Fine-grained Incremental Learning and Multi-feature Tossing Graphs to Improve Bug Triaging

Pamela Bhattacharya Iulian Neamtiu
Department of Computer Science and Engineering
University of California, Riverside
Riverside, California 92521
Email:{pamelab,neamtiu}@cs.ucr.edu

***Abstract*—Software bugs are inevitable and bug fixing is a difficult, expensive, and lengthy process. One of the primary reasons why bug fixing takes so long is the difficulty of accurately assigning a bug to the most competent developer for that bug kind or bug class. Assigning a bug to a potential developer, also known as bug triaging, is a labor-intensive, time-consuming and fault-prone process if done manually. Moreover, bugs frequently get re-assigned to multiple developers before they are resolved, a process known as bug tossing. Researchers have proposed automated techniques to facilitate bug triaging and reduce bug tossing using machine learning-based prediction and tossing graphs. While these techniques achieve good prediction accuracy for triaging and reduce tossing paths, they are vulnerable to several issues: outdated training sets, inactive developers, and imprecise, single-attribute tossing graphs. In this paper we improve triaging accuracy and reduce tossing path lengths by employing several techniques such as refined classification using additional attributes and intra-fold updates during training, a precise ranking function for recommending potential tossees in tossing graphs, and multi-feature tossing graphs. We validate our approach on two large software projects, Mozilla and Eclipse, covering 856,259 bug reports and 21 cumulative years of development. We demonstrate that our techniques can achieve up to 83.62% prediction accuracy in bug triaging. Moreover, we reduce tossing path lengths to 1.5–2 tosses for most bugs, which represents a reduction of up to 86.31% compared to original tossing paths. Our improvements have the potential to significantly reduce the bug fixing effort, especially in the context of sizable projects with large numbers of testers and developers.**

I. Introduction

Building and maintaining software is expensive. A survey by the National Institute of Standards and Technology estimated that the annual cost of software bugs is about \$59.5 billion [1]. Other studies indicate that maintenance costs are at least 50%, and sometimes more than 90%, of the total costs associated with a software product [2], [3]. These surveys suggest that making the bug fixing process more efficient would lower software production costs.

Most software projects use bug trackers to organize the bug fixing process and facilitate application maintenance. For instance, Bugzilla is a popular bug tracker used by many large projects, such as Mozilla, Eclipse, KDE, and Apache [4]. These applications receive hundreds of bug reports a day; ideally, each bug gets assigned to a developer who can fix it in the least amount of time. This process of assigning bugs, known as *bug triaging*, is complicated by several factors: if done manually, triaging is labor-intensive, time-consuming and fault-prone; moreover, for open source projects, it is difficult to keep track of active developers and their expertise. Identifying the right developer for fixing a new bug is further aggravated by growth, e.g., as projects add more components, modules, developers and testers [5]. An empirical study by Jeong et al. [6] reports that, on average, the Eclipse project takes about 40 days to assign a bug to the first developer, and then it takes an additional 100 days or more to reassign the bug to the second developer. Similarly, in the Mozilla project, on average, it takes 180 days for the first assignment and then an additional 250 days if the first assigned developer is unable to fix it. These numbers indicate that the lack of triaging and tossing techniques results in considerably high effort associated with bug resolution.

Effective bug triaging can be divided into two subgoals: (1) assigning a bug for the first time to a matching potential developer, and (2) reassigning it to another promising developer if the first assignee is unable to resolve it, then repeat this reassignment process (*bug tossing*) until the bug is fixed. Our findings indicate that at least 93% of all "fixed" bugs in both Mozilla and Eclipse have been tossed at least once (tossing path length ≥ 1). Ideally, for any bug triage event, the tossing length should be zero, i.e., the first person the bug is assigned to should be able to fix it; if that is not possible, the bug should be resolved in a minimum number of tosses. Prior automatic bug triaging approaches [7]–[10] use the history of bug reports and developers who fixed them to train a classifier.[1] Later, when keywords from new bug reports are given as an input to the classifier, it recommends a set of developers who have fixed similar classes of bugs in the past and are hence considered potential bug-fixers for the new bug. Jeong et al. [6] used a novel approach to automate bug triaging by building tossing graphs from tossing histories. While classifiers and tossing graphs are effective in improving the prediction accuracy for triaging and reducing tossing path lengths, their accuracy is threatened by several issues: outdated training sets, inactive developers, and imprecise, single-attribute tossing graphs.

In this paper, we demonstrate how bug triaging can be further improved. In particular, we propose three novel extensions to existing techniques.

[1]A *classifier* is a machine learning algorithm that can be trained using input attributes and desired output classes; after training, when presented with a set of input attributes, the classifier predicts the most likely output class.

First, we achieve higher prediction accuracy using richer feature vectors. In addition to the bug title and summaries used in prior work, we add attributes corresponding to the product–component information for a bug.

The second novel aspect of our work is employing a fine-grained incremental learning approach. Prior work has used folding[2] to train the classifier and predict developers [9]. However, this technique becomes inadequate for large fold sizes, e.g., thousands of bugs: since the classifier is only updated at the end of each fold validation, it soon becomes outdated during the next validation, which results in low accuracy. In contrast, we employ fine-grained, intra-fold updates which keep the classifier up-to-date at all times.

The third novel aspect of our work is constructing multi-feature tossing graphs. Prior work [6] has trained a classifier with fixed bug histories; for each new bug report, the classifier recommends a set of potential developers, and for each potential developer, a tossing graph—whose edges contain tossing probabilities among developers—is used to predict possible re-assignees. However, the tossing probability alone is insufficient for recommending the most competent, still active, developer (see Section III for an example). In particular, in open source projects it is difficult to keep track of active developers and their expertise. To address this, in addition to tossing probabilities, we label tossing graph edges with developer expertise and tossing graph nodes with developer activity, which help reduce tossing path lengths significantly.

Similar to prior work, we test our approach on the fixed bug data sets for Mozilla and Eclipse. We report up to 84% prediction accuracy for Mozilla and 82.59% for Eclipse. We also find that using our approach reduces the length of tossing paths by up to 86% for correct predictions.

In Section II we define terms and techniques used in bug triaging. We discuss deficiencies with existing approaches, and present a high-level description of how we address these issues in Section III. In Section IV we elaborate on our techniques and implementation details. We present our experimental setup and results in Section V, followed by threats to validity in Section VI and related work in Section VII.

In summary, the main contributions of this paper are:

1) We demonstrate that employing a larger set of attributes associated with bug reports and performing fine-grained incremental learning via intra-fold updates increase bug triaging accuracy.
2) We show that multi-featured tossing graphs and a precise ranking function further improve triaging accuracy and reduces tossing.

