Implicit Social Network Model for Predicting and Tracking the Location of Faults

Ing-Xiang Chen*, Cheng-Zen Yang*, Ting-Kun Lu*, and Hojun Jaygarl†

*Yuan Ze University, Chungli, Taiwan, 320

† Iowa State University, Ames, IA 50011

*{sean,czyang,kylelu}@syslab.cse.yzu.edu.tw †{jaygarl}@cs.iastate.edu

***Abstract*— In software testing and maintenance activities, the observed faults and bugs are reported in bug report managing systems (BRMS) for further analysis and repair. According to the information provided by bug reports, developers need to find out the location of these faults and fix them. However, bug locating usually involves intensively browsing back and forth through bug reports and software code and thus incurs unpredictable cost of labor and time. Hence, establishing a robust model to efficiently and effectively locate and track faults is crucial to facilitate software testing and maintenance. In our observation, some related bug locations are tightly associated with the implicit links among source files. In this paper, we present an implicit social network model using PageRank to establish a social network graph with the extracted links. When a new bug report arrives, the prediction model provides users with likely bug locations according to the implicit social network graph constructed from the co-cited source files. The proposed approach has been implemented in real-world software archives and can effectively predict correct bug locations.**

I. Introduction

In today's software testing and maintenance processes, when errors are found, they are generally reported in bug report managing systems (BRMS) for further bug tracking and debugging [22], [30]. In a conventional process of debugging, software engineers need to identify the bug locations from software code, and then fix them. Since bug locations are discovered according to the information of bug reports, intensive search, which usually involves browsing back and forth through bug reports and software code, is required to locate the bug. Accordingly, technology that can assist software engineers to effectively identify the location of bugs is highly regarded.

Actually, many faults do not come alone, but cooccur with other related faults. This has been justified in the past study [12]. Based on our investigations, there further exist dependency relationships among the co-occurred locations that are cited by the same bug reports. Hence, historical bug information that indicates similar bug locations can be further used to predict future bugs.

Such analysis is not the first exploration of predicting faults based on the assumption of bug dependencies. A recent study [25] has tried to employ the dependency graph of the subsystems to predict the number of faults based on the assumption that the complexity of a subsystem's dependency graph correlates with failures. The authors further applied the concept adapted from the classical graph theory to construct the dependency graph and used network measures to predict faults [27].

Beside the dependencies between subsystems and elements, however, more implicit relationships embedded between bug reports and source files can be investigated. For example, the implicit relationships between bug locations that are co-cited by the same bug reports can be further modeled as a co-citation social network to help predict faults. Based on this assumption, a co-citation graph of bug locations can be constructed according to their co-cited relationships. Figure 1 illustrates a snippet of the co-citation graph constructed from Subversion (SVN) [30].

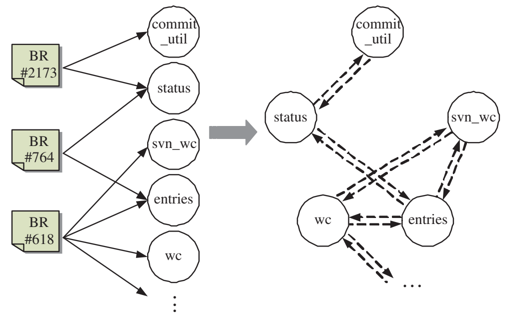
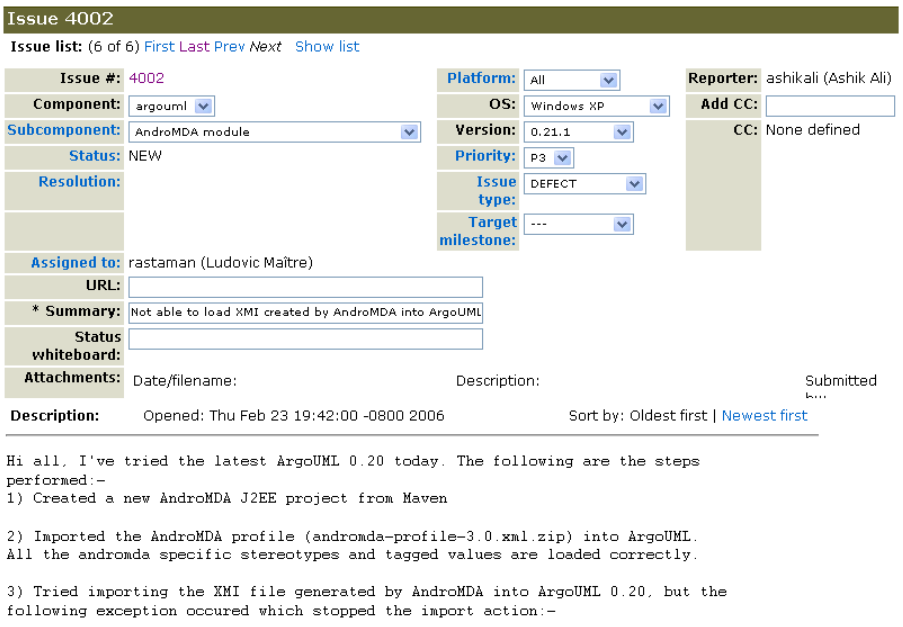


Fig. 1. A snippet of the SVN co-citation graph.

