2013 12th International Conference on Machine Learning and Applications

DECOBA: Utilizing Developers Communities in Bug Assignment

Shadi Banitaan
Department of Mathematics, Computer Science and Software Engineering
University of Detroit Mercy
Detroit, MI 48221, USA
banitash@udmercy.edu

Mamdouh Alenezi
Department of Computer Science
North Dakota State University
Fargo, ND 58108, USA
mamdouh.alenezi@ndsu.edu

***Abstract*—Bug Tracking System (BTS) is publically accessible which enables geographically distributed developers to follow the work of each other and contribute in bug fixing. Developer interactions through commenting on bug reports generate a developer social network that can be used to improve software development and maintenance activities. In large scale complex software projects, software maintenance requires larger groups to participate in its activities. Most previous bug assignments approaches assign only one developer to new bugs. However, bug fixing is a collaborative effort between several developers (i.e., many developers contribute their experience in fixing a bug report). In this work, we build developers social networks based on developers comments on bug reports and detect developers communities. We also assign a relevant community to each newly committed bug report. Moreover, we rank developers in each community based on their experience. An experimental evaluation is conducted on three open source projects namely NetBeans, Freedesktop, and Mandriva. The results show that the detected communities are significantly connected with high density. They also show that the proposed approach achieves feasible accuracy of bug assignment.**

***Keywords*—Developers Social Network, Community Detection, Bug Report Assignment, Developer Ranking**

I. Introduction

One of the important parts of the software development process is bug discovery and fixing. Bug tracking system (BTS) services as the main core for developers communication and coordination about development issues. Bug tracking systems such as Bugzilla offers a unified platform for both developers and non-developers to cooperate with each other. In this platform, they are able to submit bug reports, comment on them, track their statuses, and fix them. Such collaboration between developers and non-developers plays an important role towards producing more robust software systems. BTS is enormously useful in software development, and it is used extensively by open source software projects.

Many bug reports are received daily in most open source software projects. These reports have to be triaged and assigned to developers with relevant experience to handle them. Bug triaging is usually performed manually which is an error-prone time-consuming process. Keeping track of active developers and their expertise is not an easy task. Many bug reports get assigned to irrelevant developers which results in delaying the fixing time of these bugs. Many approaches were proposed to automate the bug triaging process. Most of these approaches predict only one developer to fix a newly coming report. Even though, bug fixing is a collaborative effort (i.e., many developers contribute their experience in fixing a bug report). Therefore, new approaches that recognize these efforts are in need.

Developers are able to contribute their thoughts in the form of comments over bug reports in a BTS. These comments reflect their interest and experience about bug reports. Developers usually discuss about the best ways to fixing a bug [1]. Most large software development teams work in a geographically distributed development environment where their discussions are commonly reflected as comments on bug reports. Developers' comments play a very important role in the life-cycle of a bug report, and most of them are relevant with how to fix the bug. Those developers, who commented on the bug, are often equipped with the relevant expertise in resolving the bug.

Social networks have been used in mining open source software repositories. Begel et. al. [2] presented the codebook framework which builds a directed graph that captures the relationships between people, code, bugs, specification, and other work artifacts. Their approach is customizable and collects data from different sources. Their aim was to build a general framework that can be used to answer inter-team coordination information. Robertsa et. al. [3] found that the core developers in Apache HTTP Server project are self-organized into subgroups that communicate repeatedly in completing the task on hand. They also observed that a few experienced developers are centrally located in the network and driving communications within the project. Hong et. al. [4] analyzed contributors social network formed by comments of Mozilla bug reports. Their results indicated a strong evidence of the community structure within developer social networks.

In this work, we propose the DECOBA (utilizing **DE**velopers **CO**mmunities in **B**ug **A**ssignment) approach to build developers social network and detect developers communities in open source projects. DECOBA also assigns a relevant community to a newly reported bug report in order to be fixed. To summarize, we make the following key contributions in this work:

- We construct developers social networks based on developers contributions in fixing bug reports.

