Who Should Fix This Bug?

John Anvik, Lyndon Hiew and Gail C. Murphy
Department of Computer Science
University of British Columbia
{janvik, lyndonh, murphy}@cs.ubc.ca

ABSTRACT

Open source development projects typically support an open bug repository to which both developers and users can report bugs. The reports that appear in this repository must be triaged to determine if the report is one which requires attention and if it is, which developer will be assigned the responsibility of resolving the report. Large open source developments are burdened by the rate at which new bug reports appear in the bug repository. In this paper, we present a semi-automated approach intended to ease one part of this process, the assignment of reports to a developer. Our approach applies a machine learning algorithm to the open bug repository to learn the kinds of reports each developer resolves. When a new report arrives, the classifier produced by the machine learning technique suggests a small number of developers suitable to resolve the report. With this approach, we have reached precision levels of 57% and 64% on the Eclipse and Firefox development projects respectively. We have also applied our approach to the gcc open source development with less positive results. We describe the conditions under which the approach is applicable and also report on the lessons we learned about applying machine learning to repositories used in open source development.

Categories and Subject Descriptors: D.2 [Software]: Software Engineering

General Terms: Management.

Keywords: Problem tracking, issue tracking, bug report assignment, bug triage, machine learning

1. INTRODUCTION

Most open source software developments incorporate an open bug repository that allows both developers and users to post problems encountered with the software, suggest possible enhancements, and comment upon existing bug reports. One potential advantage of an open bug repository is that it may allow more bugs to be identified and solved, improving the quality of the software produced [12].

However, this potential advantage also comes with a significant cost. Each bug that is reported must be *triaged* to determine if it describes a meaningful new problem or enhancement, and if it does, it must be assigned to an appropriate developer for further handling [13]. Consider the case of the Eclipse open source project[1] over a four month period (January 1, 2005 to April 30, 2005) when 3426 reports were filed, averaging 29 reports per day. Assuming that a triager takes approximately five minutes to read and handle each report, two person-hours per day is being spent on this activity. If all of these reports led to improvements in the code, this might be an acceptable cost to the project. However, since many of the reports are duplicates of existing reports or are not valid reports, much of this work does not improve the product. For instance, of the 3426 reports for Eclipse, 1190 (36%) were marked either as invalid, a duplicate, a bug that could not be replicated, or one that will not be fixed.

As a means of reducing the time spent triaging, we present an approach for semi-automating one part of the process, the assignment of a developer to a newly received report. Our approach uses a machine learning algorithm to recommend to a triager a set of developers who may be appropriate for resolving the bug. This information can help the triage process in two ways: it may allow a triager to process a bug more quickly, and it may allow triagers with less overall knowledge of the system to perform bug assignments more correctly. Our approach requires a project to have had an open bug repository for some period of time from which the patterns of who solves what kinds of bugs can be learned. Our approach also requires the specification of heuristics to interpret how a project uses the bug repository. We believe that neither of these requirements are arduous for the large projects we are targeting with this approach. Using our approach we have been able to correctly suggest appropriate developers to whom to assign a bug with a precision between 57% and 64% for the Eclipse and Firefox[2] bug repositories, which we used to develop the approach. We have also applied our approach to the gcc repository, but the results were not as encouraging, hovering around 6% precision. We believe this is in part due to a prolific bug-fixing developer who skews the learning process.

The paper makes two contributions:

[1]Eclipse provides an extensible development environment, including a Java IDE, and can be found at `www.eclipse.org` (verified 31/08/05).

[2]Firefox provides a web browser and can be found at `www.mozilla.org/products/firefox/` (verified 07/09/05).

ICSE'06, May 20–28, 2006, Shanghai, China.

1. it presents an approach with promising results for helping to automate bug assignments on open-source systems, and

2. it identifies difficulties in tracing information between the bug and source code repositories used in an open-source development that could easily be changed to support this kind of automation.

We begin by presenting background information about open bug repositories, including the information stored in a bug report, the process used to handle bug reports, and machine learning (Section 2). We then describe related efforts that attempt to automate parts of the bug handling process (Section 3). Given this background, we describe our semi-automated approach to bug assignment (Section 4) and present the results of applying the approach to the gcc project (Section 5). We then discuss issues related to the approach (Section 6) and summarize the paper (Section 7).

2. BACKGROUND

Understanding our approach requires knowledge about open bug repositories and machine learning.

2.1 Open Bug Repositories

Bug repositories, which are sometimes referred to as issue-tracking systems, provide a database of problem reports for a software project. We use the term *open bug repository* to refer to repositories in which anyone with a login and password can post a new report or comment upon an existing report. These types of repositories play an important role in open source projects because they provide a communication channel for geographically-distributed developers, and because they allow the user community to view the progress developers are making on the project.

A variety of open bug repositories are used in open source development (e.g., Bugzilla[3], GNATS[4] and JIRA[5]). We developed our approach and have analyzed projects that use Bugzilla-based repositories. Since other open bug repositories store similar information and support a similar bug reporting and solving process, we believe our approach can generalize to other bug repositories with minor changes.

2.1.1 Anatomy of a Bug Report

Figure 2 shows an example of a bug report for Eclipse stored in Bugzilla.[6] Each bug report includes pre-defined fields, free-form text, attachments, and dependencies.

The pre-defined fields provide a variety of categorical data about the bug report. Some values, such as the report identification number, creation date, and reporter, are fixed when the report is created. Other values, such as the product, component, operating system, version, priority, and severity, are selected by the reporter when the report is filed, but may also be changed over the lifetime of the report. Other fields routinely change over time, such as the person to whom the report is assigned, the current status of the report, and if resolved, its resolution state. There is also a list of the email addresses of people who have asked to be kept up to date on the activity of the bug.

