When Would This Bug Get Reported?

Ferdian Thung[1], David Lo[1], Lingxiao Jiang[1], Lucia[1], Foyzur Rahman[2], and Premkumar T. Devanbu[2]
[1]*Singapore Management University, Singapore*
[2]*University of California, Davis, USA*
{*ferdianthung,davidlo,lxjiang,lucia.2009*}*@smu.edu.sg,* {*mfrahman,devanbu*}*@cs.ucdavis.edu*

***Abstract*—Not all bugs in software would be experienced and reported by end users right away: Some bugs manifest themselves quickly and may be reported by users a few days after they get into the code base; others manifest many months or even years later, and may only be experienced and reported by a small number of users. We refer to the period of time between the time when a bug is introduced into code and the time when it is reported by a user as *bug reporting latency*. Knowledge of bug reporting latencies has an implication on prioritization of bug fixing activities—bugs with low reporting latencies may be fixed earlier than those with high latencies to shift debugging resources towards bugs highly concerning users. To investigate bug reporting latencies, we analyze bugs from three Java software systems: AspectJ, Rhino, and Lucene. We extract bug reporting data from their version control repositories and bug tracking systems, identify bug locations based on bug fixes, and back-trace bug introducing time based on change histories of the buggy code. Also, we remove non-essential changes, and most importantly, recover root causes of bugs from their treatments/fixes. We then calculate the bug reporting latencies, and find that bugs have diverse reporting latencies. Based on the calculated reporting latencies and features we extract from bugs, we build classification models that can predict whether a bug would be reported early (within 30 days) or later, which may be helpful for prioritizing bug fixing activities. Our evaluation on the three software systems shows that our bug reporting latency prediction models could achieve an AUC (Area Under the Receiving Operating Characteristics Curve) of 70.869%.**

I. Introduction

Bugs are present in most software systems. Many bugs can be uncovered when developers perform testing activities or apply bug finding tools. However, due to time pressure and budget limitations, not all such detected bugs can be fixed before they get shipped to end users. Bug fixes may also require additional regression tests to make sure the fixes do not break other program behaviors that previously work well. Thus, developers need to make an informed decision on what bugs to fix and what not to fix prior to release. If bugs are shipped to users, patches could be released later or never, depending on the effects of the bugs on users.

Not all bugs have equal effects. Some bugs only affect minor functionalities which might not be used by any user. Others affect core functionalities that are used by many. The former kind of bugs are seldom noticed early by end users, possibly only by expert users who have used the program for a long period of time. The latter kind of bugs are often noticed soon after release and may be reported early. Thus, ignoring bug reports that may not need to be fixed would also help to reduce the developer efforts and improve the utility of software testing and bug finding tools.

In this paper, we are interested in understanding the time between the introduction of a bug into the code base of a software system and the reporting of the bug by users after they start to use the program. We refer to this time period as the *bug reporting latency*. In effect, we want to provide answers to the following research question:

When would a bug get reported?

This question has implications on what bugs should be fixed first within the time budget before release, as bugs that may be reported early should be fixed first so that developers may focus on code cared more by users. Prediction models that can provide qualitative measures about bug reporting latencies could thus be useful for prioritizing bug fixing activities. The following scenarios illustrate the benefits of understanding bug reporting latencies.

Scenario 1: Ferdian noticed a bug in his code before release. He started to fix it right away. However, the fix took more time and effort than he thought, and he was not sure whether the fix would introduce other bugs. So he carried out additional regression tests and was happy to find no other bugs. Then, he went ahead to release his code, but the release was one month later than previously planned.

Scenario 2: Ferdian noticed a bug in his code before release. He would like to fix it right away, but he was aware of two complicating issues: a) fixing the bug takes time and effort, and might introduce other bugs; b) this bug may not trigger any failure for a long time in the field. So he started to estimate how likely a user would actually experience and report the bug, and he found that similar bugs in the bug databases may not be noticed by any user in three months after release. Based on the estimation, Ferdian chose to release his code as planned and delay the fix of this bug until the time when the next field upgrade would be issued. Thus, he kept the promise to release his code at certain time and had additional time to fix the bug and test the change.

Scenario 3: Ferdian noticed a bug in his code before release. He would like to fix it right away, but he was aware of the potential issues as those in Scenario 2. So he started

978-1-4673-2312-3/12/$31.00

to estimate how likely a user would actually experience and report the bug, and he found that similar bugs in the bug databases may be noticed by a user in one month after release. Thus, Ferdian chose to fix the bug right away and apply additional tests for the change. As a result, the effect of this fix was the same as Scenario 1 where the bug was fixed but the release of his code was delayed by one month.

There are studies aimed to predict the severity and priority of bugs from corresponding bug reports [32], [33], [40]. Our work is different from these studies in several aspects. First, we prioritize bug fixing activities prior to release, while they prioritize bug fixing activities after bugs are reported by end users. Second, we predict bug reporting latencies which are not provided by studies on bug severity and priority prediction. Third, we investigate the code responsible for the bug, while they investigate bug reports which may involve various kinds of information, such as text description, call stacks, core dumps, etc. There are also studies that can prioritize bug reports before releases and help to improve the accuracy of bug finding tools, such as [3], [28], [41], [48], they mostly aim to differentiate true bug reports from false positives bug reports and often make decisions based on various code features or bug histories, while we prioritize true bug reports only and make our decisions based on bug reporting latencies. To the best of our knowledge, our work on understanding and predicting bug reporting latencies is new and orthogonal to previous work.

In this paper, we study reporting latencies for 190 bugs from three Java software systems: AspectJ, Rhino, and Lucene. We collect the bug and bug fix data for these projects from iBugs [1] and JIRA [2] respectively. To find the buggy code, we compare the corresponding bug tracking system and source control system of each of the three software systems. We find the reporting time for a particular bug from the bug tracking system, and also find code commits that fix the bug in the source control system. By performing a *diff* between these commits and the ones before it, we could find the *treatments* of the bugs. We then perform a manual analysis to recover their root causes (i.e., the faulty program elements responsible for the bugs) from the *treatments* and the introduction time of the faulty program elements. We find that the bug reporting latencies for various bugs in various systems are very diverse, ranging from one day to more than three years. Also, bug severities seemingly have no effect on bug reporting latencies.

