Studying the Fix-Time for Bugs in Large Open Source Projects

Lionel Marks
School of Computing
Queen's University, Canada

Ying Zou
Dept. of Elec. and Comp. Eng.
Queen's University, Canada
ying.zou@queensu.ca

Ahmed E. Hassan
School of Computing
Queen's University, Canada
ahmed@cs.queensu.ca

ABSTRACT

Background: Bug fixing lies at the core of most software maintenance efforts. Most prior studies examine the effort needed to fix a bug (*fix-effort*). However, the effort needed to fix a bug may not correlate with the calendar time needed to fix it (*fix-time*). For example, the fix-time for bugs with low fix-effort may be long if they are considered to be of low priority.
Aims: We study the fix-time for bugs in large open source projects.
Method: We study the fix-time along three dimensions: (1) the location of the bug (e.g., which component), (2) the reporter of the bug, and (3) the description of the bug. Using these three dimensions and their associated attributes, we examine the fix-time for bugs in two large open source projects: Eclipse and Mozilla, using a random forest classifier.
Results: We show that we can correctly classify ∼65% of the time the fix-time for bugs in these projects. We perform a sensitivity analysis to identify the most important attributes in each dimension. We find that the time of the filing of a bug and its location are the most important attributes in the Mozilla project for determining the fix-time of a bug. On the other hand, the fix-time in the Eclipse project is highly dependant on the severity of the bug. Surprisingly, the priority of the bug is not an important attribute when determining the fix-time for a bug in both projects.
Conclusion: Attributes affecting the fix-time vary between projects and vary over time within the same project.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Cost estimation, Software quality assurance (SQA), Time estimation*

General Terms

Measurement, Management

Keywords

Empirical software engineering, mining software repositories, bug fix-time

PROMISE '11, September 20–21, 2011, Banff, Canada

1. INTRODUCTION

Many software development activities center around the process of fixing bugs. Bug reports are filed by various stakeholders in a project. These bug reports are then triaged by project experts who assign the bugs to the appropriate individual(s). The problem of automating the triaging process has been investigated by Anvik et al. [1] and by Cubranic and Murphy [7].

Determining the needed effort to fix a bug (fix-effort) is an important step in project planning and management. The fix-effort depends on several factors such as the complexity of the bug and the complexity of the code. Song et al. propose the use of association rule mining to classify the fix-effort for bugs in the NASA SEL defect data [21]. Zhen and Rine propose the use of neural networks to classify the fix-effort of bugs in the NASA KC1 data set [23]. Weiß et al. propose the use of a kNN approach to determine the fix-effort for bugs in the JBoss project [22].

All of the aforementioned work requires the existence of a mature software development process within a studied project, where effort tracking of each bug is recorded. Such detailed information rarely exists. Moreover, the effort needed to fix a bug (*fix-effort*) may not correlate with the calendar time needed to fix it (*fix-time*). For instance, the fix-time for bugs with low fix-effort may be long if they are assigned to an already overloaded developer or if they are considered low priority bugs.

The fix-time of a bug depends on many other attributes. For example, bugs assigned to inexperienced developers are likely to take longer to fix. Conversely, critical bugs are likely to be fixed sooner even if their fix-effort is high. Intuitively a bug report with high priority or high severity will be fixed sooner than other bugs independent of its expected fix-effort. However to date, there has not been any studies investigating bug fixing patterns in large projects.

In this paper, we study the fix-time for bugs in large open source projects. We define the fix-time as the calendar time from the triage of a bug to the time the bug is resolved and closed as fixed. We study the fix-time along three dimensions:

1. **Location:** the location of the bug (e.g., which component or operation system for multi-platform applications),
2. **Reporter:** the reporter of the bug, and
3. **Description:** the detailed description attached to the bug.

We define several attributes that capture information about each one of these three dimensions,. For example for the description of a bug, we measure the size of the initial bug report. We then use the attributes and their associated metrics to study the fix-time for bugs in two large open source projects: Eclipse and Mozilla. We classify fix-time as one of three high-level classes in an effort to derive simple and easy to communicate rules about the bug fixing process:

1. Bugs fixed within 3 months (i.e., within a quarter),

2. Bugs fixed within one year,

3. Bugs fixed within 3 years.

We use a decision tree-based algorithm to predict the class of a bug given the various attributes along the aforementioned dimensions. Instead of using simple decision tree algorithms, such as C4.5 [12], we use an advanced decision tree learner, called random forest, which is known to be robust for noisy data. Due to the large number of attributes and the large amount of studied bugs which span several years, we expect the existence of noise in our data. We perform a sensitivity analysis to identify the most important attributes in each dimension.

Our results show that we can correctly classify 65% of the time the fix-time for bugs in the Mozilla and Eclipse projects. Our sensitivity analysis indicates that the time of the filing of a bug and its location are the most important attributes in the Mozilla project for determining the fix-time of a bug; while the fix-time in the Eclipse project is highly dependant on the severity of the bug. Surprisingly, the priority of the bug is not an important attribute when determining the bug fix-time. Our results show that the attributes affecting the bug fix-time vary between projects and vary over time within the same project. Moreover using a tree-based classifier, we can correctly classify 65% the bug fix-time in these two projects.