[3] `www.bugzilla.org/`, verified 26/08/05
[4] `www.gnu.org/software/gnats/`, verified 07/09/05
[5] `www.atlassian.com/software/jira/`, verified 07/09/05
[6] Personal information has been removed from the figure.

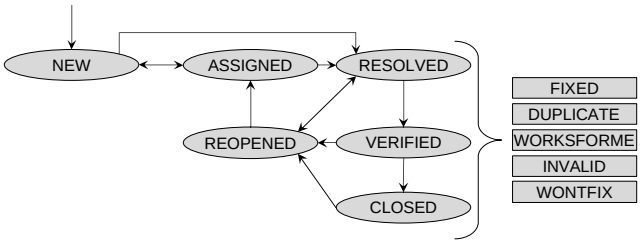


Figure 1: The life-cycle of an Eclipse bug report.

The free-form text includes the title of the report, a full description of the bug, and additional comments. The full description typically contains an elaborated description of the effects of the bug and any necessary information for a developer to reproduce the bug. The additional comments include discussions about possible approaches to fixing the bug, and pointers to other bugs that contain additional information about the problem or that appear to be duplicate reports.

Reporters and developers may provide attachments to reports to provide non-textual additional information, such as a screenshot of erroneous behaviour.

The bug repository tracks which bugs block the resolution of other bugs and the activity of each bug report. The activity log provides a historical report of how the report has changed over time, such as when the report has been reassigned, or when its priority has been changed.

2.1.2 The Life-cycle of a Bug Report

Bugs move through a series of states over their lifetime. We illustrate these states using the life-cycle of a bug report for the Eclipse bug project (Figure 1). Other projects vary slightly from this model. We describe such differences when necessary later in the paper.

When a bug report is submitted to the Eclipse repository, its status is set to NEW. Once a developer has been either assigned to or accepted responsibility for the report, the status is set to ASSIGNED. When a report is closed its status is set to RESOLVED. It may further be marked as being verified (VERIFIED) or closed for good (CLOSED). A report can be resolved in a number of ways; the resolution status in the bug report is used to record how the report was resolved. If the resolution resulted in a change to the code base, the bug is resolved as FIXED. When a developer determines that the report is a duplicate of an existing report then it is marked as DUPLICATE. If the developer was unable to reproduce the bug it is indicated by setting the resolution status to WORKSFORME. If the report describes a problem that will not be fixed, or is not an actual bug, the report is marked as WONTFIX or INVALID, respectively. A formerly resolved report may be reopened at a later date, and will have its status set to REOPENED.

2.1.3 Interactions with Bug Reports

People play different roles as they interact with reports in a bug repository. The person who submits the report is the *reporter* or the *submitter* of the report. The *triager* is the person who decides if the report is meaningful and who assigns responsibility of the report to a developer. The one that resolves the report is the *resolver*. A person that con-

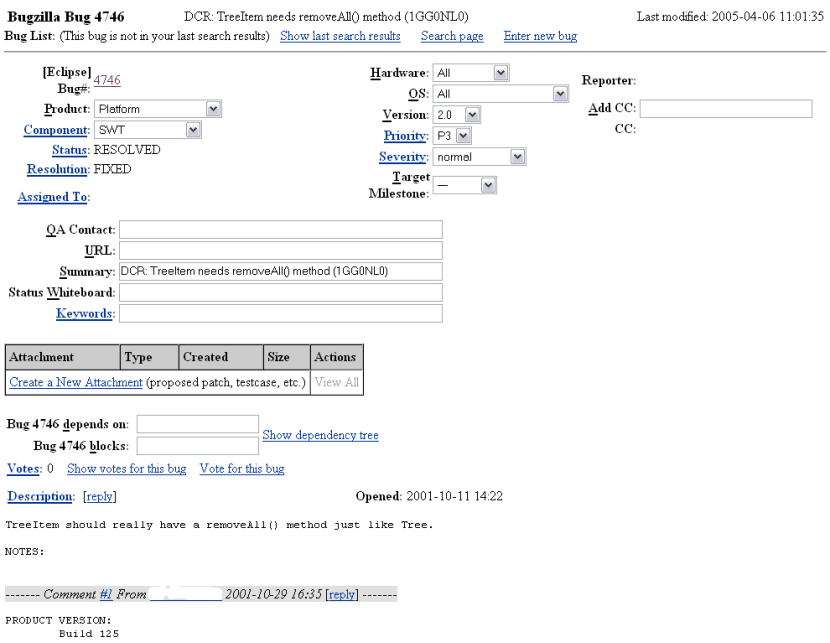


Figure 2: A sample Bugzilla bug report from Eclipse.

Table 1: Daily bug submissions around and after product release.

	Around Release			After Release		
	Mean	Min	Max	Mean	Min	Max
Eclipse	48	1	192	13	1	124
Firefox	8	1	37	5	1	37

tributes a fix for a bug is called a *contributor*. A contributor may also contribute comments about how to resolve a bug or additional information that leads to the resolution of a report.

A person may assume any one of these roles at any time. For example, a triager may resolve a report as the duplicate of an existing report. Alternatively, a developer may submit a report, assign it to himself, contribute a fix, and then resolve the report. For that report, a single person has fulfilled all the roles.

2.1.4 Bug Triage Today

In Section 1, we briefly outlined the number of reports triaged during a four month period for the Eclipse open source project. As a further analysis of the problem, Table 1 shows the rate at which bug reports were submitted for two time periods. The first time period ("Around Release") is the three months before and after the release dates for Eclipse V.3.0 (released on June 25, 2004) and Firefox V.1.0 (released on November 9, 2004). The second time period ("After Release") is between the project release dates and August 8, 2005. As one would expect, the average number of bugs submitted daily is higher around the release of a project.