II. Preliminaries

We first define several fundamental machine learning and bug triaging concepts that form the basis of our approach.

[2]Split sample validation (folding) divides the sample sets into equal-sized folds; the folds are then incrementally used for training and validation.

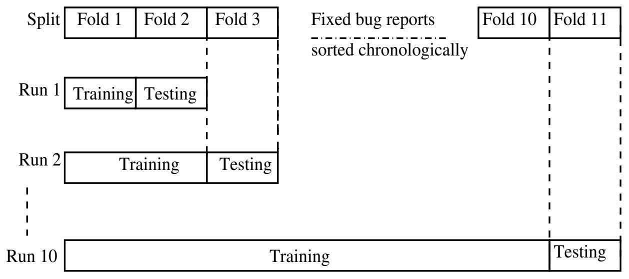


Fig. 1. Folding techniques for classification as used by Bettenburg et al.

Machine Learning for Bug Categorization: Classification is a machine learning technique for deriving a general trend from a training data set. The *training data set* (TDS) consists of pairs of input objects (called feature vectors), and their respective target outputs. The task of the supervised learner (or classifier) is to predict the output given a set of input objects, after being trained with the TDS. Feature vectors for which the desired outputs are already known form the *validation data set* (VDS) that can be used to test the accuracy of the classifier. Machine learning techniques were used by previous bug triaging works [7]–[9]: archived bug reports form feature vectors, and the developers who fixed the bugs are the outputs of the classifier. Therefore, when a new bug report is provided to the classifier, it predicts potential developers who can fix the bug based on their bug fixing history.

Feature Vectors: The accuracy of a classifier is highly dependent on the feature vectors in the TDS. Bug title and summaries have been used earlier to extract the keywords that form feature vectors. These keywords are extracted such that they represent a specific class of bugs. For example, if a bug report contains words like "icon," "image," or "display," it can be inferred that the bug is related to application layout, and is assigned to the "layout" class of bugs. We used multiple text classification techniques (`tf-idf`, stemming, stop-word and non-alphabetic word removal [11]) to extract relevant keywords from the actual bug report.

Folding: Early bug triaging approaches [6]–[8] divided the data set into two subsets: 80% for TDS and 20% for VDS. Bettenburg et al. [9] have used folding (similar to split-sample validation techniques from machine learning [12]) to achieve higher prediction accuracy. In a folding-based training and validation approach (illustrated in Figure 1), the algorithm first collects all bug reports to be used for TDS, sorts them in chronological order and then divides them into n folds. In the first run, fold 1 is used to train the classifier and then to predict the VDS. In run 2, fold 2 bug reports are added to TDS. In general, after validating the VDS from fold n, that VDS is added to the TDS for validating fold $n+1$. To reduce experimental bias [12], similar to Bettenburg et al., we chose $n = 11$ and carried out 10 iterations of the validation process.

Goal-Oriented Tossing Graphs: When a bug is assigned to a developer for the first time, and she/he is unable to fix it, the bug is assigned (tossed) to another developer. Thus a bug is tossed from one developer to another until a developer is

Tossing Paths
$A \rightarrow B \rightarrow C \rightarrow D$
$A \rightarrow E \rightarrow D \rightarrow C$
$A \rightarrow B \rightarrow E \rightarrow D$
$C \rightarrow E \rightarrow A \rightarrow D$
$B \rightarrow E \rightarrow D \rightarrow F$

Developer who tossed the bug	**Total Tosses**	**Developers who fixed the bug**					
		C		D		F	
		#	Pr	**#**	Pr	**#**	Pr
A	4	1	0.25	3	0.75	0	0
B	3	0	0	2	0.67	1	0.33
C	2	-	-	2	1.00	0	0
D	2	1	0.50	-	-	1	0.50
E	4	1	0.25	2	0.50	1	0.25

TABLE I
TOSSING PATHS AND PROBABILITIES AS USED BY JEONG ET AL.

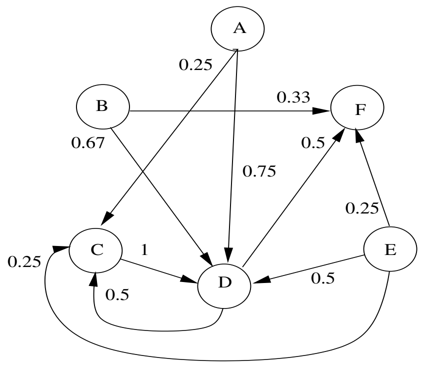


Fig. 2. Tossing graph built using tossing paths in Table I.

eventually able to fix it. Based on these tossing paths, goal-oriented tossing graphs were proposed by Jeong et al [6]; for the rest of the paper, by "tossing graph," we refer to a goal-oriented tossing graph. Tossing graphs are weighted directed graphs such that each node represents a developer, and each directed edge from D_1 to D_2 represents the fact that a bug assigned to developer D_1 was tossed and eventually fixed by developer D_2. The weight of an edge between two developers is the probability of a toss between them, based on bug tossing history. The *tossing probability*, also known as the *transaction probability*, from developer D to D_j (denoted as $D \hookrightarrow D_j$) is defined by the following equation:

$$Pr(D \hookrightarrow D_j) = \frac{\sum_1^m D \hookrightarrow D_j : D_j \text{ fixed the bug}}{\sum_{i=1}^n D \hookrightarrow D_i} \qquad (1)$$

In this equation, the numerator is the number m of tosses from developer D to D_j such that D_j fixed the bug, while the denominator is the total number of tosses from D to any other developer D_i such that D_i fixed the bug; *n* represents the total number of developers D tossed a bug to. To illustrate this, in Table I we provide sample tossing paths and show how toss probabilities are computed. For example, developer A has tossed four bugs in all, three to D and one to C, hence $Pr(A \hookrightarrow D) = 0.75$, $Pr(A \hookrightarrow C) = 0.25$, and $Pr(A \hookrightarrow F) = 0$. Note that developers who did not toss any bug (e.g., F) do not appear in the first column, and developers who did not fix any bugs (e.g., A) do not have a probability column. In Figure 2, we show the final tossing graph built using the computed tossing probabilities.

Prediction Accuracy: If the first developer in our prediction list matches the actual developer who fixed the bug, we have a hit for the Top 1 developer count. Similarly, if the second developer in our prediction list matches the actual developer who fixed the bug, we have a hit for the Top 2 developer count. For example, if there are 100 bugs in the VDS and for 20 of those bugs the actual developer is the first developer in our prediction list, the prediction accuracy for Top 1 is 20%; similarly, if the actual developer is in our Top 2 for 60 bugs, the Top 2 prediction accuracy is 60%.