In the co-citation network graph, the potential bug locations appear as vertices, and their implicit co-citation relationships are denoted by dotted bidirectional edges connecting each co-cited pair. Since some related faults usually affect each other, the bidirectional edges in the co-citation graph can further imply where the correlated side effects will happen. Through the accumulated links connecting different co-cited bug locations, the fault-prone locations and their correlations can be modeled as a social network graph.

In this paper, we first investigate the nature of the co-citation graph between bug locations. Since the implicit links between the co-cited locations imply that these locations are relevant, we consider building a co-citation graph connected with bidirectional links to denote their


Issue 4002

Issue list: (6 of 6) First Last Prev Next Show list

Issue #:	4002	Platform:	All	Reporter:	ashikali (Ashik Ali)
Component:	argouml	OS:	Windows XP	Add CC:	
Subcomponent:	AndroMDA module	Version:	0.21.1	CC:	None defined
Status:	NEW	Priority:	P3		
Resolution:		Issue type:	DEFECT		
		Target milestone:	---		
Assigned to:	rastaman (Ludovic Maître)				
URL:					
* Summary:	Not able to load XMI created by AndroMDA into ArgoUML				
Status whiteboard:					
Attachments:	Date/filename:	Description:	Submitted b...		

Description: Opened: Thu Feb 23 19:42:00 -0800 2006 Sort by: Oldest first | Newest first

```
Hi all, I've tried the latest ArgoUML 0.20 today. The following are the steps
performed:-
1) Created a new AndroMDA J2EE project from Maven

2) Imported the AndroMDA profile (andromda-profile-3.0.xml.zip) into ArgoUML.
All the andromda specific stereotypes and tagged values are loaded correctly.

3) Tried importing the XMI file generated by AndroMDA into ArgoUML 0.20, but the
following exception occured which stopped the import action:-
```

Fig. 2. A sample bug report of ArgoUML in Tigris [30].

semantic relevance. Accordingly, we model the co-cited bug locations as a social network graph, and employ information retrieval (IR) techniques to predict the potential location of faults. This paper makes the following contributions:

- It demonstrates the concept of modeling the co-cited bug locations into a social network graph.
- It defines a PageRank-based mechanism to rank the fault-prone locations. The co-citation links between bug locations are used to calculate the score of each fault-prone location.
- It designs, implements, and evaluates the implicit social network model for predicting bug locations.
- It can support good scalability to other larger projects without many changes.

The rest of this paper is organized as follows. Section 2 introduces the background behind the problem and related knowledge about bug report processing. Section 3 reviews the related work in bug prediction literature. In Section 4, we present the implicit social network model for predicting and tracking bug locations. Section 5 depicts the case study and the evaluation results. Finally, Section 6 concludes the paper and outlines the future work.

II. Background

To understand the proposed approach, background knowledge involves problem statement and bug report processing. The related background knowledge is described as follows.

A. *Problem Statement*

As in common debugging procedures, the bug locations corresponding to a bug report are usually predicted from version archives and bug report systems. To discover the potential bug locations L, the prediction of L can be denoted as $P(B \mid L)$, where an incoming bug report B is examined and predicted according to its meta-information. The predicted relationships between bug reports and locations can be viewed as 1-to-many mappings since a bug may involve several code locations. Hence, bug prediction is defined as providing users with a recommendation list of possible bug locations and establishing the traceability links between the bug reports and their locations. This study predicts the location of faults on the level of files.

As contrasted with the explicit links between bug reports and fixed locations, the implicit links between co-cited locations may potentially indicate the side effects of incoming bugs. The potential location of faults is thus predicted by establishing the implicit co-citation graph and comparing the similar fixed locations with their ranking in the graph. More formally, this study follows two fundamental hypotheses to predict the location of faults.

- *Hypothesis* 1: The location of relevant faults can be tracked by the co-cited dependency of bug locations.
- *Hypothesis* 2: New bug locations can be predicted by comparing similar bug reports and their fixed locations.

B. *Bug Report Processing*

Generally, bug reports are composed of different meta-information and textual contents to describe the faults in bug report managing systems (BRMS). To realize automated bug retrieval, the knowledge about bug report and its handling process is required.

1) The Compositions of Bug Reports: A bug report generally contains some required meta-information and short descriptions or error messages to denote the fault. After a bug report is submitted to the BRMS, there may be some comments or responses from different users appended to the original bug report. Therefore, a bug report contains more detailed textual descriptions or comments about the same bug. Figure 2 illustrates a sample bug report of ArgoUML stored in Tigris [30].

As the example in Figure 2, different types of meta-information are recorded to describe a bug such as Issue#, Summary, Reporter, etc. More descriptive information such as the descriptions and comments can also be appended in a bug report. Besides, other types of files such as software code and screenshots can be attached in the bug report to facilitate the explanation of error messages.