Each project uses a different manual strategy for performing triage. Because of the volume of reports, reports submitted to the Mozilla bug repository[7] are triaged by quality assurance volunteers, rather than the developers. A triager from the project commented:

> Everyday, almost 300 bugs appear that need triaging. This is far too much for only the Mozilla programmers to handle.[8]

Early on, the Eclipse project had a single developer triage their bug reports. However, as the task became too overwhelming for a single person, the triaging was decentralized and now each component team monitors the bug report inbox for their component.[9] Regardless of the approach used, a manual approach to bug triage takes up resources that might be better applied to other problems within the development project.

2.2 Machine Learning: Text Categorization

The area of machine learning most related to bug triage is text categorization, which involves the classification of text documents into a set of categories [15]. For the bug assignment problem, the text documents are the bug reports and the categories into which reports are classified are the names of developers suitable to resolve the report.

In machine learning, the documents are called *instances* and the attributes of an instance are called *features*. Instances may also have a label that indicates the category, or *class*, to which it belongs. A supervised machine learning algorithm takes as input a set of instances with known labels

[7]The Mozilla project comprises many projects, of which Firefox is one, and they all share the same development process and bug repository. When speaking of the development process in general we refer to the Mozilla project, and refer to Firefox when speaking of that project specifically.
[8]Personal communication with M.W., 05/03/05
[9]Personal communication with D.H., 23/02/05

and generates a *classifier*. The generated classifier can then be used to assign a label to an unknown instance. The process of creating a classifier from a set of instances is known as *training* the classifier.

Our work focuses on the the use of supervised machine learning for bug assignment.

3. RELATED WORK

We are aware of only two other efforts in automated bug assignment. Čubranić and Murphy [4] also used a text categorization approach, and with a different algorithm achieved precision levels of around 30% on Eclipse. The approach described in this paper expands on this previous work with more thorough preparation of data, the use of additional information beyond the bug description, the exploration of more algorithms, and the determination of a better performing algorithm. Canfora and Cerulo [3] outline an approach based on information retrieval in which they report recall levels of around 20% for Mozilla. In this paper, we present an approach that achieves a higher level of precision for this project and describe when the approach may apply.

Podgurski et al. also applied a machine learning algorithm to bug reports, but in their case the algorithm was applied to cluster function call profiles from automated fault reports [10]. The clusters were used to prioritize software faults and to help diagnosis their cause rather than to assign reports to appropriate developers.

In trying to determine developers with expertise in particular parts of the system, the bug assignment problem is similar to the problem of recommending experts in particular parts of the system to assist with the development process. Mockus and Herbsleb's Expertise Browser system, for example, uses source code change data from a version control system to determine experts for given elements of a software project [9]. Our approach can be viewed as trying to recommend such experts but the recommendation is based on different data for a different purpose. Specifically, when we make a recommendation on experts to solve the report, we only have available the information in the report when it is filed, which is largely a free-form description of a problem or possible enhancement.

4. A SEMI-AUTOMATED APPROACH TO BUG ASSIGNMENT

Given a new bug report, our approach uses a supervised machine learning algorithm to suggest developers who may be qualified to resolve the bug. We recommend a small list of potential resolvers because groups of developers often work on similar kinds of problems. The recommendations we make are based on bug reports that the developers have previously been assigned or resolved for the system.

Our approach is semi-automated because the triager must select the actual developer from the recommended set to whom the bug will be assigned. The triager may make this choice based on knowledge other than that available in the bug repository, such as the workloads of the developers, or who is on vacation.

Our approach consists of four steps:

1. characterizing bug reports,
2. assigning a label to each report,
3. choosing reports to train the supervised machine learning algorithm, and
4. applying the algorithm to create the classifier for recommending assignments.

We used data from two Bugzilla repositories to develop the approach: the Eclipse platform project and Firefox. Specifically, our training sets consisted of reports from these projects that had been resolved or assigned between September 1, 2004 and May 31, 2005. The training sets included 8655 reports for Eclipse and 9752 for Firefox.

As is typical in machine learning, we evaluate the performance of our approach using the measures of precision and recall. Precision measures how often the approach makes an appropriate assignment recommendation for a report (Formula 1). Recall measures how many of the developers who may be appropriate to resolve the report are actually recommended (Formula 2). Appendix A presents the details of our evaluation process. Using this process our test sets consisted of 170 reports for Eclipse and 22 reports for Firefox from May 2005 that met particular characteristics. We report the average precision and recall over all reports in the test set.

$$Precision = \frac{\#\ of\ appropriate\ recommendations}{\#\ of\ recommendations\ made} \quad (1)$$

$$Recall = \frac{\#\ of\ appropriate\ recommendations}{\#\ of\ possibly\ relevant\ developers} \quad (2)$$

4.1 Characterizing Bug Reports

Our approach requires an understanding of which bug reports are similar to each other so that we can learn the kinds of reports typically resolved by each developer. In the context of machine learning, this requirement translates to picking features to characterize a bug report. Reports with similar features can then be grouped.

As described in Section 2.1.1, each bug report contains a substantial amount of information. Our approach uses the one-line summary and full text description to characterize each report as they uniquely describe each report.

Before we can apply a machine learning algorithm to the free-form text found in the summary and description, the text must be converted into a feature vector. We follow the standard approach of first removing all stop words and non-alphabetic tokens [1]. The remaining words are used to create a feature vector indicating the frequency of the terms in the text. We then normalize the frequencies based on document length, intra-document frequency and inter-document frequency as outlined by Rennie et al. [14].

We chose not to use stemming[10] because earlier work [4] showed that it had little effect.