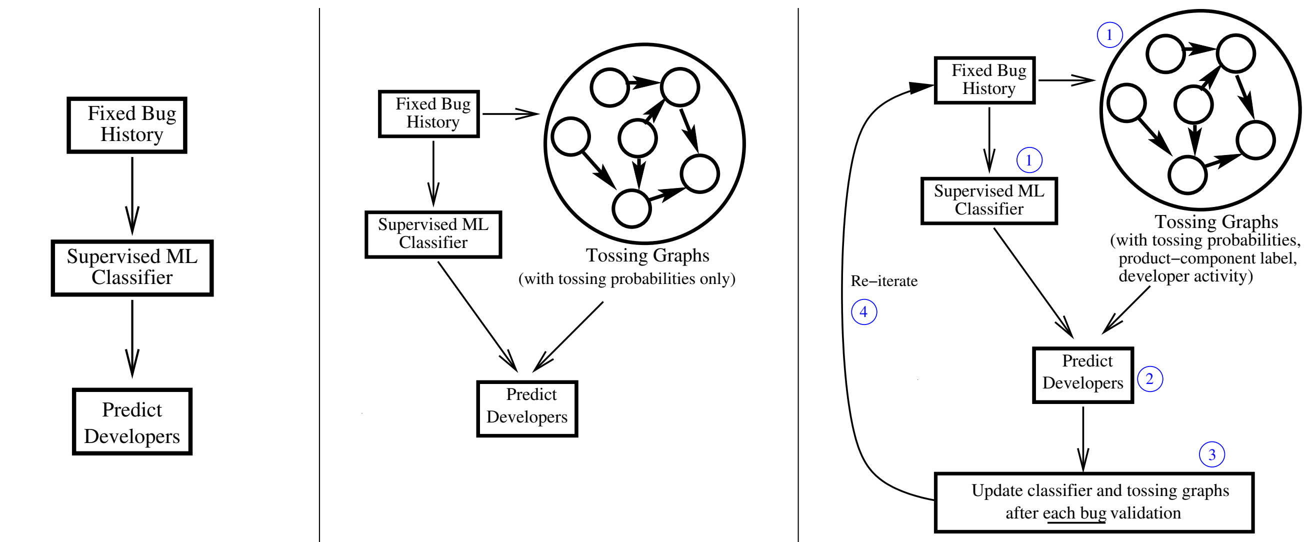
III. APPROACH

In Figure 3 we compare our approach to previous techniques. Initial works in this area (Figure 3(a)) used classifiers only [7]–[10]; more recent work by Jeong et al. [6] (Figure 3(b)) coupled classifiers with tossing graphs. Our approach (Figure 3(c)) adds fine-grained incremental learning and multi-feature tossing graphs. Our algorithm consists of four stages, as labeled in the figure: (1) initial classifier training and building the tossing graphs, (2) predicting potential developers, using the classifier and tossing graphs, (3) measuring prediction accuracy, (4) updating the training sets using the bugs which have been already validated, re-running the classifier and updating the tossing graphs. We iterate these four steps until all bugs have been validated. We present a high-level description of these steps next; the details for each step can be found in Section IV.

Data Sets: We first sort all bugs marked as "fixed" chronologically, then extract attributes associated with each of these bugs. The attributes in our TDS are: keywords from bug reports including bug title and description, the product and component the bug belongs to, and the ID of the developer who fixed the bug.

Building Tossing Graphs: Tossing graphs are built using tossing probabilities derived by analyzing bug tossing histories, as explained in Section II. Jeong et al. [6] determine potential tossees as follows: if developer A has tossed more bugs to developer B than A has tossed to D, in the future, when A cannot resolve a bug, the bug will be tossed to B, hence tossing probabilities determine tossees. However, this approach might be inaccurate in certain situations: suppose a new bug belonging to class K_1 is reported, and developer A was assigned to fix it, but he is unable to fix it; developer B has never fixed any bug of type K_1, while developer D has fixed 10 bugs of type K_1. The prior approach would recommend B as the tossee, although D is more likely to resolve the bug, rather than B. Thus, although tossing graphs reveal tossing probabilities among developers, they should also contain information about which classes of bugs were passed from one developer to another; we use multi-feature tossing graphs to capture this information.

Another problem with the classifier- and tossing graph-



(a) Classifier-based bug triaging

(b) Classifiers coupled with tossing graphs

(c) Incremental learning and multi-feature tossing graphs (our approach)

Fig. 3. Comparison of bug triaging techniques.

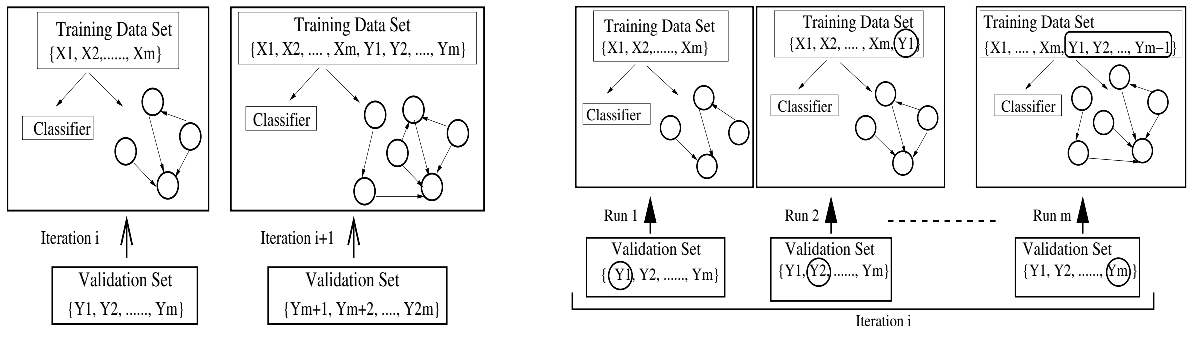
based approaches is that it is difficult to identify retired or inactive developers. This issue is aggravated in open source projects: when developers work voluntarily, it is difficult to keep track of the current set of active developers associated with the project. Anvik et al. [7] and Jeong et al. [6] have pointed out this problem and proposed solutions. Anvik et al. use a heuristic to filter out developers who have contributed to less than 9 bug resolutions in the last three months of the project. Jeong et al. assume that all developers who have numerous outgoing edges in tossing graphs are potentially retired, hence the bug will be passed on to some other developer; when a developer is not responding, the manager, group moderator or other members of the bug assignment team will re-assign the bug. Therefore, their approach permits assigning bugs to inactive developers, which increases the length of the tossing paths. In contrast, we restrict potential assignees to active developers only, and do so with a minimum number of tosses.