2) The Process of Handling Bug Reports: In the process of handling a bug report, bugs are supervised in different phases before closed [19]. Figure 3 depicts the processing flow of bug reports. When a fault or a bug is discovered, a bug report is wirtten and submitted to the BRMS. After a bug is reported in the BRMS, the report will be analyzed by an analyst, and the bug is assigned to an appropriate developer to fix it. Meanwhile, if the bug has been fixed or has been reported before, it will be moved to the corresponding fixed phase. After the fixed bug is tested and verified, the bug report will be closed. Therefore, a bug report will be given different status of resolution in its life cycle [1].

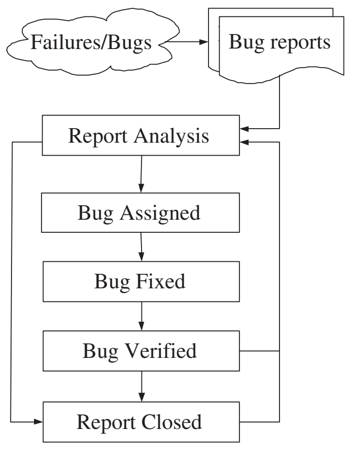


Fig. 3. The processing flow of bug reports.

III. Related Work

Over the past years, several approaches have been proposed to support automated identification of bug locations. These approaches can be briefly categorized as the following types: predicting faults from software change and cached history [8], [9], [12], [15], predicting bug locations with learning-based approaches [3], [17], [16], [18], [28], and defect detection using dependency graph and network measures [25], [27].

A. Predicting Faults from the History of Software Change and Fixed Cache

Graves et al. first attempted to predict modules that were more fault-prone by their changes and ages [8]. This model successfully improves the prediction accuracy by giving higher weights to recent changes over older ones. Therefore, their approach can predict the distribution over modules of the incidences of faults using change management data from a very large, long-lived software system.

Nagappan and Ball observed that code churn can measure the changes made to a component over a period of time and quantify the extent of software change [15]. They further proposed a technique to early predict system defect density using a set of relative code churn measures instead of absolute code churn measures. The authors conducted experiments over Windows Server 2003 binaries, and the experiments show that the proposed model achieves the prediction accuracy of 89%. However, more fine-grained predictions still need further investigations.

Hassan and Holt first proposed an approach called the top-ten list to predict fault-prone modules [9]. They were inspired by the idea of using a limited resource cache in file system and used a dynamically maintained cache to predict the most fault-prone modules. In their caching scheme, four strategies were separately considered, namely, modules that were most frequently modified, most recently modified, most frequently fixed, and most recently fixed. According to their experimental results over large open-source projects, the best Hit Rate (HR) can be achieved between 20% and 54% to the level of modules.

More recently, Kim et al. used the cached history to predict the location of faults [12]. They followed the assumption that faults usually cooccur with several related faults. In their experiment, they used a cache of 10% of the source files to predict bug locations from file level to entity level and respectively achieved the best accuracy of 95% and 72%. The results show that their FixCache approach achieves significant prediction improvements in a more fine-grained level.

B. Predicting Bugs with Learning-based Techniques

Ostrand et al. predicted the largest number of faults to the level of files with a negative binomial linear regression model [3], [17]. Their prediction model selects 20% files in each release of two systems for evaluation. The performance with the proposed regression model can predict the hightest number of faults between 71% and 92%, with the overall average being 83%.

Wong et al. used a back-propagation neural network (BPNN) model to train the inputs and the corresponding outputs of a program to further predict the suspiciousness of each executable statement. By learning the input-output relationship of a program, the proposed BPNN model can effectively predict the potential statements that contain faults by ranking their suspiciousness. Currently, the proposed BPNN model was only applied to programs with a single bug. Although a method of clustering failed executions has been mentioned to predict multiple bugs, the effectiveness of extending the proposed BPNN still needs to be verified.

Neuhaus et al. analyzed the import structure of software components and used a support vector machine (SVM) model to learn and predict vulnerable components in large software systems [16]. They further predicted which imports are most important for a component to be vulnerable. Their SVM-based predictor can correctly discover about half of all vulnerable components, and about two thirds of all predictions are correct.

Premraj et al. proposed a novel approach of using SVM with a usual suspect list to improve the prediction of potential bug locations [18]. They borrowed an idea from the criminology that the locations frequently reported as bug locations are regarded as the suspects in the usual suspect list. Premraj et al. further extracted features from bug reports and used a binary SVM model to train the historical bug reports with the usual suspect list. Their prediction results show that SVM with usual suspect list can achieve the best accuracy of 90% in smaller projects and 60% in larger projects.

C. Using Dependency Graph and Network Measures

Zimmermann and Nagappan investigated the dependencies of Windows Server 2003 to predict the number of failures [25]. They argued that an increase in complexity is accompanied by an increase in failures. Their analysis further shows that binaries in part of cycles had on average twice as many failures as the other binaries. Therefore, they computed the complexity of a subsystem's dependency graph with classical graph theory and predicted the number of failures at statistically significant levels.

In advance, Zimmermann and Nagappan applied network measures to the constructed dependency graph and predicted the number of failures [27]. They found the recall of using network measures to predict bugs is by 10% higher than using models built from the previous complexity metrics. Furthermore, network measures could identify twice the binaries that the Windows developers considered as critical.