4.2 Labeling Bug Reports

To train the classifier, we need to provide a set of reports that are labeled with the name of the developer who was either assigned to the report or who resolved it. New, unconfirmed, or reopened reports are not labeled as it is unknown who will handle the report.

[10]Stemming identifies grammatical variations of a word, such as 'see', 'seeing', and 'seen', and treats them as a being the same word.

At first glance, this step seems trivial as it seems obvious to use the value of the `assigned-to` field in the bug report. However, the problem is not this simple because each project tends to use the `status` and `assigned-to` fields of a bug report differently. For example, in both the Eclipse platform and Firefox projects, the value of the `assigned-to` field does not initally refer to a specific developer. Instead new and unconfirmed reports are first assigned to a default email address before they are assigned to an actual developer.[11] For reports with a trivial resolution, such as duplicate, the `assigned-to` field is often never changed.

Instead of using the `assigned-to` field, we use project-specific heuristics to label the reports. These heuristics can be derived either from direct knowledge of a project's process or by examining the logs of a random sample of bug reports for the project. We took the latter approach for the Eclipse platform and Firefox projects resulting in a set of heuristics, four of which we provide here.[12]

- If a report is resolved as FIXED, it was fixed by whoever submitted the last approved patch. (Firefox)
- If a report is resolved as FIXED, it was fixed by whoever marked the report as resolved. (Eclipse)
- If a report is resolved as DUPLICATE, it was resolved by whoever resolved the report of which this report is a duplicate. (Eclipse and Firefox)
- If a report is resolved as WORKSFORME, it was marked by the triager, and it is unknown which developer would have been assigned the report. The report is thus labeled as unclassifiable. (Firefox)

4.3 Selecting Training Reports

We first refine the set of training reports by filtering out those for which the project-specific heuristics cannot provide a useful label. For the Eclipse project, only 1% of the reports could not be labeled. In contrast, 49% of the Firefox reports could not labeled due to a large proportion (47%) of the training reports being marked as WORKSFORME, WONTFIX, INVALID, or the duplicate of a NEW bug report.

Even after this filtering, we must still refine the training set further to remove reports that provide information about developers who no longer work on the project and developers who have only fixed a small number of bugs. We filter for the former because it is not useful to recommend a developer who is no longer available for assignment, and we filter for the latter because the classifier does not have sufficient data to learn about a low-activity developer.

We base the second refinement on profiles of each developer's activity. Through experimentation, we have found that an effective filter is to remove the reports from any developer that has not contributed at least nine bug resolutions in the most recent three months of the project. Table 2 shows the effect of various developer activity profiles on our recommendations for the Eclipse and Firefox projects. We choose an average value of nine resolutions over three months (corresponding to the shaded row in Table 2) because this value provides the greatest stability in the precision/recall results over both of the projects.[13] This profile filtering removes 4% of the Eclipse reports and 29% of the Firefox reports.

Table 2: The effect of developer profile filtering on recommender accuracy and recall.

		# Dev.		Precision/Recall (%)	
		Firefox	Eclipse	Firefox	Eclipse
No Profile		414	146	23/1	58/7
>1 Fix in 3 mo.		94	82	59/2	57/7
Avg. Fixes Per Month Over 3 mo.	1	66	50	64/2	57/7
	2	33	42	59/2	57/7
	3	26	40	64/2	57/7
	4	21	40	64/2	58/7
	5	18	39	59/2	59/7
	6	13	37	45/1	57/7

4.4 Applying a Machine Learning Algorithm

We use Support Vector Machines (SVM) [6] in our bug assignment approach and we recommend presenting one recommendation to the human triager. We chose this algorithm and recommendation set size after comparing the results of three algorithms: Naïve Bayes [8], SVM, and C4.5 [11]. We evaluated Naïve Bayes, a probabilistic classification algorithm, because it was used in the previous effort to automate bug assignment [4] and thus provided a lower bound for finding a more appropriate algorithm. We evaluated SVM, a non-linear classification algorithm, because it has been shown to be effective for text categorization [7]. Finally, we evaluated C4.5, a popular decision tree algorithm, because the methods used by human triagers are similar to that of traversing a decision tree. Table 3 shows the precision and recall of each of these three different classifiers for increasing numbers of recommendations for each project. The data to train the classifier was prepared according to the techniques that we have outlined in this section.

5. VALIDATING WITH GCC

To determine how our approach performs on a project other than those that we used to tune the approach, we measured our ability to provide bug assignment recommendations for the gcc compiler project.[14] To train our approach for gcc, we used 2629 reports from September 2004 to April 2005, after applying the filtering described in Section 4.3.[15] We tested the approach with 194 reports from May 2005. Table 4 reports the results. Disappointingly, our results were much lower than we anticipated with a precision rate of 6% for one recommendation and 18% for two or three recommendations. We believe that this lower than expected precision value is due to both characteristics of the

[11] A user name in the Bugzilla system is an email address.

[12] The full set of heuristics is available on-line at `http://www.cs.ubc.ca/labs/spl/projects/bugTriage/assignment/heuristics.html`.

[13] We believe that the precision and recall values for Firefox with an average of one fix per developer per month is an over-estimate due to the relatively small number of reports in the test set.

[14] `www.gnu.org/software/gcc/gcc.html`, verified 26/08/05

[15] Initally there were 3440 reports; 16% were removed for inactive developers and 8% were removed for not being labeled by our heuristics.

Table 3: The effect of different machine learning algorithms on recommender accuracy and recall.

Predictions	Naïve Bayes		SVM		C4.5	
	Firefox	Eclipse	Firefox	Eclipse	Firefox	Eclipse
1	59/2	54/6	64/2	58/7	64/2	40/5
2	59/2	49/11	52/3	52/13	41/3	34/9
3	59/2	44/15	57/6	47/16	42/5	31/12

Table 4: Precision and recall for gcc.