Hence, the tossing graphs we build have additional labels compared to Jeong et al.: for each bug that contributes to an edge between two developers, we attach the bug class (product and component) to that edge; moreover, for each developer in the tossing graph, we maintain an activity count (the difference between the date of the bug being validated and the date of the last activity of that developer).

Predicting Developers: For each bug, we predict potential developers using two methods: (1) using the classifier alone, to demonstrate the advantages of incremental learning, and (2) using both the classifier and tossing graphs, to show the significance of multi-feature tossing graphs. When using the classifier alone, the input consists of bug keywords, and the classifier returns a list of developers ranked by relevance; we select the top five from this list. When using the classifier in conjunction with tossing graphs, we select the top three developers from this list, then for developers ranked 1 and 2 we use the tossing graph to recommend a potential tossee, similar to Jeong et al. For predicting potential tossees based on the tossing graph, our tossee ranking function takes into account multiple factors, in addition to the tossing probability as proposed by Jeong et al. In particular, our ranking function is also dependent on (1) the product and component of the bug, and (2) the last activity of a developer, to filter retired developers. The details of the prediction process are described in Section IV-D. Thus our final list of predicted developers contains five developer id's in both methods, classifier alone and classifier + tossing graph.

Updating Classifier and Tossing Graphs: Prior work [6], [9] has used *inter*-fold updates, i.e., the classifier and tossing graphs are updated after each fold validation, as shown in Figure 4(a). With inter-fold updates, after validating the VDS from fold n, the VDS is added to the TDS for validating fold $n + 1$. However, consider the example when the TDS contains bugs 1–100 and the VDS contains bugs 101–200. When validating bug 101, the classifier and tossing graph are trained based on bugs 1–100, but from bug 102 onwards, the classifier and tossing graph are not up-to-date any more because they do not incorporate the information from bug 101. As a result, when the validation sets contain thousands of bugs, this incompleteness affects prediction accuracy. Therefore, to achieve high accuracy, it is essential that the classifier and tossing graphs be updated with the latest bug fix; we use a fine-grained, *intra*-fold updating technique for this purpose.

We now proceed to describing intra-fold updating. After the first bug in the validation fold has been used for prediction, and accuracy has been measured, we add it to the TDS and re-train the classifier as shown in Figure 4(b). We also update the tossing graphs by adding the tossing path of the just-validated bug. This guarantees that for each bug in the validation fold,



(a) Updates after each validation set (Bettenburg et al.) (b) Updates after each bug (our approach)

Fig. 4. Comparison of training and validation techniques.

Developer ID	**Product-Component**	**Fix Count**
D_1	$\{P_1, C_2\}$	3
	$\{P_1, C_7\}$	18
	$\{P_9, C_6\}$	7

TABLE II
SAMPLE DEVELOPER PROFILE.

the classifier and the tossing graphs incorporate information about all preceding bugs. This approach has first been used in the context of machine learning by Segal et al. [13].

Folding: Similar to the Bettenburg et al.'s folding technique [9], we iterate the training and validation for all the folds. However, since our classifier and tossing graph updates are already performed during validation, we do not have to update our training data sets after each fold validation. To maintain consistency in comparing our prediction accuracies with previous approaches, we measure the average prediction accuracy over each fold.

Computational Effort: The intra-fold updates used in our approach are more computationally-intensive than inter-fold updates. However, for practical purposes this is not a concern, e.g., when deploying our approach in the actual bug tracking system, we could update the classifier and tossing graph over night. We believe that high accuracy and saving a substantial manual effort are worth the extra computational load.

IV. IMPLEMENTATION

In this section we present a detailed description of each phase of our algorithm.

A. Developer Profiles

We maintain a list of all developers and their history of bug fixes. Each developer D has a list of product-component pairs $\{P, C\}$ and their absolute count attached to his or her profile. A sample developer profile is shown in Table II, e.g., developer D_1 has fixed 3 bugs associated with product P_1 and component C_2. This information is useful beyond bug assignments; for example, while choosing moderators for a specific product or component it is a common practice to refer to the developer performance and familiarity with that product or component.

B. Classification

Given a new bug report, the classifier produces a set of potential developers who could fix the bug. We describe the classification process in the remainder of this subsection.

Choosing Fixed Bug Reports: We use the same heuristics as Anvik et al. [7] for obtaining fixed bug reports from all bug reports in Bugzilla. First, we extract all bugs marked as "verified" or "resolved"; next, we remove all bugs marked as "duplicate" or "works-for-me," which leaves us with the correct set containing fixed bugs only.

Accumulating Training Data: Prior work [7]–[9] has used keywords from the bug report and developer name/id as attributes for the training data sets; we also include the product and component the bug belongs to. For extracting relevant words from bug reports, we employ `tf-idf`, stemming, stop-word and non-alphabetic word removal [11]. We use the Weka toolkit [14] to remove stop words and form the word vectors for the dictionary (via the `StringtoWordVector` class with tf-idf enabled).

Filtering Developers for Classifier Training: Anvik et al. refine the set of training reports by using several heuristics. For example, they do not consider developers who fixed a small number of bugs, which helps remove noise from the TDS. Although this is an effective way to filter non-experts from the training data and improve accuracy, in our approach filtering is unnecessary: the ranking function is designed such that, if there are two developers A and B who have fixed bugs of the same class K, but the number of K-type bugs A has fixed is greater than the number of K-type bugs B has fixed, a K-type bug will be assigned to A.

Classifier Type: We use Weka's built-in Naïve Bayes and Bayesian Networks classifiers in our approach. Prior work has used similar classifiers, as well as Support Vector Machines and C4.5 [6]–[9], and the results have shown that classifier choice can have a slight impact on prediction accuracy. We believe that more sophisticated classifiers could further improve our accuracy, but for the scope of this paper we used Naïve Bayes and Bayesian Networks only.

C. Multi-feature Tossing Graphs

With the training data and classifier at hand, we proceed to constructing tossing graphs.

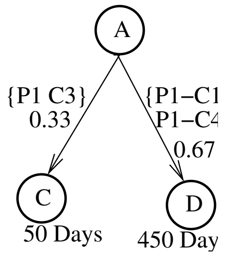


Fig. 5. Example of a multi-feature tossing graph.