Using our results, managers can allocate testing resources and plan release schedules with reasonable accuracy. Moreover, our results can be used by builders of bug tracking databases, such as Bugzilla, to simplify the information needed to file a bug report based on the impact of such information on the fix-time of bugs.

1.1 Organization of the Paper

The organization of the paper is as follows. Section 2 discusses the three dimensions, used in our study of the fix-time for bugs, along with the various attributes in these dimensions. Section 3 presents the random forest, a decision tree learner, and motivates its use to study and predict the fix-time for bugs. Section 4 presents our case study of the bugs in the Mozilla and Eclipse projects. We generate various decision trees using historical data from the bug repositories of these projects and we study the structure of these trees. Section 5 presents related work. Section 6 concludes the paper and outlines some future work.

2. DIMENSIONS FOR STUDYING THE FIX-TIME FOR BUGS

To better understand the fix-time process for bugs, we chose to study the fix-time along three dimensions. For each dimension, we defined several attributes and used metrics to quantify these attributes.

1. **Location:** the location of the bug [20 metrics for 7 attributes],

2. **Reporter:** the reporter of the bug [7 metrics for 4 attributes], and

3. **Description:** the detailed description attached to the bug [22 metrics for 12 attributes].

We now present the various attributes and their associated metrics along the three dimensions. We also motivate the use of the three dimensions and the various attributes by citing prior work and common project management wisdom.

2.1 Location Dimension

Table 1 lists the attributes in the bug location dimension. A large body of research (e.g., [9, 14, 15, 16, 18]) in software reliability shows that bugs usually lurk in specific spots with bugs reoccurring in buggy parts of a software system. The 80/20 rule in software reliability states that 80% of bugs reside in 20% of the code. We sought to determine if such a phenomena exists for the fix-time for bugs – with bugs in particular parts of the software system being fixed sooner than others. These parts could correspond to particular subsystems or operating systems or could be parts that have had a historical tendency to be fixed fast.

2.2 Reporter Dimension

Table 2 lists attributes in the people dimension. Conway's hypothesis [6], formulated by Brooks [8] as Conway's law, states that the structure of a software system matches the organization of the group that designed the system. Bowman and Holt demonstrated the effect of team structure and collaboration on the structure of several large software systems [3].

We sought to examine if the reporter of a bug had an effect on its fix-time. For example, are bugs filed by a particular person (e.g., a manager) likely to be fixed sooner than bugs filed by a project outsider.

2.3 Description Dimension

Table 3 lists the attributes in the bug description dimension. Most prior research on studying bug reports has focused primarily on this dimension. Whereas the previous two dimensions capture information about the peculiarities of a software project (e.g., its people and its architecture), this dimension captures information about the content of bug reports themselves (e.g., the size of their description) and the history of bug reports (e.g., the usual fix-time of a high priority bug). We sought to measure if the readability of a bug, its priority, its severity, or the amount of interest expressed in a bug are good indicators of its fix-time.

3. RANDOM FORESTS

The purpose of our study is to understand the most important attributes that affect the fix-time for a bug. We use a large number of attributes to determine the fix-time for a particular bug. We can model our study as a classification problem where each bug report falls in one of three classes: *fixed in* $<$ *3 Months*, *fixed in* $<$ *1 Year*, and *fixed in* $<$ *3 Years*. Given the large number of used metrics, we chose to discretize our metrics since in many instances the exact value is not critical. For example, it does not make a difference whether the number of open bugs is 101 or 102. We also chose to discretize our metrics to permit practitioners to easily use our created models. We only use three values: low, medium and high for all numerical attributes. Therefore a practitioner using our model only needs to know if the number of currently-open bugs is high, medium or low when a particular bug is filed in order to predict the likely fix-time of that bug.

There exist many machine learning techniques, such as Support Vector Machines (SVM) and neural networks, which can solve this classification problem. However, we chose to use a technique based on decision trees since decision trees build explainable models. These explainable models are essential in helping us understand the attributes affecting the fix-time for a particular bug. The use of discretized values helps as well in reducing the complexity of the generated model and ensuring their simplicity with at most 3 possible option for each numerical attribute instead of having a large number of possible splits in the created tree models.