978-0-7695-5144-9/13 $31.00 © 2013 IEEE
DOI 10.1109/ICMLA.2013.107

- We apply a community detection algorithm to detect developers communities.
- We assign a relevant community for each newly coming bug report to handle it. This will give more developers who have an experience in solving bugs.
- We rank developers in each community based on their expertise. The ranking also allows to redistribute the load between developers.
- We perform experimental evaluation using three open source projects. The results show that the detected communities are very dense which show a clear evidence of the community structure in developers social networks. They also show that the approach achieves a reasonable accuracy of bug assignment.

The rest of the paper is organized as follows: Section II describes the proposed approach. Section III briefly introduces the evaluation metrics used in this work. The experimental evaluation and discussion are presented in Section IV. Section V discusses related work. Section VI concludes the paper.

II. The DECOBA Approach

DECOBA is divided into five main steps. The first step is to create the bug term matrix of bug reports where each bug report is represented as a vector and each word in the bug report represents a feature. The second step is to build the active developers adjacency matrix where both rows and columns represent active developers, values represent the collaborative effort between developers (e.g., how much developers commented on each other). The third step is to convert the adjacency matrix into a network of developers where each node represents a developer, an undirected edge is added between two developers if they have collaborated work on bug reports. The edge is weighted by the number of collaborations between them. The forth step is to detect developers communities by applying a community detection algorithm. The fifth step is to build a predictive model in order to assign a community of developers to a newly coming bug report. The main premise is that fixing a bug is a collaborative effort between developers. Even though a bug is assigned to one developer, many developers are contributing towards fixing it. Therefore, we are predicting a group of candidate developers who are experienced in solving similar bug reports. The following sections explain these steps in detail.

A. Constructing The Vector Space Model

We use the summary of bug reports as the textual representation. The first step in DECOBA is to construct a vector space representation of bug reports. We have a collection of bug reports, B = $\{b_1, \cdots, b_{|B|}\}$. Each bug report has a collection of terms, T = $\{t_1, \cdots, t_{|T|}\}$. A weight is assigned to each term in a bug report. The weight represents the number of occurrences of the term in the bug report. This weighting scheme is known as TF. We filter out unnecessary terms which include stop-words, punctuation, white-spaces and numbers. We then apply the chi-square (X^2) feature selection method to reduce the dimensionality of the vector space and to achieve better classification results. For X^2, we select 30% as the ratio of the final number of words in the corpus [5], [6]. We select X^2 since it outperforms other feature selection methods in text classification [5].

B. Building The Adjacency Matrix

After constructing the vector space model, DECOBA extracts the collaborative efforts between active developers and saves the results in Developer-Developer Collaborative Matrix (DDCM). The DDCM matrix is a square $d \times d$ matrix, where d represents the number of active developers. The value of DDCM[i,j] is defined as follows:

$$DDCM[i,j] = Comments(i,j) + Comments(j,i) + CC(i,j)$$

where $Comments(i,j)$ represents the number of comments made by developer i on developer j bug reports, $Comments(j,i)$ represents the number of comments made by developer j on developer i bug reports, and $CC(i,j)$ represents the number of bug reports that both developer i and j commented on and do not belong to them. The main premise is that both developers have experience in solving similar bug reports if they wrote many comments on same bug reports and if they commented on each other bug reports.

C. Creating The Developers Network

After building the developers adjacency matrix, DECOBA creates a network of active developers from the DDCM matrix. Each node represents an active developer and an edge is added between two nodes (developers) if there is a collaborative effort between them. Each edge is weighted by the degree of collaboration between developers as mentioned in Section II-B.