Adding Attributes to Edges in Tossing Graphs: As discussed in Section III, tossing probabilities are a good start toward indicating potential bug fixers, but they might not be appropriate at all times. Therefore, the tossing graphs we generate have three labels in addition to the tossing probability: bug product and bug component on each edge, and number of days since a developer's last activity on each node. For example, consider three bugs that have been tossed from D_1 to D_2 and belong to three different product-component sets: $\{P_1, C_1\}$, $\{P_1, C_3\}$, and $\{P_2, C_4\}$. Therefore, in our tossing graph, the product-component set for the edge between D_1 and D_2 is $\{\{P_1, C_1\}, \{P_1, C_3\}, \{P_2, C_4\}\}$. Maintaining these additional attributes is also helpful when bugs are re-opened. Both developer profiles and tossing histories change over time, hence it is important to identify the last fixer for a bug and a potential tossee after the bug has been re-opened.

We now present three examples that demonstrate our approach and show the importance of multi-feature tossing graphs. The examples are based on the tossing graph in Figure 5; the graph indicates that developer A has been associated with developers C and D in the tossing history.

Example I. Suppose we encounter a new bug B_1 belonging to product P_1 and component C_4, and the classifier returns A as the best developer for fixing the bug. If A is unable to fix it, by considering the tossing probability and product–component match, we conclude that it should be tossed on to D.

Example II. Consider a bug B_2 belonging to product P_1 and component C_3. Although D has a higher transaction probability than C, if A is unable to fix it, because C has fixed bugs earlier from product P_1 and component C_3, he is more likely to fix it than D. Hence in this case the bug gets tossed from A to C.

Example III. Based on the last active count for D in Figure 5, i.e., 450 days, it is likely that D is a retired developer. In our approach, if a developer has been inactive for more than 100 days[3], we choose the next potential neighbor (tossee) from the reference node A. In this particular case, we choose C as the next tossee. We also use activity counts to prune inactive developers from classifier recommendations. For example, if the classifier returns n recommendations and we find that the i^{th} developer is probably retired, we do not select him, and move on to the $(i+1)^{st}$ developer.

[3]Choosing 100 days as the threshold was based on Anvik et al. [7]'s observation that developers that have been inactive for three months or more are potentially retired.

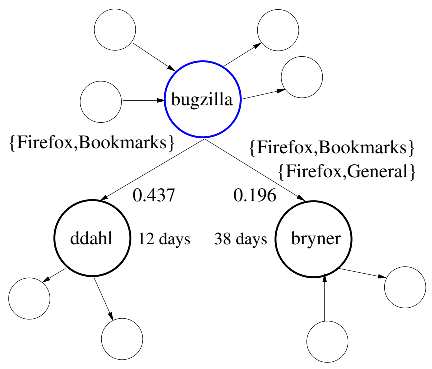


Fig. 6. Example to demonstrate significance of multi-feature tossing graphs.

Filtering Developers for Building Tossing Graphs: We do not prune the tossing graphs based on a pre-defined minimum support (frequency of contribution) for a developer, or the minimum number of tosses between two developers. Jeong et al. [6] discuss the significance of removing developers whose support is less than 10 and pruning edges between developers that have less than 15% transaction probability. Since their approach uses the probability of tossing alone to rank neighboring developers, they need the minimum support values to prune the graph. In contrast, the multiple features in our tossing graphs coupled with the ranking function (as explained in the Section IV-D) obviate the need for pruning.

D. Prediction

Example (Mozilla bug 254967). For this particular bug, the first five developers predicted by the Naïve Bayes classifier are {*bugzilla*, *fredbezies*, *myk*, *tanstaafl*, *ben.bucksch*}. However, since *bryner* is the developer who actually fixed the bug, our classifier-only prediction is inaccurate in this case. Therefore, we use the tossing graphs to select the most likely tossee for *bugzilla*, the first developer in the classifier ranked list. In Figure 6, we present the node for *bugzilla* and its neighbors.[4] If we rank the outgoing edges of *bugzilla* based on tossing probability alone, the bug should be tossed to developer *ddahl*. Though *bryner* has lower probability, he has committed patches to the product "Firefox" and component "General" that bug 254967 belong to. Hence our algorithm will choose *bryner* as the potential developer over *ddahl*, and our prediction matches the actual bug fixer. Our ranking function also takes into account developer activity; in this example, however, both developers *ddahl* and *bryner* are active, hence comparing their activities is not required. To conclude, our ranking function increases prediction accuracy while reducing tossing lengths; the actual tossing length for this particular Mozilla bug was 6, and our technique reduces it to 2.

Ranking Function: We know describe our algorithm for ranking developers. Similar to Jeong et al., we first use the classifier to predict a set of developers CP (Classifier

[4]For clarity, we only present the nodes relevant to this example, and the labels at the point of validating this bug; due to incremental learning, label values will change over time.

Developer ID	Product-Component	Fix Count
T_i	$\{P_1, C_1\}$	3
	$\{P_1, C_7\}$	18
	$\{P_9, C_6\}$	7
T_j	$\{P_1, C_1\}$	13
	$\{P_4, C_6\}$	11

TABLE III
EXAMPLE TO DEMONSTRATE TIE IN RANKS.

Predicted). Using the last activity information, we remove all developers who have not been active for the past 100 days from CP. We then sort the developers in CP using the fix counts from the developer profile (as described in Section IV-A).

Suppose the CP is $\{D_1, D_2, D_3, \ldots, D_j\}$. For each D_i in the sorted CP, we rank its tossees T_k (outgoing edges in the tossing graph) using the following ranking function:

$$\begin{aligned}\text{Rank}(T_k) = {} & Pr(D_i \hookrightarrow T_k) + \\ & MatchedProduct(T_k) + \\ & MatchedComponent(T_k) + \\ & LastActivity(T_k)\end{aligned}$$

The tossing probability, $Pr(D_i \hookrightarrow T_k)$, is computed using equation 1 (Section II). The function $MatchedProduct(T_k)$ returns 1 if the product the bug belongs to exists in developer T_k's profile, and 0 otherwise. Similarly, the function $MatchedComponent(T_k)$ returns 1 if the component the bug belongs to exists in developer T_k's profile. The $LastActivity$ function returns 1 if T_k's last activity was in the last 100 days from the date the bug was reported. As a result, $0 < \text{Rank}(T_k) \leq 4$. We then sort the tossees T_k by rank, choose the developer T_i with highest rank and add it to the new set of potential developers (ND). Thus after selecting T_i, where $i = 1, 2, \ldots, j$, the set ND becomes $\{D_1, T_1, D_2, T_2, D_3, T_3, \ldots, D_j, T_j\}$. When measuring our prediction accuracy, we use the first 5 developers in ND.

