Automated, Highly-accurate Bug Triaging Using Machine Learning

Pamela Bhattacharya[a,*], Iulian Neamtiu[a], Christian R. Shelton[a]

[a] *Department of Computer Science and Engineering, University of California, Riverside, CA, 92521, USA.*

Abstract

Empirical studies indicate that automating the bug assignment process (also known as bug triaging) has the potential to significantly reduce software evolution effort and costs. Prior work has used machine learning techniques to automate bug triaging but has employed a narrow band of tools which can be ineffective in large, long-lived software projects. To redress this situation, in this paper we employ a comprehensive set of machine learning tools and analyzes that lead to very accurate predictions, and lay the foundation for the next generation of machine learning-based bug triaging. Our work is the first to examine the impact of multiple machine learning dimensions (classifiers, attributes, and training history) on prediction accuracy in bug triaging. We employ four classifiers and perform an ablative analysis to show the relative importance of classifiers and various software process attributes on bug triaging accuracy. We propose optimization techniques that achieve high prediction accuracy while reducing training and prediction time. We validate our approach on Mozilla and Eclipse, covering 856,259 bug reports and 21 cumulative years of development. We demonstrate that our techniques can achieve up to 86.09% prediction accuracy in bug triaging and significantly reduce tossing path lengths.

Keywords: Bug triaging, bug tossing, machine learning, empirical studies

1. Introduction

Software evolution has high associated costs and effort. A survey by the National Institute of Standards and Technology estimated that the annual cost of software bugs is about $59.5 billion [1]. Some software maintenance 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], while other estimates place maintenance costs at several times the cost of the initial software

*Corresponding author

Email addresses: `pamelab@cs.ucr.edu` (Pamela Bhattacharya), `neamtiu@cs.ucr.edu` (Iulian Neamtiu), `cshelton@cs.ucr.edu` (Christian R. Shelton)

version [4]. These surveys suggest that making the bug fixing process more efficient would reduce evolution effort and 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 [5]. 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 [6], the number of bug reports submitted daily increases, and manually recommending developers based on their expertise becomes difficult. An empirical study by Jeong et al. [7] 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 effective, automatic triaging and toss reduction techniques results in considerably high effort associated with bug resolution.

Effective and automatic bug triaging can be divided into two sub-goals: (1) assigning a bug for the first time to a developer, and (2) reassigning it to another promising developer if the first assignee is unable to resolve it, then repeating 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.

In this paper, we explore the use of machine learning toward effective and automatic bug triaging along three dimensions: the choice of classification algorithms, the software process attributes that are instrumental to constructing accurate prediction models, and the efficiency–precision trade-off. Our thorough exploration along these dimensions have lead us to develop techniques that achieve unprecedented levels of bug triaging accuracy and bug tossing reduction.

Wide range of classification algorithms. Machine learning is used for recommendation purposes in various areas such as climate prediction, stock market analysis, or prediction of gene interaction in bioinformatics [8]. Machine learning techniques, in particular classifiers,[1] have also been employed earlier for automating bug triaging. These automatic bug triaging approaches [9, 10, 11, 12] use the history of bug reports and developers who fixed them to train a classifier. 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. Prior work that has used machine learning techniques for prediction or recommendation purposes has found that prediction accuracy depends on the choice of classifier, i.e., a certain classifier outperforms other classifiers for a specific kind of a problem [8]. Previous studies [9, 10, 11, 7] only used a subset of text classifiers and did not aim at analyzing which is the best classifier for this problem. Our work is the first to examine the impact of multiple machine learning dimensions (classifiers, attributes, and training history) on prediction accuracy in bug triaging and tossing. In particular, this is the first study in the area of bug triaging to consider, and compare the performance of, a broad range of classifiers:Naïve Bayes Classifier, Bayesian Networks, C4.5 and Support Vector Machines.

Ablative analysis for ensuring effective tossing graphs. Tossing graphs have been introduced by Jeong et al. [7]; they proposed automating bug triaging by building bug tossing graphs from bug 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. Prior work [7] 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 active developer (see Section 4.6.3 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. We measure the importance of additional attributes we use in tossing graphs by performing an *ablative analysis* to determine how much each of them affects the prediction accuracy. We found that each attribute is instrumental for achieving high prediction accuracy, and overall they make pruning more efficient and improve prediction accuracy by up to 22% when compared to prediction accuracy obtained in the absence of the attributes.

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

Accurate yet efficient classification. Prior works [9, 10, 11, 12, 7] have used the entire bug history of projects for classification. On one hand, this approach helps in building accurate learning models but on the other hand, is very time consuming. We propose optimization techniques for the time consuming classification process by showing how a subset of bug reports can be used to achieve high and stable prediction accuracy. As elaborated in Section 5.6 we found that by using one third of all bug reports we could achieve prediction accuracies similar to the best results of our original experiments where we used the complete bug history Therefore, our third novel contribution in this paper is that we show how by using a subset of bug reports we can achieve accurate yet efficient bug classification that reduces the associated computational effort significantly.

Similar to prior work, we test our approach on the fixed bug data sets for Mozilla and Eclipse. Our techniques achieve a bug triaging prediction accuracy of up to 85% for Mozilla and 86% for Eclipse. We also find that using our approach reduces the length of tossing paths by up to 86% for correct predictions and improves the prediction accuracy by up to 10.78% compared to previous approaches.

Our paper is structured as follows. In Section 2 we discuss prior work and how it relates to our approach. In Section 3 we define terms and techniques used in bug triaging. In Section 4 we elaborate on our contributions, techniques and implementation details. We present our experimental setup and results in Section 5. Finally, we discuss threats to validity of our study in Section 6.

2. Related Work

2.1. Machine Learning and Information Retrieval Techniques

Cubranic et al. [10] 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. 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 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 (from May 1998 to March 2010) and Eclipse (from October 2001 to March 2010)..