Predictions	Precision/Recall
1	6/0.3
2	18/2
3	18/3

project that make our approach unsuitable, and difficulties in accurately measuring precision and recall.

Project Characteristics We believe our approach may not have performed well for gcc for three reasons. First, one developer seems to dominate the bug resolution activity. Based on our labeling heuristics, the most active developer was assigned-to or resolved 1394 reports compared to the next most active developer at 160 reports. The classifier weights the active developer's category more heavily, making a new report eight times more likely to be similar to a report in that category than any other. If the reports with this developer's label are removed from the training set so that the distribution of reports is more even amongst the classes, then the precision improves to 30%.

Second, our labeling heuristics may not be sufficiently accurate.[16] In the gcc project, an automated system enters comments into the bug reports describing who has checked-in code to resolve the report. If we use this data, which goes beyond the normal data we expect in a bug report, we can improve the labeling such that the most active developer is seen by the classifier to resolve 956 reports and the second most active developer resolves 163 reports. Using this CVS information to label reports results in a modest increase in precision for one recommendation to 11%.

Third, the spread of bug resolution activity was low. Our filtering, for example, removed 63 developer names, resulting in a classifier that knew about only 29 developers. It may be that the size of the project is too small and the spread of the bug resolution activity across developers is too skewed to be suitable for our approach.

Measuring Precision and Recall We measured precision and recall according to Formulas 1 and 2 presented earlier. To calculate these measures, similar to the Eclipse and Firefox projects, we had to estimate which developers may have had the expertise to resolve a particular bug (Appendix A). For gcc, we based this estimate on data from the CVS logs and automatically generated CVS comments found in the bug reports from May 2005. The difficulty we encountered was in mapping the user names found in the CVS logs to the email addresses used by Bugzilla and our recommender. We were not able to map 32 of the 84 user names found in the CVS logs to an email address. As precision depends on the accuracy of a mapping, unresolved CVS user names lower precision.

[16]Available on-line at `http://www.cs.ubc.ca/labs/spl/projects/bugTriage/assignment/heuristics.html`.

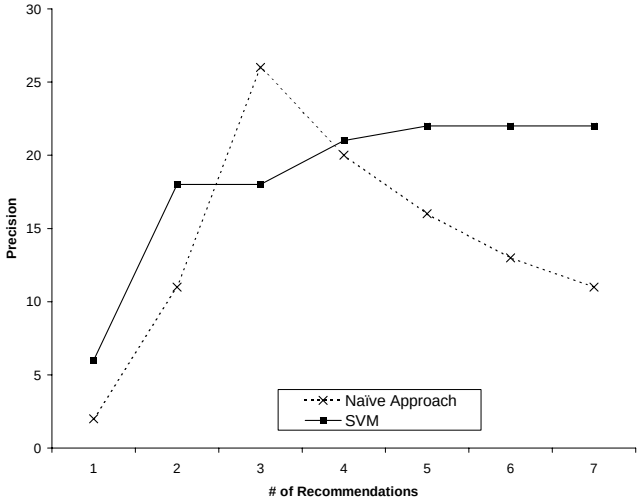


Figure 3: Using different approaches for a gcc recommender.

Using an Alternate Approach Given that a single developer seems to dominate the resolution activity for gcc, a naïve approach that recommends developers simply based on the number of reports that they have resolved may work. Figure 3 shows the results of applying this approach to the gcc data. As the figure shows, the naïve approach performs worse in general than SVM.

6. DISCUSSION

On two large software projects, Eclipse and Firefox, our approach achieved precision rates of greater than 50%. However, for gcc, the precision rate with one recommendation is only 6%. In this section, we discuss whether the precision and recall rates we are achieving on Eclipse and Firefox are good enough, and how we might better be able to judge when the approach applies. We also report on various extensions we have considered to the approach, and lessons we have learned in applying a machine learning approach to bug report information.

6.1 Assessing the Value of our Approach

6.1.1 Is >50% Precision Good Enough?

The only sure way to know whether the 57% and 64% precision rates we achieve for Firefox and Eclipse respectively help human triagers is to perform an empirical study in which we put the approach into the hands of the triagers. We believe that the precision rates we report in this paper for the two larger projects are sufficient to warrant such a study and we plan one as part of our future work.

Table 5: The size of developer groups per project.

	Min	Mean	Max
Eclipse	1	10	23
Firefox	10	30	53
gcc	2	12	34

We believe our approach warrants further study because of the context in which it can be applied. Rather than trying to completely automate bug assignment, we envision the approach being used to help new triagers make better assignments faster, to potentially reduce the time spent in performing triage, and to possibly help make the right assignment the first time. In the Eclipse project, for instance, 24% of reports are currently re-assigned before the bug is resolved. Each re-assignment requires a new developer to become acquainted with the problem. It may be that our approach can reduce some of these re-assignments, regaining time for developers to complete already assigned tasks on the system.

6.1.2 Why is the Recall so Low?

It is commonly known that for classification there is a trade-off between precision and recall. Even with this trade-off, the recall values we report are quite low, often hovering at a few percent. These values are low because of the way that we calculate recall: the set of values we report is a subset of those that we determine may be able to possibly resolve the report. Table 5 shows the minimum, average, and maximum groups size we determined for the reports used for evaluation in each project. When we report only one recommendation, the highest recall we can achieve on average is 10% (Eclipse), 3% (Firefox) and 8% (gcc).