Attribute Name	Explanation and Rationale
Product (1 metric)	The product in which the reported bug exists. Are bugs in particular products more likely to be fixed?
Version (1 metric)	The version in which the reported bug exists.
Component (1 metric)	The component in which the reported bug exists.
Operating system (1 metric)	The operating system in which the reported bug exists. Are bugs in operating systems with a large installation base (e.g., Windows) more likely to be fixed sooner than bugs in other operating systems?
Open bugs (6 metrics)	The number of bugs opened in the version, component, operating system in the last year and overall. If the number of open bugs is too large then we would expect that a recently opened bug would take longer than usual to fix, since the team is likely too busy.
Fixed bugs (6 metrics)	The number of bugs fixed in the version, component, operating system in the last year and overall. This metric along with the open bugs metric is used to gauge the fix rate of bugs in a project.
Project fix-time (4 metrics)	The average fix-time for bugs in the project up till the filing of this particular bug. The intuition is that the fix-time of a bug would be similar to the common fix-time of prior bugs in the project. We also calculate the average fix-time for the Version, Component, and Operating system of the bug.

Table 1: Bug Location Dimension (20 metrics).

Attribute Name	Explanation and Rationale
Reporter type (1 metric)	Are bugs filed by industry, people local to the project, or other open source developers likely to be fixed sooner than bugs filed by the general public? This attribute is calculated by manually classifying the developer email address field in a bug report.
Reporter popularity (4 metrics)	Is the popularity of a bug reporter likely to affect the fix-time of a bug? We use 3 metrics to capture the popularity of a reporter: 1. **Overall popularity:** Percentage of bugs fixed by the reporter relative to all bugs reported by that reporter throughout the lifetime of the project. 2. **Recent popularity:** Same as the above metric, but over the past year. This metric attempts to capture recent changes in the popularity of a reporter. 3. **Relative recent/overall popularity:** The rank of the above metrics relative to other reporters in the project.
Reporter fix-time (1 metric)	Is the fix-time for a bug dependant on the usual fix-time for bugs filed by its reporter in the past? For example, If a bug is reported by a reporter whose bugs are usually fixed quickly, will that bug be fixed quickly as well?
Reporter requests (1 metric)	Are bugs filed by reporters with a large number of open bugs likely to be delayed? We measure the number of open bugs reported by the reporter of this bug.

Table 2: Reporter Dimension (7 metrics).

Instead of using basic decision tree algorithms such as C4.5 [12], we used an advanced decision tree algorithm called Random Forest [5]. The Random Forest algorithm outperforms basic decision tree algorithms in prediction accuracy. Moreover, the Random Forest is more resistant to noise in the data. This feature of the Random Forest algorithm is very important since we expect that the data used in our study to be noisy, due to the large amount of data and the fact that there is no well-defined process in inputting data in bug repositories such as Bugzilla. Another advantage of the Random Forest algorithm is the limited number of parameters that must be configured. Also the derived models tend to be robust and stable to changes in the input attributes. Finally, often the prediction accuracy of basic decision tree algorithms suffers when many of the attributes are correlated. Given the large number of attributes in our study, we needed an algorithm that does not suffer from correlated attributes. The Random Forest algorithm deals well with correlated attributes while maintaining high accuracy in its prediction.

In contrast to simple decision tree algorithms, the Random Forest algorithm builds a large number of basic decision trees (30 trees in our case study). Each node in each tree is split using a random subset of all the attributes to ensure that all the trees have low correlation between them. The trees are built using 2/3 of the available data through sampling with replacement. The 1/3 of the remaining data is called the Out Of Bag (OOB) data and is used to test the prediction accuracy of the created forest. The use of bootstrapping and the random selection of attributes at each node has been shown to greatly improve the accuracy of tree-based classifiers [4].

In our case study, we use the OOB data to measure the accuracy of the created forests. Each sample in the OOB is pushed down all the trees in the forest and the final class of the sample is decided by aggregating the votes of each tree. One major benefit of using this technique is that we can adjust the votes accordingly based on the skew in the data. Basic decision trees are known to perform badly with highly skewed data with the tree usually predicting the

Attribute Name	Explanation and Rationale
Severity (1 metric)	The severity of the bug: blocker, critical, enhancement, major, minor, normal, and trivial. Are critical or trivial bugs fixed sooner than normal bugs? Panjer observed that the severity of a bug has a large effect on its lifetime for the Eclipse project [17]. Severity is entered by the reporter of a bug.
Priority (1 metric)	Are high priority bugs fixed sooner? Priority is specified by the project personnel during the triage of a bug.
Interest (2 metrics)	Are bugs with high interest likely to be fixed sooner? We use the **Number of CCed** people and the existence of a **QA contact** in the report as indicators of interest in a bug.
Time (4 metrics)	Does the time of reporting of a bug have an effect on its fix-time? We use four metrics to capture the reporting time: 1) **Morning, Day or Night?** 2) **Week** 3) **Month** 4) **Year**.
Bugs open in project (1 metric)	The number of bugs currently open in the project. This metric captures the workload of the whole development team.
Has comments (1 metric)	Were there any additional notes attached to the bugs? Do these additional notes have an effect on the speed of resolving a bug? Hooimeijer and Weimer's analysis of Mozilla bugs shows that the number of comments has a large effect on the speed of fixing a bug [11].
Has target milestone (1 metric)	Should the bug be resolved for a particular milestone? We expect bugs assigned to a milestone to be fixed sooner than other bugs.
Length of bug description (1 metric)	The length of the field describing the bug. Are well-documented bugs more likely to be fixed sooner?
Amount of code (1 metric)	A recent survey by Bettenburg et al. [2] shows that bug reports with code samples are fixed sooner than other bugs. We count the number of semicolons as a rough approximation of the lines of code in the description of a bug.
Readability (7 metrics)	The same survey by Bettenburg et al. [2] shows that bugs that are easy to read are considered as well-written bugs by developers. Furthermore, Hooimeijer and Weimer's analysis of the bugs in Mozilla shows that bug reports that are easier to read are fixed faster [11]. We use 7 metrics based on the Flesch and Kincaid Scores to measure the easy of readability [13].
Severity fix-time (1 metric)	The average fix-time for bugs with similar severity. The intuition here is that bugs of similar severity would be handled using the same process, independent of their complexity or other attributes.
Priority fix-time (1 metric)	The average fix-time for bugs with similar priority. Similar intuition as severity fix-time.