Anvik et al. [9] 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. They used three classifiers, SVM, Naïve

Bayes and C4.5. They observed that SVM (Support Vector Machines) performs better than the other two classifiers and reported prediction accuracy of up to 64%. Our ranking function (as described in Section 4 obviates the need to filter bugs. Similar to Anvik et al., we found that filtering bugs which are not "fixed" but "verified" or "resolved" leads to higher accuracy. They report that their initial investigation in incremental learning did not have a favorable outcome, whereas incremental learning helps in our approach; in Section 5 we explain the discrepancy between their findings and ours.

Canfora et al. used probabilistic text similarity [12] and indexing developers/modules changed due to bug fixes [13] 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. [14] 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. [15] conducted machine learning-based bug triaging on a proprietary project, SoftPM. Their experiments were based on 2,576 bug reports. They report 77.64% average prediction accuracy when considering module ID (the module a bug belongs to) as an attribute for training the classifier; the accuracy drops to 63% when module ID is not used. Their finding is similar to our observation that using product-component information for classifier training improves prediction accuracy.

Lucca et al. [16] 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. [17] 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.

2.2. Incremental Learning

Bettenburg et al. [11] demonstrate that duplicate bug reports are useful in increasing the prediction accuracy of classifiers by including them in the training set for the classifier along with the master reports of those duplicate bugs. They use folding to constantly increase the training data set during classification, 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 training data set, we use multiple text classifiers and achieve higher prediction accuracies.

2.3. Tossing Graphs

Jeong et al. [7] 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 as explained in Section 4. The set of attributes we use help improve prediction accuracy and further reduce tossing lengths, as described in Sections 5.2 and 5.3. We also perform an ablative analysis to demonstrate the significance of our attributes in the tossing graph and tossee ranking function.

3. Preliminaries

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

3.1. Machine Learning for Bug Categorization

Classification is a supervised 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. A bug report contains a description of the bug and a list of developers that were associated with a specific bug, which makes text classification applicable to bug triaging. Machine learning techniques were used by previous bug triaging works [9, 10, 11]: 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 titles 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 [18]) to extract relevant keywords from the actual bug report; these relevant keywords constitute a subset of the attributes used to train the classifier.

3.1.1. Text Classification Algorithms

We now briefly describe each classifier we used.

Naïve Bayes Classifier. Naïve Bayes is a probabilistic technique that uses Bayes' rule of conditional probability to determine the probability that an instance belongs to a certain class. Bayes' rule states that "the probability of a class conditioned on an observation is proportional to the prior probability of the class times the probability of the observation conditioned on the class" and can be denoted as follows:

$$P(class|observation) = \frac{P(observation|class) * P(class)}{P(observation)} \tag{1}$$

For example, if the word *concurrency* occurs more frequently in the reports resolved by developer A than in the reports resolved by developer B, the classifier would predict A as a potential fixer for a new bug report containing the word *concurrency*. The algorithm is called "Naïve Bayes" as it makes the strong assumption that features are independent of the label (the developer who resolved the bug). Even though this assumption does not always hold, it turns out that in practice, Naïve Bayes-based recommendation or prediction performs well [19].

Bayesian Networks. A Bayesian Network [20] is a probabilistic model that is used to represent a set of random variables and their conditional dependencies by using a directed acyclic graph (DAG). Each node in the DAG denotes a variable, and each edge corresponds to a potential direct dependence relationship between a pair of variables. Each node is associated with a conditional probability table (CPT) which gives the probability that the corresponding variable takes on a particular value given the values of its parents.

C4.5. The C4.5 algorithm [21] builds a decision tree based on the attributes of the instances in the training set. A prediction is made by following the appropriate path through the decision tree based on the attribute values of the new instance. C4.5 builds the tree recursively in a greedy fashion. Each interior node of the tree is selected to maximize the information gain of the decision at that node as estimated by the training data. The information gain is a measure of the predictability of the target class (developer who will resolve the bug report) from the decisions made along the path from the root to this node in the tree. The sub-trees end in leaf nodes at which no further useful distinctions can be made and thus a particular class is chosen.

Support Vector Machines. An SVM (Support Vector Machine [22]) is a supervised classification algorithm that finds a decision surface that maximally separates the classes of interest. That is, the closest points to the surface on each side are as far as possible from the decision surface. It employs kernels to represent non-linear mappings of the original input vectors. This allows it to build highly non-linear decision surfaces without an explicit representation of the non-linear mappings. Four kinds of kernel functions are commonly used: Linear, Polynomial, Gaussian Radial Basis Function (RBF) and Sigmoid. In our study we use Polynomial and RBF functions as they have been found to be most effective in text classification.

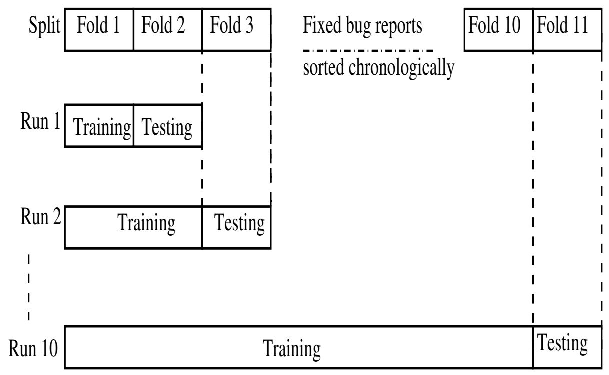


Figure 1: Folding techniques for classification as used by Bettenburg et al.

3.2. Folding

Early bug triaging approaches [7, 9, 10] divided the data set into two subsets: 80% for TDS and 20% for VDS. Bettenburg et al. [11] have used folding (similar to split-sample validation techniques from machine learning [8]) 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 the second run, 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 [8], similar to Bettenburg et al., we chose $n = 11$ and carried out 10 iterations of the validation process using incremental learning.