6.1.3 Applicability

Machine learning algorithms generally produce better results the more data there is available from which to learn. Figures 4(a) and 4(b) show respectively the accuracy and recall of our approach for differing amounts of reports per developer. To form the graphs we selected from the Eclipse data set those developers that had either been assigned to or resolved at least 200 reports. The Eclipse data was used as there were 14 developers that met this criteria; the other data sets had only a few. Classifiers were then trained with increasing numbers of reports per developer, and evaluated using the Eclipse test set. Since we used the full Eclipse test set instead of including only the test reports for the selected developers, the precision and recall in Figure 4 are lower than was seen previously (Table 3). However, as we are interested in the shape of the line, the values are not as important as the relative differences between trials. We found that at roughly 60 reports per developer the line plateaus. This may have contributed to the poor performance of our approach for gcc. Although there appeared to be sufficient data based on the total number of reports, with 39% of the reports being for one developer, only 10 of the 29 developers had more than 60 reports for training.

The applicability of our approach also depends on how well labels can be assigned to the reports selected for the training set. For the Eclipse and gcc projects, there were few reports that our heuristics could not label with a developer. In contrast, many of the reports from the Firefox project could not be assigned a label. The reports that could not be labeled for this project consisted of reports resolved as WONTFIX, WORKSFORME, INVALID, or the duplicate of NEW, UNCONFIRMED, or REOPENED bug reports. From the 9647 potential reports that could have been used for training, 4884 had to be removed because they could not be labeled with a developer's name. Compare this to the Eclipse project where only 70 of 8117 could not be labeled, or gcc where 259 of 3440 could not be labeled.

The amount of data available is one way to determine if the approach might be useful. Initally, we had chosen the Apache Ant project[17] for validation. However, an examination of the number of reports assigned or resolved from September 2004 to April 2005 resulted in 423 reports for 30 developers. We concluded that the project contained insufficient data to warrant the use of our approach, and the gcc project was selected instead. That project showed us that data is not enough, it also must have a particular profile, such as the resolved reports not being dominated by one developer. Part of our future work involves a better determination of the characteristics the reports in the bug repository need to exhibit for the approach to be expected to give good results.

6.1.4 Evaluation Using Cross Validation

The standard method for evaluating a machine learning technique is *ten-fold stratified cross validation* [17]. To evaluate a bug assignment recommender using this technique, the reports used for training would be partitioned into ten sets with random reports in each set such that each set would contain a proportionate number of reports for each developer (e.g., if a developer had ten reports, each partition would contain one of these reports). We chose not to use this approach because of the sparseness of the bug assignment information. Even for the active developers considered in our approach (see Section 4.3), there are still an insufficient number of training reports. For the Firefox and gcc projects, 75% and 86% of the developers have less than 100 reports with which to train a classifier. Holding back of a tenth of these reports for use as a test set represents a substantial loss of information for those developers, particularly if those developers work on a range of problems.

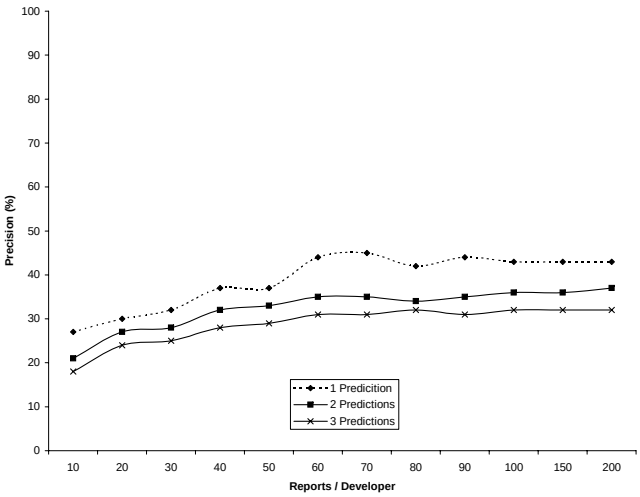
6.2 Extensions and Alternatives

6.2.1 Unsupervised Machine Learning

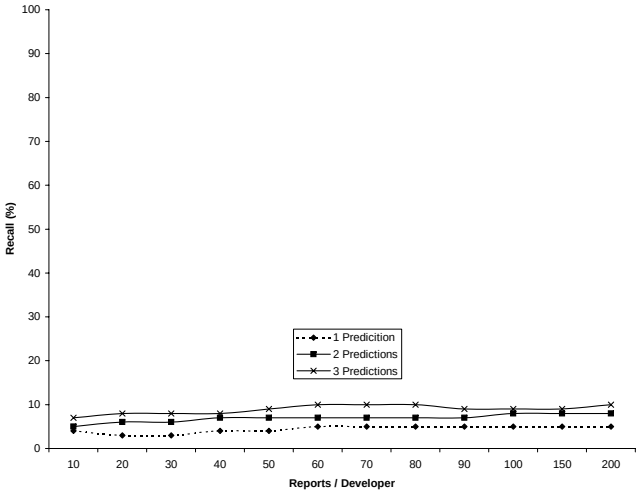
We chose to use a supervised machine-learning algorithm to recommend bug assignments. This choice means that we must provide a label for each report or not use the report for training or testing.

Alternatively, we could incorporate an unsupervised algorithm, such as Expectation Maximization [17], which does not require report labeling. Čubranić and Murphy [4] suggested the use of such an algorithm as a way to include reports that cannot be labeled. The algorithm could also be used to avoid the creation of project-specific heuristics. However, in tests that we performed using Expectation Maximization, we found it performed worse than Naïve Bayes and as a result, we abandoned the use of this algorithm.

[17] `http://ant.apache.org/` (verified 07/09/05).



(a) Precision with varying amounts of developer data.



(b) Recall with varying amounts of developer data.

Figure 4: Precision and recall for classifiers trained with differing amounts of data from the Eclipse project.

