Predicting Eclipse Bug Lifetimes

Lucas D. Panjer
Department of Computer Science
University of Victoria
Victoria, British Columbia, Canada
ldp@cs.uvic.ca

Abstract

In non-trivial software development projects planning and allocation of resources is an important and difficult task. Estimation of work time to fix a bug is commonly used to support this process. This research explores the viability of using data mining tools to predict the time to fix a bug given only the basic information known at the beginning of a bug's lifetime. To address this question, a historical portion of the Eclipse Bugzilla database is used for modeling and predicting bug lifetimes. A bug history transformation process is described and several data mining models are built and tested. Interesting behaviours derived from the models are documented. The models can correctly predict up to 34.9% of the bugs into a discretized log scaled lifetime class.

1 Introduction

Developers are often asked to estimate the amount of time they will need to fix specific bugs to aid the project planning process. Accurate estimation of task completion time can allow project planners and managers to effectively schedule releases and allocate effort to meet those targets. Since developers spend much of their time dealing with legacy code and maintenance tasks, it is important to be able to estimate the required time and effort to complete bug fixing and enhancement tasks.

This research explores the viability of predicting how long a newly confirmed bug will take to complete by exploiting prior data specific to a given project and organization. Data mining techniques are explored due to the very large size of these data sets. A bug lifetime predictive model can provide an organizational planning tool for developers and an early warning indicator for specific incoming tasks. Early warnings can be used to identify specific bugs or tasks which might never be resolved, or be used to estimate the state of a bug database at a later date.

2 Research Question

Initial literature searches revealed that there is little work that has attempted to predict how long a newly identified bug will take to fix. However, there has been research that looks at life spans and characteristics of bugs. Chou et al. [3] have shown that the average life span of certain classes of bugs in the Linux and OpenBSD kernels can extend into years. In the open-source software (OSS) development field Kim et al. [4] also study the life span of bugs: their research in two OSS projects, ArgoUML and PostgreSQL, shows that bug-fix times have a median of about 200 days and can extend into years.

This research is preliminary work designed to determine if it is possible to predict the lifetime of a bug from the time of confirmation (*NEW* state) to resolution (*RESOLVED* state), given the use of existing elementary information in a bug repository. Specifically, what levels of prediction accuracy can be achieved using these attributes? what interesting phenomena of Eclipse bugs can be derived from the models built?

3 Data Set

Eclipse's Bugzilla was chosen as a target of this research because Bugzilla has a wide installation base in the OSS and commercial development field and Eclipse is a large and mature OSS project. Bugzilla is also used by several projects that are commonly studied in a software repository mining context, and it is well studied and understood by the community.

In this approach, bugs are examined at specific points throughout their life. To accomplish this we need to examine all changes for each bug. A pre-harvested data set of Eclipse bugs was acquired from Thomas Zimmerman and imported into a database. This data import included fields contained in the Bugzilla XML data representation as well as the change history of each bug. The change history is gathered by crawling the public Eclipse Bugzilla website

Fourth International Workshop on Mining Software Repositories (MSR'07)
0-7695-2950-X/07 $20.00 © 2007 IEEE

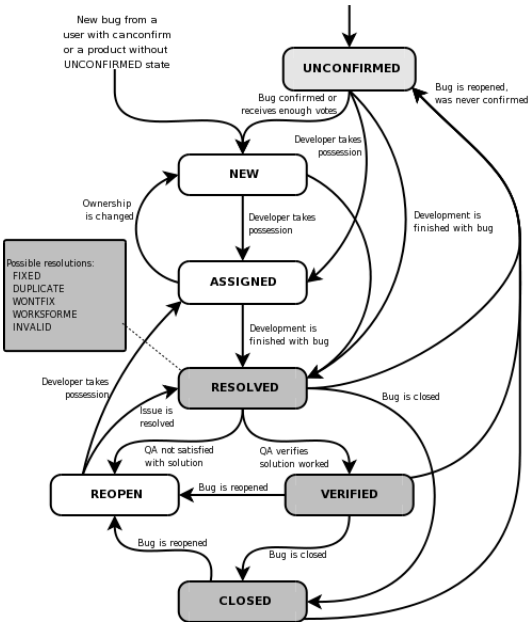

Figure 1. The life cycle of a Bugzilla bug

and parsing the change history pages for relevant fields as this information is not included in the XML representation.