Tossing paths
$A \to B \to C \to D$
$A \to E \to D \to C$
$A \to B \to E \to D$
$C \to E \to A \to D$
$B \to E \to D \to F$

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 1: Tossing paths and probabilities as used by Jeong et al.

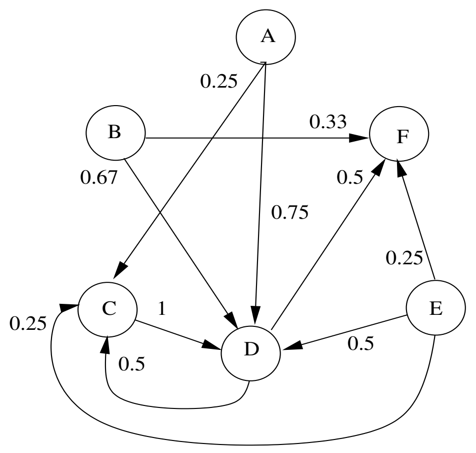


Figure 2: Tossing graph built using tossing paths in Table 1.

3.3. Goal-oriented Tossing Graphs

When a bug is assigned to a developer for the first time and she 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 eventually able to fix it. Based on these tossing paths, *goal-oriented tossing graphs* were proposed by Jeong et al .[7]; 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:

$$Pr(D \hookrightarrow D_j) = \frac{\sum_1^m D \hookrightarrow D_j : D_j \text{ fixed the bug}}{\sum_{i=1}^n D \hookrightarrow D_i} \tag{2}$$

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 1 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. It is common in open source projects that when a bug in a module is first reported, the developers associated with that module are included in the list of assignees by default. The purpose of our automatic bug triaging tool is, given a bug report, to predict developers who could be potential fixers and email them, so that human intervention is reduced as much as possible.

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%.

4. Methodology

4.1. Choosing Effective Classifiers and Features

In this section we discuss appropriate selection of machine learning algorithms and feature vectors for improving the classification process.

4.1.1. Choosing the Right Classifier

Various problems that use machine learning techniques for prediction or recommendation purposes have found that for a specific kind of a problem, one classifier outperforms other

Prior work has found that prediction accuracy depends on the choice of classifier, i.e., for a specific kind of a problem, a certain classifier outperforms other classifiers [8]. Previous bug classification and triaging studies [9, 10, 11, 7] only used a subset of text classifiers and did not aim at analyzing which is the best classifier for this problem. Our work is the first study to consider an extensive set of classifiers which are commonly used for text classification: Naïve Bayes Classifier, Bayesian Networks, C4.5 and two types of SVM classifiers (Polynomial and RBF). We found that for bug triaging it is not possible to select one classifier which is better than the rest, either for a specific project or for any project in general. Since classifier performance is also heavily dependent on the quality of bug reports, in general we could not propose choosing a specific classifier *a priori* for a given project, and found that classifier preference might change as the project evolves. Interestingly, computationally-intensive classification algorithms such as C4.5 and SVM do not consistently outperform simpler algorithms such as Naïve Bayes and Bayesian Networks. We provide details of our prediction accuracy using all five classifiers in Section 5.2.

4.1.2. Feature Selection

Classifier performance is heavily dependent on feature selection [8]. Prior work [9, 10, 11] has used keywords from the bug report and developer name or ID as features (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 [18]. We use the Weka toolkit [23] to remove stop words and form the word vectors for the dictionary (via the `StringtoWordVector` class with tf-idf enabled).

4.2. Incremental Learning

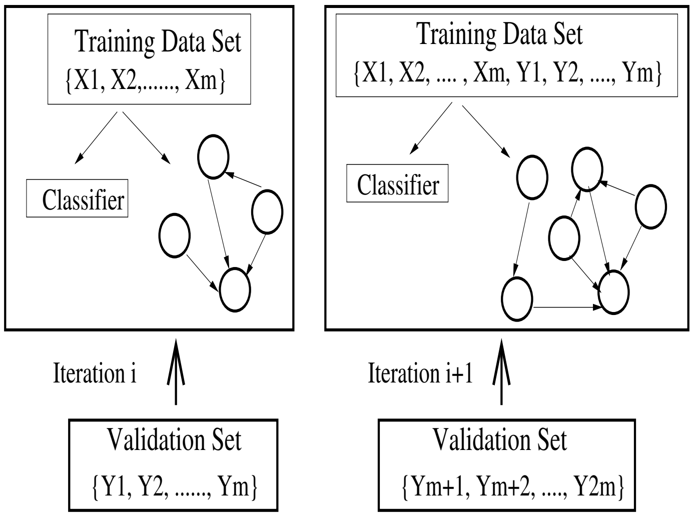
Prior work [7, 11] has used *inter*-fold updates, i.e., the classifier and tossing graphs are updated after each fold validation, as shown in Figure 3(a). With inter-fold updates, after validating the VDS from fold n, the VDS is added to the TDS for validating fold $n+1$. However, consider the example 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 9(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, 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. [24].