Table 3: Bug Description Dimension (22 metrics).

dominant class. To overcome this problem in a Random Forest, we can assign weights to votes to reflect the data skew. Based on the distribution of classes shown in Table 6, we assign the following weight: $<$ *3 Months* (1), $<$ *1 Year* (2), and $<$ *3 Years* (10) for the Eclipse data. For the Mozilla data, all classes equal are given the same weight since we do not observe a large skew in the data.

Reducing the Effect of Correlated Attributes

The use of bootstrapping and the random selection of attributes helps in reducing the negative effect of correlated attributes in tree learners. To further reduce the effect of correlated attributes and improve the performance of the random forest, we ran the random forest algorithm in two phases. In the first phase, we use all the attributes in a dimension, then we determine the top 15 most important attributes in a dimension and we re-run the random forest once more. The reduction of attributes usually improves the performance of the forest by 5-10% [5]. The procedure to determine the importance of an attribute is listed below. Our two-phase run results in 60 trees being built for each experiment. If a run has less than 15 attributes, we re-use all the attributes for the second phase. We chose 15 attributes as a reasonable (not too big and not too small) number that can capture the important attributes in a dimension. Other values are possible and could be explored in the future.

Measuring Performance and Attribute Importance

To measure the accuracy of the predictions produced by the Random Forest algorithm, we calculate the overall, $<$ *3 Months*, $<$ *1 Year*, and $<$ *3 Years* misclassification rates. We desire the lowest possible overall and per-class misclassification rates. The rates are defined using the confusion matrix, shown in Table 4, as follows:

	Classified As		
True Class	*<3 Months*	*<1 Year*	*<3 Years*
<3 Months	a	b	c
<1 Year	d	e	f
<3 Years	g	h	i

Table 4: Confusion matrix – The entries in the matrix are the classification made by a tree classifier versus the actual class in a three-class (*<3 Months*, *<1 Year*, and *<3 Years*) setting.

1. **Overall misclassification rate:** This captures the overall performance of the forest across all classes. It is defined as: $\frac{(a+e+i)}{(a+b+c+d+e+f+g+h+i)}$.
2. ***<3 Months* misclassification rate:** This captures the performance of the forest for bugs fixed within 3 months. It is defined as: $\frac{(b+c)}{(a+b+c)}$.
3. ***<1 Year* misclassification rate:** This captures the performance of the forest for bugs fixed within 1 year. It is defined as: $\frac{(d+f)}{(d+e+f)}$.
4. ***<3 Years* misclassification rate:** This captures the performance of the forest for bugs fixed within 3 years. It is defined as: $\frac{(g+h)}{(g+h+i)}$.

Sensitivity Analysis

Another benefit of using the Random Forest is that we can perform sensitivity analysis on the input attributes to determine the most important attributes in the created forest. To perform the sensitivity analysis for a particular attribute, the value of the attribute is randomly changed in all the samples in the OOB data, and the samples

are re-classified. We then measure the change in misclassification rate. If an attribute is not important then we expect that the misclassification rate would not change. Otherwise the misclassification rate would increase relative to the importance of an attribute. For the case study, we created ten random forests of 60 trees (600 trees in total) for each set of attributes then we measured the average misclassification rates. We report the average rates for all the trees in our case study.

To determine the most important attributes in the ten forests, we calculated an attribute importance for each forest, then combined the importance ranking for each forest to reach the overall importance values shown in our case study. The **attribute importance metric** shown in our case study is calculated as follows:

1. For each forest, an attribute is given a score from 1 to 10 relative to the attribute's rank: 10 for most important, 1 for tenth most important, and 0 if worst than rank ten.

2. For each attribute, we sum its score across all ten forests and we divide this sum by the maximum score that an attribute can get, i.e., 100: 10 (for highest rank) * 10 (for ten forests).

We use this metric to measure the most important attributes. In our case study, we only show the top five attributes with an importance value ≥ 60.