In summary, previous approaches have explored different aspects of ways to predict the potential location of faults. The history of software change is important evidence to discover the possible location of faults. Regression models and state-of-the-art learning techniques such as BPNN and SVM have been studied to learn past debugging knowledge and are practical to predict bugs with different strategies. The idea of using dependency graph with network measures has also been proved to be effective for bug prediction. However, the implicit co-citation relationships between bug locations have not been explored. In this study, we thus propose a novel approach using implicit social network model to predict and track the possible location of faults.

IV. Implicit Social Network Model

The proposed social network model for predicting and tracking bug location is based on implicit link analysis utilizing the co-citation links embedded between bug reports and source files. This approach is based on an important assumption that if there are two links pointing to the source location u and v from a bug report r, the bug report r indicates that the location u and v are both associated with the reported faults, and there exists a co-cited relationship between u and v. Implicit bidirectional links between u and v are added to denote the mutual co-citation. Since u and v can be further referred by other bug reports, a directed co-citation network is established through the implicit links between each co-cited location pair.

Therefore, the location of faults can be modeled as a directed co-citation graph $G = (V,E)$, where

- $V = \{ s_i \mid 1 \leq i \leq n \}$ is the set of vertices representing the potential source location of faults,
- $l_{ij} \in E$ denotes the implicit links within the source location s_i and s_j,
- and each implicit link $(i,j) \in E$ is associated with $P(s_i \mid s_j)$ denoting the conditional probability of s_j to be co-cited by the given location s_i.

In the implicit social network model, a PageRank-based algorithm [4] is presented to compute the ranking of the predicted location of faults. PageRank (PR) is adopted rather than other known social network models such as HITS [13], because PR can relieve the problems of nepotism and *topic drift* by weighting each link to the quality of the location containing the link [10], [5]. Furthermore, a damping factor is considered in PageRank to further predict the location that has never been fixed before. In the following subsections, we illustrate the proposed implicit social network model with a simple example.

A. Constructing a Co-citation Graph with Implicit Links

Based on the assumption mentioned above, the potential location of faults can be modeled as a directed graph $G = (V,E)$ according to the citation links from bug reports to their corresponding location of faults. To build up the co-citation graph, explicit co-citation links between bug reports and source locations are transformed into a directed graph with implicit links between each co-cited pair of locations. Figure 4 shows a simple example of constructing a co-citation graph with implicit links between the location pairs.

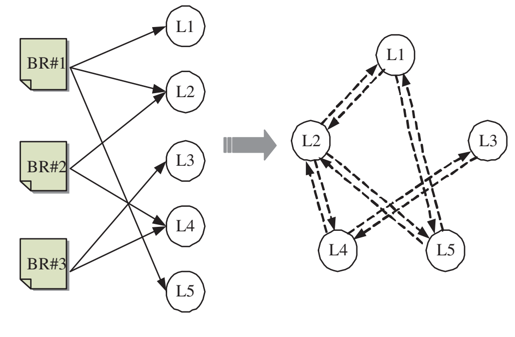


Fig. 4. An example of constructing a small co-citation graph with implicit links.

In Figure 4, for explicit links from $BR\#1$ to location $L1$, $L2$, and $L5$, the proposed approach generates location pairs ($L1$, $L2$), ($L1$, $L5$), and ($L2$, $L5$). Therefore, the possible pairs are connected by the directed implicit links l_{L1L2}, l_{L2L1}, $\cdots$, l_{L2L5}, l_{L5L2}. Likewise, explicit links from $BR\#2$ and $BR\#3$ generate the co-cited location pairs ($L2$, $L4$) and ($L3$, $L4$) that are respectively connected by their corresponding links l_{L2L4}, l_{L4L2}, and l_{L3L4}, l_{L4L3}. In Figure 4, each directed dot line denotes an implicit link from one location to another. Finally, a co-citation graph is constructed with the implicit links between each co-cited pair of fault location.

B. Applying PageRank to Implicit Links

After obtaining the co-citation graph with implicit link structure, we apply a PageRank-based algorithm that has been successfully implemented in Google search engine [29] to rank the potential location of faults. An $n \times n$ adjacency matrix is denoted by M with the rows and columns corresponding to the directed graph G of the potential location of faults. If there is a link from location j to location i, then the matrix entry $M[i, j]$ has a weight $1/N_j$, where N_j denotes the number of links that location j points to. The weights of all other entries are filled with zeros.

The adjacency matrix is used to compute the rank score of each location. The rank score $PR(i)$ of location i is recursively evaluated by a function on the rank scores of the locations that point to location i:

$$PR(i) = \sum_{j \to i} [PR(j)/N(j)] \tag{1}$$

In Equation (1), the recursive PR equation gives each location a fraction of the rank of each location pointing to it. Inversely, that location is weighted by recursively giving the strength of the links from other locations in the matrix.