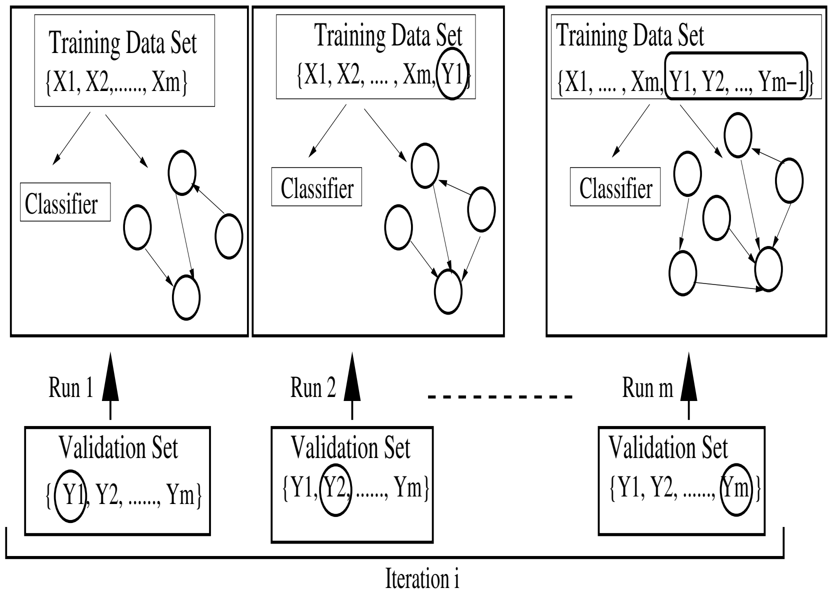
4.3. Multi-featured Tossing Graphs

Tossing graphs are built using tossing probabilities derived by analyzing bug tossing histories, as explained in Section 3.3. Jeong et al. [7] determined potential tossees as follows: if developer A has tossed more bugs to developer B than to developer D, in the future, when A cannot resolve a bug, the bug will be tossed to B, i.e., 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 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-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



(a) Updates after each validation set (Bettenburg et al.)



(b) Updates after each bug (our approach)

Figure 3: Comparison of training and validation techniques.

the project. Anvik et al. [9] and Jeong et al. [7] have pointed out this problem and proposed solutions. Anvik et al. use a heuristic to filter out developers who have contributed to fewer than 9 bug resolutions in the last 3 months of the project. Jeong et al. assume that, when within a short time span many

Product	**Component**	**Tossing paths**
P_1	C_1	$A \rightarrow B \rightarrow C$
P_1	C_3	$F \rightarrow A \rightarrow B \rightarrow E$
P_2	C_4	$B \rightarrow A \rightarrow D \rightarrow C$
P_1	C_3	$C \rightarrow E \rightarrow A \rightarrow D$
P_1	C_1	$A \rightarrow B \rightarrow E \rightarrow C$
P_1	C_3	$B \rightarrow A \rightarrow F \rightarrow D$

Developer bug assigned	**Total tosses**	**Developers who fixed the bug**					
		C		D		E	
		#	Pr	#	Pr	#	Pr
A	6	3	0.5	2	0.33	1	0.17

Developer	**Last Activity** (in days)
A	20
C	70
D	50
E	450

Table 2: Example of tossing paths, associated tossing probabilities and developer activity.

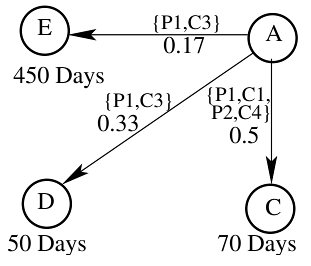


Figure 4: Multi-feature tossing graph (partial) derived from data in Table 2.

bugs get tossed from a developer D to others, leading to an increase in the number of outgoing edges in the tossing graph from D's node, D is a potentially retired developer. They suggest that this information can be used in real-world scenarios by managers to identify potentially inactive developers. Therefore, in their automatic bug triaging approach they still permit assignment of bugs to inactive developers, which increases the length of the predicted tossing paths. In contrast, we restrict potential assignees to active developers only, and do so with a minimum number of tosses.

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)[2] to that edge; moreover, for each developer

[2] *Products* are smaller projects within a large project. *Components* are sub-modules in

in the tossing graph, we maintain an activity count, i.e., the difference between the date of the bug being validated and the date of the last activity of that developer.

4.3.1. Building Multi-feature Tossing Graphs

As discussed earlier in Section 4.3, 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 expertise 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 paths, the product–component the bug belongs to, and the developer activity, as shown in Table 2. Suppose that at some point in our recommendation process for a specific bug, the classifier returns A as the best developer for fixing the bug. However, if A is unable to resolve it, we need to use the tossing graph to find the next developer. We will present three examples to illustrate which neighbor of A to choose, and how the selection depends on factors like bug source and developer activity, in addition to tossing probability. For the purpose of these examples, we just show a part of the tossing graph built from the tossing paths shown in Table 2; we show the node for developer A and its neighbors in the tossing graph in Figure 4, as the tossee selection is dependent on these nodes alone.

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 to C.

Example II. Consider a bug B_2 belonging to product P_1 and component C_3. If A is unable to fix it, although C has a higher transaction probability than D, 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 D.

Example III. Based on the last active count for E in Figure 4, i.e., 450 days, it is likely that E 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

a product. For example, Firefox is a product in Mozilla and Bookmarks is a component of Firefox.

[3]Choosing 100 days as the threshold was based on Anvik et al. [9]'s observation that

(tossee) from the reference node A. For example, consider bug B_3 which belongs to product P_1 and component C_3, which has been assigned to A and we need to find a potential tossee when A is unable to resolve it. We should never choose E as a tossee as he is a potential retired developer and hence, 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.

4.3.2. Ranking Function

As explained with examples in Section 4.3.1, the selection of a tossee depends on multiple factors. We thus use a ranking function to rank the tossees and recommend potential bug-fixer. We first show an example of our developer prediction technique for a real bug from Mozilla and then present the ranking function we use for 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. If we use the tossing graphs in addition to the classifier, we select the most likely tossee for *bugzilla*, the first developer in the classifier ranked list. In Figure 5, 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. Therefore, 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.

We now describe our algorithm for ranking developers. Similar to Jeong et al., we first use the classifier to predict a set of developers named CP (Classifier 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 4.6.1).

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:

developers that have been inactive for three months or more are potentially retired.

[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.

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

The tossing probability, $Pr(D_i \hookrightarrow T_k)$, is computed using equation 2 (Section 3). 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, named 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, and both have the same tossing probabilities for bug B (belonging to product P and component C), we use developer 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 3: 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, 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.