Based on the bug reporting latency data, we also develop a prediction engine that can predict when a bug would get reported. We first extract features from buggy code. We consider a number of code features, such as the number of conditional statements, the number of looping statements, and betweenness centralities of the nodes in call graphs. Then, these features are normalized and used to build discriminative models with the aid of classifiers. The models are then used to predict whether a bug would be reported early (in less than 30 days) or not. We evaluate our prediction models on the 190 bugs with 10-fold cross validations. Since the data is skewed (much more bugs are reported in more than 30 days) and has only two classes (early and late), we measure the accuracy of our predictions with Area Under the Receiver Operating Curve (AUC) [35], which has also been used in prior studies on predicting bug report severities [32]. Our evaluation shows that an AUC of 70.869% could be achieved when Support Vector Machines (SVM) [23] are used as the classifier.

The contributions of our work are as follows:

1) We propose a new problem of studying bug reporting latencies, namely the period of time between the introduction of a bug and the time when it is reported by end users. One goal of such a study is to help developers prioritize bugs prior releases.
2) We propose an approach to collect bug reporting latencies from software systems with version control repositories and bug tracking databases.
3) We develop an approach to predict whether the bug reporting latency for a bug would be low (below 30 days) or high. We extract features from buggy code and build a discriminative model to predict report latencies.
4) We have investigated 190 bugs extracted from AspectJ, Rhino, and Lucene and find diverse bug reporting latencies. We show that our proposed prediction engine could predict bug reporting latencies with good accuracy (AUC of 70.869%). We have also investigated a number of classifiers and evaluated their performance in predicting bug reporting latencies.

The structure of this paper is as follows. Section II presents our approach in detail. Section III describes our evaluation. We discuss related work in Section IV. Finally, we conclude with future work in Section V.

II. Proposed Approach

In this section, first we present our overall approach, and then present steps of the approach in detail.

A. Overall Approach

First, we set out to investigate and collect bug reporting latencies from known bugs from existing software systems so that we can build a ground truth about the existing latencies. We analyze several software systems using their repositories and bug tracking systems. We manually label faulty lines and find out when the last time each faulty line is changed. Using these times, we calculate the most likely bug reporting latency for each bug.We present the investigation we have performed for collecting latencies from various systems in more details in Section II-B.

Then, we construct prediction models based on the collected data. Our overall prediction flow is illustrated in Figure 1. It has training and deployment phases. In the training phase, we take in as input a set of bugs along

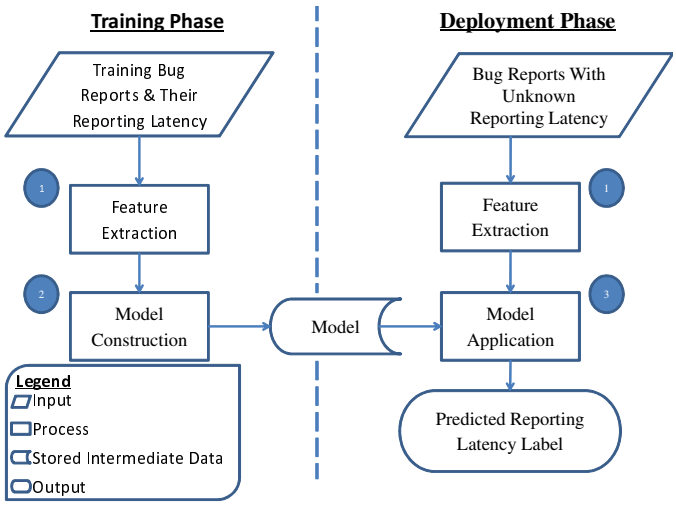

Figure 1. Proposed Bug Latency Prediction Framework

with their reporting latency labels (early or late). The bug reports are then subject to (1) feature extraction and (2) model construction. In the feature extraction step, we extract features from buggy code and use the features to build a discriminative model in the model construction step via a classifier. In the deployment phase, we take in as input a set of bugs with unknown reporting latency labels. Again, we extract features from the bugs as in the training phase. The model application process (3) finally applies the learned model to classify every bug with an unknown latency label, based on its features, as to whether it is likely to be reported in short or long period of time.

We present the feature extraction step in subsection II-C. The model construction and application steps are discussed in subsection II-D.

B. Latency Data Collection & Analysis

First, we need to get bug information from the bug databases and the source code repositories of each software system we investigate. From this information, we identify root causes of a bug, i.e., the faulty program elements that are responsible for the bug. We expect developers to have this information when they encounter the failure during the testing phase. Note that knowing where a fault is is just the first step among the many steps needed to fix the bug.

We perform the identification by manually investigating the differences between the fixed file and the buggy file before fix. We use *diff* to make this process easier. We then label the lines that we suspect are responsible for the bug. Unfortunately, *diff* is unable to directly give us the code responsible for the bug. *diff* only gives us information on what changes are made to fix the bug. Thus, they are only treatments of the bugs rather than the root causes of the bugs. For example, a piece of code may be moved around to make it easier to implement the actual fix. Also, not all changed lines are meant for fixing the bug. For example, previous studies (e.g., [25]) show that some changes are non-essential, such as changing indentation, adding comments, and code refactoring that do not change the behavior of a program. Our manual labeling process is based on personal investigation using the bug report message as guidance. We do not use automated techniques (e.g., [30]) to identify bug-inducing changes since we need all identified faulty lines to be accurate and we want to avoid biases that may be caused by possible false positives introduced by the tools. At the end of this process, we shall have the faulty lines for each bug. Using these faulty lines, we construct the features as defined in Subsection II-C.

Next, we want to find the time each bug gets introduced to the code. For this purpose, we use "git blame" or "cvs annotate" or similar scripts in the corresponding source control repositories for the software system to find out the time when the latest time each faulty line is changed. The result is a collection of the likely introduction dates for each bug. Each date in the collection of likely introduction dates corresponds to a labeled faulty line. This collection could contain some outliers, and we remove some of these outliers when we compute bug reporting latencies. Also, we need the report time for each bug. We find this from the bug tracking databases for each software system.

