in open-source software systems". In: *Journal of Systems and Software* 189 (2022), p. 111308.

[23] Clara Marie Lüders, Abir Bouraffa, and Walid Maalej. "Beyond duplicates: Towards understanding and predicting link types in issue tracking systems". In: *Proceedings of the 19th International Conference on Mining Software Repositories*. 2022, pp. 48–60.

[24] Clara Marie Lüders, Tim Pietz, and Walid Maalej. "Automated Detection of Typed Links in Issue Trackers". In: *2022 IEEE 30th International Requirements Engineering Conference (RE)*. IEEE. 2022, pp. 26–38.

[25] Murad Mamedov et al. "Building a Reusable Defect Resolution Time Prediction Model Based on a Massive Open-Source Dataset: An Industrial Report". In: *2021 IEEE International Conference on Artificial Intelligence Testing (AITest)*. IEEE. 2021, pp. 73–80.

[26] Lionel Marks, Ying Zou, and Ahmed E Hassan. "Studying the fix-time for bugs in large open source projects". In: *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*. 2011, pp. 1–8.

[27] Alexander Nicholson and Guo Jin LC. "Issue link label recovery and prediction for open source software". In: *2021 IEEE 29th International Requirements En-*

gineering Conference Workshops (REW). IEEE. 2021, pp. 126–135.

[28] Andy Podgurski and Lori A. Clarke. "A formal model of program dependences and its implications for software testing, debugging, and maintenance". In: *IEEE Transactions on software Engineering* 16.9 (1990), pp. 965–979.

[29] Ilham Mulya Rafid. "Performance evaluation for Kruskal's and Prim's Algorithm in Minimum Spanning Tree using Networkx Package and Matplotlib to visualizing the MST Result". In: *no. May* (2019).

[30] Uzma Raja. "All complaints are not created equal: text analysis of open source software defect reports". In: *Empirical Software Engineering* 18 (2013), pp. 117–138.

[31] Jeconiah Richard, Rowin Faadhilah, and Nunung Nurul Qomariyah. "Jaebot: Discord Bot for Network Analysis with NetworkX". In: *2022 International Conference on ICT for Smart Society (ICISS)*. IEEE. 2022, pp. 1–6.

[32] Reza Sepahvand, Reza Akbari, and Sattar Hashemi. "Predicting the bug fixing time using word embedding and deep long short term memories". In: *IET Software* 14.3 (2020), pp. 203–212.

[33] Meera Sharma, M Kumari, and VB Singh. "The way ahead for bug-fix time prediction". In: *Proceedings of the 3rd International Workshop on Quantitative Approaches to Software Quality*. 2015, p. 33.

[34] Renan Vieira et al. "Bayesian Analysis of Bug-Fixing Time using Report Data". In: *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 2022, pp. 57–68.

[35] Renan Vieira et al. "From Reports to Bug-Fix Commits: A 10 Years Dataset of Bug-Fixing Activity from 55 Apache's Open Source Projects". In: *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*. 2019, pp. 80–89.

[36] Stanley Wasserman and Katherine Faust. "Social network analysis: Methods and applications". In: (1994).

[37] Yibiao Yang et al. "An empirical study on dependence clusters for effort-aware fault-proneness prediction". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 2016, pp. 296–307.

[38] Wei Yuan et al. "Dependloc: A dependency-based framework for bug localization". In: *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE. 2020, pp. 61–70.

[39] Hongyu Zhang, Liang Gong, and Steve Versteeg. "Predicting bug-fixing time: an empirical study of commercial software projects". In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 1042–1051.

[40] Thomas Zimmermann and Nachiappan Nagappan. "Predicting defects using network analysis on dependency graphs". In: *Proceedings of the 30th international conference on Software engineering*. 2008, pp. 531–540.