Implementation Note

We use a Fortran implementation of the random forest algorithm [5]. Our case study requires the execution of a large number of experiments (one for each dimension and for each studied project). However, the Fortran implementation does not support command line parameters. Instead of modifying the fortran code for each experimental run, we developed a Perl program that takes as input the Fortran code and performs code transformation on the Fortran code to generate the correct code needed for each experimental run in our case study. The code transformation modifies the Fortran code inserting the parameters needed for the run.

4. CASE STUDY

To study the fix-time for bugs, we used bug data from two large open source projects: the Eclipse and Mozilla projects. Table 5 lists the various resolution types for bugs in the Mozilla and Eclipse projects. We only study bugs that have been resolved as fixed. For the Mozilla project we studied 85,616 bugs, representing 32% of all bugs in its Bugzilla database. For the Eclipse project we studied 63,402 bugs, representing 59% of all bugs in its Bugzilla database.

Project	*<3 Months*	*<1 Year*	*<3 Years*
Mozilla	46%	27%	27%
Eclipse	76%	18%	6%

Table 6: Breakdown of fix-time for bugs in the studied projects

Table 6 shows a detailed breakdown of the different classes for the studied bugs in both projects. For example, 46% of the 85,616 bugs in Mozilla are fixed in $<$ 3 Months, whereas 27% are fixed in $<$ 1 Year and 27% are fixed in $<$ 3 Years. Using the classes shown in Table 6, we conducted four experiments: one experiment for each of the three dimensions listed in Section 2 and one experiment using all attributes together. We report below the results of our experiments.

Figures 1a and 1b show the distribution of fix-time for bugs over the lifetime of the Mozilla and Eclipse projects. For the Eclipse project, most of the bugs are fixed within 3 months throughout the lifetime of the project. At the start of the Mozilla project, the fix-time for bugs was evenly distributed. However, as the project evolved, the number of quickly resolved bugs (i.e., bugs fixed in $<$ 3 months) continues to rise while the number of long-lived bugs (i.e., bugs fixed in $<$ 3 Years) continues to drop in most of the following years.

Experiment #1: Location Dimension

	Mozilla		**Eclipse**	
#	*Attribute*	*%*	*Attribute*	*%*
1	Product	97	Project fix-time	94
2	Component	87	Open bugs in product	66
3	Operating system	79		
4	Component fix-time			

Table 7: Top Attributes for the Location Dimension

For our first experiment, we use the attributes for the bug location dimension, listed in Table 1. Table 7 shows the most influential attributes for predicting fix-time for bugs in the Mozilla and Eclipse projects. The top attributes differ between both projects, however the top attributes for both projects are related to the product: product and product fix-time. For Mozilla, the product and component of a bug are more influential than the operating system under which the bug was identified. For Eclipse, the time commonly needed to fix other bugs in the same product (i.e., product fix-time) is the most influential attribute.

Experiment #2: Reporter Dimension

Our second experiment studies the dimension related to the reporter of the bug. Table 2 lists the attributes explored in that dimension. Table 8 shows the most influential attributes in this dimension for both studied projects. The most important attribute, the reporter fix-time is the same across both projects. So reporters with short fix-times will tend to have their bugs always fixed quickly. The rest of the top attributes are based on the popularity of the reporter.

	Mozilla		**Eclipse**	
#	*Attribute*	*%*	*Attribute*	*%*
1	Reporter fix-time	100	Reporter fix-time	100
2	Reporter requests	90	Overall popularity	81
3	Recent popularity	74	Recent popularity	80
4	Relative recent pop.	69	Relative recent pop.	74
4	Overall popularity	67	Relative popularity	65