Figure 5 gives an example of a small matrix M and the recursive computation for its PageRank scores. According to the co-citation graph obtained in Figure 4, each entry M_{ij} is given a weight of either zero or $1/N_j$. For instance, $L5$ has two outgoing edges to $L1$ and $L2$ ($N_{L5} = 2$), l_{L1L5} and l_{L2L5} are 1/2, and l_{L3L5}, l_{L4L5}, and l_{L5L5} are 0. Since $L5$ is pointed by $L1$ and $L2$, $PR(L5)$ is determined by $PR(L1)$ and $PR(L2)$. $L1$ and $L2$ respectively have two and three outgoing links, and thus $PR(L5)$ is the sum of $PR(L1)/2$ and $PR(L2)/3$. Let PR_k be the k^{th} intermediate PageRank state, and PR_{k+1} denote the next state. When the recursive computation of $PR_{k+1} = M \times PR_k$ converges to a set of fixed values, the converged PR_{k+1} obtains the PageRank scores of the fault-prone locations.

However, among all potential locations of faults, there may exist some locations that have never been reported as bug locations before or some bug locations that have never been co-cited with other locations. After the transformation into a co-citation graph, such kinds of locations do not have any outgoing link to other locations and thus become dangling locations. In practice, if many locations have no outgoing link, the eigenvector of the above equation is mostly zero.

Besides, if two bug locations are co-cited with only one bug report, then there exist only two directional links pointing to each other. The two co-cited bug locations thus fall into a loop. During the process of matrix computation, the loop accumulates PageRank scores but does not distribute any PageRank scores to other locations due to the lack of outgoing links. The loop forms a trap, which is called a RankSink [11]. Consequently, locations in the loop are likely to obtain higher PageRank scores than they should have.

Therefore, the basic model is modified using the concept of *random walking* with a damping factor to mitigate the problems of dangling locations and RankSinks [4], [11]. Equation (2) shows the modified form of the PageRank equation with a damping factor d.

$$PR(i) = (1 - d) + d \sum_{j \to i} [PR(j)/N(j)] \tag{2}$$

In Equation (2), the parameter d is the damping factor, which is set between zero and one. The parameter d is to predict the probability of randomly choosing one of the links on the current location and jumping to the location it links to. With the probability of $1 - d$, the PageRank readjusts the equation to model the case of randomly jumping to a location picked uniformly from the potential location of faults.

$$\begin{array}{c|ccccc|} & \multicolumn{5}{c}{\boldsymbol{M}} \\ & L1 & L2 & L3 & L4 & L5 \\ \hline L1 & 0 & 1/3 & 0 & 0 & 1/2 \\ L2 & 1/2 & 0 & 1/2 & 0 & 1/2 \\ L3 & 0 & 1/3 & 0 & 1 & 0 \\ L4 & 0 & 0 & 1/2 & 0 & 0 \\ L5 & 1/2 & 1/3 & 0 & 0 & 0 \\ \hline \end{array} \quad \overset{\textbf{Rank}}{\begin{pmatrix} 0.2 \\ 0.3 \\ 0.2 \\ 0.1 \\ 0.2 \end{pmatrix}} = \overset{\textbf{Rank}}{\begin{pmatrix} 0.2 \\ 0.3 \\ 0.2 \\ 0.1 \\ 0.2 \end{pmatrix}}$$

Fig. 5. The PageRank scores from the above co-citation graph

C. Predicting and Tracking Bugs with PageRank

After constructing the co-citation graph of bug locations using PageRank, information retrieval (IR) and natural language processing (NLP) techniques are employed to handle the processing of bug reports and the prediction of bug locations. When a new bug report comes, the proposed implicit social network model using the PageRank algorithm is applied to predict the possible location of faults by retrieving the fault-prone locations cited by similar bug reports. The detail of similarity calculation between bug reports will be defined in Section V.C. Figure 6 illustrates the process of predicting bug locations with the proposed implicit social network model.

Figure 6 shows that bug reports and their corresponding fixed locations are respectively collected from bug report managing systems (BRMS) and CVS/SVN to construct the co-citation graph of bug locations with PageRank. Based on the assumption that the explicit links between

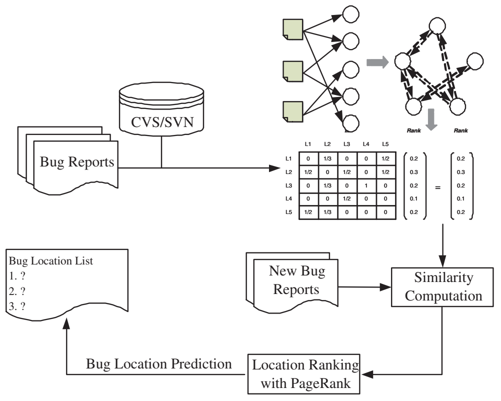


Fig. 6. Overview of the bug prediction and tracking process.

TABLE I
THE STATISTICS OF THE EXPERIMENTAL DATA.