3.1 Bugzilla Process

The life cycle of a bug in a Bugzilla tracking system follows a set of structured states and transitions [1] (See Figure 1). A bug enters the system in either the *UNCONFIRMED* or *NEW* state. If the bug is reported by an untrusted submitter it is initially classified as *UNCONFIRMED*. If the submitter is trusted, (e.g., a committer or core member of the team) the bug is initially classified as *NEW*. Bugzilla managers or triagers confirm the existence, validity, and non-duplicity of bugs and move them from *UNCONFIRMED* to *NEW*. Once a bug is triaged and assigned to the correct product, severity, and priority, bugs are assigned to developers and moved to *ASSIGNED*.

When a bug has been fixed the current assignee moves it to *RESOLVED*. Other team members may move the bug to other states, such as *VERIFIED* or *CLOSED*, or may reopen the bug during testing and release procedures. A bug is considered to be resolved when it reaches a state or status of *RESOLVED*, *VERIFIED*, or *CLOSED*. Resolutions are combined with a status to denote the reason for resolving the bug; only bugs that have resolution of *FIXED* are considered. Resolutions of *DUPLICATE*, *WONTFIX*, *WORKSFORME*, and *INVALID* denote invalid bugs and are not considered. A bug lifetime is defined as the time in days between *NEW* and *RESOLVED*.

3.2 Data Set Manipulation

The extracted data set exposes the current state of each bug. Since we want are predicting the lifetime of bugs at time of confirmation we must rolled-back to the *NEW* status. Roll-back is achieved by incrementally applying the change history of the bug in reverse order until a specific state is reached. Counts of the *cc list*, *dependent bugs*, *bug dependencies*, and *comments* of the bug are calculated while the roll-back is computed. Bugs are filtered to remove all bugs that have not been reached *RESOLVED* and the first 4907 bugs are removed from the experimental data set since they were imported into the Eclipse Bugzilla database and have invalid creation dates.

The fields included in the data set were: *assigned to*, *qa contact*, *priority*, *severity*, *product*, *component*, *operating system*, *platform*, *version*, *target milestone*, *cc count*, *bug dependencies*, *dependent bugs*, *keywords*, *comments*, and *lifetime* in days. The *votes* and *attachments* fields that are commonly used in Eclipse's Bugzilla and would be ideal candidates for exploration are missing from this data extraction. Votes are used to allow the community to express support or interest in a bug and attachments are used to post patches, stack traces, and other supporting files.

The distribution of bug lifetimes in days is a heavily skewed long-tail. The data set begins on the 11^{th} of October, 2001 and the last bug is on the 6^{th} of March, 2006. Our data set contains 118 371 usable bug data points after roll-back. The maximum number of days open is 1601 days, the mean is 66.2 days, and the standard deviation is 153.1 days. Since the bug resolution time is so heavily skewed we interpret and display it using a log_e scale.

To allow the application of data mining tools and algorithms that require a nominal target class to operate we discretized the time to resolution values using an equal-frequency binning algorithm. The size of each bin generated by the discretization algorithm roughly corresponds to the segmentation one might naturally use to reason about scheduling (e.g., day, week, month, six months, or a year). Figure 2 shows the bin sizes generated by discretization. The binning algorithm could not accommodate the large portion of the bugs that were completed in one day so there is a larger bin for the $0 < days < 1.4$ class.

4 Automatic Classification

The WEKA toolkit [2] was used to perform data mining and analysis of the constructed data set for predicting bug lifetimes. Basic algorithms, 0-R and 1-R, were initially explored to establish a baseline of prediction accuracy. Further, Naive Bayesian Networks, Decision Trees, and Logistic Regression are explored. Cross validation is used to test the models and prediction accuracy, kappa statistic, and in-