Table 8: Top Attributes for the Reporter Dimension

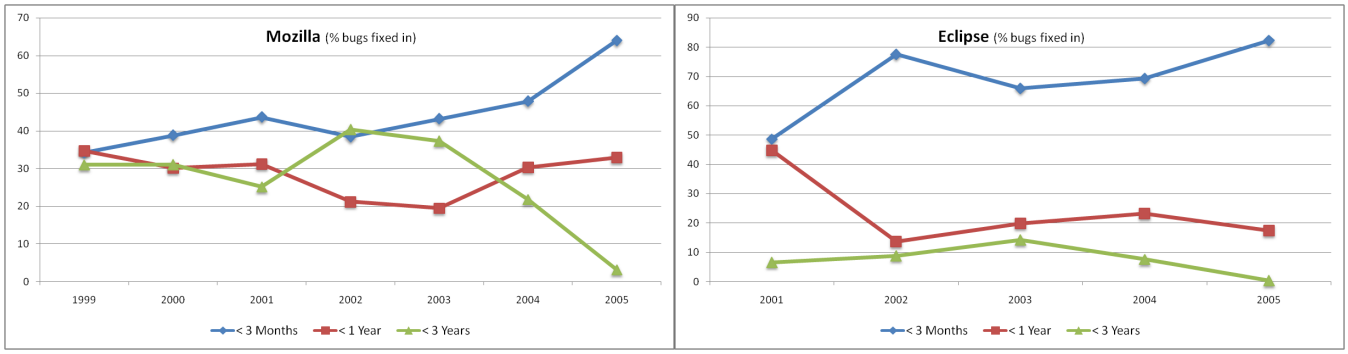
Experiment #3: Description Dimension

In our third experiment, we explore the bug description dimension. We use the attributes listed in Table 3. Table 9 shows the results of our sensitivity analysis. Our analysis shows that the year of the reporting of the bug is an influential attribute. This indicates that the bug fixing patterns vary over the lifetime of a project. Hooimeijer and Weimer also observed this dependence on the reporting year for the Mozilla project [11].

Surprisingly, the priority of a bug is not an influential attribute in either project. The severity of the bug shows up as the second most influential attribute in Eclipse, but does not show up for the

Project	Duplicate	Invalid	Moved	Won't fix	Works for me	Fixed	Total
Mozilla	99,414	29,856	103	9,512	46,659	85,616	271,160
	(37%)	(11%)	(0%)	(4%)	(17%)	(32%)	(100%)
Eclipse	19,060	7,958	0	7,141	10,013	63,402	107,574
	(18%)	(7%)	(0%)	(7%)	(9%)	(59%)	(100%)

Table 5: Statistics for the Resolution Type of Bugs for the Mozilla and Eclipse Projects



(a) Mozilla

(b) Eclipse

Figure 1: The distribution of fix-time for bugs over the lifetime of the Mozilla and Eclipse projects. The x-axis shows the specific project years, while the y-axis shows the percentage of bugs fixed within a specific class

Mozilla project. The observation about severity confirms the finding reported by Panjer using basic decision trees [17]. Our analysis shows that the year of the bug has a higher impact on the fix-time for a bug than the severity of the bug. We also note that our findings do not match with the finding by Hooimeijer and Weimer [11] that severity of bugs in Mozilla has an important effect on the time needed to fix a bug. In contrast, our results show that reporting time (e.g., Year and Week of year) have a more important influence.

We believe that this is due to the fact that Hooimeijer and Weimer measure the fix-time of a bug from the time it is entered into Bugzilla till it is fixed. In contrast, we only measure the time from the assignment of a developer (i.e., after triage) till the bug is fixed. Therefore, it might be the case that bugs with high severity are triaged faster but then their fix-time is independent of their severity level. This could be due to the fact that severity levels do not represent the true severity of a bug.

	Mozilla		Eclipse	
#	*Attribute*	%	*Attribute*	%
1	Year	100	Year	99
2	Has target milestone	82	Severity	87
3	Number of CCed	80	Number of CCed	83
4	Week of Year	77	Has target milestone	69

Table 9: Top Attributes for the Bug Description Dimension

Experiment #4:All Attributes

In our last experiment, we combine all the attributes across the three dimensions. This resulted in 50 metrics that were studied using our sensitivity analysis technique. Table 10 shows the results of our analysis. The table shows that the severity of the bug and the number of CCed project personnel are the most important attributes for the Eclipse project, while on the other hand the reporting time (year and week), and the location (product and component) have the largest influence for the Mozilla project.

	Mozilla		Eclipse	
#	*Attribute*	%	*Attribute*	%
1	Year	97	Severity	87
2	Product	92	Number of CCed	73
3	Week	71	Product fix-time	61
4	Component	63		

Table 10: Top Attributes for the Models with All Attributes

Discussion

Table 11 shows the performance of the three dimensions and all attributes for the Mozilla and Eclipse projects. The Table shows that the dimensions perform differently for both projects. For the Mozilla project, the best performing dimensions are: Location, Description, then Reporter. For the Eclipse project, the best performing dimensions are: Location, Reporter, then Description. The performance differences are statistically significant. The results suggest that the bug descriptions for the Eclipse project should be improved to help project managers in project planning and resource allocation decisions.

A random guessing approach would result in an overall misclassification rate of 60%, as we have 3 classes. Our random forest classifier shows considerable improvement over random guessing. Moreover, if we did not discretize the numerical attributes we might be able to achieve a lower misclassification rate. However, the produced model would be much harder to comprehend and use in practice for project planning as we are searching for simple and basic rules-of-thumb that practitioners could use.