6.2.2 Incremental Machine Learning

The approach we present in this paper trains the classifier using a batched set of data. Alternatively, an incremental algorithm could be used whereby instances are provided one at a time to the classifier and the classifier updates itself appropriately [16]. An incremental classifier would mimic how information flows in and alters a bug repository. It also allows the classifier to incorporate new team members or changes between teams more quickly. However, an initial investigation that we performed that used an incrementally-updated Naïve Bayes classifier resulted in a recommender that only achieved a maximum of 28% average accuracy after 400 instances were added and degraded thereafter when using four months of the Eclipse data.

6.2.3 Incorporating Additional Sources of Data

Our approach uses only data extracted from bug reports. The accuracy of our approach might be improved by using additional data sources.

An example of an additional data source is code ownership, such as from an *ownership architecture* [2]. When coupled with execution stack traces from bug reports that would provide a pointer to the code of interest for a report, code ownership may lead to better assignment recommendations. However, this approach is not likely to greatly improve the precision as we found that only 11% (1273 of 11223) of reports from the Eclipse project contained exception traces. Also exception traces are notoriously misleading as to the cause of the problem, so in fact including them in the learning process may degrade the accuracy.

The developer profiles from Section 4.3 were created by a statistical analysis of resolved reports in a bug repository. Developer profiles created via surveys could provide more accurate information about developers' experience and expertise with the system. However, creating such profiles is labour-intensive and the profiles quickly become outdated. Another means of acquiring developer profiles may be an approach such as used by the Expertise Browser [9] which uses *experience atoms* derived from source code. However, as explained in Section 6.3 correlating source code changes to bug reports is challenging for some projects.

Table 6: Number of components for which a developer has resolved a bug report.

	Project Components	Min	Mean	Max
Eclipse	17	1	1.8	9
Firefox	30	1	2.9	29
gcc	30	1	3.2	28

Table 7: Results of using a component-based classifier.

Predictions	Eclipse	Firefox	gcc
1	86/12	64/2	6/0.4
2	82/23	64/5	10/1
3	77/32	53/6	10/2

6.2.4 A Component-based Classifier

A natural structure for project teams is based on the components of a system. The Eclipse project is structured in this way. However, based on our heuristics, developers appear to handle bugs outside of their particular component. Table 6, for instance, shows the number of components comprising each project and the number of components for which developers typically resolve reports. The data shows that Eclipse developers stick closer to component boundaries with a maximum of nine components altered for a single bug than those from the other two projects.

As mentioned in Section 2.1.4, the Eclipse project first triages according to the component that the report is against, and then reports are triaged by the component team. Ta-

Table 8: Number of developers per component.

	Min	Mean	Max
Eclipse	1	8	30
Firefox	4	11	25
gcc	1	7	18

ble 7 shows the results of combining our approach with this process. We first grouped the data by reported component, and then use the groups of data to train a recommender for each component. Again, a developer profile of an average of three resolutions for each of the past three months was used. For a new report, the recommender uses the appropriate component-recommender to generate the recommendation. Table 8 shows the minimum, mean, and maximum number of developers for the component-recommenders trained for each project.

Table 7 shows that for the Eclipse project the accuracy improves, but that this component-centric approach provides no improvements for Firefox and gcc. As the component-centric approach mimics the process used by Eclipse, this is not surprising.

6.3 Lessons Learned

In developing our approach we learned a number of lessons about working with bug repository data. First, coordinating data from bug reports and CVS logs is challenging. To evaluate our approach we compared the user names from the bug reports to those found in the CVS logs to determine if our approach recommends an appropriate developer. Unfortunately, the user name in bug reports is the email address of a developer, while for Eclipse and gcc a user name in CVS is a login id for a system such as UNIX. To match the two, we had to use a mapping to enable a proper comparison for these two projects. Some mappings were trivial as part of the email address matched a login name for CVS. For Eclipse, we were able to match 69 of 83 (83%) CVS user names because of similarity in the email addresses and login names, for gcc, we were only able to match 61 of 84 (73%). This problem could be solved by a small change in process that enabled the determination of the identity of each developer across repositories by gathering mapping information or changing the identifiers in one of the repositories.

The Firefox project presented a different challenge when we tried to correlate bug reports and CVS logs. Although the user name in a CVS log is the email address of the submitter, the submitter is usually not the one who created the patch. As mentioned previously, Firefox employs a process whereby patches are reviewed before being incorporated into the source tree, and only a select group of people are authorized to check in to CVS. Consequently, when a patch is approved, the contributor often has to request that someone check in the patch, and the user name in CVS does not reflect who actually fixed the problem. To overcome this obstacle, we created a map between resolvers and fixers based on who marked the bug report resolved and who our heuristic identified as the person who fixed the problem. We assumed that the person who marked the report as resolved was the person who performed the check-in.

Another challenge that we encountered in coordinating bug reports and CVS logs is the inconsistent inclusion of references to bug report ids in the CVS comments. As the Eclipse developers were more consistent in including the bug id in their check-in comments, we were able to acquire more data for evaluating our approach with the Eclipse project, then with the Firefox project where the inclusion of the bug id was more sporadic. Again this problem could be corrected by a small procedural change whereby the inclusion of bug ids is required for CVS check-ins. The gcc project employs an alternate technique whereby a CVS comment is automatically included in the bug report showing who submitted the changes and what files were changed.

To increase the number of bug reports for our evaluation of the Firefox project, we tried to correlate reports and CVS check-ins based on a time window [5]. From the bug reports that were fixed in May 2005, we gathered CVS check-in entries in an 8-hour window around the time that the report was resolved and correlated that with the user name of the person who marked the report as resolved. Unfortunately, this resulted in a fewer number of test reports than our previous approach. Upon closer examination we found cases of a one month difference between when the report was marked as resolved and when the check-in occurred, and different bug ids associated with fixes.