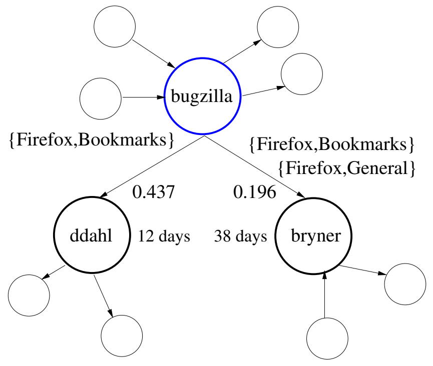


Figure 5: Actual multi-feature tossing graph extracted from Mozilla.

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 3: Sample developer profiles: developer IDs and number of bugs they fixed in each product–component pair.

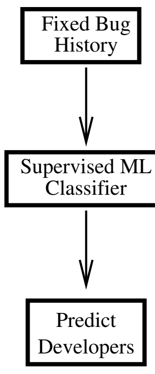
4.4. Ablative Analysis

As explained in Section 4.6.4, our ranking function for tossing graphs contains additional attributes compared to the original tossing graphs by Jeong et al. Therefore, we were interested in evaluating the importance of each attribute; to measure this, we performed an ablative analysis. We choose only two attributes out of three (product, component and developer activity) at a time and compute the decrease in prediction accuracy in the absence of the other attribute. For example, if we want to measure the significance of the "developer activity" attribute, we use only product and component attributes in our ranking function described in Section 4.6.4 and compute the decrease in prediction accuracy. In Section 5.5 we discuss the results of our ablative analysis and argue the importance of the attributes we propose.

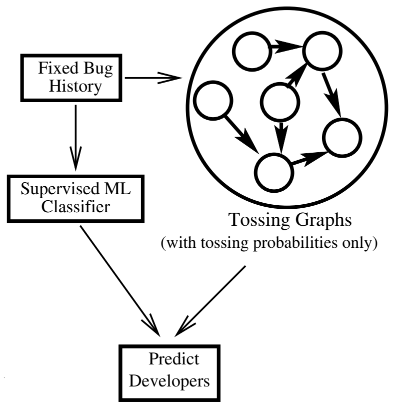
4.5. Accurate Yet Efficient Classification

One of the primary disadvantages of fine-grained incremental learning is that it is time consuming. Previous studies which used fine-grained incremental learning for other purposes [25] found that using *a part* of the bug repository history for classification might yield comparable and stable results to using the *entire* bug history. Similarly, we intended to find how many past bug reports we need to train the classifier on in order to achieve a prediction accuracy comparable to the highest prediction accuracy attained when using fold 1–10 as the TDS and fold 11 as the VDS.

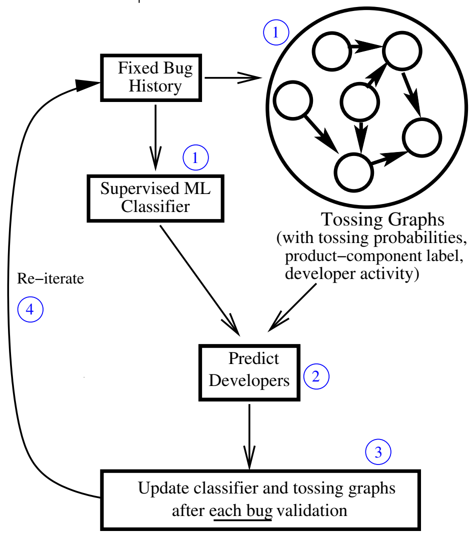
We now present the procedure we used for finding how much history is enough to yield high accuracy. We first built the tossing graphs using the TDS until fold 10; building tossing graphs and using them to rank developers is not a time consuming task, hence in our approach tossing graphs cover the entire TDS. We then incrementally started using sets of 5000 bug reports from fold 10 downwards, in descending chronological order, as our TDS for the classifier, and measured our prediction accuracy for bugs in fold 11 (VDS); we continued this process until addition of bug reports did not improve the prediction accuracy any more, implying stabilization. Note that by this method our VDS remains constant. We present the results of our optimization in Section 5.6.



(a) Classifier-based bug triaging



(b) Classifiers coupled with tossing graphs


1
Fixed Bug History
1
Supervised ML Classifier
Tossing Graphs
(with tossing probabilities, product−component label, developer activity)
Re−iterate
4
Predict Developers
2
3
Update classifier and tossing graphs after each bug validation

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

Figure 6: Comparison of bug triaging techniques.

4.6. Implementation

In Figure 6 we compare our approach to previous techniques. Initial work in this area (Figure 6(a)) used classifiers only [9, 10, 11, 12]; more recent work by Jeong et al. [7] (Figure 6(b)) coupled classifiers with tossing graphs. Our approach (Figure 6(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.

4.6.1. Developer Profiles

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

Table 4: Sample developer profile.

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 4, 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.

4.6.2. 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. [9] 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 [9, 10, 11] has used keywords from the bug report and developer name or 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 [18].

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.

4.6.3. Multi-feature Tossing Graphs

With the training data and classifier at hand, we proceed to constructing tossing graphs as explained in Section 4.3.1. We use the same bug reports used for classification to build the 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. [7] 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 4.6.4) obviate the need for pruning.

4.6.4. 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. Thus our final list of predicted developers contains five developer id's in both methods (classifier alone and classifier + tossing graph).

4.6.5. Folding

After predicting developers, similar to the Bettenburg et al.'s folding technique [11], we iterate the training and validation for all 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.

5. Results

5.1. 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 [7], 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.

5.2. Prediction Accuracy

In Tables 5 and 6 we show the results for predicting potential developers for Mozilla and Eclipse using five classifiers: Naïve Bayes, Bayesian Networks, C4.5, and SVM using Polynomial and RBF kernel functions. For our experiments, we used the classifier implementations in Weka [23] and WLSVM (for SVM) [26]. [5]

Classifier alone. To demonstrate the advantage of our fine-grained, incremental learning 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, by Anvik et al. [9]. This confirms that incremental learning is instrumental for achieving a high prediction accuracy. 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 [27] suggests that choosing a meaningful feature set is more important than the choice of classifiers, our additional attributes help improve prediction accuracy.