We verified the significance of our result differences by comparing the performance of the ten forests generated for each di-

Dimension	Mozilla				Eclipse			
	<3 Months	*<1 Year*	*<3 Years*	*Overall*	*<3 Months*	*<1 Year*	*<3 Years*	*Overall*
Reporter	39.3	80.1	44.6	**51.9**	29.5	47.0	83.4	**35.8**
Location	27.5	61.1	38.4	**39.6**	32.0	44.1	41.4	**34.7**
Description	49.2	57.9	46.2	**50.8**	42.4	43.1	48.5	**43.0**
ALL	32.3	45.4	33.6	**36.2**	32.0	35.1	37.0	**32.8**

Table 11: Misclassification (lower is better) of the Three Dimensions and all Attributes for the Mozilla and Eclipse Projects

mension against each other using a paired Wilcoxon signed-rank test. Wilcoxon is resilient to strong departures from the assumptions of the t-test [19]. Our statistical analysis assumes a 5% level of significance (i.e., $\alpha = 0.05$). We formulate the following test hypotheses:

$$\mathcal{H}_0 : \mu(Perf_A - Perf_B) = 0$$

$$\mathcal{H}_\mathcal{A} : \mu(Perf_A - Perf_B) \neq 0$$

$\mu(Perf_A - Perf_B)$ is the population mean of the difference between the performance of each of the ten forests generated for a dimension. If the null hypothesis H_0 holds (i.e., the derived p-value $> \alpha = 0.05$), then the difference in mean is not significant. If H_0 is rejected with a high probability (i.e.,the derived p-value $\leq \alpha = 0.05$), then we can be confident about the performance improvements of a particular dimension. All result differences hold for our significance level.

Figure 2a and 2b show that these identified patterns are consistent throughout the lifetime of the studied projects with the Description and Reporter dimensions alternating in being the worse performing dimensions (Models are rebuilt using data for each corresponding year). In the Figures a lower number is desirable over a higher number since the Figures show the misclassification rates.

The Bug Description dimension performs badly for both project. We believe that this is a troubling finding since it shows that various aspects of the bug report such as the readability of the report, or the severity and priority of the bug, have little effect on the fix-time of a bug. Using our findings along with a careful analysis of how these two projects make use of Bugzilla would help improve the effectiveness of such bug reporting systems. It may be desirable for builders of such tools, to explore adding additional information in the bug report to improve the performance of the bug description dimension.

Limitation of our Findings

Our results show that the attributes influencing the fix-time for bugs vary across most dimensions for the two projects that we studied. We observe a consistent trend for both projects in the people dimension. However, we have only studied two projects: Eclipse (a development environment written in Java) and Mozilla (a web browser written in C++). Further studies are needed on other types of systems to determine if we can identify general trends across projects. We defined a large number of attributes for each studied dimension. However, other attributes may be defined. These additional attributes may lead to large improvements in one dimension over the other. For our study, we discretized the fix-time using three classes (within 3 months – within a quarter, within 1 year, and within 3 years). Other discretization levels could be explored. We choose these discretization levels since we believe that they match closer to the levels used by project managers for ad hoc estimation: this quarter, this year, and when-possible. We also chose to discretize the metrics used in our models to permit the use of our models in ad hoc informal estimations. Non-discretized metrics might lead to better performing models but would be hard to use to derive rules-of-thumb for informal estimations by project managers. The discretization of the fix-time and metrics provide a clear and concise structure for comparing the results across the two different software systems.

5. RELATED WORK

Most prior work focuses on bug fix-effort as mentioned earlier in the introduction of the paper. Hooimeijer and Weimer perform linear modeling and ANOVA analysis to study the bug fix-time in the Mozilla project [11]. In contrast, we use a decision tree learner technique to perform our analysis and we explore a larger number of attributes. Moreover, the produced decision trees are much easier to comprehend and reason about due to their simplicity and intuitiveness. Hassan and Zhang [10], and Shirabad *et al.* [20] in prior work used regular decision trees, e.g., C4.5 to perform sensitivity analysis by breaking the data into ten different folds, creating a decision tree for each fold and studying the structure of the tree (i.e., the occurrence of a node at the different levels for the ten built trees). Using this analysis, we can determine the most influential attribute. However this analysis cannot determine the second most influential attribute, since the second most influential attribute is dependant on the values of the attributes higher up in the decision tree.

6. CONCLUSION

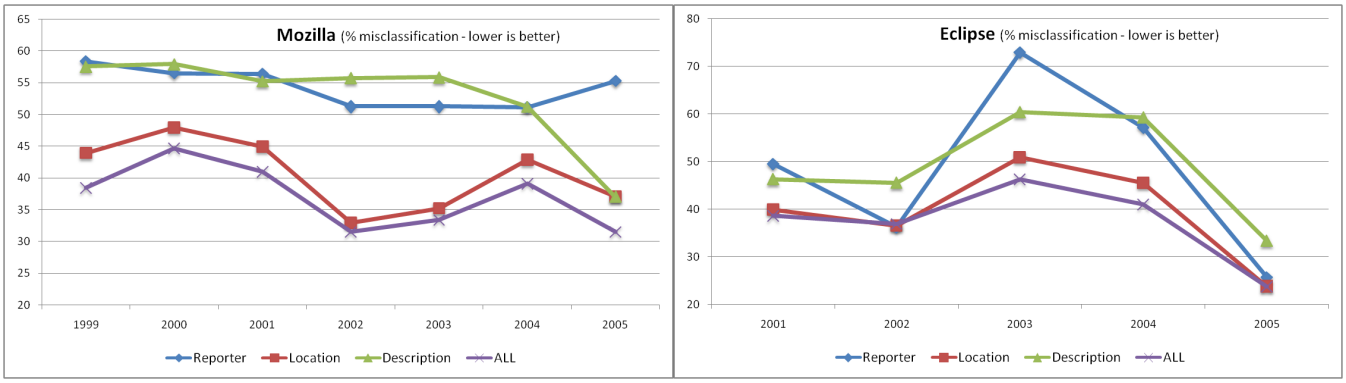
Bug fixing is an important and central activity in software development. Given a particular bug report, practitioners require assistance in gauging the effort needed to fix the bug (fix-effort) and the expected calendar time by which the bug will be fixed (fix-time). Most research efforts focus on studying the aspects of the fix-effort for a bug. In this paper, we explore the characteristics of the fix-time for a bug. Our results show that attributes such as the priority and sometimes the severity of a bug, long regarded as important attributes, have little effect on the fix-time for bugs in large open source software systems. Using a random forest algorithm we can correctly predict the class of a bug with a success rate of $\sim$65%.