D. Detecting Developers Communities

In this step, DECOBA applies the greedy optimization of Clauset et al. [7] algorithm to discover developers communities from the developers network. The greedy optimization algorithm detects dense sub-graphs, also knows as communities through optimizing a modularity score. Modularity measures the strength of community structure and it ranges from 0 to 1. Higher the value of the modularity, stronger is the community structure in the network. This algorithm is considered one of the best algorithms to detect communities in large networks. Each community represents a different aspect of systems knowledge (i.e., developers in each community have experience in resolving bug reports that have the same technical concern). All community members share the same experience since they have collaborated in each others bug reports. Since developers are free to comment on any bug in Bugzilla, the bug comments reflect developers' interest. Communities constructed based on comments reveal the communication structure between them.

E. Building A Predictive Model

After detecting developers communities, DECOBA builds a predictive model that predicts a community to solve a newly reported bug. The bug assignment is formulated as a classification task where instances represent bug reports, features represent the distinctive terms of the report, and the class label represents the community that collaborates in solving

this report. It is noteworthy that Bugzilla records the assignee as one developer who fix the bug. Since DECOBA discovered dense communities in the developer network, DECOBA replaces each developer by his/her community. As mentioned before, the fixing process is a collaborative effort. Grouping developers into communities allow us to distribute the load between them and rank their experience.

For building the predictive model, DECOBA uses two widely used machine learning techniques namely Naive Bayes and Random Forests. The Naive Bayes algorithm is a probabilistic classifier that assumes that all features are independent and it finds the class with maximum probability given a set of features values using the Bayes theorem. The Random Forests algorithm generates many decision trees such that each tree predicts a class label and it selects the class label that has the majority votes.

After assigning a community to resolve a new bug report, DECOBA ranks developers in that community to find the most appropriate developers to fix the new bug. The developers are ranked based on their experience. DECOBA gives a weight for each developers in each community based on the following formula:

$$weight(D_i|C_j) = \frac{Fixed(D_i|C_j)}{\sum_{k=1}^{N} Fixed(D_k|C_j)} + \frac{Bet(D_i|C_j)}{\sum_{k=1}^{N} Bet(D_k|C_j)}$$

where $Fixed(D_i|C_j)$ represents the number of bugs fixed by developer D_i in community C_j, $Bet(D_i|C_j)$ represents the betweenness of developer D_i in community C_j, the denominators are used to get a normalized weight. The idea is that DECOBA gives more weight to developers who have fixed more bugs in his/her community and who are the most influential in the community.

F. Illustrative Example of DECOBA

This Section illustrates the DECOBA approach using a simple toy example as shown in Figure 1. The textual summary of each bug report is converted into a vector space model then the most distinctive terms are selected by X^2. Figure 1 (a) represents this step where each row represents a vector space model for each bug, each report is represented as five distinctive terms $t_1, \cdots, t_5$ and an initial class label that represent a developer who is assigned to fix it. Then, the developers adjacency matrix, DDCM, is constructed from bug reports. Figure 1 (b) shows an example of adjacency matrix of seven developers (we just show non-zero values). Each entry in the matrix represents the summation of three different values as aforementioned. For example, DDCM[D3,D4]=14 where 14 represents the summation of three values as follows: 1) the number of comments made by developer 3 on developer 4 bug reports; 2) the number of comments made by developer 4 on developer 3 bug reports; and 3) the number of bug reports that assigned to other developers that both developer 3 and 4 commented on. After that, the adjacency matrix is converted to a weighted undirected graph as shown in Figure 1 (c). For example, an edge is added between developer (node) 3 and developer 4 with a weight of 14. Next, the community detection algorithm is applied to discover developers communities as appear in Figure 1 (d). It detects two communities where developers 1, 2, 5, and 6 belong to the first community (C1) while developers 3, 4, and 7 belong to the second community (C2). Then, as shown in Figure 1 (e), each developer (initial class label) is replaced by his/her community in the training data. Afterwards, a predictive model is built on the training data. When a new reported bug (testing instance) is received by the tracking system, the predictive model can predict a community to handle it as shown in Figure 1 (f).

III. Evaluation Metrics

In this Section, we briefly introduce the evaluation metrics used in DECOBA. Section III-A presents some metrics to perform network analysis and to measure the quality of the detected communities. Section III-B presents the metrics used to evaluate the classification results.