Classifier + tossing graphs. Columns "ML+Tossing Graphs" of Tables 5 6 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. Column 16 represents the percentage improvement of prediction accuracy obtained

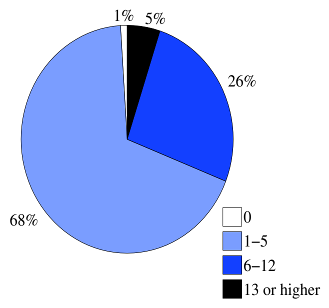
[5]The details of the parameters used for the classifiers in the experiments can be found at: `http://www.cs.ucr.edu/~pamelab/param.pdf`

ML algorithm (classifier)	**Selection**	**ML only (avg)**	**ML + Tossing Graphs (average prediction accuracy for VDS fold)**										**Avg. across all folds**	**Improv. vs. prior work**
			2	3	4	5	6	7	8	9	10	11		
Naïve Bayes	Top 1	**27.67**	13.33	20.67	22.25	25.39	24.58	30.09	30.05	33.61	35.83	40.97	**27.67**	-
	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**	8.16
	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**	11.02
	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**	6.93
	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**	9.22
Bayesian Network	Top 1	**26.71**	13.54	14.16	20.21	22.05	25.16	28.47	32.37	35.1	37.11	38.94	**26.71**	-
	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**	7.24
	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**	2.27
	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**	8.07
	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**	10.78
C4.5	Top 1	**25.46**	10.8	14.2	18.3	26.21	24.85	28.77	30.7	32.29	33.64	34.87	**25.46**	N/A
	Top 2	**31.03**	29.17	34.16	40.34	45.92	51.67	56.35	59.41	62.04	65.26	69.49	**51.38**	
	Top 3	**38.97**	33.2	38.39	43.37	51.05	56.47	62.68	66.44	69.92	73.41	75.62	**57.05**	
	Top 4	**46.43**	41.16	46.15	51.05	59.16	64.56	69.43	73.4	76.31	80.52	83.84	**64.72**	
	Top 5	**59.18**	47.04	50.49	56.67	64.25	69.07	74.68	78.74	80.37	81.59	84.82	**68.77**	
SVM (Polynomial Kernel Function, Degree=2)	Top 1	**23.82**	8.26	13.01	18.54	20.69	22.97	27.14	29.46	32.36	32.69	33.08	**23.82**	N/A
	Top 2	**28.66**	18.94	21.49	26.63	31.29	31.81	37.24	36.87	40.24	43.47	48.06	**33.6**	
	Top 3	**34.04**	23.85	23.11	27.44	32.46	39.11	32.52	41.92	44.62	45.37	48.85	**35.93**	
	Top 4	**43.92**	26.74	30.78	35.99	28.82	34.77	40.05	46.89	53.47	59.03	63.7	**42.02**	
	Top 5	**51.17**	34.83	32.85	41.14	44.4	46.94	53.76	60.3	62.69	61.01	70.95	**50.89**	
SVM (RBF Kernel Function)	Top 1	**30.98**	17.37	20.27	28.56	30.46	31.98	34.86	31.56	38.69	33.09	42.97	**30.98**	N/A
	Top 2	**39.27**	41.51	42.4	49.1	53.02	52.04	59.33	59.24	62.13	66.1	68.29	**55.32**	
	Top 3	**45.52**	43.7	44.26	49.6	53.96	61.69	54.79	62.79	66.82	67.25	69.75	**57.38**	
	Top 4	**53.42**	48.21	51.51	57.15	51.62	56.95	61.08	67.63	74.23	80.59	84.12	**63.31**	
	Top 5	**62.49**	56.07	53.99	62.2	66.13	68.54	76.23	80.74	84.69	83.04	82.71	**71.43**	

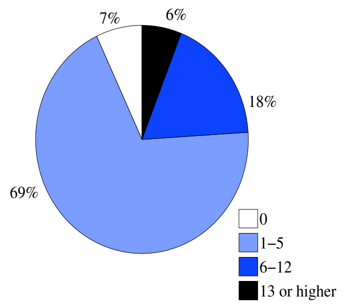
Table 5: Bug assignment prediction accuracy (percents) for Mozilla.

ML algorithm (classifier)	**Selection**	**ML only (avg)**	**ML + Tossing Graphs (average prediction accuracy for VDS fold)** 2	3	4	5	6	7	8	9	10	11	**Avg. across all folds**	**Improv. vs. prior work**
Naïve Bayes	Top 1	**32.35**	12.2	21.09	24.7	23.43	25.17	33.04	38.73	42.03	49.59	53.69	**32.35**	-
	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**	5.99
	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**	2.76
	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**	6.69
	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**	5.98
Bayesian Network	Top 1	**38.03**	24.36	29.53	31.04	36.37	34.09	40.97	40.22	43.99	48.88	50.85	**38.03**	-
	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**	3.97
	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**	8.88
	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**	5.58
	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**	6.93
C4.5	Top 1	**28.97**	11.43	21.35	24.88	28.33	25.12	30.56	31.57	35.19	38.37	42.97	**28.97**	N/A
	Top 2	**36.33**	31.07	37.65	42.24	48.23	51.75	55.54	58.13	59.44	62.61	62.98	**50.96**	
	Top 3	**48.17**	37.95	44.47	48.29	55.82	58.45	62.73	65.28	66.32	69.34	69.57	**57.82**	
	Top 4	**54.62**	44.62	51.11	55.36	61.47	65.62	69.3	71.06	72.39	75.23	76.44	**64.26**	
	Top 5	**65.98**	51.27	57.15	62.44	68.52	71.77	75.95	78.51	79.64	82.36	86.09	**71.37**	
SVM (Polynomial Kernel Function, Degree=2)	Top 1	**22.45**	9.43	13.3	15.59	20.12	24.6	24.65	26.46	30.12	31.71	29.93	**22.45**	N/A
	Top 2	**26.52**	19.51	21.4	27.1	32.02	31.04	37.33	37.24	40.13	44.1	47.29	**33.72**	
	Top 3	**30.08**	21.7	23.26	27.6	32.96	39.69	32.79	41.79	48.11	45.25	50.75	**35.88**	
	Top 4	**33.17**	26.21	30.51	35.15	29.62	34.95	40.08	46.63	53.23	59.59	63.12	**41.91**	
	Top 5	**42.92**	35.07	32.99	41.2	45.13	46.54	54.23	59.74	62.69	61.04	71.71	**51.03**	
SVM (RBF Kernel Function)	Top 1	**29.23**	16.54	22.06	22.29	28.29	26.58	31.86	31.48	33.84	36.01	43.18	**29.23**	N/A
	Top 2	**37.5**	38.4	42.85	43.39	44.41	47.2	46.98	48.29	49.49	48.8	46.68	**45.65**	
	Top 3	**47.04**	46.65	54.17	51.1	55.66	61.41	62.66	66.63	72.46	68.87	72.93	**61.25**	
	Top 4	**53.46**	48.64	49.76	55.96	54.18	59.76	64.61	70.32	74.43	74.83	78.67	**63.12**	
	Top 5	**64.77**	63.5	63.68	59.9	69.52	71.98	75.97	78.24	83.33	80.38	82.02	**72.85**	