Then, we calculate bug reporting latencies as the time difference between the bug report date and the dates in the collection of likely bug introduction dates. If the likely bug introduction date is after the bug report date, we remove the date from the collection because this is likely to be an outlier. As the result, we will have a collection of likely bug reporting latencies. We choose to use the shortest latency in the collection as the most likely reporting latency for the bug. This is based on the intuition that the latest code changed is more likely to be the direct cause of the bug, and the fact that we noticed that the earliest faulty line introduction date for a number of bugs is for a very old piece of code that does not seem to be related to the bug. By taking the shortest latency, we aim to minimize latency estimations and it may lead to more fall-backs to Scenario 1 where reporting latencies are not used for prioritizing bug fixing activities.

At this point, we have the features and the bug reporting latencies which can be used for classification. We convert each bug reporting latency into class label, either *early* or *late*. *Early* is for bug reporting latencies shorter than or equal to 30 days, and *late* is for bug reporting latencies longer than 30 days. Based on the data collected about the bug reporting latencies and other bug characteristics (e.g., bug severity as indicated by developers), we seek to understand the data by investigating two research questions (RQ1 & RQ2) in Section III-B.

C. Feature Extraction

We extract a number of features from faulty code to facilitate the construction of prediction models. These fea-

tures are listed in Table I. We make use of the counts of a number of program elements, and two measures taken from social network analysis and web search, betweenness centrality [14] and PageRank [5].

1) Program Elements Counts: Intuitively, bugs in program elements of different types may be exhibited differently. For example, a bug in a looping statement might have more chances to be exhibited as the faulty statement might be executed more frequently by users than another bug at a simple assignment statement. We would like to investigate the relationship between types of program elements and bug reporting latencies. Thus, we include a number of features that are the number of times various program elements appear in the buggy code. These include `CountLine`, `MethodDeclarationNode`, `MethodInvocationNode`, `ConditionNode`, `LoopNode`, `AssignmentNode`, `FieldAccessNode`, `ConditionExp`, `ReturnNode`, and `TryNode`.

`CountLine` is the number of lines in the buggy code. Similarly, the other features count the number of times various types of program elements, including method declarations, method invocations, conditional statements, looping statements, assignment statements, field accesses, conditional expressions, return statements, and try statements, appear in the buggy code. We normalize these numbers by dividing these counts by the corresponding `CountLine`.

2) Betweenness Centrality: Intuitively, bugs at "important" program elements are likely to be discovered earlier than those at unimportant program elements. We measure the "importance" of a piece of code by using *betweenness centrality* and *PageRank*, and investigate whether these measurements are related to bug reporting latencies by using them in our prediction engine.

Betweenness centrality is proposed in the social network analysis community to measure how important a node is in a graph [14]. The importance of a node in the graph is measured by the number of shortest paths between pairs of other nodes that pass through the node. The betweenness centrality of a node n is given by the following formula:

$$bc(n) = \sum_{a \neq b \neq n} \frac{spath(a, b, n)}{spath(a, b)}$$

In the equation, $spath(a, b, n)$ refers to the number of shortest paths between node a and node b that pass through node n. $spath(a, b)$ refers to the number of shortest paths between node a and node b.

To compute `BetweennessScore` for each bug, we construct a call graph to connect all methods in the code base and compute the betweenness centrality of the methods that contain the faulty lines. `BetweennessScore` is the logarithm of the summation of the betweenness centrality of all methods containing the faulty lines. We take the logarithm as the summation of the betweenness centralities might be very large numbers.

3) PageRank Score: PageRank is a popular algorithm used in web search to measure the importance of a web page. It was proposed by Brin and Page [5] and has been used inside Google search engine. Simply put, PageRank assumes that web pages with more links pointing to them are more important than pages with less links pointing to them. PageRank computes the likelihood of a web surfer to visit a web page starting from an arbitrary page.

PageRank is often calcluated in iterations. At the initial iteration, all pages are assigned the same PageRank score. At each of the following iterations, the score of a page p is distributed to the pages that p links to. Each linked page receives $\frac{1}{|L|}$ of the score, where L is the set of pages that p links to. The PageRank score of a web page p at iteration i can be computed by the following equation:

$$PR(p, i) = \frac{1 - r}{T} + r \sum_{q \in K(p)} \frac{PR(q, i - 1)}{|L(q)|}$$

In the equation, r is the probability that a web surfer continues to surf (a.k.a. the *damping factor*), T is the number of web pages in the database, $K(p)$ is the set of webpages that link to p, and $L(q)$ is the set of webpages that q links to.

To compute the value of `PageRankScore` for a bug, we also build a call graph connecting all methods in the code base. We then identify faulty methods that contain the faulty lines. We then compute the PageRank of all faulty methods. The `PageRankScore` value would be the maximum of all thse PageRank scores. In the PageRank score computation, the number of iterations depends on the termination condition. This condition is fulfilled if the total number of iterations is equal to or greater than maximum iterations allowed or the difference between old and new scores in one iteration is less than a tolerance.

D. Model Construction and Application

In model construction, we assume that we have a set of training data composed of bugs and their reporting latencies classified into two classes: short (within 30 days), and long (more than 30 days). Each bug is represented as a vector of feature values. We train models that can discriminate bugs with short latencies from those with long latencies.

We train a discriminative model by employing a classification algorithm [17]. Various classification algorithms have been proposed in the literature, such as decision trees, support vector machines, neural networks, naive Bayes, and many others. Given a training data, a classification algorithm would produce a discriminative model in various formats depending on the algorithm.