Lastly, we report on one perhaps unexpected lesson related to the collection of data for projects like this one. As our approach required a large quantity of data from bug repositories, we used an automated script to extract the data from the bug repositories. At one point during our data collection, the computer that was running the script was black-listed by the Eclipse bug repository firewall as our crawling was mistaken for a server attack. The situation was corrected after contacting the webmaster for the Eclipse site, but required us to be more vigilant in how often we were querying the repository.

7. SUMMARY

In this paper, we have presented an approach to semi-automating the assignment of a bug report to a developer with the appropriate expertise to resolve the report. Our approach uses a supervised machine learning algorithm that is applied to information in the bug repository. For the Eclipse and Firefox projects, we are able to achieve precision rates of over 50%, reaching 64% on one recommendation for Firefox. Our results for the gcc project were far worse, where we achieved a precision rate for one recommendation of only 6% because of characteristics of the project, such as one developer dominating the report resolution process. In addition to presenting our approach and results, we have presented an in-depth analysis of the application of machine learning to this problem and we have reported on lessons learned in trying to make use of data in the bug repository.

We believe that our approach shows promise for improving the bug assignment problem for open source software developments. Our future plans include an empirical study of the use of the approach by bug triagers on an open source system, an investigation of additional sources of information, and a prescriptive means for determining when the approach may be applicable.

8. ACKNOWLEDGMENTS

This research was funded in part by NSERC and in part by an IBM Eclipse Innovation Grant.

APPENDIX

A. EVALUATION PROCESS FOR ECLIPSE AND FIREFOX

To evaluate the approach on the Eclipse and Firefox projects, we needed to know for each bug report in the test set which developers on the project might have had the relevant expertise and might have been assigned to resolve the report. Because we did not have access to experts that could provide this information, we developed a heuristics for each project based on information in the source revision (CVS) repositories.

First, we formed the test set as consisting of all bugs in a given time period (May 2005) for which the revision's log entry specified that it resolved a specific bug. Next, we associated with each bug in the test set, the names of the modules that contained the revisions specifying the bug number. The meaning of a containing module for a revision varied with each project. For the Eclipse platform project, we used the component as the module, which resulted in 316 modules. For the Firefox project, we used the sub-component level of organization of the project as the containing module, which resulted in 1739 modules. We then compiled a list of developers who had previously committed changes to these modules by analyzing the revision history of each file in the module. We considered this list of developers as potential experts to resolve the bug because they had knowledge of this part of the system. We used the notion of containing module to derive this information because we felt that limiting the list of relevant developers to particular files that were touched was too constraining.

The result of this processing for each bug in the test set is a list of developer user names from the source revision repository. However, in the bug repository, developers are identified by a separately chosen login, often one of their email addresses. Because of the differences in these identifiers, we had to construct a mapping between the identifiers used in the bug repository and in the source repository. Differences in the conventions used in each project meant a separate manual mapping method was used in each case. The Eclipse mapping was relatively straightforward as the user names and emails differed mostly in abbreviation formats. For Eclipse, we were able to map 69 out of 83 user names used in the source revision repository. The mapping for Firefox was more challenging because the project uses a process in which fixes are reviewed and contributers must request that their changes be checked in by a committer (Section 6.3). As a result, the person who checks in a fix is rarely the person that made the fix. In constructing a mapping for this project, we assumed that the person who marked a report as resolved is generally also the person who checked in the fix. For Firefox, we were able to map 220 resolver emails from the bug reports in the training set to a list of possible fixers for which they submitted revisions. There were 35 resolver email addresses for which we were unable to create a mapping for from the CVS logs.

B. REFERENCES

[1] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. 1999.

[2] I. T. Bowman and R. C. Holt. Reconstructing ownership architectures to help understand software systems. In *Proceedings of International Workshop on Program Comprehension*, pages 28–37, 1999.

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

[4] D. Čubranić and G. C. Murphy. Automatic bug triage using text classification. In *Proceedings of Software Engineering and Knowledge Engineering*, pages 92–97, 2004.

[5] D. Čubranić, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, 2005.

[6] S. R. Gunn. Support Vector Machines for classification and regression. Technical report, University of Southampton, Faculty of Engineering, Science and Mathematics; School of Electronics and Computer Science, 1998.

[7] T. Joachims. Text categorization with support vector machines: Learning with many relevant features. In *Proceedings of the 10th European Conference on Machine Learning*, pages 137–142, 1998.

[8] G. H. John and P. Langley. Estimating continous distributions in Bayesian classifiers. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 338–345, 1995.

[9] A. Mockus and J. D. Herbsleb. Expertise browser: A quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering*, pages 503–512, 2002.

[10] A. Podgurski, D. Leon, P. Francis, Wes Masri, M. Minch, Jiayang Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering*, pages 465–475, 2003.

[11] R. Quinlan. *C4.5: Programs for Machine Learning*. 1993.

[12] E. S. Raymond. The cathedral and the bazaar. *First Monday*, 3(3), 1998.

[13] C. R. Reis and R. P. de Mattos Fortes. An overview of the software engineering process and tools in the Mozilla project. In *Proceedings of the Open Source Software Development Workshop*, pages 155–175, 2002.

[14] J. D. M. Rennie, L. Shih, J. Teevan, and D. R. Karger. Tackling the poor assumptions of Naive Bayes classifiers. In *Proceedings of International Conference on Machine Learning*, pages 616–623, 2003.

[15] F. Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys*, 34(1):1–47, 2002.

[16] R. Segal and J. Kephart. Incremental learning in SwiftFile. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 863–870, 2000.

[17] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools with Java Implementations*. 2000.