A. Network Metrics

1) Centralization: The centralization of a network calculates a graph-level centrality index based on node-level centrality measure. Centrality measures allow to find developers who are extensively indulged in relationships with other network developers.

2) Betweenness Centrality: Betweenness centrality is defined by the number of shortest paths going through a node [8]. It is a useful measure of the load and importance of a node. A node with high betweenness has great influence over other nodes in the network.

3) Average Degree: The average degree of a network is computed as the average internal degree of all nodes in the network. The degree of a node is the number of its adjacent edges. It somehow shows how much nodes are interconnected.

4) Density: The density of a network is defined as the number of the edges present in the network divided by maximum possible edges excluding self loops in the network. Community discovery algorithms strive for dense sub-networks where the higher the density of these sub-networks, the better the quality.

5) Average Community Size: The average community size refers to the average number of nodes within each community. The average community size is a key quantitative characteristic of community structure in a network as it indicates if communities are disparate or uniform in their division of a network [9].

B. Classification Metrics

In this Section, we present the widely used classification metrics namely Precision, Recall, and F-Measure. Precision is the percentage of suggested developers who actually worked toward fixing the bug. Recall is the percentage of developers who worked on the bug who were actually suggested. F-Measure is the harmonic mean of Precision and Recall.

IV. Experiments and Discussion

A. Datasets and Prepossessing

1) Datasets: We evaluate DECOBA on bug repositories of NetBeans, Freedesktop, and Mandriva. We collect the bug reports that have the status of Closed, Verified, and Resolved and the resolution of Fixed. For each bug report, we extract

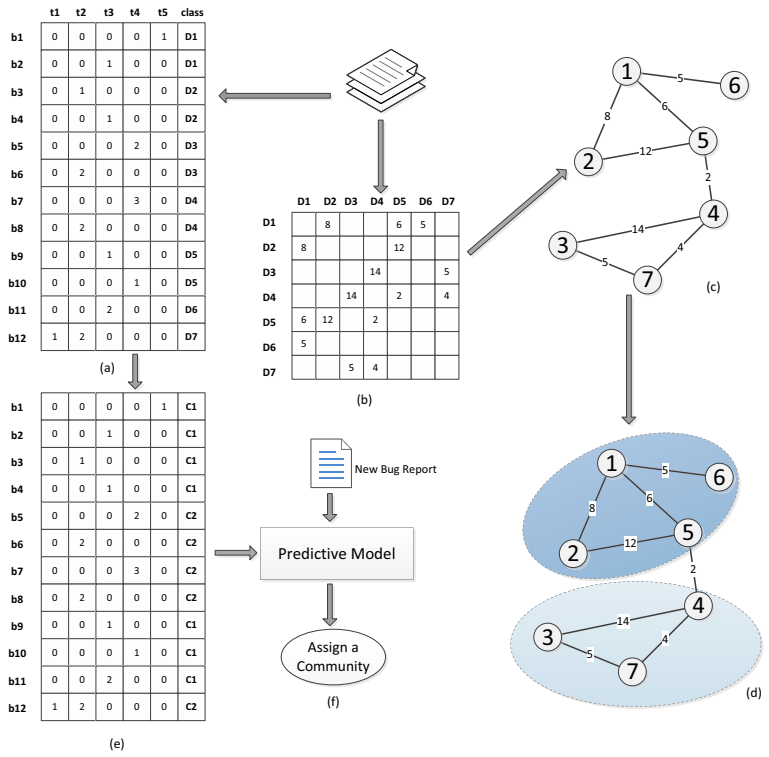


Fig. 1. A toy example of the DECOBA Approach. (a) shows a bug-term matrix reduced by X^2 where class represents the developer who is assigned to fix the bug. (b) represents developers adjacency matrix, DDCM where DDCM[D1,D2] = 8 represents the collaborative effort between developer D1 and developer D2. (c) represents the developers network. (d) shows the results of applying the community detection algorithm. (e) shows the bug-term matrix where the developer is replaced by his/her community. After creating the training data that appears in (e), a predictive model is built and then used to predict a community to a new bug report as shown in (f).