If the training data is imbalanced, we equalize the numbers of data points with the positive label and the negative label. We follow the previous work by duplicating the minority label because we do not want to lose any data point. So, if the number of data points with the positive

Table I
FEATURES USED FOR PREDICTION MODEL

Feature	Definition
CountLine	Number of faulty lines
MethodDeclarationNode	Number of method declarations containing the faulty lines divided by CountLine
MethodInvocationNode	Number of method invocation in the faulty lines divided by CountLine
ConditionNode	Number of conditional statement containing the faulty lines divided by CountLine
LoopNode	Number of loop statement containing the faulty lines divided by CountLine
AssignmentNode	Number of assignment in the faulty lines divided by CountLine
BetweennessScore	Logarithm of the summation of the betweenness centralities of methods containing faulty lines
FieldAccessNode	Number of field accesses in the faulty lines divided by CountLine
ConditionExp	Number of conditional expressions in the faulty lines divided by CountLine
ReturnNode	Number of return statements in the faulty lines divided by CountLine
TryNode	Number of try statements containing the faulty lines divided by CountLine
PageRankScore	Summation of the PageRank score of the methods containing the faulty lines

label is lower than that with the negative label, we duplicate the data points with the positive label until the number is equal to that with the negative label. We try to duplicate the positive label as evenly as possible.

During the deployment phase, we classify a new bug with unknown latency. To do so, we first extract features from this new bug. We extract the same set of features as described in the previous sub-sections. The resultant vector of feature values is then fed to the discriminative model, which would then output the latency label (short or long) for the new bug.

III. EMPIRICAL EVALUATION

In this section, we first describe our concrete dataset taken from three Java software systems: AspectJ, Rhino, and Lucene. Next, we present our research questions. Then we then present our findings that answer each of the questions. Along the way, we describe the evaluation metric, namely AUC, which is used to measure the effectiveness of our prediction. Finally, we present the threats to validity.

A. Dataset

We analyze two Java programs (Rhino and AspectJ) from iBugs repository [1] and a third Java program Lucene from JIRA [2]. Rhino is a Javascript interpreter written in Java with a code size of about 49kLOC. AspectJ is a compiler for aspect-oriented programming in Java with a code size of about 75kLOC. Lucene (version 2.9) is a text search engine library with a code size of about 88kLOC. The iBugs repository stores both pre-fix versions that contain bugs (buggy versions) and the corresponding post-fix versions where the bugs are fixed. Each of the buggy versions is assumed to contain one bug that may span across multiple lines in multiple files. Information about each fix is also provided, e.g. the numbers of changed lines, changed methods, and changed files, and the severity level of the bug. Also, we can know which lines or files are actually changed based on the differences between the pre-fix and the post-fix versions. Similar information about bugs and corresponding fixes in Lucene is collected from its JIRA repository.

As described in Subsection II-B, we process above dataset to get bug reporting latencies. To generate features, we count different types of program elements with JDT [22], build call graph with Wala [53], and compute the betweenness centralities and PageRank scores with JUNG [24]. When computing RageRank, we use the default damping factor 1 and set the maximum number of iterations to 100 and the tolerance to 0.001. We find the likely bug introduction dates with "git blame" for Lucene and "cvs annotate" for AspectJ and Rhino. We ignore bugs in the three systems where their root causes are ambiguous according to our manual analysis. We also ignore bugs having a latency less than one day (similar to the study on bug fixes by Lamkanfi and Demeyer [31]). At the end of the process, we collect 190 bugs (19 bugs from Rhino, 28 bugs from Lucene, and 143 bugs from Aspectj). 34 of them have *short* reporting latencies, and 156 have *long* reporting latencies.

B. Research Questions

Our study investigates the following five research questions (RQ1-RQ5). In RQ1, we investigate the distribution of bug reporting latencies. In RQ2, we explore the relationship between bug reporting latency and bug severity. In RQ3, we investigate the effectiveness of our approach in predicting bug reporting latencies. We investigate the factors that influence the effectiveness of our approach in RQ4. Finally, we investigate the effectiveness of a number of classification algorithms for building discriminative models to predict bug reporting latencies in RQ5.

RQ1: What is the distribution of bug reporting latencies? We are interested in investigating how bug reporting latencies vary. If all bugs have similar reporting latencies, then predicting reporting latencies would not be needed. On the other hand, if bugs have widely varying latencies, it would make the problem more challenging.

RQ2: What is the relationships between bug reporting latencies and bug severity? Do bugs with a low latency have a high severity? A bug report typically indicates the severity of the bug. This severity could be one of the following: `blocker`, `critical`, `major`, `normal`, `minor`, and `trivial`. We would like to understand the correlation between bug reporting latencies with severities.

RQ3: How accurate does our approach predict bug reporting latencies? We compare the accuracy of our approach with a trivial, random classifier.

RQ4: What are the best features that could better discriminate bugs with short and long reporting latencies? We extract many features from bug reports. Some of the features are likely to be better in discriminating bug reports than other features. We would like to find features that are more effective than the others.

RQ5: What are the effectiveness of various common classification algorithms in predicting bug reporting latencies? When answering the previous research questions (RQ1-RQ4), we only use Support Vector Machines (SVMs) as the classifier, which have been used for many software engineering problems. Our overall framework allows other classification algorithms, and thus we investigate whether SVMs could outperform other classification algorithms, such as decision trees, naive bayes, and neural networks.

C. RQ1: Bug Reporting Latency Distribution

We show the distribution of bug reporting latencies in Figure 2. From the histogram, we find that there are 34 bugs reported in one month, 20 bugs reported between one to two months, 12 bugs reported between two to three months, 33 bugs reported between three to six month, 38 bugs reported between half a year to one year, and the remaining 53 bugs reported after one year. We notice that 65.26% of bugs was reported after 3 months. We also find that only 17.89% of the bug reported *early* (within 30 days) and the others are reported *late* (later than 30 days).

D. RQ2: Bug Reporting Latency versus Severity

We divide the bugs into several groups based on their latency. For each group, we investigate the proportion of bugs of various severity levels. We show the result in Table II. From the table, we see that for each bug severity, the highest number of bugs is in latency group >365 for blocker severity, 91-180 for critical severity, 1-30, 31-60, and 91-180 for major severity, >365 for normal severity, >365 for minor severity, and >365 for trivial severity.