If two potential tossees T_i and T_j have the same rank, and both are active developers, due to same tossing probabilities for bug B (belonging to product P and component C), we use their profiles to further rank them. There can be two cases in this tie: (1) both T_i and T_j's profiles contain $\{P, C\}$, or (2) there is no match with either P or C. For the first case, consider the example in Table III: suppose a new bug B belongs to $\{P_1, C_1\}$. Assume T_i and T_j are the two potential tossees from developer D (where D has been predicted by the classifier) and suppose both T_i and T_j have the same tossing probabilities from D. From developer profiles, we find that T_j has fixed more bugs for $\{P_1, C_1\}$ than T_i and hence we choose T_j (case 1). If the developers have the same fix count, or neither has P and/or C in their profile (case 2), we randomly choose one.

E. Incremental Learning

After validation and measuring prediction accuracy (as described in Section II) we update the TDS for the classifier and the tossing graphs using the tossing path of the most recently validated bug. We incrementally update both the classifier and the tossing graphs as described in Section III.

V. RESULTS

A. Experimental Setup

We used Mozilla and Eclipse bugs to measure the accuracy of our proposed algorithm. We analyzed the entire life span of both applications. For Mozilla, our data set ranges from bug number 37 to 549,999 (May 1998 to March 2010). For Eclipse, we considered bugs numbers from 1 to 306,296 (October 2001 to March 2010). Mozilla and Eclipse bug reports have been found to be of high quality [6], which helps reduce noise when training the classifiers. We divided our bug data sets into 11 folds and executed 10 iterations to cover all the folds.

B. Prediction Accuracy

In Table IV we show the results for predicting potential developers for Mozilla and Eclipse using Naïve Bayes and Bayesian Networks.

Classifier: To demonstrate the advantage of our incremental approach, we measure the prediction accuracy of the classifier alone; column "ML only" contains the classifier-only average prediction accuracy rate. We found that our approach increases accuracy by 8.91% on average compared to the best previously-reported, no-incremental learning approach, due to Anvik et al. [7]. This confirms that incremental learning is indeed useful. Anvik et al. report that their initial investigation of incremental learning did not yield highly accurate predictions, though no details are provided. We have two explanations for why our findings differ from theirs. First, their experiments are based on 8,655 reports for Eclipse and 9,752 for Firefox, while we use many more (306,297 reports for Eclipse and 549,962 reports for Mozilla). Second, since anecdotal evidence [15] suggests that choosing a meaningful feature set is more important than the choice of classifiers, our additional attributes help improve prediction accuracy.

Classifier + Tossing Graph: Columns "ML+Tossing Graphs" of Table IV contain the average accurate predictions for each fold (top 2 to top 5 developers) when using both the classifier and the tossing graph; the Top 1 developer is predicted using the classifier only, hence rows 1, 6, 11, and 16 are empty for columns 5–15. Consider row 2, which contains prediction accuracy results for Top 2 in Mozilla using the Naïve Bayes classifier: column 5 (value 39.14) represents the percentage of correct predictions for *fold 1*; column 6 (value 44.59) represents the percentage of correct predictions for *folds 1* and *2*; column 15 (value 54.49) represents the average value for all iterations across all folds. Our best average accuracy is reached using Naïve Bayes (77.87% for Mozilla and 77.43% for Eclipse). The maximum observed accuracy is 83.62% for Mozilla (fold 9) and 82.59% for Eclipse (fold 10). We found that the average accuracy for the applications was higher using Naïve Bayes for Mozilla, although the prediction numbers were comparable for Eclipse using either classifier.

These findings confirm the effectiveness of fine-grained incremental learning with multi-feature tossing graphs, as prior

Program	ML algorithm (classifier)	Selection	ML Only Average	ML + Tossing Graphs (average prediction accuracy for VDS fold)										
				2	3	4	5	6	7	8	9	10	11	Average across all folds
Mozilla	Naïve Bayes	Top 1	**27.48**	-	-	-	-	-	-	-	-	-	-	-
		Top 2	**42.19**	39.14	44.59	47.72	49.39	52.57	57.36	59.46	62.37	64.99	67.23	**54.49**
		Top 3	**54.25**	51.34	62.77	66.15	57.50	63.14	61.33	64.65	77.54	71.76	74.66	**65.09**
		Top 4	**59.13**	64.20	75.86	79.57	70.66	69.11	69.84	67.68	82.87	68.77	69.71	**71.82**
		Top 5	**65.66**	74.63	77.69	81.12	79.91	76.15	72.33	75.76	83.62	78.05	79.47	**77.87**
	Bayesian Network	Top 1	**26.98**	-	-	-	-	-	-	-	-	-	-	-
		Top 2	**44.43**	36.98	38.9	37.46	40.89	43.53	48.18	51.7	54.29	57.57	60.43	**46.99**
		Top 3	**49.51**	47.19	49.45	46.42	51.42	53.82	49.59	53.63	59.26	61.91	63.9	**53.65**
		Top 4	**58.72**	54.31	57.01	54.77	59.88	61.7	63.47	62.11	67.64	68.81	66.08	**61.59**
		Top 5	**62.91**	59.22	59.44	61.02	68.29	64.87	68.3	71.9	76.38	77.06	78.91	**68.54**
Eclipse	Naïve Bayes	Top 1	**32.99**	-	-	-	-	-	-	-	-	-	-	-
		Top 2	**48.19**	39.53	38.66	36.03	39.16	39.29	41.82	43.2	47.94	51.65	54.18	**43.15**
		Top 3	**54.15**	47.95	50.84	48.46	49.52	59.45	62.77	61.73	68.19	74.95	69.07	**59.30**
		Top 4	**59.13**	56.29	61.16	59.88	60.81	69.64	69.37	75.64	75.3	78.22	77.31	**68.37**
		Top 5	**65.66**	66.73	69.92	74.13	77.03	77.9	81.8	82.05	80.63	82.59	81.44	**77.43**
	Bayesian Network	Top 1	**38.16**	-	-	-	-	-	-	-	-	-	-	-
		Top 2	**41.43**	36.11	41.49	41.13	44.81	46.34	47.4	48.61	53.84	59.18	63.69	**48.26**
		Top 3	**59.50**	51.16	52.8	54.62	57.38	56.39	63.26	66.68	70.34	76.72	77.34	**62.67**
		Top 4	**62.72**	62.92	59.03	63.09	68.27	68.33	71.79	73.37	74.15	76.94	77.04	**69.50**
		Top 5	**68.91**	74.04	72.41	70.92	71.52	73.5	75.61	79.28	79.68	80.61	81.38	**75.86**

TABLE IV
BUG ASSIGNMENT PREDICTION ACCURACY (PERCENTS).

work attained a maximum prediction accuracy of 72.92% for Mozilla and 77.14% for Eclipse [6].