Project Name	**Subversion (SVN)**	**ArgoUML**
– Language	C	Java
– Software Type	SCM Tool	UML Tool
– SCM	Subversion	Subversion
– BRMS	Tigris	Tigris
– Data Period	07/01-03/05	02/00-04/07
– # of Fixed BR	211	1,024
– # of Files	148	1,405
– # of Bugs	516	4,494

historical bug reports and the corresponding fixed locations can reveal the knowledge of debugging, these links are further used to predict and track the potential bug locations of a new coming bug report. Since the contents of bug reports are composed of structured meta-information and textual descriptions, the debugging knowledge embedded in the historical bug reports can be further extracted with IR and NLP techniques. After transforming bug reports into knowledge representation with a vector space model (VSM), the likely bug locations denoted by a new coming bug report can be predicted by comparing it to historical bug reports with cosine similarity measure [2]. The predicted locations are tracked through the explicit links from similar bug reports pointing to the corresponding location of faults, which are ranked by the PageRank scores. Likewise, when the new bug report and its corresponding locations are fixed, the co-citation relationships will be added to the implicit social network graph for future predictions.

V. Data Collection and Processing

To validate the performance of the proposed implicit social network model, experiments were conducted on two open-source projects collected from bug report managing systems (BRMS). The statistics of data sets and their processing are described in the following subsections.

A. Data Sets

Table I summarizes the statistics of the collected datasets. Bug reports of two open-source projects, Subversion (SVN) and ArgoUML, were collected from Tigirs for evaluation [30]. SVN and ArgoUML are respectively developed in C and Java. These two projects are selected in the experiments because their version archives are well integrated with their bug databases.

B. Identifying the Fixed Locations

To identify the fixed components from failed ones in the source files, the fixed information will be retrieved from version archives and BRMS. In this study, the links between bug reports and their corresponding fixed bugs are discovered in the following two ways that have been used in previous studies [6], [7], [24], [26].

1) Using keywords in change logs: The fixes in version archives are identified by the messages that describe changes. The proposed approach first refers to the previous method to explore references to bug reports such as ”Fixed issue # 1234”. However, the trust level of only employing this approach is low [26].

2) Using keywords in bug reports: More key words in bug reports such as “fixed”, “defect”, “bug” or matches patterns like “# and a number” are used to extract the links. Although each bug number potentially has a corresponding reference to a bug report, such references are not sufficient to extract the fixed information. In advance, text similarity between bug reports and change logs are applied to discover more embedded links.

C. Extraction of Bug Information with IR/NLP

Since the nature of bug reports is composed of structured natural language, informational retrieval (IR) and natural language processing (NLP) techniques are applied to support retrieving and tracking the location of faults. We thus employed IR and NLP techniques to extract bug information embedded in both bug reports and source archives. To employ IR and NLP techniques to extract information about debugging, bug reports need to be preprocessed as *bag-of-words* by standard IR steps, namely, tokenization, stemming, stopword removal, vector space representation, and similarity calculation [2], [14].

First, each bug report is tokenized by space and thus the sentences and passages in each report can be transformed into term space. Stemming is to reduce the inflected words into their stem, base or root form, e.g. plural nouns into singular nouns, and the past, progressive, and perfect tense into the simple tense. Stopword removal aims to remove the common words that do not carry specific information such as “the”, “a”, “this”, and “that”. The preprocessed terms are represented in a vector space by assigning different weights, and then IR techniques such as cosine measure can be further employed to retrieve similar bug reports and track the location of faults.

VI. Experimental Setup

According to the problem definition, the possibility of location (L) is predicted to file level by comparing and ranking the similarity of a new coming bug report (B) to the fixed bug reports and their locations. To evaluate the fixed locations, bug reports resolved as “FIXED” were used for experiments. The contents of bug reports are further transformed into vector space representation with $TF \cdot IDF$ weighting [20], [21] after the standard IR processing steps to properly present the data semantics.

In the experiment, bug reports were chronologically divided into 10 folds, in which the first 9-fold data were used as historical bug reports to build up the co-citation graph, and the last 1-fold data were treated as new coming bug reports for test. The experimental setup ensures that the historical data are used to predict future bug locations reported by new coming bug reports. Besides, such setup closely simulates the cases in the real world.

The proposed implicit social network model using PageRank is assessed in comparison with an SVM model to validate its performance [18]. To simulate the likelihood of jumping to an arbitrary location, the random probability in PageRank is generally set around 0.15 with a damping factor of 0.85 [4]. Both models are implemented in the same experimental environment to provide users with a recommendation list of 1-100 potential locations for each coming bug report. This study makes an assessment of prediction accuracy that has been used in other studies [12], [18] and regards the correct predictions as hit. Therefore, accuracy is to measure the percentage of correct hits by the following equation:

$$Prediction\ accuracy = \frac{\#\ of\ hit}{\#\ of\ hit\ +\ \#\ of\ miss}$$

VII. Results and Discussion

In this section, the prediction accuracy for both models was assessed. Experimental results and the threats to validity are discussed as follows.

A. Evaluation

Figure 7 depicts the prediction accuracy of the social network model and the SVM model for SVN. The results show that the social network model using PageRank can correctly predict the potential location of faults up to 100% in a recommendation list of 100 files. However, the SVM model only achieves the prediction accuracy of 81.8% in the same number of predictions.

Similar to the behavior of browsing search results on the Web, over 80% users usually only look at the first page of 10 retrieved items [23]. Therefore, a recommendation list of 10 files was further investigated for its prediction power. In top 10 predictions, the proposed model with PageRank can achieve the accuracy of 40.9%, but the SVM model only achieves the prediction accuracy of about 31.8%.