We also divide the bugs into those reported within 30 days and those reported more than the 30 days. For each group, again we investigate the proportion of bug reports of various severity labels. We show the result in Table III. From the table we can note that 75% of blocker bug reported late, 90.9% of critical bug reported late, 75.9% of major bug reported late, 84.3% of normal bug reported late, 76.2% of minor bug reported late, and 75% of trivial bug reported late. Across different severity levels, we see that over 70% of the bugs is reported late, even for bugs with blocker, critical, and major severity.

We also calculate the correlation between bug reporting latencies and their severities. Note that severity is ordinal categorial, we then treat different levels as an integer number between 1 and 6, and compute Pearson product-moment correlation coefficient for this purpose [9]. It is a measure of linear dependence between two variables. The value is ranged from -1 to 1. A value -1 means all instances of variable X increase as variable Y decreases, and value 1 means all instances of variable X increase as variable Y increases. A zero value means there are no linear correlation between the two variables. We utilize SPSS [49] to compute the correlation coefficient between latencies (in days) and severities. We find that the Pearson coefficient is -0.013. This number is lower than the critical value of Pearson coefficient at the significance level of 0.05. It means there are almost no correlation between bug reporting latencies and severities.

Table II
DISTRIBUTION OF SEVERITY PER LATENCY GROUP

Group	Blocker	Critical	Major	Normal	Minor	Trivial
1-30	1	1	7	19	5	1
31-60	0	1	7	9	3	0
61-90	0	0	0	7	4	1
91-180	0	4	7	19	3	0
181-365	1	2	4	31	0	0
>365	2	3	4	36	6	2

Table III
DISTRIBUTION OF SEVERITY PER LATENCY GROUP

Group	Blocker	Critical	Major	Normal	Minor	Trivial
$\leq$ 30 days	1	1	7	19	5	1
$>$ 30 days	3	10	22	102	16	3

E. RQ3: Bug Report Latency Prediction Results

To measure the effectiveness of our latency prediction, we adopt the Area Under the Receiver Operating Characteristic (ROC) Curve (AUC) [12], which is also a metric used in previous studies in predicting bug severities [32], and compare our prediction with a random prediction.

1) Evaluation Metric: We use Area Under the Receiver Operating Characteristic Curve (AUC) [35] to evaluate our prediction performance because our data is skewed. Receiver Operating Characteristic Curve (ROC) is a two-dimensional measure of classification performance. It is a plot of the true positive rates versus false positive rates. For evaluation measure, when AUC is equal to 1, the classifier achieves perfect accuracy. The higher the AUC, the better the performance achieved by the classifier.

To compute AUC, we need to find the points in ROC curve for each classifier. Each point corresponds to the true positive and false positive rates of the classifier when a particular classification threshold is used to decide whether a data point would be labeled as positive or negative. From these points (including (0,0) and (1,1)), we compute the AUC by using trapezoid area formula as follows:

$$AUC = \sum_{i=1}^{n-1} \frac{(TPR_i + TPR_{i+1}) * (FPR_{i+1} - FPR_i)}{2}$$

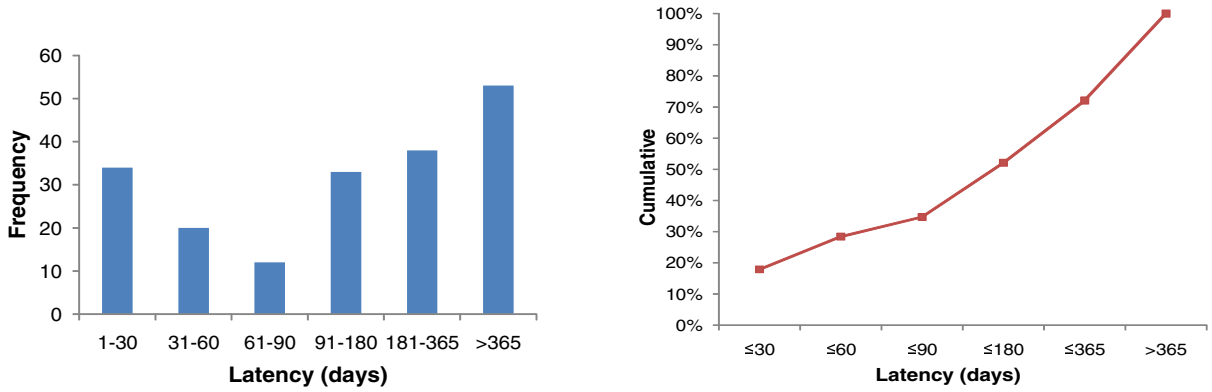

Figure 2. Bug Report Latency Time Histogram (Left) and Its Corresponding Cumulative Graph (Right)

where n is the number of points we have in the ROC curve including point (0,0). TPR_i and FPR_i is the true positive rate and the false positive rate for point i respectively.

In our evaluation, we perform k-fold cross validation to evaluate the classifier. This approach randomly splits the whole set of bugs into k buckets and perform k iterations. In each iteration, one bucket is used as testing data and the remaining buckets are used as training data. We train the classifier using the training data and evaluate it using the test data. We compute AUC for each of the iterations, and take the average of these AUCs. This average AUC is used as the performance measure of our classifier. We set k=10.

2) Prediction Effectiveness: The performance measurement when SVM is used as a classifier is shown in Table IV. We use svm_perf implementation [23]. It achieves on average an AUC of 70.869%. AUC of 0.5 indicates that a technique is as good as a random technique, while higher than 0.5 indicates that the technique performs better than the random technique [34], [46]. Thus, compared with a random classifier which randomly guesses the latency label for a bug, a 41.738% improvement of AUC could be achieved. We use the random classifier as baseline because as far as we know, there are no other study on bug reporting latencies for us to compare.