Table 6: Bug assignment prediction accuracy (percents) for Eclipse.



(a) Mozilla



(b) Eclipse

Figure 7: Original tossing length distribution for "fixed" bugs.

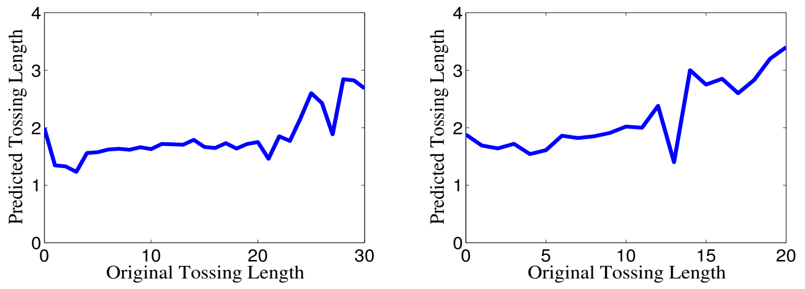
by our technique when compared to using tossing graphs with tossing probabilities only. Our best average accuracy is reached using C4.5 (84.82% for Mozilla and 86.09% for Eclipse). We found that this prediction accuracy is higher than the prediction accuracy we obtained in our earlier work [28] where we used Naïve Bayes and Bayesian Networks only. When compared to prior work [7] (where Naïve Bayes and Bayesian Networks were used as ML algorithms and tossing probabilities alone were used in the tossing graphs) our technique improved prediction accuracy by up to 11.02%.

5.3. Tossing Length Reduction

We compute the original tossing path lengths for "fixed" bugs in Mozilla and Eclipse, and present them 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 triaging 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 by up to 86.67% in Mozilla and 83.28% in Eclipse. For correctly-predicted bugs with original tossing length less than 13, prior work [7] has reduced tossing path lengths to 2–4 tosses, while our approach reduces them to an average of 1.5 tosses for Mozilla and 1.8 tosses for Eclipse, hence multi-feature tossing graphs prove to be very effective.

5.4. 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



(a) Mozilla (b) Eclipse

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

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,” “works-for-me,” “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.

5.5. Ablative Analysis

Since our ranking function for tossing graphs contains additional attributes compared to the original tossing graphs by Jeong et al., we were interested to evaluate the importance of each attribute using ablative analysis as described in Section 4.4. In Table 7 we show the maximum percentage reduction in prediction accuracy when one of the attributes is removed from the ranking function. The decrease in prediction accuracy shows that the removal of product attribute affects the prediction accuracy the most, followed by the developer activity label and component label respectively. These accuracy reductions underline the importance of using all attributes in the ranking function, and more generally, the advantage of the richer feature vectors our approach relies on.

Attribute removed from ranking function	**Maximum reduction in prediction accuracy (%)**	
	Mozilla	**Eclipse**
Product	16.44	22.07
Component	8.11	10.93
Developer activity	19.83	12.62

Table 7: Impact of individual ranking function attributes on prediction accuracy.

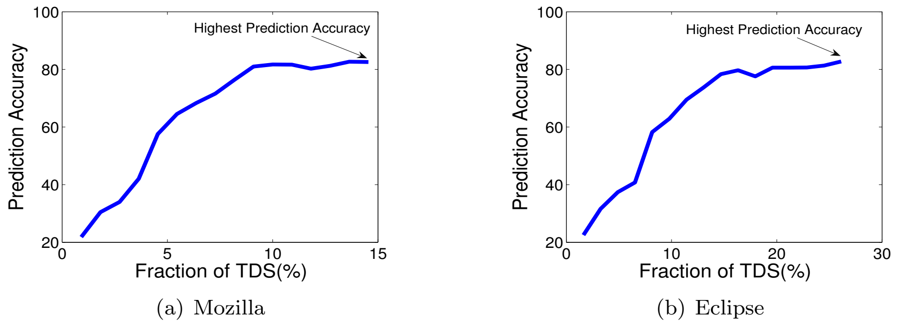


Figure 9: Change in prediction accuracy when using subsets of bug reports using Naïve Bayes classifier.