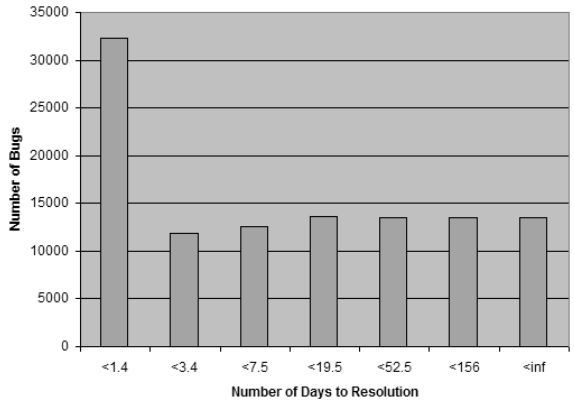

Figure 2. Distribution of resolution times

teresting traits of each result are documented. Prediction accuracy and the kappa statistic are used instead of precision, recall, or f-measure of each class because the target classification for this data set is nominal with seven values. The kappa statistic is presented as a measure of the agreement between the prediction and the true classification. This is used as an overall measure of accuracy across all output classes. A summary of result measures is shown in Table 1.

4.1 0-R and 1-R

The 0-R algorithm correctly classifies 29.1% of bugs and the kappa statistic is 0. 0-R predicts the class *<1.4 days* in all cases. The 1-R algorithm builds a one level decision tree by first generating a set of rules that each test a particular attribute, then simply selecting the best-performing rule. The 1-R algorithm correctly classifies 31.0% of bugs and the kappa statistic is 0.0747. The attribute selected to classify every instance is the *comments*. The rule states that bugs with $0 < comments < 1.5$ are resolved in $1.4 < days < 3.5$, bugs with $1.5 < comments < 6.5$ are resolved in $0 < days < 1.4$, and that bugs with $comments \geq 6.5$ are resolved in $days \geq 52$.

4.2 Decision Trees

The C4.5 decision tree algorithm [5] correctly classifies 31.9% of bugs and the kappa statistic is 0.0938. The top node of these trees is always the *comments* (count), followed by *assigned to*. Reviewing the resultant trees and subsequently the source data it was noted that the *assigned to* field often contained **-inbox@eclipse.org* addresses that are used as placeholders for specific components until bugs are actually assigned to human developers.

Further models were built with the *assigned to* field removed. In these model we correctly classify 31.9% of bugs and the kappa statistic is 0.0938. The prediction accuracy of these models is the same but the rules generated are more interesting. The top node is still always *comments* followed by *severity* and version *version*.

Algorithm	Predicted %	Kappa
0-R	29.1%	0.0000
1-R	31.0%	0.0747
C4.5 Decision Tree	31.9%	0.0938
Naive Bayes	32.5%	0.1195
Logistic Regression	34.9%	0.1577

Table 1. Summary of prediction results

4.3 Naive Bayes

Naive Bayes is a rule generator based on Bayes' rule of conditional probability which treats all attributes as if they were independent. The Naive Bayes algorithm correctly classifies 32.5% of bugs and the kappa statistic is 0.1195. The Naive Bayes algorithm provides a broader distribution of predictions than the 0-R or 1-R algorithms, classifying at least several thousand bugs into each class.

4.4 Logistic Regression

Logistic regressions are used to provide a regression analysis where many of the dependent variables are nominal or if the class is nominal. Due to computational constraints a set of 496 randomly selected bugs is considered, or 0.42% of the original data set. Using the logistic regression algorithm a model is built and correctly classifies 34.9% of bugs and the kappa statistic is 0.1577.

5 Discussion

0-R provides a baseline of prediction accuracy for resolution time of bugs in the source data set. 0-R predicts that all bugs will be resolved within $0 < days < 1.4$. This is simply because the most common actual outcome is $0 < days < 1.4$.

The 1-R algorithm selects the most important variable by generating a decision tree stump. This is the first rule that would exist in a larger decision tree. In this case, 1-R selects a rule defining $comments \leq 1.5$ to predict bug resolution times of $1.4 < days < 3.4$ and $1.5 < comments < 6.5$ to predict bug resolution times of $0 < days < 1.4$. Higher values for *comments* always imply resolution times of greater than 52 days.