TABLE I. STATISTICS ABOUT THE DATASETS

Name	# bugs	From	to
NetBeans	14861	Apr, 01, 2011	Dec 25, 2012
Freedesktop	9981	Jan, 01, 2011	Dec 25, 2012
Mandriva	9199	Apr, 01, 2008	Dec 25, 2012

the bug ID, the assignee, the summary, and the commenters. Table I shows a statistics about the selected datasets.

To consider only active developers, reports that are assigned to inactive developers (i.e., developers who no longer work on the project or developers who have only fixed a small number of bugs) are removed. In addition, developers who have fixed at least 15 bug reports in the dataset are considered active. Table II shows a summary of the refined datasets.

TABLE II. SUMMARY OF THE DATASETS

Name	# of Bug Reports	# developers
NetBeans	14354	72
Freedesktop	8730	117
Mandriva	8582	103

B. Results

1) Developers Communities Results: Table IV presents some analysis measures of the constructed networks. where *Centrality* represents the graph level centrality index, *Betweenness* represents the average betweenness of all nodes, and *Degree* represents the average degree of all nodes. Table IV shows that Freedesktop has more influential developers than other projects (the average betwenness of all developers is 77.67). It also shows that NetBeans has the highest connectivity between developers (the average degree is 43.19).

TABLE IV. ANALYSIS OF DEVELOPERS NETWORK

Name	Centrality	Betweenness	Degree
NetBeans	0.35	43.62	43.19
Freedesktop	0.33	77.67	15.06
Mandriva	0.34	42.77	37.63

In order to measure the quality of the detected communities, we calculate several metrics namely NOC, ACS, Density, and AD. Table V shows the results where NOC denotes the number of communities, ACS denotes the average community size, Density represents the average density of all communities, and AD denotes the average internal degree of nodes in all communities. It is clear from Table V that the average density of communities are high which indicates strong interconnected communities. NetBeans has the highest average density (0.82) while Freedesktop has the lowest average density (0.59). Detected communities in all developers networks share a similar average internal degree of nodes (around 8).

TABLE III. COMMUNITIES ANALYSIS

Name	Community	Density	Max Betweenness	Developer
NetBeans	C1	0.76	144	pgebauer
	C2	0.80	127	AlyonaStashkova
	C3	0.67	147	av-nb
	C4	0.85	162	jsedek
	C5	1	38	jkovalsky
	C6	0.83	174	vkvashin
Freedesktop	C1	0.35	622	daniel
	C2	0.35	285	will.thompson
	C3	0.48	484	tfheen
	C4	0.86	145	mesa-dev
	C5	0.91	288	programming
Mandriva	C1	0.59	318	supp
	C2	0.66	148	tpg
	C3	0.90	91	tmb
	C4	0.73	69	jvictor
	C5	0.53	53	andreas

TABLE V. COMMUNITIES RESULTS

Name	NOC	ACS	Density	AD
NetBeans	6	12	0.82	8.80
Freedesktop	5	15.8	0.59	8.30
Mandriva	5	12.8	0.68	8.53

Table III shows some interesting findings about the detected communities. For each detected community, we report its density, maximum betweeness value, and the most influential developer (i.e., the developer who has the highest betweenness in that community). The most influential developer in each community will be used later to rank developers in each community. For NetBeans, community C5 has a density value of one which forms a clique (i.e., every two nodes are connected by an edge). The lowest density (0.67) is observed in community C3. For Freedesktop, community C5 has the highest density (0.91) while both C1 and C2 have the lowest density (0.35). For Mandariva, community C3 has the highest density (0.90) while C5 has the lowest density (0.53).