Table IV
PREDICTION MODEL PERFORMANCE MEASURE

Approach	AUC
Ours	0.70869
Random	0.5

F. RQ4: Most Discriminative Features

We want to know which feature is the most discriminative feature. We compute Fisher score for this purpose [7], [11], [27], [36]. Fisher score is a classifier independent criterion to measure the discrimination between a feature and the label. Given data instances x_i, i= $1, \ldots, n$, the Fisher score $F(j)$ for j^{th} feature can be computed by this equation:

$$F(j) = \frac{(\bar{x}_j^{(+)} - \bar{x}_j)^2 + (\bar{x}_j^{(-)} - \bar{x}_j)^2}{\frac{1}{n_+ - 1}\sum_{i=1}^{n_+}(x_{i,j}^{(+)} - \bar{x}_j^{(+)})^2 + \frac{1}{n_- - 1}\sum_{i=1}^{n_-}(x_{i,j}^{(-)} - \bar{x}_j^{(-)})^2}$$

In above equation, n_+ and n_- are the numbers of data points with positive and negative labels respectively. $\bar{x}_j$, $\bar{x}_j^{(+)}$, and $\bar{x}_j^{(-)}$ are the average of the j^{th} feature of all, positive-labeled, and negative-labeled data points. $x_{i,j}^{(+)}$ and $x_{i,j}^{(-)}$ is the j^{th} feature of the i^{th} positive/negative data point.

The score for each feature is in Table V. Based on Fisher scores, the most discriminative feature to predict whether a bug would be reported *early* or *late* is the normalized number of loops contained in the bug. The second best feature is line count. The third one is normalized number of try statements contained in the bug. The fourth one is normalized number of conditionals. The fifth is PageRank. Other special feature we add for the classification is betweenness centralities. It ranks ninth based on its Fisher score.

Table V
FISHER SCORE FOR EACH FEATURE

Feature	Fisher Score
LoopNode	0.228958166
CountLine	0.090998204
TryNode	0.064501643
ConditionNode	0.040184208
PageRankScore	0.024143757
FieldAccessNode	0.016027146
MethodDeclarationNode	0.015991601
AssignmentNode	0.010149085
BetweennessScore	0.006147854
ConditionExpNode	0.002262019
MethodInvocationNode	0.000442194
ReturnNode	0.000028648

G. RQ5: Best Classification Algorithm

We want to compare the performance of other classification algorithms in classifying bug reporting latencies. We choose ADTree, Naive Bayes, and Voted Perceptron to be compared with SVM. ADTree (alternating decision tree) is an algorithm that combines decision trees and boosting [16]. It works like normal decision trees, but instead of just following one path, it follows all paths in which the condition is true. Each of the paths will give a score. The sum of all these scores is used as the final score. A data point is classified as positive if its final score is positive and vice verse. Naive Bayes [45] is a classifier based on

Bayes theorem. The classifier computes the probability of each class label for a data point. The label with the highest probability is the predicted label for that data point. Voted Perceptron is a single layer perceptron variant which apply voting process to the original algorithm [15].

We use the Weka implementation of each of these algorithms [56]. The performance of these algorithms is shown in Table VI. None of these algorithms achieves a better AUC score than SVM.

Table VI
OTHER CLASSIFICATION ALGORITHM PERFORMANCE MEASURE

Algorithm	AUC
ADTree	0.6211
Naive Bayes	0.6552
Voted Perceptron	0.6252

We also investigate the efficiency of the classification algorithms. We show the runtime of the classification algorithms in Table VII. We calculate these values by running the classification algorithm ten times and average the runtime costs. We notice that Voted Perceptron is the fastest algorithm followed by Naive Bayes, ADTree, and lastly SVM. Even though SVM has the best performance, it takes a long time to run. The time needed for SVM is about nine times of that of Voted Perceptron. It is somewhat understandable considering that SVM usually needs many iterations until it finds the optimal solutions.

Table VII
RUNTIMES OF VARIOUS CLASSIFICATION ALGORITHM

Classification Algorithm	Time (in Seconds)
SVM	21.16
ADTree	2.64
Naive Bayes	2.45
Voted Perceptron	2.36

H. Threats to Validity

There are threats to construct validity, threats to internal validity, and threats to external validity.

Threats to construct validity refers to the suitability of our evaluation measure. We use AUC [35], which is a standard measure proposed to evaluate the accuracy of a classification task for imbalanced data with two class labels and used by Lamkanfi et al. [32] to evaluate the accuracy of predicting the severity labels of bug reports.

Threats of internal validity refers to evaluation errors and bias. We manually extract root causes of bugs and bug introduce time from their fixes. This process might be error prone. We have tried to reduce this error by re-checking the resultant root causes. Also, the bug tracking systems we use have no clear indication whether a bug reported is pre-release or post-release, and thus we assume the distribution of the predictors to use is the same between the pre-release and post-release bugs.

Threats of external validity refers to the generalizability of our findings. Due to the manual labor involved in the identification of bugs, we only analyze 190 bugs. We have tried to reduce this threat by analyzing bugs from various software systems. In the future, we plan to reduce this threat by investigating more bugs from more software systems.

IV. RELATED WORK

In the following, we present a number of related studies on bug severity prediction, bug fix time prediction,

Bug Severity Prediction. This line of work tries to predict the severity and priority of a bug report. Some examples of this line of work are the work by Menzies and Marcus [40], Lamkanfi et al. [32], [33], etc.

The pioneer of this work is Menzies and Marcus who predict the severity of bug reports from NASA [40]. They use information retrieval technique to extract a set of word tokens from the reports and sort the tokens based on their importance. To find the importance of the word tokens, the term frequency, inverse document frequency, and information gain of the word tokens are computed. The top-k important tokens are then input to a classification approach Ripper [10] to produce a set of classification rules. Their approach could predict the severity label of a new bug report.

Extending the work of Menzies and Marcus, Lamkanfi et al. investigate bug reports of various open source projects from their corresponding Bugzilla's [32]. Instead of predicting the exact severity label of a new bug report, they group bug reports into severe and not severe. They group blocker, critical, and major into severe bug reports. Minor and trivial labels are grouped into non-severe bug reports. They drop all bug reports classified as normal. Following the above work, Lamkanfi et al. also investigate the effectiveness of a number of classification algorithms in predicting the severity of bugs [33]. Their experiments show that Naive Bayes outperforms other classification algorithms on a dataset containing 29,204 bug reports. The work by Herraiz et al. [18] suggests that there are too many severity levels used in Eclipse project and there is a need to simplify them.