C. Reduction in Tossing Lengths

We compute the original tossing path lengths for "fixed" bugs in Mozilla and Eclipse, and present it in Figure 7; we observe that most bugs have tossing length less than 13 for both applications. Note that tossing length is zero if the first assigned developer is able to resolve the bug. Ideally, a bug triage model should be able to recommend bug fixers such that tossing lengths are zero. However, this is unlikely to happen in practice due to the unique nature of bugs. Though Jeong et al. measured tossing lengths for both "assigned" and "verified" bugs, we ignore "assigned" bugs because they are still open, hence we do not know the final tossing length yet.

In Figure 8, we present the average reduced tossing lengths of the bugs for which we could correctly predict the developer. We find that the predicted tossing lengths are reduced significantly, especially for bugs which have original tossing lengths less than 13. Our approach reports reductions in tossing lengths up to 86.67% in Mozilla and 83.28% in Eclipse. For correctly predicted bugs with original tossing length less than 13, prior work [6] has reduced tossing path lengths to 2–4 tosses, while our approach reduces them to 1.5–2 tosses, hence multi-feature tossing graphs prove to be very effective.

D. Filtering Noise in Bug Reports

We found that when training sets comprise bugs with resolution "verified" or "resolved" and arbitrary status, the noise is much higher than when considering bugs with resolution "verified" or "resolved" and status "fixed". In fact, we found that, when considering arbitrary-status bugs, the accuracy is on average 23% lower than the accuracy attained when considering fixed-status bugs only. Jeong et al. considered all bugs with resolution "verified" and arbitrary-status for their training and validation purposes. They found that tossing graphs are noisy, hence they chose to prune developers with support less than 10 and edges with transaction probability less than 15%.

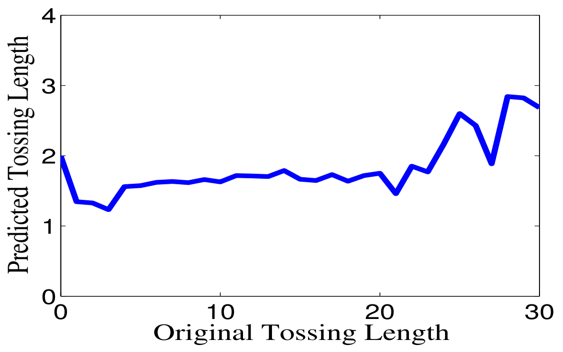
Our analysis suggests that bugs whose status changes from "new" or "open" to "fixed" are actual bugs which have been resolved, even though various other kinds of bugs, such as "invalid," "worksforme," "wontfix," "incomplete" or "duplicate" may be categorized as "verified" or "resolved". We conjecture that developers who submit patches are more competent than developers who only verify the validity of a bug and mark them as "invalid" or developers who find a temporary solution and change the bug status to "works-for-me". Anvik et al. made a similar distinction between message repliers and contributors/maintainers. They found that only a subset of those replying to bug messages are actually submitting patches and contributing to the source code, hence they only retain the contributing repliers for their TDS.

VI. THREATS TO VALIDITY

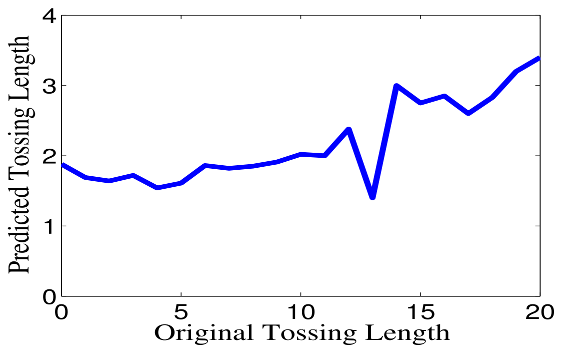
We now present possible threats to the validity of our study.

Generalization to Other Systems: The high quality of bug reports found in Mozilla and Eclipse [6] facilitates the use of classification methods. However, we cannot claim that our findings generalize to bug databases for other projects. Additionally, we have validated our approach on open source projects only, but commercial software might have different assignment policies and our approach might not be a good predictor in those cases.

Small Projects: We used two large and widely-used open source projects for our experiments, Mozilla and Eclipse. Both

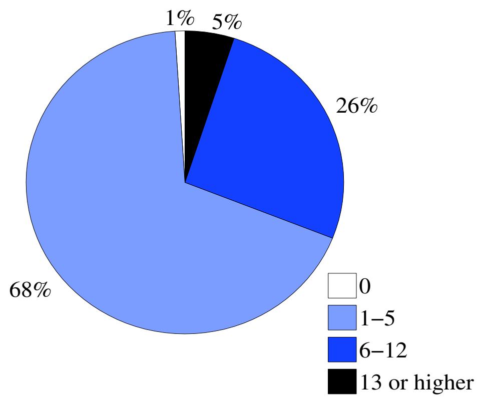


(a) Mozilla

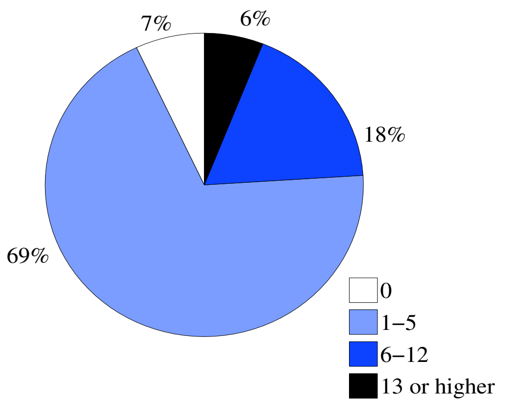


(b) Eclipse

Fig. 8. Average reduction in tossing lengths for correctly predicted bugs when using ML + Tossing Graphs (using both classifiers).



(a) Mozilla



(b) Eclipse

Fig. 7. Original tossing length distribution for “fixed” bugs.

these projects have multiple products and components, hence we could use this information as attributes for our classifier and labels in our tossing graphs. For comparatively smaller projects which do not have products or components, the lack of product-component labels on edges would reduce accuracy. Nevertheless, fine-grained incremental learning and pruning inactive developers would still be beneficial.

VII. Related Work

A. Machine Learning and Information Retrieval Techniques

Cubranic et al. [8] were the first to propose the idea of using text classification methods (similar to methods used in machine learning) to semi-automate the process of bug triaging. The authors used keywords extracted from the title and description of the bug report, as well as developer ID’s as attributes, and trained a Naïve Bayes classifier. Thus with new bug reports, the classifier suggests one or more potential developers for fixing the bug. Their method used bug reports for Eclipse from January 1, 2002 to September 1, 2002 for training, and reported a prediction accuracy of up to 30%. While we use classification as a part of our bug triage approach, in addition, we employ incremental learning and tossing graphs to reach higher accuracy. Moreover, our data sets are much larger, covering the entire lifespan of both Mozilla and Eclipse until March 2010.