5.6. Accurate Yet Efficient Classification

One of the primary disadvantages of fine-grained incremental learning is that it is very time consuming. As described in Section 4.5, we performed a study to find how many past bug reports we need to train the classifier to achieve approximately similar prediction accuracy when compared to the highest prediction accuracy attained when using folds 1–10 as the TDS and fold 11 as the VDS. We used the Naïve Bayes classifier as our ML algorithm in this case. We present our results in Figure 9. We found that Firefox required approximately 14% and Eclipse required about 26% of all bug reports (in descending chronological order) to achieve prediction accuracies greater than 80%, i.e., similar to the best results of our original experiments where we used the complete bug history to train our classifier. Therefore, a practical way to reduce the computational effort associated with learning, yet maintain high prediction accuracy, is to prune the bug report set and only use a recent subset (e.g., the most recent 14% to 26% of bug reports).

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 because very few bugs get fixed the day they are reported. Before we use the algorithm to predict developers, we will train it with all fixed bug reports in the history. Whenever a new bug gets fixed, the TDS needs to be updated and we need to re-train the classifier. However, while about one hundreds of bugs are reported every day for large projects like Mozilla and Eclipse, less than 1% get fixed every day [7]. Since we use fixed bug reports only, if we update the TDS overnight with the new fixed bug reports and retrain the classifier, we can still achieve high prediction accuracies.

6. 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 [7] 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 we might require considering different attribute sets.

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

7. Conclusions

Machine learning and tossing graphs have proved to be promising for automating bug triaging. In this paper we lay the foundation for future work that uses machine learning techniques to improve automatic bug triaging by examining the impact of multiple machine learning dimensions on triaging accuracy.

We used a broad range of text classifiers and found that unlike many problems which use specific machine learning algorithms, we could not select a specific classifier for the bug triaging problem. We show that, for bug triaging, computationally-intensive classification algorithms such as C4.5 and SVM do not always perform better than their simple counterparts such as Naïve Bayes and Bayesian Networks. We performed an ablative analysis to measure the relative importance of various software process attributes in prediction accuracy. Our study indicates that to avoid the time-consuming classification process we can use a subset of the bug reports from the bug databases and yet achieve stable-high prediction accuracy.

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.

References

[1] NIST, The economic impacts of inadequate infrastructure for software testing, Planning Report, 2002.

[2] J. Koskinen, `http://users.jyu.fi/~koskinen/smcosts.htm`, 2003.

[3] R. C. Seacord, D. Plakosh, G. A. Lewis, Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices, Addison-Wesley, ISBN 0321118847, 2003.

[4] I. Sommerville, Software Engineering (7th Edition), Pearson Addison Wesley, ISBN 0321210263, 2004.

[5] Bugzilla User Database, `http://www.bugzilla.org/installation-list/`, 2010.

[6] Increase in Open Source Growth, `http://software.intel.com/en-us/blogs/2009/08/04/idc-reports-an-increase-in-open-source-growth/`, 2009.

[7] G. Jeong, S. Kim, T. Zimmermann, Improving Bug Triage with Bug Tossing Graphs, in: FSE, 2009.

[8] I. Witten, E. Frank, Data Mining: Practical Machine Learning Tools and Techniques, Morgan Kaufmann, second edn., 2005.

[9] J. Anvik, L. Hiew, G. C. Murphy, Who should fix this bug?, in: ICSE, 361–370, 2006.

[10] D. Cubranic, G. C. Murphy, Automatic bug triage using text categorization, in: SEKE, 2004.

[11] N. Bettenburg, R. Premraj, T. Zimmermann, S. Kim, Duplicate Bug Reports Considered Harmful... Really?, in: ICSM, 2008.

[12] G. Canfora, L. Cerulo, Supporting change request assignment in open source development, in: SAC, 1767–1772, 2006.

[13] G. Canfora, L. Cerulo, How software repositories can help in resolving a new change request, in: Workshop on Empirical Studies in Reverse Engineering, 2005.

[14] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, B. Wang, Automated support for classifying software failure reports, in: ICSE, 465–475, 2003.

[15] Z. Lin, F. Shu, Y. Yang, C. Hu, Q. Wang, An empirical study on bug assignment automation using Chinese bug data, in: ESEM, 2009.

[16] G. A. D. Lucca, M. D. Penta, S. Gradara, An Approach to Classify Software Maintenance Requests, in: ICSM, 93–102, 2002.

[17] D. Matter, A. Kuhn, O. Nierstrasz, Assigning bug reports using a vocabulary-based expertise model of developers, MSR .

[18] C. D. Manning, P. Raghavan, H. Schtze, Introduction to Information Retrieval, Cambridge University Press, 2008.

[19] P. Domingos, M. Pazzani, Beyond Independence: Conditions for the Optimality of the Simple Bayesian Classifier, in: Machine Learning, Morgan Kaufmann, 105–112, 1996.

[20] D. Koller, N. Friedman, Probabilistic Graphical Models: Principles and Techniques, The MIT Press, 2009.

[21] J. R. Quinlan, C4.5: programs for machine learning, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, ISBN 1-55860-238-0, 1993.

[22] B. E. Boser, I. M. Guyon, V. N. Vapnik, A Training Algorithm for Optimal Margin Classifiers, in: Proceedings of the Fifth Annual Workshop on Computational Learning Theory, 144–152, 1992.

[23] Weka Toolkit 3.6, `http://www.cs.waikato.ac.nz/ml/weka/`, 2010.

[24] R. B. Segal, J. O. Kephart, Incremental Learning in SwiftFile, in: ICML, 863–870, 2000.

[25] A. Lamkanfi, S. Demeyer, E. Giger, B. Goethals, Predicting the severity of a reported bug, in: MSR, 1–10, 2010.

[26] Y. EL-Manzalawy, V. Honavar, WLSVM: Integrating LibSVM into Weka Environment, Software available at `http://www.cs.iastate.edu/~yasser/wlsvm`, 2005.

[27] E. Keogh, Personal communication, 2010.

[28] P. Bhattacharya, I. Neamtiu, Fine-grained Incremental Learning and Multi-feature Tossing Graphs to Improve Bug Triaging, in: IEEE Conference on Software Maintenance, 2010.