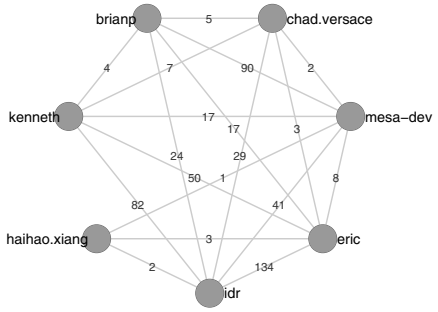


Fig. 2. One of the communities in the Freedesktop project. The density is 0.86, the average betweenness is 2.86 and the average degree is 5.14.

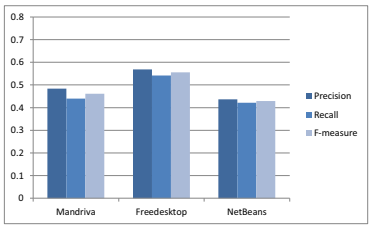
Figure 2 shows an example of one of the communities (C4) in the Freedesktop project. It also shows high weights of many of the connection between developers. For instance, the weight of the connection between eric and idr is 134 which indicates a high volume of collaboration. After a deep analysis of this community, we notice that developers in this community fix bugs belong to common products and components. The developers in this community have a high number of connections between each other. The developer, mesa-dev, is the most influential developer in this community with a betweenness value of 145.

2) Classification Results: For evaluation, we use two popular machine learning techniques namely Naive Bayes and Random Forests. The dataset is divided into training and testing sets. To obtain unbiased evaluation results, we perform a 10-fold cross-validation. Figure 3 (a) shows Precision, Recall, and F-measure with Naive Bayes. The lowest F-measure is obtained for NetBeans (0.43) while the highest F-measure is obtained for Freedesktop (0.56). Figure 3 (b) shows Precision, Recall, and F-measure with Random Forests. The lowest F-measure is obtained for Mandriva (0.49) while the highest F-measure is obtained for Freedesktop (0.69). It is clear from Figure 3 that Random Forests is slightly better than Naive Bayes in all projects.

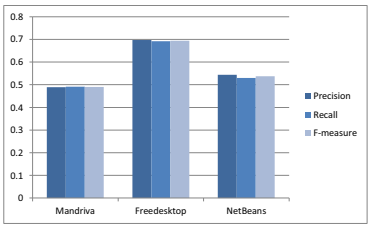
V. RELATED WORK

Recently, developers' social networks have been used to support several software activities. Nodes in these networks represent developers while edges represent the interaction among them. These interactions can represent several relationships such as contribution of fixing same bugs, communication through emails, and changing the same artifact. Bird et al. [10] built a social network of email correspondents (developers and non-developers). One of their important findings is that social network measures such as in-degree, out-degree and betweenness show that developers who actually commit changes, play important roles in the email community than non-developers. In later study [11] they analyzed the email communication of open source projects and found that sub-communities manifest most strongly in technical discussions, and are considerably connected with collaboration behavior. Hong et al. [4] studied the evolution of developers' social networks of Mozilla. They found that while most social networks demonstrate power law degree distributions, developers' social networks do not. They also found strong community structure in them. Crowston et al. [12] studied large number of open source teams interactions for their communications centralization. They found that there is no common pattern of communication centralization of these projects where some projects are highly cantered on one developer while others are not. In this paper, DECOBA constructs developers social networks based on their comments in bug reports.

Many approaches adopted both machine learning and information retrieval techniques to solve the bug assignment



(a) NaiveBayes



(b) Random Forest

Fig. 3. Classification results using Naive Bayes and Random Forests.