Work by Kim et al. [30] identify the changes in the program that potentially introduce bugs. Buse and Weimer [6] propose a statistical model to predict the execution frequencies of program paths in a program which may be used for bug finding or bug latency prediction.

Different from the above studies, our work do not analyze bug reports rather buggy code. We output information on predicted bug reporting latency period which is not output by any of the existing studies. Most importantly, we consider another problem setting namely on the prioritization of bugs that are found post-release. The above studies would be very useful to prioritize post-release bugs. However, they could not be used to prioritize pre-release bugs as there are not bug reports to analyze prior to a release.

Bug Fix Time Prediction. This line of work predicts the time needed to fix a particular bug. Weiß et al. predict the number of developer hours needed to fix a bug by looking for

similar existing bug reports and investigate the time needed to resolve the issues for prior similar bug reports [55]. These times are used to predict the time needed to predict a new bug report. A more recent study was performed by Hosseini et al. [19]. An empirical study on how long developers took to fix bugs has also been done by Kim and Whitehead [29]. They report some statistics of bug fixing time and highlight the top files in terms of highest bug fixing time.

Duplicate Bug Report Prediction. This line of work tries to detect if a new bug report is a duplicate of an existing bug report. There are many studies that take in a new bug report and returns the top-k most similar bug reports to it. Developers then check if the new bug report is a duplicate or not by looking at the top-k bug report list. This is referred to as duplicate bug report retrieval problem. Studies work on another problem that is referred to as duplicate bug report identification problem, where the task is, given a new bug report, to predict if it is a duplicate or not.

There are a number of existing studies on duplicate bug report retrieval. One of the first study was performed by Runeson et al. which proposes a formula that computes the similarity of two bug reports based on the concept of term frequency [47]. Wang et al. propose a formula that consider both term frequency and inverse document frequency [54]. Wang et al. not only consider the similarity of words between two bug reports but also the similarity of the execution traces corresponding to the bug reports. Jalbert and Weimer propose yet another formula based on term frequency for duplicate bug report retrieval; they also propose a solution for bug report identification [21]. Recently Sun et al. make use of Support Vector Machine (SVM) for finding similar bug reports [51]. Sun et al. later extend this work by proposing a new measure based on BM25F to compare two bug reports [50]. Tian et al. extend the work of Jalbert and Weimer by proposing an effective set of features that could better predict if a bug report is a duplicate or not [52].

Warning Prioritization. This line of work tries to identify whether a warning reported by an automated bug finding tool is really true or just a false positive [3], [28], [41], [48]. In this work, we analyze real bugs rather than warnings and our goal is to estimate how long it takes until the real bugs get noticed and reported.

Bug Categorization. This line of work assign bugs into various categories. Huang et al. propose a work that assign bug reports into one of the following labels: capability, security, performance, reliability, requirement, and usability [20]. Pordguski et al [44] and Francis et al. [13] group reported software failures such that reports of each group share the same or similar causes by analyzing the corresponding execution traces. Kim et al. [26] predict top crashes that occur before a new software release based on crashes that frequently occur in past releases.

Empirical Study on Bugs. There are number of empirical studies on bugs. Pan et al. perform an empirical study to find out what kinds of bug fixes have been performed in a number of open source software systems [43]. Chou et al. perform an empirical study on errors in operating systems [8]. Palix et al. investigate different faults in Linux ten years after the study by Chou et al. [42]. Bird et al. [4] investigate bug feature and commit feature bias in defect datasets used to evaluate the performance of defect prediction techniques and that bug feature bias could impact the performance of defect prediction techniques.

V. Conclusion and Future Work

We investigate the problem of prioritizing bugs by predicting the latency between the time when a bug is introduced to the code base and the time when a bug is likely to be experienced and reported by end users. We refer to this as bug reporting latency problem. We have collected bug reporting latencies from three Java software systems: AspectJ, Rhino, and Lucene, and find that reporting latencies are diverse and not correlated with bug severities. We find that only 17.89% of the bug reported *early* (within 30 days) and the others are reported *late*.

We also propose a classification-based approach to predict bug reporting latencies. To do that, we extract features from buggy code and use these features to build discriminative models via classifiers. The discriminative models are then used to label future bugs to identify if they would be reported in a short period of time (within 30 days) or not.

We have evaluated our prediction models on bugs from AspectJ, Rhino, and Lucene. The results show that we could predict bug reporting latencies with an AUC of 70.869%. We also find that the best five most discriminative features are the normalized number of loops contained in the bug, the line count of the bug, the normalized number of try statements contained in the bug, the normalized number of conditionals contained in the bug, and the PageRank scores. Lastly, we find that SVM is the most accurate classification algorithm in our solution, even though it has the longest runtime than others.

In the future, we plan to improve the accuracy of our proposed approach by leveraging other features that we could extract from bugs (e.g., semantics or functionality of the related code) and by proposing a better classification approach (e.g., by considering more fine grained labels or even predicting actual reporting latencies). We could potentially leverage frequent patterns and rules, c.f. [7], [37]–[39]. We also plan to evaluate on a larger set of bugs from more real programs.

References

[1] "iBUGS," http://www.st.cs.uni-saarland.de/ibugs/.
[2] "Lucene-JIRA," https://issues.apache.org/jira/browse/LUCENE.
[3] N. Ayewah and W. Pugh, "The Google FindBugs fixit," in *ISSTA*, 2010, pp. 241–252.

[4] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. T. Devanbu, "Fair and balanced?: Bias in bug-fix datasets," in *ESEC/SIGSOFT FSE*, 2009.

[5] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *WWW*, 1998, pp. 107–117.

[6] R. P. L. Buse and W. Weimer, "The road not taken: Estimating path execution frequency statically." in *ICSE*, 2009.

[7] H. Cheng, X. Yan, J. Han, and C.-W. Hsu, "Discriminative frequent pattern analysis for effective classification," in *ICDE*, 2007, pp. 716–725.