Acknowledgments

We are grateful to Thomas Zimmermann for giving us access to the bug repositories used in our case study. We also acknowledge the help of Lucas Panjar who developed the initial version of the scripts used to process the bug repositories.

7. REFERENCES

[1] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering*, pages 361–370, Shanghai, China, May 2006.



(a) Mozilla (b) Eclipse

Figure 2: The Yearly Misclassification Rate for the Mozilla and Eclipse projects. The x-axis shows the specific project years while the y-axis shows the misclassification rate for each dimension and the overall misclassification rate. A lower number is desirable over a higher number since we are showing the misclassification rates.

[2] N. Bettenburg, S. Just, A. Schröter, C. Weiß, R. Premraj, and T. Zimmermann. What Makes a Good Bug Report. Technical report, Saarland University, 2008.

[3] I. T. Bowman, R. C. Holt, and N. V. Brewster. Reconstructing Ownership Architectures To Help Understand Software Systems. In *Proceedings of the 7th International Workshop on Program Comprehension*, Pittsburgh, USA, May 1999.

[4] L. Breiman. Bagging Predictors. *Machine Learning*, 26(1):123–140, 1996.

[5] L. Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

[6] M. E. Conway. How do comittees invent? 14(4):28–31, 1968.

[7] D. Cubranic and G. C. Murphy. Automatic bug triage using text categorization. In *Proceedings of the Sixteenth International Conference on Software Engineering and Knowledge Engineering*, pages 92–97, Banff, Canada, 2004.

[8] J. Frederick P. Brooks. *The Mythical Man-Month*. Addison Wesley Professional, 1974.

[9] T. L. Graves, A. F. Karr, J. S. Marron, and H. P. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[10] A. E. Hassan and K. Zhang. Using decision trees to predict the certification result of a build. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 189–198, Washington, DC, USA, 2006. IEEE Computer Society.

[11] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 34–43, New York, NY, USA, 2007. ACM.

[12] J. Quinlan. *Programs for Machine Learning*. Morgan Kaufmann, 1993.

[13] J. P. Kincaid, R. P. Fishburne, R. L. Rogers, and B. S. Chissom. Derivation of new readability formulas (automated readability index, fog count and flesch reading ease formula) for navy enlisted personnel. Research Branch Report 8-75, Saarland University,, 1975.

[14] M. Leszak, D. E. Perry, and D. Stoll. Classification and evaluation of defects in a project retrospective. *The Journal of Systems and Software*, 61(3):173–187, 2002.

[15] J. Munson and T. Khoshgoftaar. The Detection of Fault-Prone Programs. *IEEE Transactions on Software Engineering*, 18(5):423–433, 1992.

[16] N. Ohlsson and H. Alberg. Predicting Fault-Prone Software Modules in Telephone Switches. *IEEE Transactions on Software Engineering*, 22(12):886–894, dec 1996.

[17] L. D. Panjer. Predicting eclipse bug lifetimes. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, May 2007.

[18] D. E. Perry and C. S.Steig. Software Faults in Evolving a Large, Real-Time System: a Case Study. In *Proceedings of the 4th European Software Engineering Conference*, Garmisch, Germany, Sept. 1993.

[19] J. Rice. *Mathematical Statisitcs and Data Analysis*. Duxbury press, 1995.

[20] J. S. Shirabad. *Supporting Software Maintenance by Mining Software Update Records*. PhD thesis, University of Ottawa, 2003.

[21] Q. Song, M. Shepperd, M. Cartwright, and C. Mair. Software defect association mining and defect correction effort prediction. *IEEE Transactions on Software Engineering*, 32(2):69–82, feb 2006.

[22] C. Weiß, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, May 2007.

[23] H. Zeng and D. Rine. Estimation of software defects fix effort using neural networks. In *Annual International Computer Software And Applications Conference*, Sept 2004.