problem. Čubranić et al. [13] were the first to formulate the bug assignment problem as a text classification. Anvik et al. [14] enhanced the approach proposed by Čubranić et al. by 1) removing inactive developers; 2) using project-specific heuristics to label bug reports; and 3) applying different machine learning techniques namely SVM, Naive Bayes and Decision Trees. Zou et al. [6] proposed an enhancement to bug assignment problem by using both feature selection and instance selection techniques. They evaluated their approach on the Eclipse project and showed that the combinations of feature selection and instance selection achieved better accuracy. Aljarah et al. [15] investigated the effect of some term selection approaches on the classification effectiveness. Their results indicated that Log Odds Ratio outperforms Latent Semantic Analysis and Information Gain. Banitaan and Alenezi [16] proposed TRAM, an approach to raise the prediction accuracy of bug triage by using the most distinguished terms of bug reports, the components in which the bugs belong to, and the reporter who filed the bug. Their experimental evaluation showed that TRAM outperforms many existing machine learning-based approaches in terms of classification accuracy. Alenezi et al. [5] investigated which of five state-of-the-art term selection method is better to use in bug assignment. Their experimental evaluation showed that X^2 term selection method achieved the best results a cross five different open source projects. In this work, DECOBA builds a predictive model using 30% of the textual corpus and predicts a community to handle a new bug report instead of one developer.

VI. Conclusions

In this paper, we presented DECOBA , an approach that builds a developers social network based on their collaboration comments and detects strong developers communities. These detected communities are then utilized in bug assignment. Two different machine learning techniques are used to build predictive models. The predictive model assigns a relevant community to a newly submitted bug report in order to fix it. We conducted experimental evaluation on three open source projects. We have shown the feasibility of the DECOBA approach and applied it to bug assignment. Our results give strong evidence of the community structure within developer's social network. They also showed that DECOBA achieved feasible prediction accuracy of bug assignment. Developers communities built in DECOBA can be used in several applications such as propagating information to some relevant communities and studying the evolution of leadership between developers.

References

[1] A. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 344–353.

[2] A. Begel, Y. P. Khoo, and T. Zimmermann, "Codebook: discovering and exploiting relationships in software repositories," in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 1. IEEE, 2010, pp. 125–134.

[3] J. Robertsa, I.-H. Hann, and S. Slaughter, "Communication networks in an open source software project," *Open Source Systems*, pp. 297–306, 2006.

[4] Q. Hong, S. Kim, S. Cheung, and C. Bird, "Understanding a developer social network and its evolution," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 2011, pp. 323–332.

[5] M. Alenezi, K. Magel, and S. Banitaan, "Efficient bug triaging using text mining," *Journal of Software*, vol. 8, no. 9, pp. 2185–2190, 2013.

[6] W. Zou, Y. Hu, J. Xuan, and H. Jiang, "Towards training set reduction for bug triage," in *Proceedings of the 2011 IEEE 35th Annual Computer Software and Applications Conference*, ser. COMPSAC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 576–581.

[7] A. Clauset, M. E. Newman, and C. Moore, "Finding community structure in very large networks," *Physical review E*, vol. 70, no. 6, p. 066111, 2004.

[8] L. C. Freeman, "Centrality in social networks conceptual clarification," *Social networks*, vol. 1, no. 3, pp. 215–239, 1979.

[9] S.-Y. Chan, P. Hui, and K. Xu, "Community detection of time-varying mobile social networks," *Complex Sciences*, pp. 1154–1159, 2009.

[10] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *Proceedings of the 2006 international workshop on Mining software repositories*. ACM, 2006, pp. 137–143.

[11] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu, "Latent social structure in open source projects," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 24–35.

[12] K. Crowston and J. Howison, "The social structure of free and open source software development," *First Monday*, vol. 10, no. 2-7, 2005.

[13] D. Čubranić and G. C. Murphy, "Automatic bug triage using text categorization," in *In SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering*. Citeseer, 2004, pp. 92–97.

[14] J. Anvik and G. C. Murphy, "Reducing the effort of bug report triage: Recommenders for development-oriented decisions," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 3, p. 10, 2011.

[15] I. Aljarah, S. Banitaan, S. Abufardeh, W. Jin, and S. Salem, "Selecting discriminating terms for bug assignment: a formal analysis," in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*. ACM, 2011, p. 12.

[16] S. Banitaan and M. Alenezi, "Tram: An approach for assigning bug reports using their metadata," in *Communications and Information Technology (ICCIT), 2013 Third International Conference on*. IEEE, 2013, pp. 215–219.