This behaviour could exist because most bugs start with one comment and that comment is provided by the submitter of the bug while describing the issue. Given the

0-7695-2950-X/07 $20.00

heavily skewed distribution of resolution times favouring the bug resolution time of $0 < days < 1.4$ this is not a surprising result. The second portion of the rule, which notes that bugs tend to close in less than 1.4 days when $1.5 < comments < 6.5$ is interesting as it might imply that bugs with early discussion are resolved quickly.

5.1 Advanced Predictions

Top node analysis of the decision trees shows that *comments* is the most influential variable. After *assigned to* is removed due to inaccurate data we see that *severity* as assigned by either the bug submitter or triager followed by *version* are the next important variables.

Analysis of the decision tree rules provides several interesting observations. Bugs with little discussion, fewer than four comments, tend to be resolved in less than 1.4 days; bugs that can be resolved with little discussion are resolved quickly. When bugs require more conversation, greater than four comments, the resolution times become dependent on *severity*. Bugs assigned a *severity* of *blocker*, *critical*, *trivial* result in a resolution time of less than 1.4 days; simple and very important bugs are fixed very quickly. Bugs assigned a *severity* of *enhancement* or *future* require six months or more to resolve; less severe bugs are fixed at a slower pace. Bugs assigned a *severity* of *major* have more comments and longer resolution times, specifically where greater numbers of comments exist.

It seems that triagers and assignees heed the assigned severity rating or that the triaging team is correctly assigning *severity*. It is also interesting that the *priority* field is not seen as important in the decision trees. Perhaps the Eclipse team's process places less importance on the priority field.

Naive Bayes generates a prediction model assuming that all variables are strongly independent. This technique can work surprisingly well on real-world data sets where there exists strong dependence between variables [6]. The application of the naive Bayes shows little improvement in correctly predicted lifetimes. This might imply that the variables in this data set tend to not be either strongly independent or strongly functionally dependent variables. Our prediction accuracy is slightly better than when using a decision tree approach, but does not allow easy comprehension of the rules used to predict the outcomes.

Logistic regression was used on a small randomly selected sample of the data set. The best rate for correctly classified instances is achieved using this method. A set of coefficients is calculated for each target class. For all classes the influential variables are *severity*, *operating system*, *product*, *component*, *version*, *number of watchers*, and *comments*. These variables seem to intuitively match the variables that I consider most important when filing bug reports or structuring queries. Perhaps this is true for other users of this Bugzilla database.

None of the modeling approaches used yielded particularly high correct classification or kappa values, nor did the advanced data mining algorithms significantly exceed the baseline predictions. The accuracy achieved is not likely to be of value to a bug triager. It seems that there are other attributes or metrics that may have greater influence of the resolution time of bugs. There is merit in predicting lifetimes at a later point in time removing the bugs that close quickly, increasing the relevance of the remaining bugs.

6 Conclusion

This research explores modeling of bug lifetimes using information available at the beginning of a bug's lifetime. A technique to roll-back a bug to a specific state is presented and used to create data sets. Data mining techniques are explored and evaluated to predict lifetimes and to learn which features are salient for Eclipse bugs.

Exploring the Eclipse bug set with various data mining algorithms reveal that an accuracy of 34.9% can be acheived using only the primitive attributes associated with a bug. The most influential affecting length of bug lifetime are commenting activity, severity of bug as determined by the software development team, product, component, and version. Speculation suggests that higher level attributes such as average lifetime of bugs in a specific product or component may have greater predictive power.

7 Acknowledgments

I am grateful to Thomas Zimmermann for providing an extracted Eclipse Bugzilla data set with change histories and to Ahmed Hassan for his guidance in this research.

References

[1] *Bugzilla Documentation – Life Cycle of a Bug. Available online at http://www.bugzilla.org/docs/tip/html/lifecycle.html.*

[2] *Weka 3 – Machine Learning Software in Java. Available online at http://www.cs.waikato.ac.nz/ ml/weka/index.html.*

[3] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. *ACM SIGOPS Operating Systems Review*, 35(5):73–88, 2001.

[4] S. Kim and J. E. James Whitehead. How long did it take to fix bugs? In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 173–174, New York, NY, USA, 2006. ACM Press.

[5] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[6] I. Rish. An empirical study of the naive Bayes classifier. *Proceedings of IJCAI-01 Workshop on Empirical Methods in Artificial Intelligence*, 335, 2001.