Figure 8 shows the accuracy results of both prediction models for a larger open-source project, ArgoUML. The SVM-based approach consistently achieves slightly better prediction accuracy of about 5% to 6% than the social network approach in a recommendation list of 10 to 100 files. In the second experiment, we have explored the reason that degrades the performance of the social network approach. One reason is that the ArgoUML project does not provide abundant co-citation information

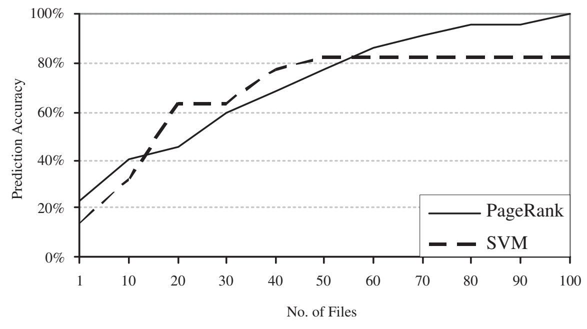


Fig. 7. The prediction accuracy for SVN.

because currently less than 60% faults could be collected to build its social network graph. Many implicit links embedded in the rest of the uncollected bug reports have not been explored to rank the significant fault-proneness. Another reason is that the weight $d = 0.85$ used in the current experiment cannot reflect the co-citation behavior in ArgoUML and still need to be fine-tuned. These observations will help to suggest whether the learning-based approach or the social network approach be used for the tradeoff between high accuracy.

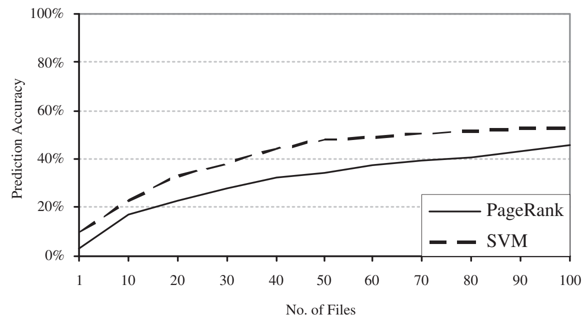


Fig. 8. The prediction accuracy for ArgoUML.

B. Discussion

Although the implicit social network model can correctly predict the potential location of faults in SVN, there still exists much room to further improve the prediction power to larger projects and different types of projects. Currently, the prediction models are only validated in two open-source projects. Hence, there may exist some threats to validity. For example, the systems examined in the experiment might not be representative enough since many closed-source projects may result in different types of bug localities. Besides, the partially collected faults in these two projects may also be threats to affect the prediction performance.

However, we believe that the prediction accuracy can be further improved with bug information of better quality. Especially for a recommendation list of 10 locations, the quality of the collected bug reports significantly affects its accuracy. Since an accurate recommendation list of 10 locations will be of most help for software developers and analysts to rapidly and correctly locate the bugs, it is the highest prioritized goal for further improvements. Moreover, data semantics about bug reports need to be explored and extracted with advanced IR/NLP techniques to retrieve more semantically relevant bug locations.

VIII. Conclusions and Future Work

In this paper, we present an implicit social network model to predict and rank bug locations in the co-cited bug graph. The performance of correctly predicting bug locations has been validated. For a smaller project, the implicit social network model with PageRank is highly promising to correctly hit the potential bug locations. For a larger project, the proposed model has achieved comparable prediction accuracy for incremental bug prediction. This model can primitively facilitate software engineers in the task of locating bugs and support project managers to allocate limited resources for quality assurance.

This paper has revealed that the co-citation relationship is an important clue among various hints to predict the potential location of faults. An implicit social network model using a PageRank-based algorithm and IR/NLP techniques has been investigated to be one promising way to explore the significant bug locations. Nevertheless, much work still needs further investigations.

- To discover similar bug locations of more semantic relevance, advanced IR techniques such as building domain-specific ontologies and semantic similarity measures need to be further studied.
- Advanced filtering and weighting approaches need to be studied to filter out the unimportant co-citation links and further enhance valuable bug links.
- Various types of software projects and larger systems need to be validated in the future to support the prediction model.

Future work involves the investigations on not only naive link analysis, but profound social network relationships and advanced IR/NLP techniques. More larger case studies also need to be studied for validation.

References

[1] J. Anvik, L. Hiew, and G. C. Murphy, "Who Should Fix this Bug?," *Proc. of the 28th Int'l Conf. on Software Eng. (ICSE 2006),* pp. 361–370, 2006.

[2] R. A., Baeza-Yates, and B. A., Ribeiro-Neto, *Modern Information Retrieval,* Addison Wesley, 1999.

[3] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "Predicting the Location and Number of Faults in Large Software Systems," *IEEE Trans. on Software Eng.,* vol. 31, no. 4, pp. 340–355, 2005.

[4] S. Brin and L. Page, "The Anatomy of Large-Scale Hypertextual Web Search Engine," *Computer Networks and ISDN Systems,* vol. 30, no. 1-7, pp. 107–117, 1998.