[8] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler, "An empirical study of operating system errors," in *SOSP*, 2001, pp. 73–88.

[9] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*, 2nd ed. Routledge, 1988.

[10] W. Cohen, "Fast effective rule induction," in *ICML*, 1995.

[11] R. Duda, P. Hart, and D. Stork, *Pattern Classification*. Wiley Interscience, 2000.

[12] T. Fawcett, "An introduction to roc analysis," *Pattern Recogn. Lett.*, vol. 27, no. 8, pp. 861–874, Jun 2006.

[13] P. Francis, D. Leon, and M. Minch, "Tree-based methods for classifying software failures," in *ISSRE*, 2004.

[14] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, vol. 40, no. 1, pp. 35–41, Mar 1977.

[15] Y. Freund and R. E. Schapire, "Large margin classification using the perceptron algorithm," in *Proceedings of the 11th Annual Conference on Computational Learning Theory*, 1998.

[16] Y. Freund and L. Mason, "The alternating decision tree learning algorithm," in *ICML*, 1999, pp. 124–133.

[17] J. Han and M. Kamber, *Data Mining Concepts and Techniques*, 2nd ed. Morgan Kaufmann, 2006.

[18] I. Herraiz, D. M. German, J. M. Gonzalez-Barahona, and G. Robles, "Towards a simplification of the bug report form in eclipse," in *MSR*, 2008, pp. 145–148.

[19] H. Hosseini, R. Nguyen, and M. Godfrey, "A market-based bug allocation mechanism using predictive bug lifetimes," in *CSMR*, 2012.

[20] L. Huang, V. Ng, I. Persing, R. Geng, X. Bai, and J. Tian, "AutoODC: Automated generation of orthogonal defect classifications," in *ASE*, 2011.

[21] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in *DSN*, 2008.

[22] JDT, "Java development tools," http://www.eclipse.org/jdt/.

[23] T. Joachims, "Training linear SVMs in linear time," in *KDD*, 2006. [Online]. Available: http://svmlight.joachims.org/

[24] JUNG, "Java universal network graph framework," http://jung.sourceforge.net/index.html.

[25] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *ICSE*, 2011, pp. 351–360.

[26] D. Kim, X. Wang, S. Kim, A. Zeller, S.-C. Cheung, and S. Park, "Which crashes should I fix first?: Predicting top crashes at an early stage to prioritize debugging efforts." in *TSE*, 2011.

[27] H. Kim, S. Kim, T. Weninger, J. Han, and T. F. Abdelzaher, "Ndpmine: Efficiently mining discriminative numerical features for pattern-based classification," in *ECML/PKDD (2)*, 2010, pp. 35–50.

[28] S. Kim and M. D. Ernst, "Which warnings should I fix first?" in *ESEC/FSE*, 2007, pp. 45–54.

[29] S. Kim and E. J. W. Jr., "How long did it take to fix bugs?" in *MSR*, 2006, pp. 173–174.

[30] S. Kim, T. Zimmermann, K. Pan, and E. J. W. Jr, "Automatic identification of bug-introducing changes." in *ASE*, 2006.

[31] A. Lamkanfi and S. Demeyer, "Filtering bug reports for fix-time analysis," in *CSMR*, 2012.

[32] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *MSR*, 2010.

[33] A. Lamkanfi, S. Demeyer, Q. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *CSMR*, 2011.

[34] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings." in *TSE*, 2008.

[35] C. X. Ling, J. Huang, and H. Zhang, "AUC: A better measure than accuracy in comparing learning algorithms," in *Canadian Conference on AI*, 2003, pp. 329–341.

[36] D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun, "Classification of software behaviors for failure detection: a discriminative pattern mining approach," in *KDD*, 2009, pp. 557–566.

[37] D. Lo, S.-C. Khoo, and J. Li, "Mining and ranking generators of sequential patterns," in *SDM*, 2008, pp. 553–564.

[38] D. Lo, S.-C. Khoo, and C. Liu, "Efficient mining of recurrent rules from a sequence database," in *DASFAA*, 2008, pp. 67–83.

[39] D. Lo, S.-C. Khoo, and L. Wong, "Non-redundant sequential rules - theory and algorithm," *Inf. Syst.*, vol. 34, no. 4-5, pp. 438–453, 2009.

[40] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *ICSM*, 2008.

[41] M. G. Nanda, M. Gupta, S. Sinha, S. Chandra, D. Schmidt, and P. Balachandran, "Making defect-finding tools work for you," in *ICSE*, 2010, pp. 99–108.

[42] N. Palix, G. Thomas, S. Saha, C. Calvès, J. L. Lawall, and G. Muller, "Faults in linux: ten years later," in *ASPLOS*, 2011, pp. 305–318.

[43] K. Pan, S. Kim, and E. J. W. Jr., "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, 2009.

[44] 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.

[45] J. D. M. Rennie, L. Shih, J. Teevan, and D. R. Karger, "Tackling the poor assumptions of naive bayes text classifiers," in *ICML*, 2003, pp. 616–623.

[46] D. Romano and M.Pinzger, "Using source code metrics to predict change-prone java interfaces." in *ICSM*, 2011.

[47] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *ICSE*, 2007, pp. 499–510.

[48] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel, "Predicting accurate and actionable static analysis warnings: An experimental approach," in *ICSE*, 2008, pp. 341–350.

[49] SPSS, "Predictive analytics software and solutions," http://www-01.ibm.com/software/analytics/spss/.

[50] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *ASE*, 2011.

[51] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *ICSE*, 2010.

[52] Y. Tian, D. Lo, and C. Sun, "Improved duplicate bug report identification," in *CSMR*, 2012.

[53] WALA, "T.J. watson libraries for analysis," http://wala.sourceforge.net/wiki/index.php.

[54] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *ICSE*, 2008, pp. 461–470.

[55] C. Weiß, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in *MSR*, 2007, p. 1.

[56] I. H. Witten and E. Frank, *Data Mining: Practical machine learning tools and techniques*, 2nd ed. Morgan Kaufmann, 2005.