Anvik et al. [7] improved the machine learning approach proposed by Cubranic et al. by using filters when collecting training data: (1) filtering out bug reports labeled invalid,” “wontfix,” or “worksforme,” (2) removing developers who no longer work on the project or do not contribute significantly, and (3) filtering developers who fixed less than 9 bugs. Our ranking function obviates the need to filter bugs. They used three classifiers, SVM, Naïve Bayes and C4.5, and reported prediction accuracy of up to 64%. They observed that SVM performs better than the other two classifiers. Since our idea is to include additional information in the training models and tossing graphs, better classifiers could increase our prediction accuracy, an exploration we leave to future work. Similar to them, we found that filtering bugs which are not “fixed” but “verified” or “resolved” leads to higher accuracy. The authors also report that their initial investigation in incremental learning did not have a favorable outcome; in Section V we explained the discrepancy between their findings and ours.

Canfora et al. used probabilistic text similarity [10] and indexing developers/modules changed due to bug fixes [16] to automate bug triaging. When using information retrieval based bug triaging, they report up to 50% top 1 recall accuracy and when indexing source file changes with developers they achieve 30%-50% top 1 recall for KDE and 10%–20% top 1 recall for Mozilla.

Podgurski et al. [17] also used machine learning techniques to classify bug reports but their study was not targeted at bug triaging; rather, their study focused on classifying and prioritizing various kinds of software faults.

Lin et al. [18] conducted machine learning-based bug triaging on a proprietary project, SoftPM. Their experiments were based on 2576 bug reports. They report 77.64% average prediction accuracy when considering module ID (the module a bug belongs to) as an attribute in the TDS; the accuracy drops to 63% when module ID is not used. Their finding is similar to our observation that using product-component information in the TDS improves prediction accuracy.

Lucca et al. [19] used information retrieval approaches to classify maintenance requests via classifiers. However, the end goal of their approach is bug classification, not bug triaging. They achieved up to 84% classification accuracy by using both split-sample and cross-sample validation techniques.

Matter et al. [20] model a developer's expertise using the vocabulary found in the developer's source code. They recommend potential developers by extracting information from new bug reports and looking it up in the vocabulary. Their approach was tested on 130,769 Eclipse bug reports and reported prediction accuracies of 33.6% for top 1 developers and 71% for top 10 developers.

B. *Incremental Learning*

Bettenburg et al. [9] demonstrate that duplicate bug reports are useful in increasing the prediction accuracy of classifiers by including them in the TDS along with the master reports of those duplicate bugs. They use folding to constantly increase the TDS, and show how this incremental approach achieves prediction accuracies of up to 56%; they do not need tossing graphs, because reducing tossing path lengths is not one of their goals. We use the same general approach for the classification part, though we improve it by using more attributes in the TDS.

C. *Tossing Graphs*

Jeong et al. [6] introduced the idea of using bug tossing graphs to predict a set of suitable developers for fixing a bug. The authors use classifiers and tossing graphs (Markov model based) to recommend potential developers. We use fine-grained, intra-fold updates and extra attributes for classification; our tossing graphs are similar to theirs, but we use additional attributes on edges and nodes. Our additions help improve accuracy and further reduce tossing lengths, as described in Sections V-B and V-C.

VIII. Conclusion

Machine learning techniques and tossing graphs has proved to be promising for automating bug triaging. We employed three novel extensions to prior triaging approaches and showed that we could achieve higher prediction accuracy in recommending potential developers and higher reductions in tossing path lengths. In particular, we show how intra-fold updates are beneficial for achieving higher prediction accuracy in bug triaging when using classifiers in isolation. We also show that developer recommendation is improved when classifying developers based on the product-component a bug belongs to, in addition to the bug types they have fixed in the past.

We validated our approach on two large, long-lived open-source projects; in the future, we plan to test how our current model generalizes to projects of different scale and lifespan. We also intend to test our approach on proprietary software. Since classifiers are often domain-based, we plan to investigate how different classifiers, and different feature sets would affect prediction accuracy.

Acknowledgments

We thank Eamonn Keogh, Dennis Jeffrey, Changhui Lin, Sudipto Das and the anonymous referees for their helpful comments on this work; David Miller from Mozilla and Denis Roy from Eclipse for the help with data collection via Bugzilla. This research was supported in part by a UC Regents' Faculty Fellowship.

References

[1] NIST, "The economic impacts of inadequate infrastructure for software testing," Planning Report, May 2002.
[2] J. Koskinen, http://users.jyu.fi/~koskinen/smcosts.htm.
[3] R. C. Seacord, D. Plakosh, and G. A. Lewis, *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley, 2003.
[4] Bugzilla User Database, http://www.bugzilla.org/installation-list/.
[5] Increase in Open Source Growth, http://software.intel.com/en-us/blogs/2009/08/04/idc-reports-an-increase-in-open-source-growth/.
[6] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *FSE*, August 2009.
[7] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *ICSE*, 2006, pp. 361–370.
[8] D. Cubranic and G. C. Murphy, "Automatic bug triage using text categorization," in *SEKE*, 2004.
[9] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful... really?" in *ICSM*, 2008.
[10] G. Canfora and L. Cerulo, "Supporting change request assignment in open source development," in *SAC*, 2006, pp. 1767–1772.
[11] C. D. Manning, P. Raghavan, and H. Schtze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.
[12] I. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*, 2nd ed. Morgan Kaufmann, 2005.
[13] R. B. Segal and J. O. Kephart, "Incremental learning in swiftfile," in *ICML*, 2000, pp. 863–870.
[14] Weka Toolkit, http://www.cs.waikato.ac.nz/ml/weka/.
[15] E. Keogh, Personal communication, March 2010.
[16] G. Canfora and L. Cerulo, "How software repositories can help in resolving a new change request," in *Workshop on Empirical Studies in Reverse Engineering*, 2005.
[17] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *ICSE*, 2003, pp. 465–475.
[18] Z. Lin, F. Shu, Y. Yang, C. Hu, and Q. Wang, "An empirical study on bug assignment automation using chinese bug data," in *ESEM*, 2009.
[19] G. A. D. Lucca, M. D. Penta, and S. Gradara, "An approach to classify software maintenance requests," in *ICSM*, 2002, pp. 93–102.
[20] D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning bug reports using a vocabulary-based expertise model of developers," *MSR*, 2009.