[5] S. Chakrabarti, M. Joshi, and V. Tawde, "Enhanced Topic Distillation using Text, Markup Tags, and Hyperlinks," *Proc. of the 24th ACM SIGIR Conf. on Research and Development in Information Retrieval (SIGIR 2001),* pp. 208–216, 2001.

[6] D. Cubranic and G. C. Murphy, "Hipikat: Recommending Pertinent Software Development Artifacts," *Proc. of the 25th Int'l Conf. on Software Eng. (ICSE 2003),* pp. 408–418, 2003.

[7] M. Fischer, M. Pinzger, and H. Gall, "Populating a Release History Database from Version Control and Bug Tracking Systems," *Proc. of Int'l Conf. on Software Maintenance (ICSM 2003),* pp. 23, 2003.

[8] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History," *IEEE Trans. on Software Eng.,* vol. 26, no. 7, pp. 653–661, 2000.

[9] A. E. Hassan and R. C. Holt, "The Top Ten List: Dynamic Fault Prediction," *Proc. of Int'l Conf. on Software Maintenance (ICSM 2005),* pp. 263–272, 2005.

[10] M. R. Henzinger, "Hyperlink Analysis for the Web," *IEEE Internet Computing,* vol. 5, no.1 , pp. 45–50, 2001.

[11] S. J. Kim and S. H. Lee, "An Improved Computation of the PageRank Algorithm," *Lecture Notes in Computer Science,* vol. 2291, pp. 73–85, 2002.

[12] S. Kim, T. Zimmermann, E. J. Whitehead, Jr., and A. Zeller, "Predicting Faults from Cached History," *Proc. of the 29th Int'l Conf. on Software Eng. (ICSE 2007),* pp. 489–498, 2007.

[13] J. M. Kleinberg, "Authoritative Sources in a Hyperlinked Environment," *Journal of the ACM,* vol. 46, no. 5, pp. 604–632, 1999.

[14] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. "Have Things Changed Now? an Empirical Study of Bug Characteristics in Modern Open Source Software. *Proc. of the 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID 2006),* pp. 25–33, 2006.

[15] N. Nagappan and T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density," *Proc. of the 27th Int'l Conf. on Software Eng. (ICSE 2005),*

[16] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting Vulnerable Software Components," *Proc. of the 14th ACM Conf. on Comp. and Communications Security (CCS 2007),* pp. 529–540, 2007.

[17] T. J. Ostrand, E. J.Weyuker, and R. M. Bell. "Where the Bugs Are," *Proc. of the 2004 ACM SIGSOFT Int'l Symp. on Software testing and analysis (ISSTA 2004),* pp. 86–96, 2004.

[18] R. Premraj, I.-X., Chen, H. Jaygarl, T. Nguyen, T. Zimmermann, and S. Kim, and A. Zeller, "Where Should I Fix This Bug?," Saarland University, Saarbrücken, Germany, Feb., 2008.

[19] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of Duplicate Defect Reports Using Natural Language Processing," *Proc. of the 29th Int'l Conf. on Software Eng. (ICSE 2007),* pp. 499–510, 2007.

[20] G. Salton, and M.J. McGill, *Introduction to Modern Information Retrieval,* McGraw-Hill, 1983.

[21] G. Salton, and M.J. McGill, *Automatic Text Processing: The Transformation Analysis and Retrieval of Information by Computer,* Addison Wesley, 1989.

[22] M. Serrano, and I. Ciordia, "Bugzilla, ITracker, and Other Bug Trackers," *IEEE Software,* vol. 22, no. 2, pp. 11–13, 2005.

[23] C. Silverstein, M. Henzinger, H. Marais, and M. Moricz, "Analysis of a Very Large AltaVista Query Log," *ACM SIGIR Forum,* vol. 33, no. 1, pp. 6–12, 1999.

[24] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do Changes Induce Fixes? On Fridays," *Proc. of the 1st Int'l Workshop on Mining Software Repositories (MSR 2005),* 2005.

[25] T. Zimmermann and N. Nagappan, "Predicting Subsystem Defects using Dependency Graph Complexities," *Proc. of the 18th IEEE Int'l Symp. on Software Reliability Eng. (ISSRE 2007),* pp. 227–236, 2007.

[26] T. Zimmermann, R. Premraj and A. Zeller, "Predicting Defects for Eclipse," *Proc. of 3rd Int'l Workshop on Predictor Models in Software Engineering (PROMISE 2007),* pp. 9, 2007.

[27] T. Zimmermann and N. Nagappan, "Predicting Defects using Social Network Analysis on Dependency Graphs," *Proc. of the 30th Int'l Conf. on Software Eng. (ICSE 2008),* (Accepted), May 2008.

[28] W. E. Wong, L. Zhao, Y. Qi, K.-Y. Cai, and J. Dong, "Effective Fault Localization using BP Neural Networks," *Proc. of the 19th Int'l Conf. on Software Eng and Knowledge Eng. (SEKE 2007),* pp. 374–379, 2007.

[29] Google Search Engine, http://www.google.com.

[30] Tigris, http://www.tigris.org/.